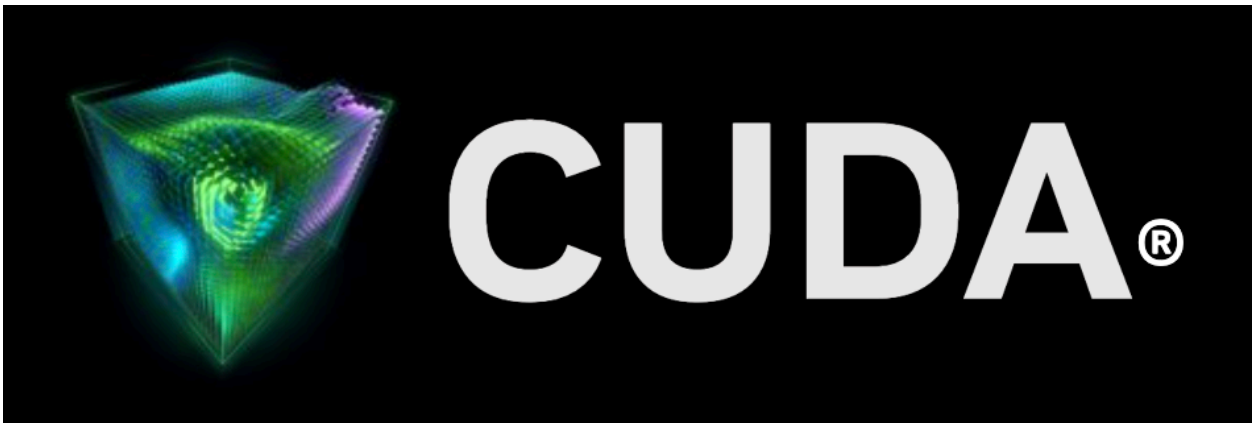


CUDA Runtime API Reference Manual

Release 13.5

NVIDIA Corporation

Nov 02, 2026

Contents

1 Difference between the driver and runtime APIs **1**

2 API Synchronization behavior **3**

3 Stream synchronization behavior **5**

4 Graph object thread safety **7**

5 Rules for version mixing **9**

6 APIs **11**

6.1 C++ API Routines . . . 11
6.1.1 Functions . . . 20
6.2 Data types used by CUDA Runtime . . . 71
6.2.1 Macros . . . 85
6.2.2 Enumerations . . . 92
6.2.3 Typedefs . . . 169
6.3 Device Management . . . 171
6.3.1 Functions . . . 173
6.4 Device Management [DEPRECATED] . . . 200
6.4.1 Functions . . . 201
6.5 Direct3D 10 Interoperability . . . 203
6.5.1 Enumerations . . . 203
6.5.2 Functions . . . 204
6.6 Direct3D 10 Interoperability [DEPRECATED] . . . 207
6.6.1 Enumerations . . . 208
6.6.2 Functions . . . 208
6.7 Direct3D 11 Interoperability . . . 217
6.7.1 Enumerations . . . 218
6.7.2 Functions . . . 218
6.8 Direct3D 11 Interoperability [DEPRECATED] . . . 221
6.8.1 Functions . . . 222
6.9 Direct3D 9 Interoperability . . . 223
6.9.1 Enumerations . . . 223
6.9.2 Functions . . . 224
6.10 Direct3D 9 Interoperability [DEPRECATED] . . . 228
6.10.1 Enumerations . . . 229
6.10.2 Functions . . . 230
6.11 Driver Entry Point Access . . . 238
6.11.1 Functions . . . 239
6.12 EGL Interoperability . . . 242
6.12.1 Functions . . . 243
6.13 Error Handling . . . 249
6.13.1 Functions . . . 250

6.14 Error Log Management Functions . . . 252
6.14.1 Functions . . . 253
6.14.2 Typedefs . . . 255
6.15 Event Management . . . 255
6.15.1 Functions . . . 256
6.16 Execution Context Management . . . 263
6.16.1 Overview . . . 263
6.16.2 Lifetime of CUDA Resources . . . 264
6.16.3 APIs Operating on Current Context . . . 264
6.16.3.1 APIs Operating on Device Resources . . . 264
6.16.3.2 APIs Operating on Context Resources . . . 264
6.16.4 Green Contexts . . . 264
6.16.5 Functions . . . 267
6.17 Execution Control . . . 281
6.17.1 Functions . . . 282
6.18 Execution Control [DEPRECATED] . . . 296
6.18.1 Functions . . . 297
6.19 External Resource Interoperability . . . 298
6.19.1 Functions . . . 299
6.20 Graph Management . . . 311
6.20.1 Functions . . . 318
6.21 Graphics Interoperability . . . 411
6.21.1 Functions . . . 412
6.22 Interactions with the CUDA Driver API . . . 418
6.22.1 Execution Contexts . . . 418
6.22.2 Primary Context (aka Device Execution Context) . . . 418
6.22.3 Initialization and Tear-Down . . . 418
6.22.4 CUcontext Interoperability . . . 419
6.22.5 Interactions between CUstream and cudaStream_t . . . 419
6.22.6 Interactions between CUevent and cudaEvent_t . . . 419
6.22.7 Interactions between CUarray and cudaArray_t . . . 420
6.22.8 Interactions between CUgraphicsResource and cudaGraphicsResource_t . . . 420
6.22.9 Interactions between CUtexObject and cudaTextureObject_t . . . 420
6.22.10 Interactions between CUsurfObject and cudaSurfaceObject_t . . . 420
6.22.11 Interactions between CUfunction and cudaFunction_t . . . 421
6.22.12 Interactions between CUkernel and cudaKernel_t . . . 421
6.22.13 Functions . . . 421
6.23 Library Management . . . 422
6.23.1 Functions . . . 423
6.24 Logical Endpoint . . . 430
6.24.1 Functions . . . 430
6.25 Memory Management . . . 431
6.25.1 Functions . . . 435
6.26 Memory Management [DEPRECATED] . . . 508
6.26.1 Functions . . . 509
6.27 Occupancy . . . 514
6.27.1 Functions . . . 515
6.28 OpenGL Interoperability . . . 521
6.28.1 Enumerations . . . 522
6.28.2 Functions . . . 522
6.28.3 Typedefs . . . 525
6.29 OpenGL Interoperability [DEPRECATED] . . . 525
6.29.1 Enumerations . . . 526
6.29.2 Functions . . . 527

6.30 Peer Device Memory Access . . . 531
6.30.1 Functions . . . 532
6.31 Profiler Control . . . 534
6.31.1 Functions . . . 534
6.32 Stream Management . . . 535
6.32.1 Functions . . . 537
6.32.2 Typedefs . . . 557
6.33 Stream Ordered Memory Allocator . . . 558
6.33.1 overview . . . 558
6.33.2 Supported Platforms . . . 558
6.33.3 Functions . . . 559
6.34 Surface Object Management . . . 571
6.34.1 Functions . . . 572
6.35 Texture Object Management . . . 573
6.35.1 Functions . . . 574
6.36 Unified Addressing . . . 582
6.36.1 Overview . . . 582
6.36.2 Supported Platforms . . . 583
6.36.3 Looking Up Information from Pointer Values . . . 583
6.36.4 Automatic Mapping of Host Allocated Host Memory . . . 583
6.36.5 Direct Access of Peer Memory . . . 583
6.36.6 Exceptions, Disjoint Addressing . . . 584
6.36.7 Functions . . . 584
6.37 VDPAU Interoperability . . . 585
6.37.1 Functions . . . 586
6.38 Version Management . . . 589
6.38.1 Functions . . . 589

7 Structs **593**
7.1 CUuuid_st . . . 593
7.2 cudaAccessPolicyWindow . . . 593
7.3 cudaArrayMemoryRequirements . . . 594
7.4 cudaArraySparseProperties . . . 594
7.5 cudaAsyncNotificationInfo_t . . . 595
7.6 cudaChannelFormatDesc . . . 596
7.7 cudaChildGraphNodeParams . . . 596
7.8 cudaConditionalNodeParams . . . 597
7.9 cudaDevResource . . . 598
7.10 cudaDevSmResource . . . 599
7.11 cudaDevSmResourceGroupParams . . . 599
7.12 cudaDevWorkqueueConfigResource . . . 600
7.13 cudaDevWorkqueueResource . . . 601
7.14 cudaDeviceProp . . . 601
7.15 cudaEglFrame . . . 607
7.16 cudaEglPlaneDesc . . . 608
7.17 cudaEventRecordNodeParams . . . 609
7.18 cudaEventWaitNodeParams . . . 610
7.19 cudaExtent . . . 610
7.20 cudaExternalMemoryBufferDesc . . . 610
7.21 cudaExternalMemoryHandleDesc . . . 611
7.22 cudaExternalMemoryMipmappedArrayDesc . . . 612
7.23 cudaExternalSemaphoreHandleDesc . . . 613
7.24 cudaExternalSemaphoreSignalNodeParams . . . 614
7.25 cudaExternalSemaphoreSignalNodeParamsV2 . . . 615

7.26 cudaExternalSemaphoreSignalParams . . . 615
7.27 cudaExternalSemaphoreWaitNodeParams . . . 616
7.28 cudaExternalSemaphoreWaitNodeParamsV2 . . . 616
7.29 cudaExternalSemaphoreWaitParams . . . 617
7.30 cudaFuncAttributes . . . 618
7.31 cudaGraphEdgeData . . . 621
7.32 cudaGraphExecUpdateResultInfo . . . 622
7.33 cudaGraphInstantiateParams . . . 622
7.34 cudaGraphKernelNodeUpdate . . . 623
7.35 cudaGraphNodeParams . . . 624
7.36 cudaGraphRecaptureCallbackData . . . 625
7.37 cudaHostNodeParams . . . 626
7.38 cudaHostNodeParamsV2 . . . 626
7.39 cudaIpcEventHandle_t . . . 627
7.40 cudaIpcMemHandle_t . . . 627
7.41 cudaKernelNodeParams . . . 627
7.42 cudaKernelNodeParamsV2 . . . 628
7.43 cudaLaunchAttribute . . . 629
7.44 cudaLaunchConfig_t . . . 629
7.45 cudaLaunchMemSyncDomainMap . . . 630
7.46 cudaMemAccessDesc . . . 630
7.47 cudaMemAllocNodeParams . . . 631
7.48 cudaMemAllocNodeParamsV2 . . . 631
7.49 cudaMemFabricHandle_t . . . 632
7.50 cudaMemFreeNodeParams . . . 632
7.51 cudaMemLocation . . . 632
7.52 cudaMemPoolProps . . . 633
7.53 cudaMemPoolPtrExportData . . . 634
7.54 cudaMemcpy3DBatchOp . . . 634
7.55 cudaMemcpy3DOperand . . . 635
7.56 cudaMemcpy3DParms . . . 636
7.57 cudaMemcpy3DPeerParms . . . 637
7.58 cudaMemcpyAttributes . . . 637
7.59 cudaMemcpyBatchNodeParams . . . 638
7.60 cudaMemcpyNodeParams . . . 639
7.61 cudaMemsetParams . . . 640
7.62 cudaMemsetParamsV2 . . . 640
7.63 cudaOffset3D . . . 641
7.64 cudaPitchedPtr . . . 642
7.65 cudaPointerAttributes . . . 642
7.66 cudaPos . . . 643
7.67 cudaResourceDesc . . . 644
7.68 cudaResourceViewDesc . . . 645
7.69 cudaTextureDesc . . . 646
7.70 cudalibraryHostUniversalFunctionAndDataTable . . . 647

8 Deprecated List **651**

9 Notices **655**
9.1 Notice . . . 655
9.2 OpenCL . . . 656
9.3 Trademarks . . . 656

Chapter 1. Difference between the driver and runtime APIs

The driver and runtime APIs are very similar and can for the most part be used interchangeably. However, there are some key differences worth noting between the two.

Complexity vs. control

The runtime API eases device code management by providing implicit primary context initialization and management, and implicit module management. This leads to simpler code, but it also lacks the level of control that the driver API has.

In comparison, the driver API offers more fine-grained control, especially over module loading. Kernel launches are much more complex to implement, as the execution configuration and kernel parameters must be specified with explicit function calls. However, unlike the runtime, where all the kernels are automatically loaded during initialization and stay loaded for as long as the program runs, with the driver API it is possible to only keep the modules that are currently needed loaded, or even dynamically reload modules. The driver API is also language-independent as it only deals with cubin objects.

Context management

Unless an execution context cudaExecutionContext_t is specified, the runtime API decides itself which context to use for a thread: if a context has been made current to the calling thread through the driver API, the runtime will use that, but if there is no such context, it uses the device execution context which is a “primary context.” Primary contexts are created as needed, one per device per process, are reference-counted, and are then destroyed when there are no more references to them. Within one process, all users of the runtime API will share the primary context, unless a context has been made current to each thread or an explicit execution context is specified to the runtime APIs.

Using the runtime API with primary contexts has its tradeoffs, however. It can cause trouble for users writing plug-ins for larger software packages, for example, because if all plug-ins run in the same process, they will all share a context but will likely have no way to communicate with each other. So, if one of them calls cudaDeviceReset() after finishing all its CUDA work, the other plug-ins will fail because the context they were using was destroyed without their knowledge. To avoid this issue, CUDA clients can use the driver API to create and set the current context, and then use the runtime API to work with it. However, contexts may consume significant resources, such as device memory, extra host threads, and performance costs of context switching on the device. This runtime-driver context sharing is important when using the driver API in conjunction with libraries built on the runtime API, such as cuBLAS or cuFFT.

Chapter 2. API Synchronization behavior

The API provides memcpy/memset functions in both synchronous and asynchronous forms, the latter having an “Async” suffix. This is a misnomer as each function may exhibit synchronous or asynchronous behavior depending on the arguments passed to the function. The synchronous forms of these APIs issue these copies through the default stream.

Any CUDA API call may block or synchronize for various reasons such as contention for or unavailability of internal resources. Such behavior is subject to change and undocumented behavior should not be relied upon.

Memcpy

In the reference documentation, each memcpy function is categorized as synchronous or asynchronous, corresponding to the definitions below.

Synchronous

1. For transfers from pageable host memory to device memory, a stream sync is performed before the copy is initiated. The function will return once the pageable buffer has been copied to the staging memory for DMA transfer to device memory, but the DMA to final destination may not have completed.
2. For transfers from pinned host memory to device memory, the function is synchronous with respect to the host.
3. For transfers from device to either pageable or pinned host memory, the function returns only once the copy has completed.
4. For transfers from device memory to device memory, no host-side synchronization is performed.
5. For transfers from any host memory to any host memory, the function is fully synchronous with respect to the host.

Asynchronous

1. For transfers between device memory and pageable host memory, the function might be synchronous with respect to host.
2. For transfers from any host memory to any host memory, the function is fully synchronous with respect to the host.
3. If pageable memory must first be staged to pinned memory, the driver may synchronize with the stream and stage the copy into pinned memory.
4. For all other transfers, the function should be fully asynchronous.

Memset

The cudaMemset functions are asynchronous with respect to the host except when the target memory is pinned host memory. The Async versions are always asynchronous with respect to the host.

Kernel Launches

Kernel launches are asynchronous with respect to the host. Details of concurrent kernel execution and data transfers can be found in the CUDA Programmers Guide.

Chapter 3. Stream synchronization behavior

Default stream

The default stream, used when `0` is passed as a `cudaStream_t` or by APIs that operate on a stream implicitly, can be configured to have either legacy or per-thread synchronization behavior as described below.

The behavior can be controlled per compilation unit with the `--default-stream` nvcc option. Alternatively, per-thread behavior can be enabled by defining the `CUDA_API_PER_THREAD_DEFAULT_STREAM` macro before including any CUDA headers. Either way, the `CUDA_API_PER_THREAD_DEFAULT_STREAM` macro will be defined in compilation units using per-thread synchronization behavior.

Legacy default stream

The legacy default stream is an implicit stream which synchronizes with all other streams in the same `CUcontext` except for non-blocking streams, described below. (For applications using the runtime APIs only, there will be one context per device.) When an action is taken in the legacy stream such as a kernel launch or `cudaStreamWaitEvent()`, the legacy stream first waits on all blocking streams, the action is queued in the legacy stream, and then all blocking streams wait on the legacy stream.

For example, the following code launches a kernel `k_1` in stream `s`, then `k_2` in the legacy stream, then `k_3` in stream `s`:

```
k_1<<<1, 1, 0, s>>>();
k_2<<<1, 1>>>();
k_3<<<1, 1, 0, s>>>();
```

The resulting behavior is that *k_2* will block on `k_1` and `k_3` will block on `k_2`.

Non-blocking streams which do not synchronize with the legacy stream can be created using the `cudaStreamNonBlocking` flag with the stream creation APIs.

The legacy default stream can be used explicitly with the `CUstream` (`cudaStream_t`) handle `CU_STREAM_LEGACY` (`cudaStreamLegacy`).

Per-thread default stream

The per-thread default stream is an implicit stream local to both the thread and the `CUcontext`, and which does not synchronize with other streams (just like explicitly created streams). The per-thread default stream is not a non-blocking stream and will synchronize with the legacy default stream if both are used in a program.

The per-thread default stream can be used explicitly with the `CUstream` (`cudaStream_t`) handle `CU_STREAM_PER_THREAD` (`cudaStreamPerThread`).

Chapter 4. Graph object thread safety

Graph objects (`cudaGraph_t`, `CUgraph`) are not internally synchronized and must not be accessed concurrently from multiple threads. API calls accessing the same graph object must be serialized externally.

Note that **this includes APIs which may appear to be read-only**, such as `cudaGraphClone()` (`cuGraphClone()`) and `cudaGraphInstantiate()` (`cuGraphInstantiate()`). No API or pair of APIs is guaranteed to be safe to call on the same graph object from two different threads without serialization.

Chapter 5. Rules for version mixing

1. Starting with CUDA 11.0, the ABI version for the CUDA runtime is bumped every major release. CUDA-defined types, whether opaque handles or structures like `cudaDeviceProp`, have their ABI tied to the major release of the CUDA runtime. It is unsafe to pass them from function A to function B if those functions have been compiled with different major versions of the toolkit and linked together into the same device executable.
2. The CUDA Driver API has a per-function ABI denoted with a _v* extension. CUDA-defined types (e.g structs) should not be passed across different ABI versions. For example, an application calling `cuMemcpy2D_v2(const CUDA_MEMCPY2D_v2 *pCopy)` and using the older version of the struct `CUDA_MEMCPY2D_v1` instead of `CUDA_MEMCPY2D_v2`.
3. Users should not arbitrarily mix different API versions during the lifetime of a resource. These resources include IPC handles, memory, streams, contexts, events, etc. For example, a user who wants to allocate CUDA memory using `cuMemAlloc_v2` should free the memory using `cuMemFree_v2` and not `cuMemFree`.

Chapter 6. APIs

6.1. C++ API Routines

impl_private

This section describes the C++ high level API functions of the CUDA runtime application programming interface. To use these functions, your application needs to be compiled with the nvcc compiler.

C++-style interface built on top of CUDA runtime API

Classes

__cudaOccupancyB2DHelper
Helper functor for cudaOccupancyMaxPotentialBlockSize.

Functions

__host__ cudaChannelFormatDesc *cudaCreateChannelDesc*(void)
Returns a channel descriptor using the specified format.

__host__ cudaChannelFormatDesc *cudaCreateChannelDesc< char >*(void)

__host__ cudaChannelFormatDesc *cudaCreateChannelDesc< char1 >*(void)

__host__ cudaChannelFormatDesc *cudaCreateChannelDesc< char2 >*(void)

__host__ cudaChannelFormatDesc *cudaCreateChannelDesc< char4 >*(void)

__host__ cudaChannelFormatDesc *cudaCreateChannelDesc< cudaChannelFormatKindNV12 >*(void)

__host__ cudaChannelFormatDesc *cudaCreateChannelDesc< cudaChannelFormatKindSignedBlock-Compressed4 >*(void)

__host__ cudaChannelFormatDesc *cudaCreateChannelDesc< cudaChannelFormatKindSignedBlock-Compressed5 >*(void)

__host__ cudaChannelFormatDesc *cudaCreateChannelDesc< cudaChannelFormatKindSignedBlockCompressed6H >*(void)

__host__ cudaChannelFormatDesc *cudaCreateChannelDesc< cudaChannelFormatKindSignedNormalized16X1 >*(void)

__host__ cudaChannelFormatDesc *cudaCreateChannelDesc< cudaChannelFormatKindSignedNormalized16X2 >*(void)

__host__ cudaChannelFormatDesc *cudaCreateChannelDesc< cudaChannelFormatKindSignedNormalized16X4 >*(void)

__host__ cudaChannelFormatDesc *cudaCreateChannelDesc< cudaChannelFormatKindSignedNormalized8X1 >*(void)

__host__ cudaChannelFormatDesc *cudaCreateChannelDesc< cudaChannelFormatKindSignedNormalized8X2 >*(void)

__host__ cudaChannelFormatDesc *cudaCreateChannelDesc< cudaChannelFormatKindSignedNormalized8X4 >*(void)

__host__ cudaChannelFormatDesc *cudaCreateChannelDesc< cudaChannelFormatKindUnsigned16Planar420 >*(void)

__host__ cudaChannelFormatDesc *cudaCreateChannelDesc< cudaChannelFormatKindUnsigned16Planar422 >*(void)

__host__ cudaChannelFormatDesc *cudaCreateChannelDesc< cudaChannelFormatKindUnsigned16Planar444 >*(void)

__host__ cudaChannelFormatDesc *cudaCreateChannelDesc< cudaChannelFormatKindUnsigned16SemiPlanar420 >*(void)

__host__ cudaChannelFormatDesc *cudaCreateChannelDesc< cudaChannelFormatKindUnsigned16SemiPlanar422 >*(void)

__host__ cudaChannelFormatDesc *cudaCreateChannelDesc< cudaChannelFormatKindUnsigned16SemiPlanar444 >*(void)

__host__ cudaChannelFormatDesc *cudaCreateChannelDesc< cudaChannelFormatKindUnsigned8Packed422 >*(void)

__host__ cudaChannelFormatDesc *cudaCreateChannelDesc< cudaChannelFormatKindUnsigned8Packed444 >*(void)

__host__ cudaChannelFormatDesc *cudaCreateChannelDesc< cudaChannelFormatKindUnsigned8Planar420 >*(void)

__host__ cudaChannelFormatDesc *cudaCreateChannelDesc< cudaChannelFormatKindUnsigned8Planar422 >*(void)

__host__ cudaChannelFormatDesc *cudaCreateChannelDesc< cudaChannelFormatKindUnsigned8Planar444 >*(void)

__host__ cudaChannelFormatDesc *cudaCreateChannelDesc< cudaChannelFormatKindUnsigned8SemiPlanar420 >*(void)

__host__ cudaChannelFormatDesc *cudaCreateChannelDesc< cudaChannelFormatKindUnsigned8SemiPlanar422 >*(void)

__host__ cudaChannelFormatDesc *cudaCreateChannelDesc< cudaChannelFormatKindUnsigned8SemiPlanar444 >*(void)

__host__ cudaChannelFormatDesc *cudaCreateChannelDesc< cudaChannelFormatKindUnsignedBlockCompressed1 >*(void)

__host__ cudaChannelFormatDesc *cudaCreateChannelDesc< cudaChannelFormatKindUnsignedBlockCompressed1SRGB >*(void)

__host__ cudaChannelFormatDesc *cudaCreateChannelDesc< cudaChannelFormatKindUnsignedBlockCompressed2 >*(void)

__host__ cudaChannelFormatDesc *cudaCreateChannelDesc< cudaChannelFormatKindUnsignedBlockCompressed2SRGB >*(void)

__host__ cudaChannelFormatDesc *cudaCreateChannelDesc< cudaChannelFormatKindUnsignedBlockCompressed3 >*(void)

__host__ cudaChannelFormatDesc *cudaCreateChannelDesc< cudaChannelFormatKindUnsignedBlockCompressed3SRGB >*(void)

__host__ cudaChannelFormatDesc *cudaCreateChannelDesc< cudaChannelFormatKindUnsignedBlockCompressed4 >*(void)

__host__ cudaChannelFormatDesc *cudaCreateChannelDesc< cudaChannelFormatKindUnsignedBlockCompressed5 >*(void)

__host__ cudaChannelFormatDesc *cudaCreateChannelDesc< cudaChannelFormatKindUnsignedBlockCompressed6H >*(void)

__host__ cudaChannelFormatDesc *cudaCreateChannelDesc< cudaChannelFormatKindUnsigned-BlockCompressed7 >*(void)

__host__ cudaChannelFormatDesc *cudaCreateChannelDesc< cudaChannelFormatKindUnsigned-BlockCompressed7SRGB >*(void)

__host__ cudaChannelFormatDesc *cudaCreateChannelDesc< cudaChannelFormatKindUnsignedNormalized1010102 >*(void)

__host__ cudaChannelFormatDesc *cudaCreateChannelDesc< cudaChannelFormatKindUnsignedNormalized16X1 >*(void)

__host__ cudaChannelFormatDesc *cudaCreateChannelDesc< cudaChannelFormatKindUnsignedNormalized16X2 >*(void)

__host__ cudaChannelFormatDesc *cudaCreateChannelDesc< cudaChannelFormatKindUnsignedNormalized16X4 >*(void)

__host__ cudaChannelFormatDesc *cudaCreateChannelDesc< cudaChannelFormatKindUnsignedNormalized8X1 >*(void)

__host__ cudaChannelFormatDesc *cudaCreateChannelDesc< cudaChannelFormatKindUnsignedNormalized8X2 >*(void)

__host__ cudaChannelFormatDesc *cudaCreateChannelDesc< cudaChannelFormatKindUnsignedNormalized8X4 >*(void)

else __host__ cudaChannelFormatDesc cudaCreateChannelDesc< float >(void)

__host__ cudaChannelFormatDesc *cudaCreateChannelDesc< float1 >*(void)

__host__ cudaChannelFormatDesc *cudaCreateChannelDesc< float2 >*(void)

__host__ cudaChannelFormatDesc *cudaCreateChannelDesc< float4 >*(void)

__host__ cudaChannelFormatDesc *cudaCreateChannelDesc< int >*(void)

__host__ cudaChannelFormatDesc *cudaCreateChannelDesc< int1 >*(void)

__host__ cudaChannelFormatDesc *cudaCreateChannelDesc< int2 >*(void)

__host__ cudaChannelFormatDesc *cudaCreateChannelDesc< int4 >*(void)

__host__ cudaChannelFormatDesc *cudaCreateChannelDesc< long >*(void)

__host__ cudaChannelFormatDesc *cudaCreateChannelDesc< long1 >*(void)

__host__ cudaChannelFormatDesc *cudaCreateChannelDesc< long2 >*(void)

__host__ cudaChannelFormatDesc *cudaCreateChannelDesc< long4 >*(void)

__host__ cudaChannelFormatDesc *cudaCreateChannelDesc< short >*(void)

__host__ cudaChannelFormatDesc *cudaCreateChannelDesc< short1 >*(void)

__host__ cudaChannelFormatDesc *cudaCreateChannelDesc< short2 >*(void)

__host__ cudaChannelFormatDesc *cudaCreateChannelDesc< short4 >*(void)

__host__ cudaChannelFormatDesc *cudaCreateChannelDesc< signed char >*(void)

__host__ cudaChannelFormatDesc *cudaCreateChannelDesc< uchar1 >*(void)

__host__ cudaChannelFormatDesc *cudaCreateChannelDesc< uchar2 >*(void)

__host__ cudaChannelFormatDesc *cudaCreateChannelDesc< uchar4 >*(void)

__host__ cudaChannelFormatDesc *cudaCreateChannelDesc< uint1 >*(void)

__host__ cudaChannelFormatDesc *cudaCreateChannelDesc< uint2 >*(void)

__host__ cudaChannelFormatDesc *cudaCreateChannelDesc< uint4 >*(void)

__host__ cudaChannelFormatDesc *cudaCreateChannelDesc< ulong1 >*(void)

__host__ cudaChannelFormatDesc *cudaCreateChannelDesc< ulong2 >*(void)

__host__ cudaChannelFormatDesc *cudaCreateChannelDesc< ulong4 >*(void)

__host__ cudaChannelFormatDesc *cudaCreateChannelDesc< unsigned char >*(void)

__host__ cudaChannelFormatDesc *cudaCreateChannelDesc< unsigned int >*(void)

__host__ cudaChannelFormatDesc *cudaCreateChannelDesc< unsigned long >*(void)

__host__ cudaChannelFormatDesc *cudaCreateChannelDesc< unsigned short >*(void)

__host__ cudaChannelFormatDesc *cudaCreateChannelDesc< ushort1 >*(void)

__host__ cudaChannelFormatDesc *cudaCreateChannelDesc< ushort2 >*(void)

__host__ cudaChannelFormatDesc *cudaCreateChannelDesc< ushort4 >*(void)

__host__ cudaChannelFormatDesc *cudaCreateChannelDescHalf*(void)

__host__ cudaChannelFormatDesc *cudaCreateChannelDescHalf1*(void)

__host__ cudaChannelFormatDesc *cudaCreateChannelDescHalf2*(void)

__host__ cudaChannelFormatDesc *cudaCreateChannelDescHalf4*(void)

__host__ cudaChannelFormatDesc *cudaCreateChannelDescNV12*(void)

__host__ cudaError_t *cudaEventCreate*(cudaEvent_t *event, unsigned int flags)
Creates an event object with the specified flags.

__host__ cudaError_t *cudaFuncGetAttributes*(struct cudaFuncAttributes *attr, T *entry)
Find out attributes for a given function.

__host__ cudaError_t *cudaFuncGetName*(const char **name, T *func)
Returns the function name for a device entry function pointer.

__host__ cudaError_t *cudaFuncSetAttribute*(T *func, enum cudaFuncAttribute attr, int value)
Set attributes for a given function.

__host__ cudaError_t *cudaFuncSetCacheConfig*(T *func, enum cudaFuncCache cacheConfig)
Sets the preferred cache configuration for a device function.

__host__ cudaError_t *cudaFuncSetSharedMemConfig*(T *func, enum cudaSharedMemConfig config)

__host__ cudaError_t *cudaGetKernel*(cudaKernel_t *kernelPtr, T *func)
Get pointer to device kernel that matches entry function `entryFuncAddr` .

__host__ cudaError_t *cudaGetSymbolAddress*(void **devPtr, const T &symbol)
Finds the address associated with a CUDA symbol.

__host__ cudaError_t *cudaGetSymbolSize*(size_t *size, const T &symbol)
Finds the size of the object associated with a CUDA symbol.

__host__ cudaError_t *cudaGraphAddMemcpyNodeFromSymbol*(cudaGraphNode_t *pGraphNode, cudaGraph_t graph, const cudaGraphNode_t *pDependencies, size_t numDependencies, void *dst, const T &symbol, size_t count, size_t offset, enum cudaMemcpyKind kind)
Creates a memcpy node to copy from a symbol on the device and adds it to a graph.

__host__ cudaError_t *cudaGraphAddMemcpyNodeToSymbol*(cudaGraphNode_t *pGraphNode, cudaGraph_t graph, const cudaGraphNode_t *pDependencies, size_t numDependencies, const T &symbol, const void *src, size_t count, size_t offset, enum cudaMemcpyKind kind)
Creates a memcpy node to copy to a symbol on the device and adds it to a graph.

__host__ cudaError_t *cudaGraphExecMemcpyNodeSetParamsFromSymbol*(cudaGraphExec_t hGraphExec, cudaGraphNode_t node, void *dst, const T &symbol, size_t count, size_t

offset, enum cudaMemcpyKind kind)
Sets the parameters for a memcpy node in the given graphExec to copy from a symbol on the device.

__host__ cudaError_t *cudaGraphExecMemcpyNodeSetParamsToSymbol*(cudaGraphExec_t hGraphExec, cudaGraphNode_t node, const T &symbol, const void *src, size_t count, size_t offset, enum cudaMemcpyKind kind)
Sets the parameters for a memcpy node in the given graphExec to copy to a symbol on the device.

__host__ cudaError_t *cudaGraphExecUpdate*(cudaGraphExec_t hGraphExec, cudaGraph_t hGraph, cudaGraphNode_t *hErrorNode_out, enum cudaGraphExecUpdateResult *updateResult_out)

__host__ cudaError_t *cudaGraphInstantiate*(cudaGraphExec_t *pGraphExec, cudaGraph_t graph, cudaGraphNode_t *pErrorNode, char *pLogBuffer, size_t bufferSize)
Creates an executable graph from a graph.

__host__ cudaError_t *cudaGraphMemcpyNodeSetParamsFromSymbol*(cudaGraphNode_t node, void *dst, const T &symbol, size_t count, size_t offset, enum cudaMemcpyKind kind)
Sets a memcpy node's parameters to copy from a symbol on the device.

__host__ cudaError_t *cudaGraphMemcpyNodeSetParamsToSymbol*(cudaGraphNode_t node, const T &symbol, const void *src, size_t count, size_t offset, enum cudaMemcpyKind kind)
Sets a memcpy node's parameters to copy to a symbol on the device.

__host__ cudaError_t *cudaHostAlloc*(T **ptr, size_t size, unsigned int flags)

__host__ cudaError_t *cudaHostGetDevicePointer*(T **pDevice, void *pHost, unsigned int flags)

__host__ cudaError_t *cudaLaunchCooperativeKernel*(T *func, dim3 gridDim, dim3 blockDim, void **args, size_t sharedMem=0, cudaStream_t stream=0)
Launches a device function.

__host__ cudaError_t *cudaLaunchKernel*(T *func, dim3 gridDim, dim3 blockDim, void **args, size_t sharedMem=0, cudaStream_t stream=0)
Launches a device function.

__host__ cudaError_t *cudaLaunchKernelEx*(const cudaLaunchConfig_t *config, const cudaKernel_t kernel, ActTypes &&... args)
Launches a CUDA function with launch-time configuration.

__host__ cudaError_t *cudaLaunchKernelEx*(const cudaLaunchConfig_t *config, void(*kernel)(ExpTypes...), ActTypes &&... args)
Launches a CUDA function with launch-time configuration.

__host__ cudaError_t *cudaLibraryGetGlobal*(T **dptr, size_t *bytes, cudaLibrary_t library, const char *name)
Returns a global device pointer.

__host__ cudaError_t *cudaLibraryGetManaged*(T **dptr, size_t *bytes, cudaLibrary_t library, const char *name)
Returns a pointer to managed memory.

__host__ cudaError_t *cudaLibraryGetUnifiedFunction*(T **fptr, cudaLibrary_t library, const char *symbol)
Returns a pointer to a unified function.

__host__ cudaError_t *cudaMalloc*(T **devPtr, size_t size)

__host__ cudaError_t *cudaMallocAsync*(T **ptr, size_t size, cudaStream_t stream)

__host__ cudaError_t *cudaMallocAsync*(void **ptr, size_t size, cudaMemPool_t memPool, cudaStream_t stream)
Allocate from a pool.

__host__ cudaError_t *cudaMallocAsync*(T **ptr, size_t size, cudaMemPool_t memPool, cudaStream_t stream)

__host__ cudaError_t *cudaMallocFromPoolAsync*(T **ptr, size_t size, cudaMemPool_t memPool, cudaStream_t stream)

__host__ cudaError_t *cudaMallocHost*(T **ptr, size_t size, unsigned int flags=0)

__host__ cudaError_t *cudaMallocHost*(void **ptr, size_t size, unsigned int flags)
Allocates page-locked memory on the host.

__host__ cudaError_t *cudaMallocManaged*(T **devPtr, size_t size, unsigned int flags=0x01)
Allocates memory that will be automatically managed by the Unified Memory system.

__host__ cudaError_t *cudaMallocPitch*(T **devPtr, size_t *pitch, size_t width, size_t height)

__host__ cudaError_t *cudaMemDiscardAndPrefetchBatchAsync*(T **dptrs, size_t *sizes, size_t count, struct cudaMemLocation prefetchLocs, unsigned long long flags, cudaStream_t stream)
Performs a batch of memory discard and prefetches asynchronously.

__host__ cudaError_t *cudaMemDiscardAndPrefetchBatchAsync*(T **dptrs, size_t *sizes, size_t count, struct cudaMemLocation *prefetchLocs, size_t *prefetchLocIdxs, size_t numPrefetchLocs, unsigned long long flags, cudaStream_t stream)
Performs a batch of memory discard and prefetches asynchronously.

__host__ cudaError_t *cudaMemPrefetchBatchAsync*(T **dptrs, size_t *sizes, size_t count, struct cudaMemLocation prefetchLocs, unsigned long long flags, cudaStream_t stream)
Performs a batch of memory prefetches asynchronously.

__host__ cudaError_t *cudaMemPrefetchBatchAsync*(T **dptrs, size_t *sizes, size_t count, struct cudaMemLocation *prefetchLocs, size_t *prefetchLocIdxs, size_t numPrefetchLocs, unsigned long long flags, cudaStream_t stream)
Performs a batch of memory prefetches asynchronously.

__host__ cudaError_t *cudaMemcpyAsync*(T dst, U src, const size_t size, struct cudaMemcpyAttributes *attr, cudaStream_t hStream)
Performs a memory copy asynchronously.

__host__ cudaError_t *cudaMemcpyBatchAsync*(T *const *dsts, U *const *srcs, const size_t *sizes, size_t count, struct cudaMemcpyAttributes *attrs, size_t *attrsIdxs, size_t numAttrs, cudaStream_t hStream)
Performs a batch of memory copies asynchronously.

__host__ cudaError_t *cudaMemcpyBatchAsync*(T *const *dsts, U *const *srcs, const size_t *sizes, size_t count, struct cudaMemcpyAttributes attr, cudaStream_t hStream)
Performs a batch of memory copies asynchronously.

__host__ cudaError_t *cudaMemcpyFromSymbol*(void *dst, const T &symbol, size_t count, size_t offset=0, enum cudaMemcpyKind kind=cudaMemcpyDeviceToHost)
Copies data from the given symbol on the device.

__host__ cudaError_t *cudaMemcpyFromSymbolAsync*(void *dst, const T &symbol, size_t count, size_t offset=0, enum cudaMemcpyKind kind=cudaMemcpyDeviceToHost, cudaStream_t stream=0)
Copies data from the given symbol on the device.

__host__ cudaError_t *cudaMemcpyToSymbol*(const T &symbol, const void *src, size_t count, size_t offset=0, enum cudaMemcpyKind kind=cudaMemcpyHostToDevice)
Copies data to the given symbol on the device.

__host__ cudaError_t *cudaMemcpyToSymbolAsync*(const T &symbol, const void *src, size_t count, size_t offset=0, enum cudaMemcpyKind kind=cudaMemcpyHostToDevice, cudaStream_t stream=0)
Copies data to the given symbol on the device.

__host__ cudaError_t *cudaOccupancyAvailableDynamicSMemPerBlock*(size_t *dynamicSmemSize, T *func, int numBlocks, int blockSize)
Returns dynamic shared memory available per block when launching `numBlocks` blocks on SM.

__host__ cudaError_t *cudaOccupancyMaxActiveBlocksPerMultiprocessor*(int *numBlocks, T func, int blockSize, size_t dynamicSMemSize)
Returns occupancy for a device function.

__host__ cudaError_t *cudaOccupancyMaxActiveBlocksPerMultiprocessorWithFlags*(int *numBlocks, T func, int blockSize, size_t dynamicSMemSize, unsigned int flags)
Returns occupancy for a device function with the specified flags.

__host__ cudaError_t *cudaOccupancyMaxActiveClusters*(int *numClusters, T *func, const cudaLaunchConfig_t *config)
Given the kernel function (`func`) and launch configuration (`config`), return the maximum number of clusters that could co-exist on the target device in `*numClusters` .

__host__ __device__ cudaError_t *cudaOccupancyMaxPotentialBlockSize*(int *minGridSize, int *blockSize, T func, size_t dynamicSMemSize=0, int blockSizeLimit=0)
Returns grid and block size that achieves maximum potential occupancy for a device function.

__host__ __device__ cudaError_t *cudaOccupancyMaxPotentialBlockSizeVariableSMem*(int *minGridSize, int *blockSize, T func, UnaryFunction blockSizeToDynamicSMemSize, int blockSizeLimit=0)
Returns grid and block size that achieves maximum potential occupancy for a device function.

__host__ __device__ cudaError_t *cudaOccupancyMaxPotentialBlockSizeVariableSMemWithFlags*(int *minGridSize, int *blockSize, T func, UnaryFunction blockSizeToDynamicSMemSize, int blockSizeLimit=0, unsigned int flags=0)
Returns grid and block size that achieves maximum potential occupancy for a device function.

__host__ __device__ cudaError_t *cudaOccupancyMaxPotentialBlockSizeWithFlags*(int *minGridSize, int *blockSize, T func, size_t dynamicSMemSize=0, int blockSizeLimit=0, unsigned int flags=0)
Returns grid and block size that achived maximum potential occupancy for a device function with the specified flags.

__host__ cudaError_t *cudaOccupancyMaxPotentialClusterSize*(int *clusterSize, T *func, const cudaLaunchConfig_t *config)
Given the kernel function (`func`) and launch configuration (`config`), return the maximum cluster size in `*clusterSize` .

__host__ cudaError_t *cudaStreamAttachMemAsync*(cudaStream_t stream, T *devPtr, size_t length=0, unsigned int flags=0x04)
Attach memory to a stream asynchronously.

6.1.1. Functions

template<class **T**>
__host__ *cudaChannelFormatDesc* **cudaCreateChannelDesc**(void)

Returns a channel descriptor using the specified format.

Returns a channel descriptor with format `f` and number of bits of each component `x`, `y`, `z`, and `w`. The *cudaChannelFormatDesc* is defined as:

```
struct cudaChannelFormatDesc {
  int x, y, z, w;
  enum cudaChannelFormatKind f;
};
```

where *cudaChannelFormatKind* is one of *cudaChannelFormatKindSigned*, *cudaChannelFormatKindUnsigned*, cudaChannelFormatKindFloat, *cudaChannelFormatKindSignedNormalized8X1*, *cudaChannelFormatKindSignedNormalized8X2*, *cudaChannelFormatKindSignedNormalized8X4*, *cudaChannelFormatKindUnsignedNormalized8X1*, *cudaChannelFormatKindUnsignedNormalized8X2*, *cudaChannelFormatKindUnsignedNormalized8X4*, *cudaChannelFormatKindSignedNormalized16X1*, *cudaChannelFormatKindSignedNormalized16X2*, *cudaChannelFormatKindSignedNormalized16X4*, *cudaChannelFormatKindUnsignedNormalized16X1*, *cudaChannelFormatKindUnsignedNormalized16X2*, *cudaChannelFormatKindUnsignedNormalized16X4*, *cudaChannelFormatKindUnsignedNormalized1010102* or *cudaChannelFormatKindNV12*. *cudaChannelFormatKindUnsigned8Packed422*, *cudaChannelFormatKindUnsigned8Packed444*, *cudaChannelFormatKindUnsigned8SemiPlanar420*, *cudaChannelFormatKindUnsigned16SemiPlanar420*, *cudaChannelFormatKindUnsigned8SemiPlanar422*, *cudaChannelFormatKindUnsigned16SemiPlanar422*, *cudaChannelFormatKindUnsigned8SemiPlanar444*, *cudaChannelFormatKindUnsigned16SemiPlanar444*, *cudaChannelFormatKindUnsigned8Planar420*, *cudaChannelFormatKindUnsigned16Planar420*, *cudaChannelFormatKindUnsigned8Planar422*, *cudaChannelFormatKindUnsigned16Planar422*, *cudaChannelFormatKindUnsigned8Planar444*, and *cudaChannelFormatKindUnsigned16Planar444*.

The format is specified by the template specialization.

The template function specializes for the following scalar types: char, signed char, unsigned char, short, unsigned short, int, unsigned int, long, unsigned long, and float. The template function specializes for the following vector types: char{1|2|4}, uchar{1|2|4}, short{1|2|4}, ushort{1|2|4}, int{1|2|4}, uint{1|2|4}, long{1|2|4}, ulong{1|2|4}, float{1|2|4}. The template function specializes for following cudaChannelFormatKind enum values: *cudaChannelFormatKind*{Uns|S}ignedNormalized{8|16}X{1|2|4}, *cudaChannelFormatKindUnsignedNormalized1010102* *cudaChannelFormatKindNV12*, *cudaChannelFormatKindUnsigned8Packed422*, *cudaChannelFormatKindUnsigned8Packed444*, *cudaChannelFormatKindUnsigned8SemiPlanar420*, *cudaChannelFormatKindUnsigned16SemiPlanar420*, *cudaChannelFormatKindUnsigned8SemiPlanar422*, *cudaChannelFormatKindUnsigned16SemiPlanar422*, *cudaChannelFormatKindUnsigned8SemiPlanar444*, *cudaChannelFormatKindUnsigned16SemiPlanar444*, *cudaChannelFormatKindUnsigned8Planar420*, *cudaChannelFormatKindUnsigned16Planar420*, *cudaChannelFormatKindUnsigned8Planar422*, *cudaChannelFormatKindUnsigned16Planar422*, *cudaChannelFormatKindUnsigned8Planar444*, and *cudaChannelFormatKindUnsigned16Planar444*.

Invoking the function on a type without a specialization defaults to creating a channel format of kind *cudaChannelFormatKindNone*

See also:

cudaCreateChannelDesc (Low level), *cudaGetChannelDesc*,

Returns
Channel descriptor with format f

template<>
__host__ *cudaChannelFormatDesc* **cudaCreateChannelDesc**<char>(void)

template<>
__host__ *cudaChannelFormatDesc* **cudaCreateChannelDesc**<char1>(void)

template<>
__host__ *cudaChannelFormatDesc* **cudaCreateChannelDesc**<char2>(void)

template<>
__host__ *cudaChannelFormatDesc* **cudaCreateChannelDesc**<char4>(void)

template<>
__host__ *cudaChannelFormatDesc* **cudaCreateChannelDesc**<*cudaChannelFormatKindNV12*>(void)

template<>
__host__ *cudaChannelFormatDesc* **cudaCreateChannelDesc**<*cudaChannelFormatKindSignedBlockCompressed4*>(v

template<>
__host__ *cudaChannelFormatDesc* **cudaCreateChannelDesc**<*cudaChannelFormatKindSignedBlockCompressed5*>(v

template<>
__host__ *cudaChannelFormatDesc* **cudaCreateChannelDesc**<*cudaChannelFormatKindSignedBlockCompressed6H*>(

template<>
__host__ *cudaChannelFormatDesc* **cudaCreateChannelDesc**<*cudaChannelFormatKindSignedNormalized16X1*>(void

template<>
__host__ *cudaChannelFormatDesc* **cudaCreateChannelDesc**<*cudaChannelFormatKindSignedNormalized16X2*>(void

template<>
__host__ *cudaChannelFormatDesc* **cudaCreateChannelDesc**<*cudaChannelFormatKindSignedNormalized16X4*>(void

template<>
__host__ *cudaChannelFormatDesc* **cudaCreateChannelDesc**<*cudaChannelFormatKindSignedNormalized8X1*>(void)

template<>
__host__ *cudaChannelFormatDesc* **cudaCreateChannelDesc**<*cudaChannelFormatKindSignedNormalized8X2*>(void)

template<>
__host__ *cudaChannelFormatDesc* **cudaCreateChannelDesc**<*cudaChannelFormatKindSignedNormalized8X4*>(void)

template<>
__host__ *cudaChannelFormatDesc* **cudaCreateChannelDesc**<*cudaChannelFormatKindUnsigned16Planar420*>(void)

template<>

__host__ *cudaChannelFormatDesc* **cudaCreateChannelDesc**<*cudaChannelFormatKindUnsigned16Planar422*>(void)

template<>
__host__ *cudaChannelFormatDesc* **cudaCreateChannelDesc**<*cudaChannelFormatKindUnsigned16Planar444*>(void)

template<>
__host__ *cudaChannelFormatDesc* **cudaCreateChannelDesc**<*cudaChannelFormatKindUnsigned16SemiPlanar420*>(

template<>
__host__ *cudaChannelFormatDesc* **cudaCreateChannelDesc**<*cudaChannelFormatKindUnsigned16SemiPlanar422*>(

template<>
__host__ *cudaChannelFormatDesc* **cudaCreateChannelDesc**<*cudaChannelFormatKindUnsigned16SemiPlanar444*>(

template<>
__host__ *cudaChannelFormatDesc* **cudaCreateChannelDesc**<*cudaChannelFormatKindUnsigned8Packed422*>(void)

template<>
__host__ *cudaChannelFormatDesc* **cudaCreateChannelDesc**<*cudaChannelFormatKindUnsigned8Packed444*>(void)

template<>
__host__ *cudaChannelFormatDesc* **cudaCreateChannelDesc**<*cudaChannelFormatKindUnsigned8Planar420*>(void)

template<>
__host__ *cudaChannelFormatDesc* **cudaCreateChannelDesc**<*cudaChannelFormatKindUnsigned8Planar422*>(void)

template<>
__host__ *cudaChannelFormatDesc* **cudaCreateChannelDesc**<*cudaChannelFormatKindUnsigned8Planar444*>(void)

template<>
__host__ *cudaChannelFormatDesc* **cudaCreateChannelDesc**<*cudaChannelFormatKindUnsigned8SemiPlanar420*>(vo

template<>
__host__ *cudaChannelFormatDesc* **cudaCreateChannelDesc**<*cudaChannelFormatKindUnsigned8SemiPlanar422*>(vo

template<>
__host__ *cudaChannelFormatDesc* **cudaCreateChannelDesc**<*cudaChannelFormatKindUnsigned8SemiPlanar444*>(vo

template<>
__host__ *cudaChannelFormatDesc* **cudaCreateChannelDesc**<*cudaChannelFormatKindUnsignedBlockCompressed1*>

template<>
__host__ *cudaChannelFormatDesc* **cudaCreateChannelDesc**<*cudaChannelFormatKindUnsignedBlockCompressed1S*

template<>
__host__ *cudaChannelFormatDesc* **cudaCreateChannelDesc**<*cudaChannelFormatKindUnsignedBlockCompressed2*>

template<>
__host__ *cudaChannelFormatDesc* **cudaCreateChannelDesc**<*cudaChannelFormatKindUnsignedBlockCompressed2S*

template<>
__host__ *cudaChannelFormatDesc* **cudaCreateChannelDesc**<*cudaChannelFormatKindUnsignedBlockCompressed3*>

template<>
__host__ *cudaChannelFormatDesc* **cudaCreateChannelDesc**<*cudaChannelFormatKindUnsignedBlockCompressed3S*

template<>

__host__ *cudaChannelFormatDesc* **cudaCreateChannelDesc**<*cudaChannelFormatKindUnsignedBlockCompressed4*>

template<>
__host__ *cudaChannelFormatDesc* **cudaCreateChannelDesc**<*cudaChannelFormatKindUnsignedBlockCompressed5*>

template<>
__host__ *cudaChannelFormatDesc* **cudaCreateChannelDesc**<*cudaChannelFormatKindUnsignedBlockCompressed6H*

template<>
__host__ *cudaChannelFormatDesc* **cudaCreateChannelDesc**<*cudaChannelFormatKindUnsignedBlockCompressed7*>

template<>
__host__ *cudaChannelFormatDesc* **cudaCreateChannelDesc**<*cudaChannelFormatKindUnsignedBlockCompressed7S*

template<>
__host__ *cudaChannelFormatDesc* **cudaCreateChannelDesc**<*cudaChannelFormatKindUnsignedNormalized1010102*

template<>
__host__ *cudaChannelFormatDesc* **cudaCreateChannelDesc**<*cudaChannelFormatKindUnsignedNormalized16X1*>(vo

template<>
__host__ *cudaChannelFormatDesc* **cudaCreateChannelDesc**<*cudaChannelFormatKindUnsignedNormalized16X2*>(vo

template<>
__host__ *cudaChannelFormatDesc* **cudaCreateChannelDesc**<*cudaChannelFormatKindUnsignedNormalized16X4*>(vo

template<>
__host__ *cudaChannelFormatDesc* **cudaCreateChannelDesc**<*cudaChannelFormatKindUnsignedNormalized8X1*>(voi

template<>
__host__ *cudaChannelFormatDesc* **cudaCreateChannelDesc**<*cudaChannelFormatKindUnsignedNormalized8X2*>(voi

template<>
__host__ *cudaChannelFormatDesc* **cudaCreateChannelDesc**<*cudaChannelFormatKindUnsignedNormalized8X4*>(voi

Warning: doxygenfunction: Unable to resolve function "cudaCreateChannelDesc< float >" with arguments "(void)". Candidate function could not be parsed. Parsing error is Error when parsing function declaration. If the function has no return type: Error in declarator or parameters-and-qualifiers Invalid C++ declaration: Expected identifier in nested name, got keyword: else [error at 15] template<> else __host__ cudaChannelFormatDesc cudaCreateChannelDesc< float > (void) ——————^ If the function has a return type: Invalid C++ declaration: Expected identifier in nested name, got keyword: else [error at 15] template<> else __host__ cudaChannelFormatDesc cudaCreateChannelDesc< float > (void) ——————^

template<>
__host__ *cudaChannelFormatDesc* **cudaCreateChannelDesc**<float1>(void)

template<>
__host__ *cudaChannelFormatDesc* **cudaCreateChannelDesc**<float2>(void)

template<>
__host__ *cudaChannelFormatDesc* **cudaCreateChannelDesc**<float4>(void)

template<>

__host__ *cudaChannelFormatDesc* **cudaCreateChannelDesc**<int>(void)

template<>
__host__ *cudaChannelFormatDesc* **cudaCreateChannelDesc**<int1>(void)

template<>
__host__ *cudaChannelFormatDesc* **cudaCreateChannelDesc**<int2>(void)

template<>
__host__ *cudaChannelFormatDesc* **cudaCreateChannelDesc**<int4>(void)

template<>
__host__ *cudaChannelFormatDesc* **cudaCreateChannelDesc**<long>(void)

template<>
__host__ *cudaChannelFormatDesc* **cudaCreateChannelDesc**<long1>(void)

template<>
__host__ *cudaChannelFormatDesc* **cudaCreateChannelDesc**<long2>(void)

template<>
__host__ *cudaChannelFormatDesc* **cudaCreateChannelDesc**<long4>(void)

template<>
__host__ *cudaChannelFormatDesc* **cudaCreateChannelDesc**<short>(void)

template<>
__host__ *cudaChannelFormatDesc* **cudaCreateChannelDesc**<short1>(void)

template<>
__host__ *cudaChannelFormatDesc* **cudaCreateChannelDesc**<short2>(void)

template<>
__host__ *cudaChannelFormatDesc* **cudaCreateChannelDesc**<short4>(void)

template<>
__host__ *cudaChannelFormatDesc* **cudaCreateChannelDesc**<signed char>(void)

template<>
__host__ *cudaChannelFormatDesc* **cudaCreateChannelDesc**<uchar1>(void)

template<>
__host__ *cudaChannelFormatDesc* **cudaCreateChannelDesc**<uchar2>(void)

template<>
__host__ *cudaChannelFormatDesc* **cudaCreateChannelDesc**<uchar4>(void)

template<>
__host__ *cudaChannelFormatDesc* **cudaCreateChannelDesc**<uint1>(void)

template<>
__host__ *cudaChannelFormatDesc* **cudaCreateChannelDesc**<uint2>(void)

template<>
__host__ *cudaChannelFormatDesc* **cudaCreateChannelDesc**<uint4>(void)

template<>

__host__ *cudaChannelFormatDesc* **cudaCreateChannelDesc**<ulong1>(void)

template<>
__host__ *cudaChannelFormatDesc* **cudaCreateChannelDesc**<ulong2>(void)

template<>
__host__ *cudaChannelFormatDesc* **cudaCreateChannelDesc**<ulong4>(void)

template<>
__host__ *cudaChannelFormatDesc* **cudaCreateChannelDesc**<unsigned char>(void)

template<>
__host__ *cudaChannelFormatDesc* **cudaCreateChannelDesc**<unsigned int>(void)

template<>
__host__ *cudaChannelFormatDesc* **cudaCreateChannelDesc**<unsigned long>(void)

template<>
__host__ *cudaChannelFormatDesc* **cudaCreateChannelDesc**<unsigned short>(void)

template<>
__host__ *cudaChannelFormatDesc* **cudaCreateChannelDesc**<ushort1>(void)

template<>
__host__ *cudaChannelFormatDesc* **cudaCreateChannelDesc**<ushort2>(void)

template<>
__host__ *cudaChannelFormatDesc* **cudaCreateChannelDesc**<ushort4>(void)

__host__ *cudaChannelFormatDesc* **cudaCreateChannelDescHalf**(void)

__host__ *cudaChannelFormatDesc* **cudaCreateChannelDescHalf1**(void)

__host__ *cudaChannelFormatDesc* **cudaCreateChannelDescHalf2**(void)

__host__ *cudaChannelFormatDesc* **cudaCreateChannelDescHalf4**(void)

__host__ *cudaChannelFormatDesc* **cudaCreateChannelDescNV12**(void)

__host__ *cudaError_t* **cudaEventCreate**(*cudaEvent_t* *event, unsigned int flags)

Creates an event object with the specified flags.

Creates an event object with the specified flags. Valid flags include:

- *cudaEventDefault*: Default event creation flag.
- *cudaEventBlockingSync*: Specifies that event should use blocking synchronization. A host thread that uses *cudaEventSynchronize()* to wait on an event created with this flag will block until the event actually completes.
- *cudaEventDisableTiming*: Specifies that the created event does not need to record timing data. Events created with this flag specified and the *cudaEventBlockingSync* flag not specified will provide the best performance when used with *cudaStreamWaitEvent()* and *cudaEventQuery()*.

See also:

cudaEventCreate (C API), *cudaEventCreateWithFlags*, *cudaEventRecord*, *cudaEventQuery*, *cudaEventSynchronize*, *cudaEventDestroy*, *cudaEventElapsedTime*, *cudaStreamWaitEvent*

Note: Note that this function may also return error codes from previous, asynchronous launches.

Note: Note that this function may also return *cudaErrorInitializationError*, *cudaErrorInsufficientDriver* or *cudaErrorNoDevice* if this call tries to initialize internal CUDA RT state.

Note: Note that as specified by *cudaStreamAddCallback* no CUDA function may be called from callback. *cudaErrorNotPermitted* may, but is not guaranteed to, be returned as a diagnostic in such case.

Parameters

- **event** – - Newly created event
- **flags** – - Flags for new event

Returns

cudaSuccess, *cudaErrorInvalidValue*, *cudaErrorLaunchFailure*, *cudaErrorMemoryAllocation*

template<class **T**>
__host__ *cudaError_t* **cudaFuncGetAttributes**(struct *cudaFuncAttributes* *attr, *T* *entry)

Find out attributes for a given function.

This function obtains the attributes of a function specified via `entry`. The parameter `entry` must be a pointer to a function that executes on the device. The parameter specified by `entry` must be declared as a `__global__` function. The fetched attributes are placed in `attr`. If the specified function does not exist, then *cudaErrorInvalidDeviceFunction* is returned.

Note that some function attributes such as *maxThreadsPerBlock* may vary based on the device that is currently being used.

cudaLaunchKernel (C++ API), *cudaFuncSetCacheConfig (C++ API)*, *cudaFuncGetAttributes (C API)*, ::cudaSetDoubleForDevice, ::cudaSetDoubleForHost

Note: Note that this function may also return error codes from previous, asynchronous launches.

Note: Note that this function may also return *cudaErrorInitializationError*, *cudaErrorInsufficientDriver* or *cudaErrorNoDevice* if this call tries to initialize internal CUDA RT state.

Note: Note that as specified by *cudaStreamAddCallback* no CUDA function may be called from callback. *cudaErrorNotPermitted* may, but is not guaranteed to, be returned as a diagnostic in such case.

Note: The API can also be used with a kernel *cudaKernel_t* by querying the handle using *cudaLibraryGetKernel()* or *cudaGetKernel* and then passing it to the API by casting to void*. The symbol

`entryFuncAddr` passed to *cudaGetKernel* should be a symbol that is registered with the same CUDA Runtime instance.

Note: Passing a symbol that belongs that belongs to a different runtime instance will result in undefined behavior. The only type that can be reliably passed to a different runtime instance is *cudaKernel_t*

Parameters

- **attr** – - Return pointer to function's attributes
- **entry** – - Function to get attributes of

Returns

cudaSuccess, *cudaErrorInvalidDeviceFunction*

template<class **T**>
__host__ *cudaError_t* **cudaFuncGetName**(const char **name, *T* *func)

Returns the function name for a device entry function pointer.

Returns in `**name` the function name associated with the symbol `func` . The function name is returned as a null-terminated string. This API may return a mangled name if the function is not declared as having C linkage. If `**name` is NULL, *cudaErrorInvalidValue* is returned. If `func` is not a device entry function, *cudaErrorInvalidDeviceFunction* is returned.

cudaFuncGetName (C API)

Note: Note that this function may also return error codes from previous, asynchronous launches.

Note: Note that this function may also return *cudaErrorInitializationError*, *cudaErrorInsufficientDriver* or *cudaErrorNoDevice* if this call tries to initialize internal CUDA RT state.

Note: Note that as specified by *cudaStreamAddCallback* no CUDA function may be called from callback. *cudaErrorNotPermitted* may, but is not guaranteed to, be returned as a diagnostic in such case.

Note: The API can also be used with a kernel *cudaKernel_t* by querying the handle using *cudaLibraryGetKernel()* or *cudaGetKernel* and then passing it to the API by casting to void*. The symbol `entryFuncAddr` passed to *cudaGetKernel* should be a symbol that is registered with the same CUDA Runtime instance.

Note: Passing a symbol that belongs that belongs to a different runtime instance will result in undefined behavior. The only type that can be reliably passed to a different runtime instance is *cudaKernel_t*

Parameters

- **name** – - The returned name of the function
- **func** – - The function pointer to retrieve name for

Returns
cudaSuccess, *cudaErrorInvalidValue*, *cudaErrorInvalidDeviceFunction*

template<class **T**>
__host__ *cudaError_t* **cudaFuncSetAttribute**(*T* *func, enum *cudaFuncAttribute* attr, int value)

Set attributes for a given function.

This function sets the attributes of a function specified via `entry`. The parameter `entry` must be a pointer to a function that executes on the device. The parameter specified by `entry` must be declared as a `__global__` function. The enumeration defined by `attr` is set to the value defined by `value`. If the specified function does not exist, then *cudaErrorInvalidDeviceFunction* is returned. If the specified attribute cannot be written, or if the value is incorrect, then *cudaErrorInvalidValue* is returned.

Valid values for `attr` are:

- *cudaFuncAttributeMaxDynamicSharedMemorySize* - The requested maximum size in bytes of dynamically-allocated shared memory. The sum of this value and the function attribute ::sharedSizeBytes cannot exceed the device attribute *cudaDevAttrMaxSharedMemoryPerBlockOptin*. The maximal size of requestable dynamic shared memory may differ by GPU architecture.
- *cudaFuncAttributePreferredSharedMemoryCarveout* - On devices where the L1 cache and shared memory use the same hardware resources, this sets the shared memory carveout preference, in percent of the total shared memory. See *cudaDevAttrMaxSharedMemoryPerMultiprocessor*. This is only a hint, and the driver can choose a different ratio if required to execute the function.
- *cudaFuncAttributeRequiredClusterWidth*: The required cluster width in blocks. The width, height, and depth values must either all be 0 or all be positive. The validity of the cluster dimensions is checked at launch time. If the value is set during compile time, it cannot be set at runtime. Setting it at runtime will return cudaErrorNotPermitted.
- *cudaFuncAttributeRequiredClusterHeight*: The required cluster height in blocks. The width, height, and depth values must either all be 0 or all be positive. The validity of the cluster dimensions is checked at launch time. If the value is set during compile time, it cannot be set at runtime. Setting it at runtime will return cudaErrorNotPermitted.
- *cudaFuncAttributeRequiredClusterDepth*: The required cluster depth in blocks. The width, height, and depth values must either all be 0 or all be positive. The validity of the cluster dimensions is checked at launch time. If the value is set during compile time, it cannot be set at runtime. Setting it at runtime will return cudaErrorNotPermitted.
- *cudaFuncAttributeNonPortableClusterSizeAllowed*: Indicates whether the function can be launched with non-portable cluster size. 1 is allowed, 0 is disallowed.
- *cudaFuncAttributeClusterSchedulingPolicyPreference*: The block scheduling policy of a function. The value type is cudaClusterSchedulingPolicy.

cudaLaunchKernel (C++ API), *cudaFuncSetCacheConfig (C++ API)*, *cudaFuncGetAttributes (C API)*, ::cudaSetDoubleForDevice, ::cudaSetDoubleForHost

Note: Note that this function may also return error codes from previous, asynchronous launches.

Note: Note that this function may also return *cudaErrorInitializationError*, *cudaErrorInsufficientDriver* or *cudaErrorNoDevice* if this call tries to initialize internal CUDA RT state.

Note: Note that as specified by *cudaStreamAddCallback* no CUDA function may be called from callback. *cudaErrorNotPermitted* may, but is not guaranteed to, be returned as a diagnostic in such case.

Note: The API can also be used with a kernel *cudaKernel_t* by querying the handle using *cudaLibraryGetKernel()* or *cudaGetKernel* and then passing it to the API by casting to void*. The symbol `entryFuncAddr` passed to *cudaGetKernel* should be a symbol that is registered with the same CUDA Runtime instance.

Note: Passing a symbol that belongs that belongs to a different runtime instance will result in undefined behavior. The only type that can be reliably passed to a different runtime instance is *cudaKernel_t*

Parameters

- **entry** – - Function to get attributes of
- **attr** – - Attribute to set
- **value** – - Value to set

Returns

cudaSuccess, *cudaErrorInvalidDeviceFunction*, *cudaErrorInvalidValue*

template<class **T**>
__host__ *cudaError_t* **cudaFuncSetCacheConfig**(*T* *func, enum *cudaFuncCache* cacheConfig)

Sets the preferred cache configuration for a device function.

On devices where the L1 cache and shared memory use the same hardware resources, this sets through `cacheConfig` the preferred cache configuration for the function specified via `func`. This is only a preference. The runtime will use the requested configuration if possible, but it is free to choose a different configuration if required to execute `func`.

`func` must be a pointer to a function that executes on the device. The parameter specified by `func` must be declared as a `__global__` function. If the specified function does not exist, then *cudaErrorInvalidDeviceFunction* is returned.

This setting does nothing on devices where the size of the L1 cache and shared memory are fixed.

Launching a kernel with a different preference than the most recent preference setting may insert a device-side synchronization point.

The supported cache configurations are:

- *cudaFuncCachePreferNone*: no preference for shared memory or L1 (default)
- *cudaFuncCachePreferShared*: prefer larger shared memory and smaller L1 cache
- *cudaFuncCachePreferL1*: prefer larger L1 cache and smaller shared memory

cudaLaunchKernel (C++ API), *cudaFuncSetCacheConfig (C API)*, *cudaFuncGetAttributes (C++ API)*, ::cudaSetDoubleForDevice, ::cudaSetDoubleForHost, ::cudaThreadGetCacheConfig, ::cudaThreadSetCacheConfig

---

Note: Note that this function may also return error codes from previous, asynchronous launches.

Note: Note that this function may also return *cudaErrorInitializationError*, *cudaErrorInsufficientDriver* or *cudaErrorNoDevice* if this call tries to initialize internal CUDA RT state.

Note: Note that as specified by *cudaStreamAddCallback* no CUDA function may be called from callback. *cudaErrorNotPermitted* may, but is not guaranteed to, be returned as a diagnostic in such case.

Parameters

- **func** – - device function pointer
- **cacheConfig** – - Requested cache configuration

Returns

cudaSuccess, *cudaErrorInvalidDeviceFunction*

template<class **T**>
__host__ *cudaError_t* **cudaFuncSetSharedMemConfig**(*T* *func, enum *cudaSharedMemConfig* config)

template<class **T**>
__host__ *cudaError_t* **cudaGetKernel**(*cudaKernel_t* *kernelPtr, *T* *func)

Get pointer to device kernel that matches entry function `entryFuncAddr`.

Returns in `kernelPtr` the device kernel corresponding to the entry function `entryFuncAddr`.

See also:

cudaGetKernel (C API)

Parameters

- **kernelPtr** – - Returns the device kernel
- **entryFuncAddr** – - Address of device entry function to search kernel for

Returns

cudaSuccess

template<class **T**>

__host__ *cudaError_t* **cudaGetSymbolAddress**(void **devPtr, const *T* &symbol)

Finds the address associated with a CUDA symbol.

Returns in `*devPtr` the address of symbol `symbol` on the device. `symbol` can either be a variable that resides in global or constant memory space. If `symbol` cannot be found, or if `symbol` is not declared in the global or constant memory space, `*devPtr` is unchanged and the error *cudaErrorInvalidSymbol* is returned.

See also:

cudaGetSymbolAddress (C API), *cudaGetSymbolSize (C++ API)*

Note: Note that this function may also return error codes from previous, asynchronous launches.

Note: Note that this function may also return *cudaErrorInitializationError*, *cudaErrorInsufficientDriver* or *cudaErrorNoDevice* if this call tries to initialize internal CUDA RT state.

Note: Note that as specified by *cudaStreamAddCallback* no CUDA function may be called from callback. *cudaErrorNotPermitted* may, but is not guaranteed to, be returned as a diagnostic in such case.

Parameters

- **devPtr** – - Return device pointer associated with symbol
- **symbol** – - Device symbol reference

Returns

cudaSuccess, *cudaErrorInvalidSymbol*, *cudaErrorNoKernelImageForDevice*

template<class **T**>
__host__ *cudaError_t* **cudaGetSymbolSize**(size_t *size, const *T* &symbol)

Finds the size of the object associated with a CUDA symbol.

Returns in `*size` the size of symbol `symbol`. `symbol` must be a variable that resides in global or constant memory space. If `symbol` cannot be found, or if `symbol` is not declared in global or constant memory space, `*size` is unchanged and the error *cudaErrorInvalidSymbol* is returned.

See also:

cudaGetSymbolAddress (C++ API), *cudaGetSymbolSize (C API)*

Note: Note that this function may also return error codes from previous, asynchronous launches.

Note: Note that this function may also return *cudaErrorInitializationError*, *cudaErrorInsufficientDriver* or *cudaErrorNoDevice* if this call tries to initialize internal CUDA RT state.

Note: Note that as specified by *cudaStreamAddCallback* no CUDA function may be called from callback. *cudaErrorNotPermitted* may, but is not guaranteed to, be returned as a diagnostic in such case.

Parameters

- **size** – - Size of object associated with symbol
- **symbol** – - Device symbol reference

Returns

cudaSuccess, *cudaErrorInvalidSymbol*, *cudaErrorNoKernelImageForDevice*

template<class **T**>
__host__ *cudaError_t* **cudaGraphAddMemcpyNodeFromSymbol**(*cudaGraphNode_t* *pGraphNode, *cudaGraph_t* graph, const *cudaGraphNode_t* *pDependencies, size_t numDependencies, void *dst, const *T* &symbol, size_t count, size_t offset, enum *cudaMemcpyKind* kind)

Creates a memcpy node to copy from a symbol on the device and adds it to a graph.

Creates a new memcpy node to copy from `symbol` and adds it to `graph` with `numDependencies` dependencies specified via `pDependencies`. It is possible for `numDependencies` to be 0, in which case the node will be placed at the root of the graph. `pDependencies` may not have any duplicate entries. A handle to the new node will be returned in `pGraphNode`.

When the graph is launched, the node will copy `count` bytes from the memory area pointed to by `offset` bytes from the start of symbol `symbol` to the memory area pointed to by `dst`. The memory areas may not overlap. `symbol` is a variable that resides in global or constant memory space. `kind` can be either *cudaMemcpyDeviceToHost*, *cudaMemcpyDeviceToDevice*, or *cudaMemcpyDefault*. Passing *cudaMemcpyDefault* is recommended, in which case the type of transfer is inferred from the pointer values. However, *cudaMemcpyDefault* is only allowed on systems that support unified virtual addressing.

Memcpy nodes have some additional restrictions with regards to managed memory, if the system contains at least one device which has a zero value for the device attribute *cudaDevAttrConcurrentManagedAccess*.

See also:

cudaMemcpyFromSymbol, *cudaGraphAddMemcpyNode*, *cudaGraphAddMemcpyNodeToSymbol*, *cudaGraphMemcpyNodeGetParams*, *cudaGraphMemcpyNodeSetParams*, *cudaGraphMemcpyNodeSetParamsFromSymbol*, *cudaGraphMemcpyNodeSetParamsToSymbol*, *cudaGraphCreate*, *cudaGraphDestroyNode*, *cudaGraphAddChildGraphNode*, *cudaGraphAddEmptyNode*, *cudaGraphAddKernelNode*, *cudaGraphAddHostNode*, *cudaGraphAddMemsetNode*

Note: Graph objects are not threadsafe. More here.

Note: Note that this function may also return error codes from previous, asynchronous launches.

Note: Note that this function may also return *cudaErrorInitializationError*, *cudaErrorInsufficientDriver* or *cudaErrorNoDevice* if this call tries to initialize internal CUDA RT state.

Note: Note that as specified by *cudaStreamAddCallback* no CUDA function may be called from callback. *cudaErrorNotPermitted* may, but is not guaranteed to, be returned as a diagnostic in such case.

Parameters

- **pGraphNode** – - Returns newly created node
- **graph** – - Graph to which to add the node
- **pDependencies** – - Dependencies of the node
- **numDependencies** – - Number of dependencies
- **dst** – - Destination memory address
- **symbol** – - Device symbol address
- **count** – - Size in bytes to copy
- **offset** – - Offset from start of symbol in bytes
- **kind** – - Type of transfer

Returns
cudaSuccess, *cudaErrorInvalidValue*

template<class **T**>
__host__ *cudaError_t* **cudaGraphAddMemcpyNodeToSymbol**(*cudaGraphNode_t* *pGraphNode, *cudaGraph_t* graph, const *cudaGraphNode_t* *pDependencies, size_t numDependencies, const *T* &symbol, const void *src, size_t count, size_t offset, enum *cudaMemcpyKind* kind)

Creates a memcpy node to copy to a symbol on the device and adds it to a graph.

Creates a new memcpy node to copy to `symbol` and adds it to `graph` with `numDependencies` dependencies specified via `pDependencies`. It is possible for `numDependencies` to be 0, in which case the node will be placed at the root of the graph. `pDependencies` may not have any duplicate entries. A handle to the new node will be returned in `pGraphNode`.

When the graph is launched, the node will copy `count` bytes from the memory area pointed to by `src` to the memory area pointed to by `offset` bytes from the start of symbol `symbol`. The memory areas may not overlap. `symbol` is a variable that resides in global or constant memory space. `kind` can be either *cudaMemcpyHostToDevice*, *cudaMemcpyDeviceToDevice*, or *cudaMemcpyDefault*. Passing *cudaMemcpyDefault* is recommended, in which case the type of transfer is inferred from the pointer values. However, *cudaMemcpyDefault* is only allowed on systems that support unified virtual addressing.

Memcpy nodes have some additional restrictions with regards to managed memory, if the system contains at least one device which has a zero value for the device attribute *cudaDevAttrConcurrentManagedAccess*.

See also:

cudaMemcpyToSymbol, *cudaGraphAddMemcpyNode*, *cudaGraphAddMemcpyNodeFromSymbol*, *cudaGraphMemcpyNodeGetParams*, *cudaGraphMemcpyNodeSetParams*, *cudaGraphMemcpyNodeSetParamsToSymbol*, *cudaGraphMemcpyNodeSetParamsFromSymbol*, *cudaGraphCreate*, *cudaGraphDestroyNode*, *cudaGraphAddChildGraphNode*, *cudaGraphAddEmptyNode*, *cudaGraphAddKernelNode*, *cudaGraphAddHostNode*, *cudaGraphAddMemsetNode*

Note: Graph objects are not threadsafe. More here.

Note: Note that this function may also return error codes from previous, asynchronous launches.

Note: Note that this function may also return *cudaErrorInitializationError*, *cudaErrorInsufficientDriver* or *cudaErrorNoDevice* if this call tries to initialize internal CUDA RT state.

Note: Note that as specified by *cudaStreamAddCallback* no CUDA function may be called from callback. *cudaErrorNotPermitted* may, but is not guaranteed to, be returned as a diagnostic in such case.

Parameters

- **pGraphNode** – - Returns newly created node
- **graph** – - Graph to which to add the node
- **pDependencies** – - Dependencies of the node
- **numDependencies** – - Number of dependencies
- **symbol** – - Device symbol address
- **src** – - Source memory address
- **count** – - Size in bytes to copy
- **offset** – - Offset from start of symbol in bytes
- **kind** – - Type of transfer

Returns

cudaSuccess, *cudaErrorInvalidValue*

template<class **T**>
__host__ *cudaError_t* **cudaGraphExecMemcpyNodeSetParamsFromSymbol**(*cudaGraphExec_t* hGraphExec, *cudaGraphNode_t* node, void *dst, const *T* &symbol, size_t count, size_t offset, enum *cudaMemcpyKind* kind)

Sets the parameters for a memcpy node in the given graphExec to copy from a symbol on the device.

Updates the work represented by `node` in `hGraphExec` as though `node` had contained the given params at instantiation. `node` must remain in the graph which was used to instantiate `hGraphExec`. Changed edges to and from `node` are ignored.

`symbol` and `dst` must be allocated from the same contexts as the original source and destination memory. The instantiation-time memory operands must be 1-dimensional. Zero-length operations are not supported.

The modifications only affect future launches of `hGraphExec`. Already enqueued or running launches of `hGraphExec` are not affected by this call. `node` is also not modified by this call.

Returns *cudaErrorInvalidValue* if the memory operands' mappings changed or the original memory operands are multidimensional.

See also:

cudaGraphAddMemcpyNode, *cudaGraphAddMemcpyNodeFromSymbol*, *cudaGraphMemcpyNodeSetParams*, *cudaGraphMemcpyNodeSetParamsFromSymbol*, *cudaGraphInstantiate*, *cudaGraphExecMemcpyNodeSetParams*, *cudaGraphExecMemcpyNodeSetParamsToSymbol*, *cudaGraphExecKernelNodeSetParams*, *cudaGraphExecMemsetNodeSetParams*, *cudaGraphExecHostNodeSetParams*

Note: Graph objects are not threadsafe. More here.

Note: Note that this function may also return error codes from previous, asynchronous launches.

Note: Note that this function may also return *cudaErrorInitializationError*, *cudaErrorInsufficientDriver* or *cudaErrorNoDevice* if this call tries to initialize internal CUDA RT state.

Note: Note that as specified by *cudaStreamAddCallback* no CUDA function may be called from callback. *cudaErrorNotPermitted* may, but is not guaranteed to, be returned as a diagnostic in such case.

Parameters

- **`hGraphExec`** – - The executable graph in which to set the specified node
- **`node`** – - Memcpy node from the graph which was used to instantiate graphExec
- **`dst`** – - Destination memory address
- **`symbol`** – - Device symbol address
- **`count`** – - Size in bytes to copy
- **`offset`** – - Offset from start of symbol in bytes
- **`kind`** – - Type of transfer

Returns
cudaSuccess, *cudaErrorInvalidValue*

template<class **T**>
__host__ *cudaError_t* **cudaGraphExecMemcpyNodeSetParamsToSymbol**(*cudaGraphExec_t* hGraphExec, *cudaGraphNode_t* node, const *T* &symbol, const void *src, size_t count, size_t offset, enum *cudaMemcpyKind* kind)

Sets the parameters for a memcpy node in the given graphExec to copy to a symbol on the device.

Updates the work represented by `node` in `hGraphExec` as though `node` had contained the given params at instantiation. `node` must remain in the graph which was used to instantiate `hGraphExec`. Changed edges to and from `node` are ignored.

`src` and `symbol` must be allocated from the same contexts as the original source and destination memory. The instantiation-time memory operands must be 1-dimensional. Zero-length operations are not supported.

The modifications only affect future launches of `hGraphExec`. Already enqueued or running launches of `hGraphExec` are not affected by this call. `node` is also not modified by this call.

Returns *cudaErrorInvalidValue* if the memory operands' mappings changed or the original memory operands are multidimensional.

See also:

cudaGraphAddMemcpyNode, *cudaGraphAddMemcpyNodeToSymbol*, *cudaGraphMemcpyNodeSetParams*, *cudaGraphMemcpyNodeSetParamsToSymbol*, *cudaGraphInstantiate*, *cudaGraphExecMemcpyNodeSetParams*, *cudaGraphExecMemcpyNodeSetParamsFromSymbol*, *cudaGraphExecKernelNodeSetParams*, *cudaGraphExecMemsetNodeSetParams*, *cudaGraphExecHostNodeSetParams*

Note: Graph objects are not threadsafe. More here.

Note: Note that this function may also return error codes from previous, asynchronous launches.

Note: Note that this function may also return *cudaErrorInitializationError*, *cudaErrorInsufficientDriver* or *cudaErrorNoDevice* if this call tries to initialize internal CUDA RT state.

Note: Note that as specified by *cudaStreamAddCallback* no CUDA function may be called from callback. *cudaErrorNotPermitted* may, but is not guaranteed to, be returned as a diagnostic in such case.

Parameters

- **hGraphExec** – - The executable graph in which to set the specified node

- **node** – - Memcpy node from the graph which was used to instantiate graphExec
- **symbol** – - Device symbol address
- **src** – - Source memory address
- **count** – - Size in bytes to copy
- **offset** – - Offset from start of symbol in bytes
- **kind** – - Type of transfer

Returns

cudaSuccess, *cudaErrorInvalidValue*

__host__ *cudaError_t* **cudaGraphExecUpdate**(*cudaGraphExec_t* hGraphExec, *cudaGraph_t* hGraph, *cudaGraphNode_t* *hErrorNode_out, enum *cudaGraphExecUpdateResult* *updateResult_out)

__host__ *cudaError_t* **cudaGraphInstantiate**(*cudaGraphExec_t* *pGraphExec, *cudaGraph_t* graph, *cudaGraphNode_t* *pErrorNode, char *pLogBuffer, size_t bufferSize)

Creates an executable graph from a graph.

Instantiates `graph` as an executable graph. The graph is validated for any structural constraints or intra-node constraints which were not previously validated. If instantiation is successful, a handle to the instantiated graph is returned in `pGraphExec`.

If there are any errors, diagnostic information may be returned in `pErrorNode` and `pLogBuffer`. This is the primary way to inspect instantiation errors. The output will be null terminated unless the diagnostics overflow the buffer. In this case, they will be truncated, and the last byte can be inspected to determine if truncation occurred.

See also:

cudaGraphInstantiateWithFlags, *cudaGraphCreate*, *cudaGraphUpload*, *cudaGraphLaunch*, *cudaGraphExecDestroy*

Note: Graph objects are not threadsafe. More here.

Note: Note that this function may also return error codes from previous, asynchronous launches.

Note: Note that this function may also return *cudaErrorInitializationError*, *cudaErrorInsufficientDriver* or *cudaErrorNoDevice* if this call tries to initialize internal CUDA RT state.

Note: Note that as specified by *cudaStreamAddCallback* no CUDA function may be called from callback. *cudaErrorNotPermitted* may, but is not guaranteed to, be returned as a diagnostic in such case.

Parameters

- **pGraphExec** – - Returns instantiated graph
- **graph** – - Graph to instantiate
- **pErrorNode** – - In case of an instantiation error, this may be modified to indicate a node contributing to the error
- **pLogBuffer** – - A character buffer to store diagnostic messages
- **bufferSize** – - Size of the log buffer in bytes

Returns

cudaSuccess, *cudaErrorInvalidValue*

template<class **T**>
__host__ *cudaError_t* **cudaGraphMemcpyNodeSetParamsFromSymbol**(*cudaGraphNode_t* node, void *dst, const *T* &symbol, size_t count, size_t offset, enum *cudaMemcpyKind* kind)

Sets a memcpy node's parameters to copy from a symbol on the device.

Sets the parameters of memcpy node `node` to the copy described by the provided parameters.

When the graph is launched, the node will copy `count` bytes from the memory area pointed to by `offset` bytes from the start of symbol `symbol` to the memory area pointed to by `dst`. The memory areas may not overlap. `symbol` is a variable that resides in global or constant memory space. `kind` can be either *cudaMemcpyDeviceToHost*, *cudaMemcpyDeviceToDevice*, or *cudaMemcpyDefault*. Passing *cudaMemcpyDefault* is recommended, in which case the type of transfer is inferred from the pointer values. However, *cudaMemcpyDefault* is only allowed on systems that support unified virtual addressing.

See also:

cudaMemcpyFromSymbol, *cudaGraphMemcpyNodeSetParams*, *cudaGraphMemcpyNodeSetParamsToSymbol*, *cudaGraphAddMemcpyNode*, *cudaGraphMemcpyNodeGetParams*

Note: Graph objects are not threadsafe. More here.

Note: Note that this function may also return error codes from previous, asynchronous launches.

Note: Note that this function may also return *cudaErrorInitializationError*, *cudaErrorInsufficientDriver* or *cudaErrorNoDevice* if this call tries to initialize internal CUDA RT state.

Note: Note that as specified by *cudaStreamAddCallback* no CUDA function may be called from callback. *cudaErrorNotPermitted* may, but is not guaranteed to, be returned as a diagnostic in such case.

Parameters

- **node** – - Node to set the parameters for

- **dst** – - Destination memory address
- **symbol** – - Device symbol address
- **count** – - Size in bytes to copy
- **offset** – - Offset from start of symbol in bytes
- **kind** – - Type of transfer

Returns

cudaSuccess, *cudaErrorInvalidValue*

template<class **T**>
__host__ *cudaError_t* **cudaGraphMemcpyNodeSetParamsToSymbol**(*cudaGraphNode_t* node, const *T* &symbol, const void *src, size_t count, size_t offset, enum *cudaMemcpyKind* kind)

Sets a memcpy node's parameters to copy to a symbol on the device.

Sets the parameters of memcpy node `node` to the copy described by the provided parameters.

When the graph is launched, the node will copy `count` bytes from the memory area pointed to by `src` to the memory area pointed to by `offset` bytes from the start of symbol `symbol`. The memory areas may not overlap. `symbol` is a variable that resides in global or constant memory space. `kind` can be either *cudaMemcpyHostToDevice*, *cudaMemcpyDeviceToDevice*, or *cudaMemcpyDefault*. Passing *cudaMemcpyDefault* is recommended, in which case the type of transfer is inferred from the pointer values. However, *cudaMemcpyDefault* is only allowed on systems that support unified virtual addressing.

See also:

cudaMemcpyToSymbol, *cudaGraphMemcpyNodeSetParams*, *cudaGraphMemcpyNodeSetParamsFromSymbol*, *cudaGraphAddMemcpyNode*, *cudaGraphMemcpyNodeGetParams*

Note: Graph objects are not threadsafe. More here.

Note: Note that this function may also return error codes from previous, asynchronous launches.

Note: Note that this function may also return *cudaErrorInitializationError*, *cudaErrorInsufficientDriver* or *cudaErrorNoDevice* if this call tries to initialize internal CUDA RT state.

Note: Note that as specified by *cudaStreamAddCallback* no CUDA function may be called from callback. *cudaErrorNotPermitted* may, but is not guaranteed to, be returned as a diagnostic in such case.

Parameters

- **node** – - Node to set the parameters for
- **symbol** – - Device symbol address

- **src** – - Source memory address
- **count** – - Size in bytes to copy
- **offset** – - Offset from start of symbol in bytes
- **kind** – - Type of transfer

Returns
cudaSuccess, *cudaErrorInvalidValue*

template<class **T**>
__host__ *cudaError_t* **cudaHostAlloc**(*T* **ptr, size_t size, unsigned int flags)

template<class **T**>
__host__ *cudaError_t* **cudaHostGetDevicePointer**(*T* **pDevice, void *pHost, unsigned int flags)

template<class **T**>
__host__ *cudaError_t* **cudaLaunchCooperativeKernel**(*T* *func, dim3 gridDim, dim3 blockDim, void **args, size_t sharedMem = 0, *cudaStream_t* stream = 0)

Launches a device function.

The function invokes kernel `func` on `gridDim` (`gridDim.xgridDim.ygridDim.z`) grid of blocks. Each block contains `blockDim` (`blockDim.xblockDim.yblockDim.z`) threads.

The device on which this kernel is invoked must have a non-zero value for the device attribute *cudaDevAttrCooperativeLaunch*.

The total number of blocks launched cannot exceed the maximum number of blocks per multiprocessor as returned by *cudaOccupancyMaxActiveBlocksPerMultiprocessor* (or *cudaOccupancyMaxActiveBlocksPerMultiprocessorWithFlags*) times the number of multiprocessors as specified by the device attribute *cudaDevAttrMultiProcessorCount*.

The kernel cannot make use of CUDA dynamic parallelism.

If the kernel has N parameters the `args` should point to array of N pointers. Each pointer, from `args[0]` to `args[N - 1]`, point to the region of memory from which the actual parameter will be copied.

`sharedMem` sets the amount of dynamic shared memory that will be available to each thread block.

`stream` specifies a stream the invocation is associated to.

cudaLaunchCooperativeKernel (C API)

Note: Note that this function may also return error codes from previous, asynchronous launches.

Note: This function exhibits asynchronous behavior for most use cases.

Note: This function uses standard default stream semantics.

Note: Note that this function may also return *cudaErrorInitializationError*, *cudaErrorInsufficientDriver* or *cudaErrorNoDevice* if this call tries to initialize internal CUDA RT state.

Note: Note that as specified by *cudaStreamAddCallback* no CUDA function may be called from callback. *cudaErrorNotPermitted* may, but is not guaranteed to, be returned as a diagnostic in such case.

Note: The API can also be used with a kernel *cudaKernel_t* by querying the handle using *cudaLibraryGetKernel()* or *cudaGetKernel* and then passing it to the API by casting to void*. The symbol `entryFuncAddr` passed to *cudaGetKernel* should be a symbol that is registered with the same CUDA Runtime instance.

Note: Passing a symbol that belongs that belongs to a different runtime instance will result in undefined behavior. The only type that can be reliably passed to a different runtime instance is *cudaKernel_t*

Parameters

- **func** – - Device function symbol
- **gridDim** – - Grid dimentions
- **blockDim** – - Block dimentions
- **args** – - Arguments
- **sharedMem** – - Shared memory (defaults to 0)
- **stream** – - Stream identifier (defaults to NULL)

Returns

cudaSuccess, *cudaErrorInvalidDeviceFunction*, *cudaErrorInvalidConfiguration*, *cudaErrorLaunchFailure*, *cudaErrorLaunchTimeout*, *cudaErrorLaunchOutOfResources*, *cudaErrorSharedObjectInitFailed*

template<class **T**>
__host__ *cudaError_t* **cudaLaunchKernel**(*T* *func, dim3 gridDim, dim3 blockDim, void **args, size_t sharedMem = 0, *cudaStream_t* stream = 0)

Launches a device function.

The function invokes kernel `func` on `gridDim` (`gridDim.xgridDim.ygridDim.z`) grid of blocks. Each block contains `blockDim` (`blockDim.xblockDim.yblockDim.z`) threads.

If the kernel has N parameters the `args` should point to array of N pointers. Each pointer, from `args[0]` to `args[N - 1]`, point to the region of memory from which the actual parameter will be copied.

`sharedMem` sets the amount of dynamic shared memory that will be available to each thread block.

`stream` specifies a stream the invocation is associated to.

cudaLaunchKernel (C API)

Note: Note that this function may also return error codes from previous, asynchronous launches.

Note: This function exhibits asynchronous behavior for most use cases.

Note: This function uses standard default stream semantics.

Note: Note that this function may also return *cudaErrorInitializationError*, *cudaErrorInsufficientDriver* or *cudaErrorNoDevice* if this call tries to initialize internal CUDA RT state.

Note: Note that as specified by *cudaStreamAddCallback* no CUDA function may be called from callback. *cudaErrorNotPermitted* may, but is not guaranteed to, be returned as a diagnostic in such case.

Note: The API can also be used with a kernel *cudaKernel_t* by querying the handle using *cudaLibraryGetKernel()* or *cudaGetKernel* and then passing it to the API by casting to void*. The symbol `entryFuncAddr` passed to *cudaGetKernel* should be a symbol that is registered with the same CUDA Runtime instance.

Note: Passing a symbol that belongs that belongs to a different runtime instance will result in undefined behavior. The only type that can be reliably passed to a different runtime instance is *cudaKernel_t*

Parameters

- **func** – - Device function symbol
- **gridDim** – - Grid dimentions
- **blockDim** – - Block dimentions
- **args** – - Arguments
- **sharedMem** – - Shared memory (defaults to 0)
- **stream** – - Stream identifier (defaults to NULL)

Returns

cudaSuccess, *cudaErrorInvalidDeviceFunction*, *cudaErrorInvalidConfiguration*, *cudaErrorLaunchFailure*, *cudaErrorLaunchTimeout*, *cudaErrorLaunchOutOfResources*, *cudaErrorSharedObjectInitFailed*, *cudaErrorInvalidPtx*, *cudaErrorUnsupportedPtxVersion*, *cudaErrorNoKernelImageForDevice*, *cudaErrorJitCompilerNotFound*, *cudaErrorJitCompilationDisabled*

template<typename ...**ActTypes**>

__host__ *cudaError_t* **cudaLaunchKernelEx**(const *cudaLaunchConfig_t* *config, const *cudaKernel_t* kernel, *ActTypes*&&... args)

Launches a CUDA function with launch-time configuration.

Invokes the kernel `kernel` on `config->gridDim` (`config->gridDim.xconfig->gridDim.yconfig->gridDim.z`) grid of blocks. Each block contains `config->blockDim` (`config->blockDim.xconfig->blockDim.yconfig->blockDim.z`) threads.

`config->dynamicSmemBytes` sets the amount of dynamic shared memory that will be available to each thread block.

`config->stream` specifies a stream the invocation is associated to.

Configuration beyond grid and block dimensions, dynamic shared memory size, and stream can be provided with the following two fields of `config:`

`config->attrs` is an array of `config->numAttrs` contiguous *cudaLaunchAttribute* elements. The value of this pointer is not considered if `config->numAttrs` is zero. However, in that case, it is recommended to set the pointer to NULL. `config->numAttrs` is the number of attributes populating the first `config->numAttrs` positions of the `config->attrs` array.

The kernel arguments should be passed as arguments to this function via the `args` parameter pack.

The C API version of this function, `cudaLaunchKernelExC`, is also available for pre-C++11 compilers and for use cases where the ability to pass kernel parameters via void* array is preferable.

See also:

cudaLaunchKernelEx (C API), ::cuLaunchKernelEx

Note: This function uses standard default stream semantics.

Note: Note that this function may also return error codes from previous, asynchronous launches.

Note: Note that this function may also return *cudaErrorInitializationError*, *cudaErrorInsufficientDriver* or *cudaErrorNoDevice* if this call tries to initialize internal CUDA RT state.

Note: Note that as specified by *cudaStreamAddCallback* no CUDA function may be called from callback. *cudaErrorNotPermitted* may, but is not guaranteed to, be returned as a diagnostic in such case.

Parameters

- **config** – - Launch configuration
- **func** – - Kernel to launch
- **args** – - Parameter pack of kernel parameters

Returns

cudaSuccess, *cudaErrorInvalidDeviceFunction*, *cudaErrorInvalidConfiguration*, *cudaErrorLaunchFailure*, *cudaErrorLaunchTimeout*, *cudaErrorLaunchOutOfResources*, *cudaErrorSharedObjectInitFailed*, *cudaErrorInvalidPtx*, *cudaErrorUnsupportedPtxVersion*, *cudaErrorNoKernelImageForDevice*, *cudaErrorJitCompilerNotFound*, *cudaErrorJitCompilationDisabled*

template<typename ...**ExpTypes**, typename ...**ActTypes**>
__host__ *cudaError_t* **cudaLaunchKernelEx**(const *cudaLaunchConfig_t* *config, void (*kernel)(*ExpTypes*...), *ActTypes*&&... args)

Launches a CUDA function with launch-time configuration.

Invokes the kernel `kernel` on `config->gridDim` (`config->gridDim.xconfig->gridDim.yconfig->gridDim.z`) grid of blocks. Each block contains `config->blockDim` (`config->blockDim.xconfig->blockDim.yconfig->blockDim.z`) threads.

`config->dynamicSmemBytes` sets the amount of dynamic shared memory that will be available to each thread block.

`config->stream` specifies a stream the invocation is associated to.

Configuration beyond grid and block dimensions, dynamic shared memory size, and stream can be provided with the following two fields of `config:`

`config->attrs` is an array of `config->numAttrs` contiguous *cudaLaunchAttribute* elements. The value of this pointer is not considered if `config->numAttrs` is zero. However, in that case, it is recommended to set the pointer to NULL. `config->numAttrs` is the number of attributes populating the first `config->numAttrs` positions of the `config->attrs` array.

The kernel arguments should be passed as arguments to this function via the `args` parameter pack.

The C API version of this function, `cudaLaunchKernelExC`, is also available for pre-C++11 compilers and for use cases where the ability to pass kernel parameters via void* array is preferable.

See also:

cudaLaunchKernelEx (C API), ::cuLaunchKernelEx

Note: This function uses standard default stream semantics.

Note: Note that this function may also return error codes from previous, asynchronous launches.

Note: Note that this function may also return *cudaErrorInitializationError*, *cudaErrorInsufficientDriver* or *cudaErrorNoDevice* if this call tries to initialize internal CUDA RT state.

Note: Note that as specified by *cudaStreamAddCallback* no CUDA function may be called from callback. *cudaErrorNotPermitted* may, but is not guaranteed to, be returned as a diagnostic in such case.

Note: The API can also be used with a kernel *cudaKernel_t* by querying the handle using *cudaLibraryGetKernel()* or *cudaGetKernel* and then passing it to the API by casting to void*. The symbol `entryFuncAddr` passed to *cudaGetKernel* should be a symbol that is registered with the same CUDA Runtime instance.

Note: Passing a symbol that belongs that belongs to a different runtime instance will result in undefined behavior. The only type that can be reliably passed to a different runtime instance is *cudaKernel_t*

Parameters

- **config** – - Launch configuration
- **kernel** – - Kernel to launch
- **args** – - Parameter pack of kernel parameters

Returns

cudaSuccess, *cudaErrorInvalidDeviceFunction*, *cudaErrorInvalidConfiguration*, *cudaErrorLaunchFailure*, *cudaErrorLaunchTimeout*, *cudaErrorLaunchOutOfResources*, *cudaErrorSharedObjectInitFailed*, *cudaErrorInvalidPtx*, *cudaErrorUnsupportedPtxVersion*, *cudaErrorNoKernelImageForDevice*, *cudaErrorJitCompilerNotFound*, *cudaErrorJitCompilationDisabled*

template<class **T**>
__host__ *cudaError_t* **cudaLibraryGetGlobal**(*T* **dptr, size_t *bytes, *cudaLibrary_t* library, const char *name)

Returns a global device pointer.

Returns in `*dptr` and `*bytes` the base pointer and size of the global with name `name` for the requested library `library` and the current device. If no global for the requested name `name` exists, the call returns *cudaErrorSymbolNotFound*. One of the parameters `dptr` or `bytes` (not both) can be NULL in which case it is ignored.

See also:

cudaLibraryLoadData, *cudaLibraryLoadFromFile*, *cudaLibraryUnload*, *cudaLibraryGetManaged*

Parameters

- **dptr** – - Returned global device pointer for the requested library
- **bytes** – - Returned global size in bytes
- **library** – - Library to retrieve global from
- **name** – - Name of global to retrieve

Returns

cudaSuccess, *cudaErrorCudartUnloading*, *cudaErrorInitializationError*, *cudaErrorInvalidValue*, *cudaErrorInvalidResourceHandle*, *cudaErrorSymbolNotFound* *cudaErrorDeviceUninitialized*, *cudaErrorContextIsDestroyed*

template<class **T**>
__host__ *cudaError_t* **cudaLibraryGetManaged**(*T* **dptr, size_t *bytes, *cudaLibrary_t* library, const char *name)

Returns a pointer to managed memory.

Returns in `*dptr` and `*bytes` the base pointer and size of the managed memory with name `name` for the requested library `library`. If no managed memory with the requested name `name` exists, the call returns *cudaErrorSymbolNotFound*. One of the parameters `dptr` or `bytes` (not both) can be NULL in which case it is ignored. Note that managed memory for library `library` is shared across devices and is registered when the library is loaded.

See also:

cudaLibraryLoadData, *cudaLibraryLoadFromFile*, *cudaLibraryUnload*, *cudaLibraryGetGlobal*

Parameters

- **dptr** – - Returned pointer to the managed memory
- **bytes** – - Returned memory size in bytes
- **library** – - Library to retrieve managed memory from
- **name** – - Name of managed memory to retrieve

Returns
cudaSuccess, *cudaErrorCudartUnloading*, *cudaErrorInitializationError*, *cudaErrorInvalidValue*, *cudaErrorInvalidResourceHandle*, *cudaErrorSymbolNotFound*

template<class **T**>
__host__ *cudaError_t* **cudaLibraryGetUnifiedFunction**(*T* **fptr, *cudaLibrary_t* library, const char *symbol)

Returns a pointer to a unified function.

Returns in `*fptr` the function pointer to a unified function denoted by `symbol`. If no unified function with name `symbol` exists, the call returns *cudaErrorSymbolNotFound*. If there is no device with attribute *cudaDeviceProp::unifiedFunctionPointers* present in the system, the call may return *cudaErrorSymbolNotFound*.

See also:

cudaLibraryLoadData, *cudaLibraryLoadFromFile*, *cudaLibraryUnload*

Parameters

- **fptr** – - Returned pointer to a unified function
- **library** – - Library to retrieve function pointer memory from
- **symbol** – - Name of function pointer to retrieve

Returns
cudaSuccess, *cudaErrorCudartUnloading*, *cudaErrorInitializationError*, *cudaErrorInvalidValue*, *cudaErrorInvalidResourceHandle*, *cudaErrorSymbolNotFound*

template<class **T**>

__host__ *cudaError_t* **cudaMalloc**(*T* **devPtr, size_t size)

template<class **T**>
__host__ *cudaError_t* **cudaMallocAsync**(*T* **ptr, size_t size, *cudaStream_t* stream)

__host__ *cudaError_t* **cudaMallocAsync**(void **ptr, size_t size, *cudaMemPool_t* memPool, *cudaStream_t* stream)

Allocate from a pool.

This is an alternate spelling for cudaMallocFromPoolAsync made available through function overloading.

See also:

cudaMallocFromPoolAsync, *cudaMallocAsync (C API)*

template<class **T**>
__host__ *cudaError_t* **cudaMallocAsync**(*T* **ptr, size_t size, *cudaMemPool_t* memPool, *cudaStream_t* stream)

template<class **T**>
__host__ *cudaError_t* **cudaMallocFromPoolAsync**(*T* **ptr, size_t size, *cudaMemPool_t* memPool, *cudaStream_t* stream)

template<class **T**>
__host__ *cudaError_t* **cudaMallocHost**(*T* **ptr, size_t size, unsigned int flags = 0)

__host__ *cudaError_t* **cudaMallocHost**(void **ptr, size_t size, unsigned int flags)

Allocates page-locked memory on the host.

Allocates `size` bytes of host memory that is page-locked and accessible to the device. The driver tracks the virtual memory ranges allocated with this function and automatically accelerates calls to functions such as *cudaMemcpy()*. Since the memory can be accessed directly by the device, it can be read or written with much higher bandwidth than pageable memory obtained with functions such as ::malloc(). Allocating excessive amounts of pinned memory may degrade system performance, since it reduces the amount of memory available to the system for paging. As a result, this function is best used sparingly to allocate staging areas for data exchange between host and device.

The `flags` parameter enables different options to be specified that affect the allocation, as follows.

- *cudaHostAllocDefault*: This flag's value is defined to be 0.
- *cudaHostAllocPortable*: The memory returned by this call will be considered as pinned memory by all CUDA contexts, not just the one that performed the allocation.
- *cudaHostAllocMapped*: Maps the allocation into the CUDA address space. The device pointer to the memory may be obtained by calling cudaHostGetDevicePointer().
- *cudaHostAllocWriteCombined*: Allocates the memory as write-combined (WC). WC memory can be transferred across the PCI Express bus more quickly on some system configurations, but cannot be read efficiently by most CPUs. WC memory is a good option for buffers that will be written by the CPU and read by the device via mapped pinned memory or host->device transfers.

All of these flags are orthogonal to one another: a developer may allocate memory that is portable, mapped and/or write-combined with no restrictions.

cudaSetDeviceFlags() must have been called with the *cudaDeviceMapHost* flag in order for the *cudaHostAllocMapped* flag to have any effect.

The *cudaHostAllocMapped* flag may be specified on CUDA contexts for devices that do not support mapped pinned memory. The failure is deferred to cudaHostGetDevicePointer() because the memory may be mapped into other CUDA contexts via the *cudaHostAllocPortable* flag.

Memory allocated by this function must be freed with *cudaFreeHost()*.

See also:

cudaSetDeviceFlags, *cudaMallocHost (C API)*, *cudaFreeHost*, cudaHostAlloc

---

Note: Note that this function may also return error codes from previous, asynchronous launches.

Note: Note that this function may also return *cudaErrorInitializationError*, *cudaErrorInsufficientDriver* or *cudaErrorNoDevice* if this call tries to initialize internal CUDA RT state.

Note: Note that as specified by *cudaStreamAddCallback* no CUDA function may be called from callback. *cudaErrorNotPermitted* may, but is not guaranteed to, be returned as a diagnostic in such case.

Parameters

- **ptr** – - Device pointer to allocated memory
- **size** – - Requested allocation size in bytes
- **flags** – - Requested properties of allocated memory

Returns

cudaSuccess, *cudaErrorMemoryAllocation*

template<class **T**>
__host__ *cudaError_t* **cudaMallocManaged**(*T* **devPtr, size_t size, unsigned int flags = 0x01)

Allocates memory that will be automatically managed by the Unified Memory system.

Allocates `size` bytes of managed memory on the device and returns in `*devPtr` a pointer to the allocated memory. If the device doesn't support allocating managed memory, *cudaErrorNotSupported* is returned. Support for managed memory can be queried using the device attribute *cudaDevAttrManagedMemory*. The allocated memory is suitably aligned for any kind of variable. The memory is not cleared. If `size` is 0, *cudaMallocManaged* returns *cudaErrorInvalidValue*. The pointer is valid on the CPU and on all GPUs in the system that support managed memory. All accesses to this pointer must obey the Unified Memory programming model.

`flags` specifies the default stream association for this allocation. `flags` must be one of *cudaMemAttachGlobal* or *cudaMemAttachHost*. The default value for `flags` is *cudaMemAttachGlobal*. If *cudaMemAttachGlobal* is specified, then this memory is accessible from any stream on any device. If *cudaMemAttachHost* is specified, then the allocation should not be accessed from devices that have a zero value for the device attribute *cudaDevAttrConcurrentManagedAccess*; an explicit call to *cudaStreamAttachMemAsync* will be required to enable access on such devices.

If the association is later changed via *cudaStreamAttachMemAsync* to a single stream, the default association, as specifed during *cudaMallocManaged*, is restored when that stream is destroyed. For **managed** variables, the default association is always *cudaMemAttachGlobal*. Note that destroying a stream is an asynchronous operation, and as a result, the change to default association won't happen until all work in the stream has completed.

Memory allocated with *cudaMallocManaged* should be released with *cudaFree*.

Device memory oversubscription is possible for GPUs that have a non-zero value for the device attribute *cudaDevAttrConcurrentManagedAccess*. Managed memory on such GPUs may be evicted from device memory to host memory at any time by the Unified Memory driver in order to make room for other allocations.

In a multi-GPU system where all GPUs have a non-zero value for the device attribute *cudaDevAttrConcurrentManagedAccess*, managed memory may not be populated when this API returns and instead may be populated on access. In such systems, managed memory can migrate to any processor's memory at any time. The Unified Memory driver will employ heuristics to maintain data locality and prevent excessive page faults to the extent possible. The application can also guide the driver about memory usage patterns via *cudaMemAdvise*. The application can also explicitly migrate memory to a desired processor's memory via *cudaMemPrefetchAsync*.

In a multi-GPU system where all of the GPUs have a zero value for the device attribute *cudaDevAttrConcurrentManagedAccess* and all the GPUs have peer-to-peer support with each other, the physical storage for managed memory is created on the GPU which is active at the time *cudaMallocManaged* is called. All other GPUs will reference the data at reduced bandwidth via peer mappings over the PCIe bus. The Unified Memory driver does not migrate memory among such GPUs.

In a multi-GPU system where not all GPUs have peer-to-peer support with each other and where the value of the device attribute *cudaDevAttrConcurrentManagedAccess* is zero for at least one of those GPUs, the location chosen for physical storage of managed memory is system-dependent.

- On Linux, the location chosen will be device memory as long as the current set of active contexts are on devices that either have peer-to-peer support with each other or have a non-zero value for the device attribute *cudaDevAttrConcurrentManagedAccess*. If there is an active context on a GPU that does not have a non-zero value for that device attribute and it does not have peer-to-peer support with the other devices that have active contexts on them, then the location for physical storage will be 'zero-copy' or host memory. Note that this means that managed memory that is located in device memory is migrated to host memory if a new context is created on a GPU that doesn't have a non-zero value for the device attribute and does not support peer-to-peer with at least one of the other devices that has an active context. This in turn implies that context creation may fail if there is insufficient host memory to migrate all managed allocations.
- On Windows, the physical storage is always created in 'zero-copy' or host memory. All GPUs will reference the data at reduced bandwidth over the PCIe bus. In these circumstances, use of the environment variable CUDA_VISIBLE_DEVICES is recommended to restrict CUDA to only use those GPUs that have peer-to-peer support. Alternatively, users can also set CUDA_MANAGED_FORCE_DEVICE_ALLOC to a non-zero value to force the driver to always use device memory for physical storage. When this environment variable is set to a non-zero value, all devices used in that process that support managed memory have to be peer-to-peer compatible with each other. The error *cudaErrorInvalidDevice* will be returned if a device that supports managed memory is used and it is not peer-to-peer compatible with any of the other managed memory supporting devices that were previously used in that process, even if *cudaDeviceReset* has been called on those devices. These environment variables are described in the CUDA programming guide under the "CUDA environment variables" section.
- On ARM, managed memory is not available on discrete gpu with Drive PX-2.

See also:

cudaMallocPitch, *cudaFree*, *cudaMallocArray*, *cudaFreeArray*, *cudaMalloc3D*, *cudaMalloc3DArray*, *cudaMallocHost (C API)*, *cudaFreeHost*, cudaHostAlloc, *cudaDeviceGetAttribute*, *cudaStreamAttachMemAsync*

Note: Note that this function may also return *cudaErrorInitializationError*, *cudaErrorInsufficientDriver* or *cudaErrorNoDevice* if this call tries to initialize internal CUDA RT state.

Note: Note that as specified by *cudaStreamAddCallback* no CUDA function may be called from callback. *cudaErrorNotPermitted* may, but is not guaranteed to, be returned as a diagnostic in such case.

Parameters

- **flags** – Memory can be accessed by any stream on any device
- **devPtr** – - Pointer to allocated device memory
- **size** – - Requested allocation size in bytes
- **flags** – - Must be either *cudaMemAttachGlobal* or *cudaMemAttachHost* (defaults to *cudaMemAttachGlobal*)

Returns

cudaSuccess, *cudaErrorMemoryAllocation*, *cudaErrorNotSupported*, *cudaErrorInvalidValue*

template<class **T**>
__host__ *cudaError_t* **cudaMallocPitch**(*T* **devPtr, size_t *pitch, size_t width, size_t height)

template<typename **T**>
__host__ *cudaError_t* **cudaMemDiscardAndPrefetchBatchAsync**(*T* **dptrs, size_t *sizes, size_t count, struct *cudaMemLocation* prefetchLocs, unsigned long long flags, *cudaStream_t* stream)

Performs a batch of memory discard and prefetches asynchronously.

This is an alternate spelling for cudaMemDiscardAndPrefetchBatchAsync made available through function overloading.

The *cudaMemLocation* specified by `prefetchLocs` are applicable for all the operations in the batch.

See also:

cudaMemDiscardAndPrefetchBatchAsync

template<typename **T**>

__host__ *cudaError_t* **cudaMemDiscardAndPrefetchBatchAsync**(*T* **dptrs, size_t *sizes, size_t count, struct *cudaMemLocation* *prefetchLocs, size_t *prefetchLocIdxs, size_t numPrefetchLocs, unsigned long long flags, *cudaStream_t* stream)

Performs a batch of memory discard and prefetches asynchronously.

This is an alternate spelling for cudaMemDiscardAndPrefetchBatchAsync made available through function overloading.

See also:

cudaMemDiscardAndPrefetchBatchAsync

template<typename **T**>
__host__ *cudaError_t* **cudaMemPrefetchBatchAsync**(*T* **dptrs, size_t *sizes, size_t count, struct *cudaMemLocation* prefetchLocs, unsigned long long flags, *cudaStream_t* stream)

Performs a batch of memory prefetches asynchronously.

This is an alternate spelling for cudaMemPrefetchBatchAsync made available through function overloading.

The *cudaMemLocation* specified by `prefetchLocs` are applicable for all the prefetches specified in the batch.

See also:

cudaMemPrefetchBatchAsync

template<typename **T**>
__host__ *cudaError_t* **cudaMemPrefetchBatchAsync**(*T* **dptrs, size_t *sizes, size_t count, struct *cudaMemLocation* *prefetchLocs, size_t *prefetchLocIdxs, size_t numPrefetchLocs, unsigned long long flags, *cudaStream_t* stream)

Performs a batch of memory prefetches asynchronously.

This is an alternate spelling for cudaMemPrefetchBatchAsync made available through function overloading.

See also:

cudaMemPrefetchBatchAsync

template<typename **T**, typename **U**>
__host__ *cudaError_t* **cudaMemcpyAsync**(*T* dst, *U* src, const size_t size, struct *cudaMemcpyAttributes* *attr, *cudaStream_t* hStream)

Performs a memory copy asynchronously.

Allows specifying attributes for the copy.

This is an alternate spelling for cudaMemcpyAsync made available through function overloading.

See also:

cudaMemcpyAsync *cudaMemcpyBatchAsync* *cudaMemcpyWithAttributesAsync*

template<typename **T**, typename **U**>
__host__ *cudaError_t* **cudaMemcpyBatchAsync**(*T* *const *dsts, *U* *const *srcs, const size_t *sizes, size_t count, struct *cudaMemcpyAttributes* *attrs, size_t *attrsIdxs, size_t numAttrs, *cudaStream_t* hStream)

Performs a batch of memory copies asynchronously.

This is an alternate spelling for cudaMemcpyBatchAsync made available through function overloading.

See also:

cudaMemcpyBatchAsync

template<typename **T**, typename **U**>
__host__ *cudaError_t* **cudaMemcpyBatchAsync**(*T* *const *dsts, *U* *const *srcs, const size_t *sizes, size_t count, struct *cudaMemcpyAttributes* attr, *cudaStream_t* hStream)

Performs a batch of memory copies asynchronously.

This is an alternate spelling for cudaMemcpyBatchAsync made available through function overloading.

The *cudaMemcpyAttributes* specified by `attr` are applicable for all the copies specified in the batch.

See also:

cudaMemcpyBatchAsync

template<class **T**>
__host__ *cudaError_t* **cudaMemcpyFromSymbol**(void *dst, const *T* &symbol, size_t count, size_t offset = 0, enum *cudaMemcpyKind* kind = *cudaMemcpyDeviceToHost*)

Copies data from the given symbol on the device.

Copies `count` bytes from the memory area `offset` bytes from the start of symbol `symbol` to the memory area pointed to by `dst`. The memory areas may not overlap. `symbol` is a variable that resides in global or constant memory space. `kind` can be either *cudaMemcpyDeviceToHost* or *cudaMemcpyDeviceToDevice*.

See also:

cudaMemcpy, *cudaMemcpy2D*, *cudaMemcpy2DToArray*, *cudaMemcpy2DFromArray*, *cudaMemcpy2DArrayToArray*, *cudaMemcpyToSymbol*, *cudaMemcpyAsync*, *cudaMemcpy2DAsync*, *cudaMemcpy2DToArrayAsync*, *cudaMemcpy2DFromArrayAsync*, *cudaMemcpyToSymbolAsync*, *cudaMemcpyFromSymbolAsync*

Note: Note that this function may also return error codes from previous, asynchronous launches.

Note: This function exhibits synchronous behavior for most use cases.

Note: Use of a string naming a variable as the `symbol` parameter was deprecated in CUDA 4.1 and removed in CUDA 5.0.

Note: Note that this function may also return *cudaErrorInitializationError*, *cudaErrorInsufficientDriver* or *cudaErrorNoDevice* if this call tries to initialize internal CUDA RT state.

Note: Note that as specified by *cudaStreamAddCallback* no CUDA function may be called from callback. *cudaErrorNotPermitted* may, but is not guaranteed to, be returned as a diagnostic in such case.

Parameters

- **dst** – - Destination memory address
- **symbol** – - Device symbol reference
- **count** – - Size in bytes to copy
- **offset** – - Offset from start of symbol in bytes
- **kind** – - Type of transfer

Returns

cudaSuccess, *cudaErrorInvalidValue*, *cudaErrorInvalidSymbol*, *cudaErrorInvalidMemcpyDirection*, *cudaErrorNoKernelImageForDevice*

template<class **T**>
__host__ *cudaError_t* **cudaMemcpyFromSymbolAsync**(void *dst, const *T* &symbol, size_t count, size_t offset = 0, enum *cudaMemcpyKind* kind = *cudaMemcpyDeviceToHost*, *cudaStream_t* stream = 0)

Copies data from the given symbol on the device.

Copies `count` bytes from the memory area `offset` bytes from the start of symbol `symbol` to the memory area pointed to by `dst`. The memory areas may not overlap. `symbol` is a variable that resides in global or constant memory space. `kind` can be either *cudaMemcpyDeviceToHost* or *cudaMemcpyDeviceToDevice*.

cudaMemcpyFromSymbolAsync() is asynchronous with respect to the host, so the call may return before the copy is complete. The copy can optionally be associated to a stream by passing a non-zero `stream` argument. If `kind` is *cudaMemcpyDeviceToHost* and `stream` is non-zero, the copy may overlap with operations in other streams.

See also:

cudaMemcpy, *cudaMemcpy2D*, *cudaMemcpy2DToArray*, *cudaMemcpy2DFromArray*, *cudaMemcpy2DArrayToArray*, *cudaMemcpyToSymbol*, *cudaMemcpyFromSymbol*, *cudaMemcpyAsync*, *cud-*

aMemcpy2DAsync, *cudaMemcpy2DToArrayAsync*, *cudaMemcpy2DFromArrayAsync*, *cudaMemcpyToSymbolAsync*

Note: Note that this function may also return error codes from previous, asynchronous launches.

Note: This function exhibits asynchronous behavior for most use cases.

Note: Use of a string naming a variable as the `symbol` parameter was deprecated in CUDA 4.1 and removed in CUDA 5.0.

Note: Note that this function may also return *cudaErrorInitializationError*, *cudaErrorInsufficientDriver* or *cudaErrorNoDevice* if this call tries to initialize internal CUDA RT state.

Note: Note that as specified by *cudaStreamAddCallback* no CUDA function may be called from callback. *cudaErrorNotPermitted* may, but is not guaranteed to, be returned as a diagnostic in such case.

Parameters

- **`dst`** – - Destination memory address
- **`symbol`** – - Device symbol reference
- **`count`** – - Size in bytes to copy
- **`offset`** – - Offset from start of symbol in bytes
- **`kind`** – - Type of transfer
- **`stream`** – - Stream identifier

Returns

cudaSuccess, *cudaErrorInvalidValue*, *cudaErrorInvalidSymbol*, *cudaErrorInvalidMemcpyDirection*, *cudaErrorNoKernelImageForDevice*

template<class **T**>
__host__ *cudaError_t* **`cudaMemcpyToSymbol`**(const *T* &symbol, const void *src, size_t count, size_t offset = 0, enum *cudaMemcpyKind* kind = *cudaMemcpyHostToDevice*)

Copies data to the given symbol on the device.

Copies `count` bytes from the memory area pointed to by `src` to the memory area `offset` bytes from the start of symbol `symbol`. The memory areas may not overlap. `symbol` is a variable that resides in global or constant memory space. `kind` can be either *cudaMemcpyHostToDevice* or *cudaMemcpyDeviceToDevice*.

See also:

cudaMemcpy, *cudaMemcpy2D*, *cudaMemcpy2DToArray*, *cudaMemcpy2DFromArray*, *cudaMemcpy2DArrayToArray*, *cudaMemcpyFromSymbol*, *cudaMemcpyAsync*, *cudaMemcpy2DAsync*, *cudaMemcpy2DToArrayAsync*, *cudaMemcpy2DFromArrayAsync*, *cudaMemcpyToSymbolAsync*, *cudaMemcpyFromSymbolAsync*

Note: Note that this function may also return error codes from previous, asynchronous launches.

Note: This function exhibits synchronous behavior for most use cases.

Note: Use of a string naming a variable as the `symbol` parameter was deprecated in CUDA 4.1 and removed in CUDA 5.0.

Note: Note that this function may also return *cudaErrorInitializationError*, *cudaErrorInsufficientDriver* or *cudaErrorNoDevice* if this call tries to initialize internal CUDA RT state.

Note: Note that as specified by *cudaStreamAddCallback* no CUDA function may be called from callback. *cudaErrorNotPermitted* may, but is not guaranteed to, be returned as a diagnostic in such case.

Parameters

- **`symbol`** – - Device symbol reference
- **`src`** – - Source memory address
- **`count`** – - Size in bytes to copy
- **`offset`** – - Offset from start of symbol in bytes
- **`kind`** – - Type of transfer

Returns

cudaSuccess, *cudaErrorInvalidValue*, *cudaErrorInvalidSymbol*, *cudaErrorInvalidMemcpyDirection*, *cudaErrorNoKernelImageForDevice*

template<class **T**>
__host__ *cudaError_t* **cudaMemcpyToSymbolAsync**(const *T* &symbol, const void *src, size_t count, size_t offset = 0, enum *cudaMemcpyKind* kind = *cudaMemcpyHostToDevice*, *cudaStream_t* stream = 0)

Copies data to the given symbol on the device.

Copies `count` bytes from the memory area pointed to by `src` to the memory area `offset` bytes from the start of symbol `symbol`. The memory areas may not overlap. `symbol` is a variable that resides in global or constant memory space. `kind` can be either *cudaMemcpyHostToDevice* or *cudaMemcpyDeviceToDevice*.

cudaMemcpyToSymbolAsync() is asynchronous with respect to the host, so the call may return before the copy is complete. The copy can optionally be associated to a stream by passing a non-zero `stream` argument. If `kind` is *cudaMemcpyHostToDevice* and `stream` is non-zero, the copy may overlap with operations in other streams.

See also:

cudaMemcpy, *cudaMemcpy2D*, *cudaMemcpy2DToArray*, *cudaMemcpy2DFromArray*, *cudaMemcpy2DArrayToArray*, *cudaMemcpyToSymbol*, *cudaMemcpyFromSymbol*, *cudaMemcpyAsync*, *cudaMemcpy2DAsync*, *cudaMemcpy2DToArrayAsync*, *cudaMemcpy2DFromArrayAsync*, *cudaMemcpyFromSymbolAsync*

Note: Note that this function may also return error codes from previous, asynchronous launches.

Note: This function exhibits asynchronous behavior for most use cases.

Note: Use of a string naming a variable as the `symbol` parameter was deprecated in CUDA 4.1 and removed in CUDA 5.0.

Note: Note that this function may also return *cudaErrorInitializationError*, *cudaErrorInsufficientDriver* or *cudaErrorNoDevice* if this call tries to initialize internal CUDA RT state.

Note: Note that as specified by *cudaStreamAddCallback* no CUDA function may be called from callback. *cudaErrorNotPermitted* may, but is not guaranteed to, be returned as a diagnostic in such case.

Parameters

- **`symbol`** – - Device symbol reference
- **`src`** – - Source memory address
- **`count`** – - Size in bytes to copy
- **`offset`** – - Offset from start of symbol in bytes
- **`kind`** – - Type of transfer
- **`stream`** – - Stream identifier

Returns

cudaSuccess, *cudaErrorInvalidValue*, *cudaErrorInvalidSymbol*, *cudaErrorInvalidMemcpyDirection*, *cudaErrorNoKernelImageForDevice*

template<class **T**>
__host__ *cudaError_t* **cudaOccupancyAvailableDynamicSMemPerBlock**(size_t *dynamicSmemSize, *T* *func, int numBlocks, int blockSize)

Returns dynamic shared memory available per block when launching `numBlocks` blocks on SM.

Returns in `*dynamicSmemSize` the maximum size of dynamic shared memory to allow `numBlocks` blocks per SM.

See also:

cudaOccupancyMaxPotentialBlockSize

See also:

cudaOccupancyMaxPotentialBlockSizeWithFlags

See also:

cudaOccupancyMaxActiveBlocksPerMultiprocessor

See also:

cudaOccupancyMaxActiveBlocksPerMultiprocessorWithFlags

See also:

cudaOccupancyMaxPotentialBlockSizeVariableSMem

See also:

cudaOccupancyMaxPotentialBlockSizeVariableSMemWithFlags

Note: Note that this function may also return error codes from previous, asynchronous launches.

Note: Note that this function may also return *cudaErrorInitializationError*, *cudaErrorInsufficientDriver* or *cudaErrorNoDevice* if this call tries to initialize internal CUDA RT state.

Note: Note that as specified by *cudaStreamAddCallback* no CUDA function may be called from callback. *cudaErrorNotPermitted* may, but is not guaranteed to, be returned as a diagnostic in such case.

Note: The API can also be used with a kernel *cudaKernel_t* by querying the handle using *cudaLibraryGetKernel()* or *cudaGetKernel* and then passing it to the API by casting to void*. The symbol `entryFuncAddr` passed to *cudaGetKernel* should be a symbol that is registered with the same CUDA Runtime instance.

Note: Passing a symbol that belongs that belongs to a different runtime instance will result in undefined behavior. The only type that can be reliably passed to a different runtime instance is *cudaKernel_t*

Parameters

- **`dynamicSmemSize`** – - Returned maximum dynamic shared memory

- **func** – - Kernel function for which occupancy is calculated
- **numBlocks** – - Number of blocks to fit on SM
- **blockSize** – - Size of the block

Returns

cudaSuccess, *cudaErrorInvalidDevice*, *cudaErrorInvalidDeviceFunction*, *cudaErrorInvalidValue*, *cudaErrorUnknown*,

template<class **T**>
__host__ *cudaError_t* **cudaOccupancyMaxActiveBlocksPerMultiprocessor**(int *numBlocks, *T* func, int blockSize, size_t dynamicSMemSize)

Returns occupancy for a device function.

Returns in `*numBlocks` the maximum number of active blocks per streaming multiprocessor for the device function.

See also:

cudaOccupancyMaxActiveBlocksPerMultiprocessorWithFlags

See also:

cudaOccupancyMaxPotentialBlockSize

See also:

cudaOccupancyMaxPotentialBlockSizeWithFlags

See also:

cudaOccupancyMaxPotentialBlockSizeVariableSMem

See also:

cudaOccupancyMaxPotentialBlockSizeVariableSMemWithFlags

See also:

cudaOccupancyAvailableDynamicSMemPerBlock

Note: Note that this function may also return error codes from previous, asynchronous launches.

Note: Note that this function may also return *cudaErrorInitializationError*, *cudaErrorInsufficientDriver* or *cudaErrorNoDevice* if this call tries to initialize internal CUDA RT state.

Note: Note that as specified by *cudaStreamAddCallback* no CUDA function may be called from callback. *cudaErrorNotPermitted* may, but is not guaranteed to, be returned as a diagnostic in such case.

Note: The API can also be used with a kernel *cudaKernel_t* by querying the handle using *cudaLibraryGetKernel()* or *cudaGetKernel* and then passing it to the API by casting to void*. The symbol

`entryFuncAddr` passed to *cudaGetKernel* should be a symbol that is registered with the same CUDA Runtime instance.

Note: Passing a symbol that belongs that belongs to a different runtime instance will result in undefined behavior. The only type that can be reliably passed to a different runtime instance is *cudaKernel_t*

Parameters

- **numBlocks** – - Returned occupancy
- **func** – - Kernel function for which occupancy is calulated
- **blockSize** – - Block size the kernel is intended to be launched with
- **dynamicSMemSize** – - Per-block dynamic shared memory usage intended, in bytes

Returns

cudaSuccess, *cudaErrorInvalidDevice*, *cudaErrorInvalidDeviceFunction*, *cudaErrorInvalidValue*, *cudaErrorUnknown*,

template<class **T**>
__host__ *cudaError_t* **cudaOccupancyMaxActiveBlocksPerMultiprocessorWithFlags**(int *numBlocks, *T* func, int blockSize, size_t dynamicSMemSize, unsigned int flags)

Returns occupancy for a device function with the specified flags.

Returns in `*numBlocks` the maximum number of active blocks per streaming multiprocessor for the device function.

The `flags` parameter controls how special cases are handled. Valid flags include:

- *cudaOccupancyDefault*: keeps the default behavior as *cudaOccupancyMaxActiveBlocksPerMultiprocessor*
- *cudaOccupancyDisableCachingOverride*: suppresses the default behavior on platform where global caching affects occupancy. On such platforms, if caching is enabled, but per-block SM resource usage would result in zero occupancy, the occupancy calculator will calculate the occupancy as if caching is disabled. Setting this flag makes the occupancy calculator

to return 0 in such cases. More information can be found about this feature in the “Unified L1/Texture Cache” section of the Maxwell tuning guide.

See also:

cudaOccupancyMaxActiveBlocksPerMultiprocessor

See also:

cudaOccupancyMaxPotentialBlockSize

See also:

cudaOccupancyMaxPotentialBlockSizeWithFlags

See also:

cudaOccupancyMaxPotentialBlockSizeVariableSMem

See also:

cudaOccupancyMaxPotentialBlockSizeVariableSMemWithFlags

See also:

cudaOccupancyAvailableDynamicSMemPerBlock

Note: Note that this function may also return error codes from previous, asynchronous launches.

Note: Note that this function may also return *cudaErrorInitializationError*, *cudaErrorInsufficientDriver* or *cudaErrorNoDevice* if this call tries to initialize internal CUDA RT state.

Note: Note that as specified by *cudaStreamAddCallback* no CUDA function may be called from callback. *cudaErrorNotPermitted* may, but is not guaranteed to, be returned as a diagnostic in such case.

Note: The API can also be used with a kernel *cudaKernel_t* by querying the handle using *cudaLibraryGetKernel()* or *cudaGetKernel* and then passing it to the API by casting to void*. The symbol `entryFuncAddr` passed to *cudaGetKernel* should be a symbol that is registered with the same CUDA Runtime instance.

Note: Passing a symbol that belongs that belongs to a different runtime instance will result in undefined behavior. The only type that can be reliably passed to a different runtime instance is *cudaKernel_t*

Parameters

- **`numBlocks`** – - Returned occupancy
- **`func`** – - Kernel function for which occupancy is calulated

- **blockSize** – - Block size the kernel is intended to be launched with
- **dynamicSMemSize** – - Per-block dynamic shared memory usage intended, in bytes
- **flags** – - Requested behavior for the occupancy calculator

Returns

cudaSuccess, *cudaErrorInvalidDevice*, *cudaErrorInvalidDeviceFunction*, *cudaErrorInvalidValue*, *cudaErrorUnknown*,

template<class **T**>
__host__ *cudaError_t* **cudaOccupancyMaxActiveClusters**(int *numClusters, *T* *func, const *cudaLaunchConfig_t* *config)

Given the kernel function (`func`) and launch configuration (`config`), return the maximum number of clusters that could co-exist on the target device in `*numClusters`.

If the function has required cluster size already set (see *cudaFuncGetAttributes*), the cluster size from config must either be unspecified or match the required size. Without required sizes, the cluster size must be specified in config, else the function will return an error.

Note that various attributes of the kernel function may affect occupancy calculation. Runtime environment may affect how the hardware schedules the clusters, so the calculated occupancy is not guaranteed to be achievable.

See also:

cudaFuncGetAttributes

Note: Note that this function may also return error codes from previous, asynchronous launches.

Note: Note that this function may also return *cudaErrorInitializationError*, *cudaErrorInsufficientDriver* or *cudaErrorNoDevice* if this call tries to initialize internal CUDA RT state.

Note: Note that as specified by *cudaStreamAddCallback* no CUDA function may be called from callback. *cudaErrorNotPermitted* may, but is not guaranteed to, be returned as a diagnostic in such case.

Note: The API can also be used with a kernel *cudaKernel_t* by querying the handle using *cudaLibraryGetKernel()* or *cudaGetKernel* and then passing it to the API by casting to void*. The symbol `entryFuncAddr` passed to *cudaGetKernel* should be a symbol that is registered with the same CUDA Runtime instance.

Note: Passing a symbol that belongs that belongs to a different runtime instance will result in undefined behavior. The only type that can be reliably passed to a different runtime instance is *cudaKernel_t*

Parameters

- **numClusters** – - Returned maximum number of clusters that could co-exist on the target device
- **func** – - Kernel function for which maximum number of clusters are calculated
- **config** – - Launch configuration for the given kernel function

Returns

cudaSuccess, *cudaErrorInvalidDeviceFunction*, *cudaErrorInvalidValue*, *cudaErrorInvalidClusterSize*, *cudaErrorUnknown*,

template<class **T**>
__host__ __device__ *cudaError_t* **cudaOccupancyMaxPotentialBlockSize**(int *minGridSize, int *blockSize, *T* func, size_t dynamicSMemSize = 0, int blockSizeLimit = 0)

Returns grid and block size that achieves maximum potential occupancy for a device function.

Returns in `*minGridSize` and `*blocksize` a suggested grid / block size pair that achieves the best potential occupancy (i.e. the maximum number of active warps with the smallest number of blocks).

Use

See also:

cudaOccupancyMaxPotentialBlockSizeVariableSMem if the amount of per-block dynamic shared memory changes with different block sizes.

See also:

cudaOccupancyMaxPotentialBlockSizeWithFlags

See also:

cudaOccupancyMaxActiveBlocksPerMultiprocessor

See also:

cudaOccupancyMaxActiveBlocksPerMultiprocessorWithFlags

See also:

cudaOccupancyMaxPotentialBlockSizeVariableSMem

See also:

cudaOccupancyMaxPotentialBlockSizeVariableSMemWithFlags

See also:

cudaOccupancyAvailableDynamicSMemPerBlock

Note: Note that this function may also return error codes from previous, asynchronous launches.

Note: Note that this function may also return *cudaErrorInitializationError*, *cudaErrorInsufficientDriver* or *cudaErrorNoDevice* if this call tries to initialize internal CUDA RT state.

Note: Note that as specified by *cudaStreamAddCallback* no CUDA function may be called from callback. *cudaErrorNotPermitted* may, but is not guaranteed to, be returned as a diagnostic in such case.

Parameters

- **minGridSize** – - Returned minimum grid size needed to achieve the best potential occupancy
- **blockSize** – - Returned block size
- **func** – - Device function symbol
- **dynamicSMemSize** – - Per-block dynamic shared memory usage intended, in bytes
- **blockSizeLimit** – - The maximum block size `func` is designed to work with. 0 means no limit.

Returns

cudaSuccess, *cudaErrorInvalidDevice*, *cudaErrorInvalidDeviceFunction*, *cudaErrorInvalidValue*, *cudaErrorUnknown*,

template<typename **UnaryFunction**, class **T**>
__host__ __device__ *cudaError_t* **cudaOccupancyMaxPotentialBlockSizeVariableSMem**(int *minGridSize, int *blockSize, *T* func, *UnaryFunction* blockSizeToDynamicSMemSize, int blockSizeLimit = 0)

Returns grid and block size that achieves maximum potential occupancy for a device function.

Returns in `*minGridSize` and `*blocksize` a suggested grid / block size pair that achieves the best potential occupancy (i.e. the maximum number of active warps with the smallest number

of blocks).

See also:

cudaOccupancyMaxPotentialBlockSizeVariableSMemWithFlags

See also:

cudaOccupancyMaxActiveBlocksPerMultiprocessor

See also:

cudaOccupancyMaxActiveBlocksPerMultiprocessorWithFlags

See also:

cudaOccupancyMaxPotentialBlockSize

See also:

cudaOccupancyMaxPotentialBlockSizeWithFlags

See also:

cudaOccupancyAvailableDynamicSMemPerBlock

Note: Note that this function may also return error codes from previous, asynchronous launches.

Note: Note that this function may also return *cudaErrorInitializationError*, *cudaErrorInsufficientDriver* or *cudaErrorNoDevice* if this call tries to initialize internal CUDA RT state.

Note: Note that as specified by *cudaStreamAddCallback* no CUDA function may be called from callback. *cudaErrorNotPermitted* may, but is not guaranteed to, be returned as a diagnostic in such case.

Parameters

- **minGridSize** – - Returned minimum grid size needed to achieve the best potential occupancy
- **blockSize** – - Returned block size
- **func** – - Device function symbol
- **blockSizeToDynamicSMemSize** – - A unary function / functor that takes block size, and returns the size, in bytes, of dynamic shared memory needed for a block
- **blockSizeLimit** – - The maximum block size `func` is designed to work with. 0 means no limit.

Returns

cudaSuccess, *cudaErrorInvalidDevice*, *cudaErrorInvalidDeviceFunction*, *cudaErrorInvalidValue*, *cudaErrorUnknown*,

template<typename **UnaryFunction**, class **T**>

__host__ __device__ *cudaError_t* **cudaOccupancyMaxPotentialBlockSizeVariableSMemWithFlags**(int *minGridSize, int *blockSize, *T* func, *UnaryFunction* blockSizeToDynamicSMemSize, int blockSizeLimit = 0, unsigned int flags = 0)

Returns grid and block size that achieves maximum potential occupancy for a device function.

Returns in `*minGridSize` and `*blocksize` a suggested grid / block size pair that achieves the best potential occupancy (i.e. the maximum number of active warps with the smallest number of blocks).

The `flags` parameter controls how special cases are handled. Valid flags include:

- *cudaOccupancyDefault*: keeps the default behavior as *cudaOccupancyMaxPotentialBlockSizeVariableSMemWithFlags*
- *cudaOccupancyDisableCachingOverride*: This flag suppresses the default behavior on platform where global caching affects occupancy. On such platforms, if caching is enabled, but per-block SM resource usage would result in zero occupancy, the occupancy calculator will calculate the occupancy as if caching is disabled. Setting this flag makes the occupancy calculator to return 0 in such cases. More information can be found about this feature in the "Unified L1/Texture Cache" section of the Maxwell tuning guide.

See also:

cudaOccupancyMaxPotentialBlockSizeVariableSMem

See also:

cudaOccupancyMaxActiveBlocksPerMultiprocessor

See also:

cudaOccupancyMaxActiveBlocksPerMultiprocessorWithFlags

See also:

cudaOccupancyMaxPotentialBlockSize

See also:

cudaOccupancyMaxPotentialBlockSizeWithFlags

See also:

cudaOccupancyAvailableDynamicSMemPerBlock

Note: Note that this function may also return error codes from previous, asynchronous launches.

Note: Note that this function may also return *cudaErrorInitializationError*, *cudaErrorInsufficientDriver* or *cudaErrorNoDevice* if this call tries to initialize internal CUDA RT state.

Note: Note that as specified by *cudaStreamAddCallback* no CUDA function may be called from callback. *cudaErrorNotPermitted* may, but is not guaranteed to, be returned as a diagnostic in such case.

Parameters

- **`minGridSize`** – - Returned minimum grid size needed to achieve the best potential occupancy
- **`blockSize`** – - Returned block size
- **`func`** – - Device function symbol
- **`blockSizeToDynamicSMemSize`** – - A unary function / functor that takes block size, and returns the size, in bytes, of dynamic shared memory needed for a block
- **`blockSizeLimit`** – - The maximum block size `func` is designed to work with. 0 means no limit.
- **`flags`** – - Requested behavior for the occupancy calculator

Returns

cudaSuccess, *cudaErrorInvalidDevice*, *cudaErrorInvalidDeviceFunction*, *cudaErrorInvalidValue*, *cudaErrorUnknown*,

template<class **T**>

__host__ __device__ *cudaError_t* **cudaOccupancyMaxPotentialBlockSizeWithFlags**(int *minGridSize, int *blockSize, *T* func, size_t dynamicSMemSize = 0, int blockSizeLimit = 0, unsigned int flags = 0)

Returns grid and block size that achived maximum potential occupancy for a device function with the specified flags.

Returns in `*minGridSize` and `*blocksize` a suggested grid / block size pair that achieves the best potential occupancy (i.e. the maximum number of active warps with the smallest number of blocks).

The `flags` parameter controls how special cases are handle. Valid flags include:

- *cudaOccupancyDefault*: keeps the default behavior as *cudaOccupancyMaxPotentialBlockSize*
- *cudaOccupancyDisableCachingOverride*: This flag suppresses the default behavior on platform where global caching affects occupancy. On such platforms, if caching is enabled, but per-block SM resource usage would result in zero occupancy, the occupancy calculator will calculate the occupancy as if caching is disabled. Setting this flag makes the occupancy calculator to return 0 in such cases. More information can be found about this feature in the "Unified L1/Texture Cache" section of the Maxwell tuning guide.

Use

See also:

cudaOccupancyMaxPotentialBlockSizeVariableSMem if the amount of per-block dynamic shared memory changes with different block sizes.

See also:

cudaOccupancyMaxPotentialBlockSize

See also:

cudaOccupancyMaxActiveBlocksPerMultiprocessor

See also:

cudaOccupancyMaxActiveBlocksPerMultiprocessorWithFlags

See also:

cudaOccupancyMaxPotentialBlockSizeVariableSMem

See also:

cudaOccupancyMaxPotentialBlockSizeVariableSMemWithFlags

See also:

cudaOccupancyAvailableDynamicSMemPerBlock

Note: Note that this function may also return error codes from previous, asynchronous launches.

Note: Note that this function may also return *cudaErrorInitializationError*, *cudaErrorInsufficientDriver* or *cudaErrorNoDevice* if this call tries to initialize internal CUDA RT state.

Note: Note that as specified by *cudaStreamAddCallback* no CUDA function may be called from callback. *cudaErrorNotPermitted* may, but is not guaranteed to, be returned as a diagnostic in such case.

Parameters

- **minGridSize** – - Returned minimum grid size needed to achieve the best potential occupancy
- **blockSize** – - Returned block size
- **func** – - Device function symbol
- **dynamicSMemSize** – - Per-block dynamic shared memory usage intended, in bytes
- **blockSizeLimit** – - The maximum block size `func` is designed to work with. 0 means no limit.
- **flags** – - Requested behavior for the occupancy calculator

Returns

cudaSuccess, *cudaErrorInvalidDevice*, *cudaErrorInvalidDeviceFunction*, *cudaErrorInvalidValue*, *cudaErrorUnknown*,

template<class **T**>
__host__ *cudaError_t* **cudaOccupancyMaxPotentialClusterSize**(int *clusterSize, *T* *func, const *cudaLaunchConfig_t* *config)

Given the kernel function (`func`) and launch configuration (`config`), return the maximum cluster size in `*clusterSize`.

The cluster dimensions in `config` are ignored. If func has a required cluster size set (see *cudaFuncGetAttributes*),`*clusterSize` will reflect the required cluster size.

By default this function will always return a value that's portable on future hardware. A higher value may be returned if the kernel function allows non-portable cluster sizes.

This function will respect the compile time launch bounds.

See also:

cudaFuncGetAttributes

Note: Note that this function may also return error codes from previous, asynchronous launches.

Note: Note that this function may also return *cudaErrorInitializationError*, *cudaErrorInsufficientDriver* or *cudaErrorNoDevice* if this call tries to initialize internal CUDA RT state.

Note: Note that as specified by *cudaStreamAddCallback* no CUDA function may be called from callback. *cudaErrorNotPermitted* may, but is not guaranteed to, be returned as a diagnostic in such case.

Note: The API can also be used with a kernel *cudaKernel_t* by querying the handle using *cudaLibraryGetKernel()* or *cudaGetKernel* and then passing it to the API by casting to void*. The symbol `entryFuncAddr` passed to *cudaGetKernel* should be a symbol that is registered with the same CUDA Runtime instance.

Note: Passing a symbol that belongs that belongs to a different runtime instance will result in undefined behavior. The only type that can be reliably passed to a different runtime instance is *cudaKernel_t*

Parameters

- **`clusterSize`** – - Returned maximum cluster size that can be launched for the given kernel function and launch configuration
- **`func`** – - Kernel function for which maximum cluster size is calculated
- **`config`** – - Launch configuration for the given kernel function

Returns

cudaSuccess, *cudaErrorInvalidDeviceFunction*, *cudaErrorInvalidValue*, *cudaErrorUnknown*,

template<class **T**>
__host__ *cudaError_t* **`cudaStreamAttachMemAsync`**(*cudaStream_t* stream, *T* *devPtr, size_t length = 0, unsigned int flags = 0x04)

Attach memory to a stream asynchronously.

Enqueues an operation in `stream` to specify stream association of `length` bytes of memory starting from `devPtr`. This function is a stream-ordered operation, meaning that it is dependent on, and will only take effect when, previous work in stream has completed. Any previous association is automatically replaced.

`devPtr` must point to an one of the following types of memories:

- managed memory declared using the **managed** keyword or allocated with *cudaMallocManaged*.
- a valid host-accessible region of system-allocated pageable memory. This type of memory may only be specified if the device associated with the stream reports a non-zero value for the device attribute *cudaDevAttrPageableMemoryAccess*.

For managed allocations, `length` must be either zero or the entire allocation's size. Both indicate that the entire allocation's stream association is being changed. Currently, it is not possible to change stream association for a portion of a managed allocation.

For pageable allocations, `length` must be non-zero.

The stream association is specified using `flags` which must be one of *cudaMemAttachGlobal*, *cudaMemAttachHost* or *cudaMemAttachSingle*. The default value for `flags` is *cudaMemAttachSingle* If the *cudaMemAttachGlobal* flag is specified, the memory can be accessed by any stream on any device. If the *cudaMemAttachHost* flag is specified, the program makes a guarantee that it won't access the memory on the device from any stream on a device that has a zero value for the device attribute *cudaDevAttrConcurrentManagedAccess*. If the *cudaMemAttachSingle* flag is specified and `stream` is associated with a device that has a zero value for the device attribute *cudaDevAttrConcurrentManagedAccess*, the program makes a guarantee that it will only access the memory on the device from `stream`. It is illegal to attach singly to the NULL stream, because the NULL stream is a virtual global stream and not a specific stream. An error will be returned in this case.

When memory is associated with a single stream, the Unified Memory system will allow CPU access to this memory region so long as all operations in `stream` have completed, regardless of whether other streams are active. In effect, this constrains exclusive ownership of the managed memory region by an active GPU to per-stream activity instead of whole-GPU activity.

Accessing memory on the device from streams that are not associated with it will produce undefined results. No error checking is performed by the Unified Memory system to ensure that kernels launched into other streams do not access this region.

It is a program's responsibility to order calls to *cudaStreamAttachMemAsync* via events, synchronization or other means to ensure legal access to memory at all times. Data visibility and coherency will be changed appropriately for all kernels which follow a stream-association change.

If `stream` is destroyed while data is associated with it, the association is removed and the association reverts to the default visibility of the allocation as specified at *cudaMallocManaged*. For **managed** variables, the default association is always *cudaMemAttachGlobal*. Note that destroying a stream is an asynchronous operation, and as a result, the change to default association won't happen until all work in the stream has completed.

See also:

cudaStreamCreate, *cudaStreamCreateWithFlags*, *cudaStreamWaitEvent*, *cudaStreamSynchronize*, *cudaStreamAddCallback*, *cudaStreamDestroy*, *cudaMallocManaged*

Note: Note that this function may also return error codes from previous, asynchronous launches.

Note: Note that this function may also return *cudaErrorInitializationError*, *cudaErrorInsufficientDriver* or *cudaErrorNoDevice* if this call tries to initialize internal CUDA RT state.

Note: Note that as specified by *cudaStreamAddCallback* no CUDA function may be called from callback. *cudaErrorNotPermitted* may, but is not guaranteed to, be returned as a diagnostic in such case.

Parameters

- **`flags`** – Memory can only be accessed by a single stream on the associated device

- **stream** – - Stream in which to enqueue the attach operation
- **devPtr** – - Pointer to memory (must be a pointer to managed memory or to a valid host-accessible region of system-allocated memory)
- **length** – - Length of memory (defaults to zero)
- **flags** – - Must be one of *cudaMemAttachGlobal*, *cudaMemAttachHost* or *cudaMemAttachSingle* (defaults to *cudaMemAttachSingle*)

Returns

cudaSuccess, *cudaErrorNotReady*, *cudaErrorInvalidValue*, *cudaErrorInvalidResourceHandle*

6.2. Data types used by CUDA Runtime

Macros

CUDART_CB

CUDA_EGL_MAX_PLANES
Maximum number of planes per frame.

CUDA_IPC_HANDLE_SIZE
CUDA IPC Handle Size.

CU_UUID_HAS_BEEN_DEFINED
CUDA UUID types.

RESOURCE_ABI_BYTES

cudaArrayColorAttachment
Must be set in cudaExternalMemoryGetMappedMipmappedArray if the mipmapped array is used as a color target in a graphics API.

cudaArrayCubemap
Must be set in cudaMalloc3DArray to create a cubemap CUDA array.

cudaArrayDefault
Default CUDA array allocation flag.

cudaArrayDeferredMapping
Must be set in cudaMallocArray, cudaMalloc3DArray or cudaMallocMipmappedArray in order to create a deferred mapping CUDA array or CUDA mipmapped array.

cudaArrayLayered
Must be set in cudaMalloc3DArray to create a layered CUDA array.

cudaArraySparse
Must be set in cudaMallocArray, cudaMalloc3DArray or cudaMallocMipmappedArray in order to create a sparse CUDA array or CUDA mipmapped array.

cudaArraySparsePropertiesSingleMipTail
Indicates that the layered sparse CUDA array or CUDA mipmapped array has a single mip tail region for all layers.

cudaArraySurfaceLoadStore
Must be set in cudaMallocArray or cudaMalloc3DArray in order to bind surfaces to the CUDA array.

cudaArrayTextureGather
Must be set in cudaMallocArray or cudaMalloc3DArray in order to perform texture gather operations on the CUDA array.

cudaCpuDeviceId
Device id that represents the CPU.

cudaDevSmResourceGroupLocalityDomainIdAny
Special value for localityDomainId when cudaDevSmResourceSplit is called with cudaDevSmResourceGroupLocalityDomainId. Selects SMs from the union of all available locality domains in the input SM resource. See also: cudaDevSmResourceGroupLocalityDomainId for the set of SMs in a single locality domain.

cudaDeviceBlockingSync
Device flag - Use blocking synchronization.

cudaDeviceLmemResizeToMax
Device flag - Keep local memory allocation after launch.

cudaDeviceMapHost
Device flag - Support mapped pinned allocations.

cudaDeviceMask
Device flags mask.

cudaDeviceScheduleAuto
Device flag - Automatic scheduling.

cudaDeviceScheduleBlockingSync
Device flag - Use blocking synchronization.

cudaDeviceScheduleMask
Device schedule flags mask.

cudaDeviceScheduleSpin
Device flag - Spin default scheduling.

cudaDeviceScheduleYield
Device flag - Yield default scheduling.

cudaDeviceSyncMemops
Device flag - Ensure synchronous memory operations on this context will synchronize.

cudaEventBlockingSync
Event uses blocking synchronization.

cudaEventDefault
Default event flag.

cudaEventDisableTiming
Event will not record timing data.

cudaEventInterprocess
Event is suitable for interprocess use.

cudaEventRecordDefault
Default event record flag.

cudaEventRecordExternal
Event is captured in the graph as an external event node when performing stream capture.

cudaEventWaitDefault
Default event wait flag.

cudaEventWaitExternal
Event is captured in the graph as an external event node when performing stream capture.

cudaExternalMemoryDedicated
Indicates that the external memory object is a dedicated resource.

cudaExternalSemaphoreSignalSkipNvSciBufMemSync
When the /p flags parameter of *cudaExternalSemaphoreSignalParams* contains this flag, it indicates that signaling an external semaphore object should skip performing appropriate memory synchronization operations over all the external memory objects that are imported as *cudaExternalMemoryHandleTypeNvSciBuf* , which otherwise are performed by default to ensure data coherency with other importers of the same NvSciBuf memory objects.

cudaExternalSemaphoreWaitSkipNvSciBufMemSync
When the /p flags parameter of *cudaExternalSemaphoreWaitParams* contains this flag, it indicates that waiting an external semaphore object should skip performing appropriate memory synchronization operations over all the external memory objects that are imported as *cudaExternalMemoryHandleTypeNvSciBuf* , which otherwise are performed by default to ensure data coherency with other importers of the same NvSciBuf memory objects.

cudaGraphKernelNodePortDefault
This port activates when the kernel has finished executing.

cudaGraphKernelNodePortLaunchCompletion
This port activates when all blocks of the kernel have begun execution.

cudaGraphKernelNodePortProgrammatic
This port activates when all blocks of the kernel have performed *cudaTriggerProgrammaticLaunchCompletion()* or have terminated.

cudaHostAllocDefault
Default page-locked allocation flag.

cudaHostAllocMapped
Map allocation into device space.

cudaHostAllocPortable
Pinned memory accessible by all CUDA contexts.

cudaHostAllocWriteCombined
Write-combined memory.

cudaHostRegisterDefault
Default host memory registration flag.

cudaHostRegisterIoMemory
Memory-mapped I/O space.

cudaHostRegisterMapped
Map registered memory into device space.

cudaHostRegisterPortable
Pinned memory accessible by all CUDA contexts.

cudaHostRegisterReadOnly
Memory-mapped read-only.

cudaInitDeviceFlagsAreValid
Tell the CUDA runtime that DeviceFlags is being set in cudaInitDevice call.

cudaInvalidDeviceId
Device id that represents an invalid device.

cudaIpcMemLazyEnablePeerAccess
Automatically enable peer access between remote devices as needed.

cudaKernelNodeAttrID

cudaKernelNodeAttrValue

cudaKernelNodeAttributeAccessPolicyWindow

cudaKernelNodeAttributeClusterDimension

cudaKernelNodeAttributeClusterSchedulingPolicyPreference

cudaKernelNodeAttributeCooperative

cudaKernelNodeAttributeDeviceUpdatableKernelNode

cudaKernelNodeAttributeMemSyncDomain

cudaKernelNodeAttributeMemSyncDomainMap

cudaKernelNodeAttributeNvlinkUtilCentricScheduling

cudaKernelNodeAttributePreferredSharedMemoryCarveout

cudaKernelNodeAttributePriority

cudaMemAttachGlobal
Memory can be accessed by any stream on any device.

cudaMemAttachHost
Memory cannot be accessed by any stream on any device.

cudaMemAttachSingle
Memory can only be accessed by a single stream on the associated device.

cudaMemPoolCreateUsageHwDecompress
This flag, if set, indicates that the memory will be used as a buffer for hardware accelerated decompression.

cudaNvSciSyncAttrSignal
When /p flags of *cudaDeviceGetNvSciSyncAttributes* is set to this, it indicates that application need signaler specific NvSciSyncAttr to be filled by *cudaDeviceGetNvSciSyncAttributes* .

cudaNvSciSyncAttrWait
When /p flags of *cudaDeviceGetNvSciSyncAttributes* is set to this, it indicates that application need waiter specific NvSciSyncAttr to be filled by *cudaDeviceGetNvSciSyncAttributes* .

cudaOccupancyDefault
Default behavior.

cudaOccupancyDisableCachingOverride
Assume global caching is enabled and cannot be automatically turned off.

cudaPeerAccessDefault
Default peer addressing enable flag.

cudaStreamAttrID

cudaStreamAttrValue

cudaStreamAttributeAccessPolicyWindow

cudaStreamAttributeMemSyncDomain

cudaStreamAttributeMemSyncDomainMap

cudaStreamAttributePriority

cudaStreamAttributeSynchronizationPolicy

cudaStreamDefault
Default stream flag.

cudaStreamLegacy
Legacy stream handle.

cudaStreamNonBlocking
Stream does not synchronize with stream 0 (the NULL stream)

cudaStreamPerThread
Per-thread stream handle.

cudaSurfaceType1D

cudaSurfaceType1DLayered

cudaSurfaceType2D

cudaSurfaceType2DLayered

cudaSurfaceType3D

cudaSurfaceTypeCubemap

cudaSurfaceTypeCubemapLayered

cudaTextureType1D

cudaTextureType1DLayered

cudaTextureType2D

cudaTextureType2DLayered

cudaTextureType3D

cudaTextureTypeCubemap

cudaTextureTypeCubemapLayered

Enumerations

cudaAccessProperty
Specifies performance hint with *cudaAccessPolicyWindow* for hitProp and missProp members.

cudaAsyncNotificationType
Types of async notification that can occur.

cudaAtomicOperation
CUDA-valid Atomic Operations.

cudaAtomicOperationCapability
CUDA-valid Atomic Operation capabilities.

cudaCGScope
CUDA cooperative group scope.

cudaChannelFormatKind
Channel format kind.

cudaClusterSchedulingPolicy
Cluster scheduling policies.

cudaComputeMode
CUDA device compute modes.

cudaDevResourceType
Type of resource.

cudaDevSmResourceGroup_flags
Flags for a CUdevSmResource group.

cudaDevSmResourceSplitByCount_flags

cudaDevWorkqueueConfigScope
Sharing scope for workqueues.

cudaDeviceAttr
CUDA device attributes.

cudaDeviceNumaConfig
CUDA device NUMA config.

cudaDeviceP2PAttr
CUDA device P2P attributes.

cudaDriverEntryPointQueryResult
Enum for status from obtaining driver entry points, used with ::cudaApiGetDriverEntryPoint.

cudaEglColorFormat
CUDA EGL Color Format - The different planar and multiplanar formats currently supported for CUDA_EGL interops.

cudaEglFrameType
CUDA EglFrame type - array or pointer.

cudaEglResourceLocationFlags
Resource location flags- sysmem or vidmem.

cudaError
impl_private

cudaExternalMemoryHandleType
External memory handle types.

cudaExternalSemaphoreHandleType
External semaphore handle types.

cudaFabricOpStatusInfo
Fabric operation status info.

cudaFabricOpStatusSource
Fabric operation status source.

cudaFlushGPUDirectRDMAWritesOptions
CUDA GPUDirect RDMA flush writes APIs supported on the device.

cudaFlushGPUDirectRDMAWritesScope
CUDA GPUDirect RDMA flush writes scopes.

cudaFlushGPUDirectRDMAWritesTarget
CUDA GPUDirect RDMA flush writes targets.

cudaFuncAttribute
CUDA function attributes that can be set using *cudaFuncSetAttribute* .

cudaFuncCache
CUDA function cache configurations.

cudaGPUDirectRDMAWritesOrdering
CUDA GPUDirect RDMA flush writes ordering features of the device.

cudaGetDriverEntryPointFlags
Flags to specify search options to be used with *cudaGetDriverEntryPoint* For more details see ::cuGetProcAddress.

cudaGraphChildGraphNodeOwnership
Child graph node ownership.

cudaGraphConditionalHandleFlags

cudaGraphConditionalNodeType
CUDA conditional node types.

cudaGraphDebugDotFlags
CUDA Graph debug write options.

cudaGraphDependencyType
Type annotations that can be applied to graph edges as part of *cudaGraphEdgeData* .

cudaGraphExecUpdateResult
CUDA Graph Update error types.

cudaGraphInstantiateFlags
Flags for instantiating a graph.

cudaGraphInstantiateResult
Graph instantiation results.

cudaGraphKernelNodeField
Specifies the field to update when performing multiple node updates from the device.

cudaGraphMemAttributeType
Graph memory attributes.

cudaGraphNodeType
CUDA Graph node types.

cudaGraphRecaptureStatus
Possible recapture statuses that can be returned to the user callback.

cudaGraphicsCubeFace
CUDA graphics interop array indices for cube maps.

cudaGraphicsMapFlags
CUDA graphics interop map flags.

cudaGraphicsRegisterFlags
CUDA graphics interop register flags.

cudaHostTaskSyncMode
Flags for host task sync mode.

cudaJitOption
Online compiler and linker options.

cudaJit_CacheMode
Caching modes for dlcm.

cudaJit_Fallback
Cubin matching fallback strategies.

cudaKernelFunctionType
CUDA Kernel Function Handle Type.

cudaLaunchAttributeID
Launch attributes enum; used as id field of *cudaLaunchAttribute* .

cudaLaunchAttributePortableClusterMode
Enum for defining applicability of portable cluster size, used with *cudaLaunchKernelEx* .

cudaLaunchMemSyncDomain
Memory Synchronization Domain.

cudaLibraryOption
Library options to be specified with *cudaLibraryLoadData()* or *cudaLibraryLoadFromFile()*

cudaLimit
CUDA Limits.

cudaLogLevel

cudaMemAccessFlags
Specifies the memory protection flags for mapping.

cudaMemAllocationHandleType
Flags for specifying particular handle types.

cudaMemAllocationType
Defines the allocation types available.

cudaMemLocationType
Specifies the type of location.

cudaMemPoolAttr
CUDA memory pool attributes.

cudaMemRangeAttribute
CUDA range attributes.

cudaMemcpy3DOperandType
These flags allow applications to convey the operand type for individual copies specified in *cudaMemcpy3DBatchAsync* .

cudaMemcpyFlags
Flags to specify for copies within a batch.

cudaMemcpyKind
CUDA memory copy types.

cudaMemcpySrcAccessOrder

cudaMemoryAdvise
CUDA Memory Advise values.

cudaMemoryType
CUDA memory types.

cudaResourceType
CUDA resource types.

cudaResourceViewFormat
CUDA texture resource view formats.

cudaSharedCarveout
Shared memory carveout configurations.

cudaSharedMemConfig
CUDA shared memory configuration.

cudaSharedMemoryMode
Shared memory related attributes for use with ::cuLaunchKernelEx.

cudaStreamCaptureMode
Possible modes for stream capture thread interactions.

cudaStreamCaptureStatus
Possible stream capture statuses returned by *cudaStreamIsCapturing* .

cudaStreamUpdateCaptureDependenciesFlags
Flags for *cudaStreamUpdateCaptureDependencies* .

cudaSurfaceBoundaryMode
CUDA Surface boundary modes.

cudaSurfaceFormatMode
CUDA Surface format modes.

cudaSynchronizationPolicy

cudaTextureAddressMode
CUDA texture address modes.

cudaTextureFilterMode
CUDA texture filter modes.

cudaTextureReadMode
CUDA texture read modes.

cudaUserObjectFlags
Flags for user objects for graphs.

cudaUserObjectRetainFlags
Flags for retaining user object references for graphs.

Structs

CUuuid_st

cudaAccessPolicyWindow
Specifies an access policy for a window, a contiguous extent of memory beginning at base_ptr and ending at base_ptr + num_bytes.

cudaArrayMemoryRequirements
CUDA array and CUDA mipmapped array memory requirements.

cudaArraySparseProperties
Sparse CUDA array and CUDA mipmapped array properties.

cudaAsyncNotificationInfo_t
Information describing an async notification event.

cudaChannelFormatDesc
CUDA Channel format descriptor.

cudaChildGraphNodeParams
Child graph node parameters.

cudaConditionalNodeParams
CUDA conditional node parameters.

cudaDevResource
A tagged union describing different resources identified by the type field.

cudaDevSmResource
Data for SM-related resources All parameters in this structure are OUTPUT only.

cudaDevSmResourceGroupParams
Input data for splitting SMs.

cudaDevWorkqueueConfigResource
Data for workqueue configuration related resources.

cudaDevWorkqueueResource
Handle to a pre-existing workqueue related resource.

cudaDeviceProp
CUDA device properties.

cudaEglFrame
CUDA EGLFrame Descriptor - structure defining one frame of EGL.

cudaEglPlaneDesc
CUDA EGL Plane Descriptor - structure defining each plane of a CUDA EGLFrame.

cudaEventRecordNodeParams
Event record node parameters.

cudaEventWaitNodeParams
Event wait node parameters.

cudaExtent
CUDA extent.

cudaExternalMemoryBufferDesc
External memory buffer descriptor.

cudaExternalMemoryHandleDesc
External memory handle descriptor.

cudaExternalMemoryMipmappedArrayDesc
External memory mipmap descriptor.

cudaExternalSemaphoreHandleDesc
External semaphore handle descriptor.

cudaExternalSemaphoreSignalNodeParams
External semaphore signal node parameters.

cudaExternalSemaphoreSignalNodeParamsV2
External semaphore signal node parameters.

cudaExternalSemaphoreSignalParams
External semaphore signal parameters, compatible with driver type.

cudaExternalSemaphoreWaitNodeParams
External semaphore wait node parameters.

cudaExternalSemaphoreWaitNodeParamsV2
External semaphore wait node parameters.

cudaExternalSemaphoreWaitParams
External semaphore wait parameters, compatible with driver type.

cudaFuncAttributes
CUDA function attributes.

cudaGraphEdgeData
Optional annotation for edges in a CUDA graph.

cudaGraphExecUpdateResultInfo
Result information returned by cudaGraphExecUpdate.

cudaGraphInstantiateParams
Graph instantiation parameters.

cudaGraphKernelNodeUpdate
Struct to specify a single node update to pass as part of a larger array to *cudaGraphKernelNode-UpdatesApply* .

cudaGraphNodeParams
Graph node parameters.

cudaHostNodeParams
CUDA host node parameters.

cudaHostNodeParamsV2
CUDA host node parameters.

cudaIpcEventHandle_t
CUDA IPC event handle.

cudaIpcMemHandle_t
CUDA IPC memory handle.

cudaKernelNodeParams
CUDA GPU kernel node parameters.

cudaKernelNodeParamsV2
CUDA GPU kernel node parameters.

cudaLaunchAttribute
Launch attribute.

cudaLaunchConfig_t
CUDA extensible launch configuration.

cudaLaunchMemSyncDomainMap
Memory Synchronization Domain map.

cudaMemAccessDesc
Memory access descriptor.

cudaMemAllocNodeParams
Memory allocation node parameters.

cudaMemAllocNodeParamsV2
Memory allocation node parameters.

cudaMemFabricHandle_t

cudaMemFreeNodeParams
Memory free node parameters.

cudaMemLocation
Specifies a memory location.

cudaMemPoolProps
Specifies the properties of allocations made from the pool.

cudaMemPoolPtrExportData
Opaque data for exporting a pool allocation.

cudaMemcpy3DBatchOp

cudaMemcpy3DOperand
Struct representing an operand for copy with *cudaMemcpy3DBatchAsync* .

cudaMemcpy3DParms
CUDA 3D memory copying parameters.

cudaMemcpy3DPeerParms
CUDA 3D cross-device memory copying parameters.

cudaMemcpyAttributes
Attributes specific to copies within a batch.

cudaMemcpyBatchNodeParams
CUDA Batch Memcpy Node Parameters.

cudaMemcpyNodeParams
Memcpy node parameters.

cudaMemsetParams
CUDA Memset node parameters.

cudaMemsetParamsV2
CUDA Memset node parameters.

cudaOffset3D
Struct representing offset into a *cudaArray_t* in elements.

cudaPitchedPtr
CUDA Pitched memory pointer.

cudaPointerAttributes
CUDA pointer attributes.

cudaPos
CUDA 3D position.

cudaResourceDesc
CUDA resource descriptor.

cudaResourceViewDesc
CUDA resource view descriptor.

cudaTextureDesc
CUDA texture descriptor.

cudalibraryHostUniversalFunctionAndDataTable

Typedefs

cudaArray_const_t
CUDA array (as source copy argument)

cudaArray_t
CUDA array.

cudaAsyncCallback

cudaAsyncCallbackHandle_t
CUDA async callback handle.

cudaDevResourceDesc_t
An opaque descriptor handle.

cudaEglStreamConnection
CUDA EGLSream Connection.

cudaError_t
CUDA Error types.

cudaEvent_t
CUDA event types.

cudaExecutionContext_t
An opaque handle to a CUDA execution context.

cudaExternalMemory_t
CUDA external memory.

cudaExternalSemaphore_t
CUDA external semaphore.

cudaFunction_t
CUDA function.

cudaGraphConditionalHandle
CUDA handle for conditional graph nodes.

cudaGraphDeviceNode_t
CUDA device node handle for device-side node update.

cudaGraphExec_t
CUDA executable (launchable) graph.

cudaGraphNode_t
CUDA graph node.

cudaGraph_t
CUDA graph.

cudaGraphicsResource_t
CUDA graphics resource types.

cudaHostFn_t
CUDA host function.

cudaKernel_t
CUDA kernel.

cudaLibrary_t
CUDA library.

cudaLogIterator

cudaLogsCallbackHandle

cudaMemPool_t
CUDA memory pool.

cudaMipmappedArray_const_t
CUDA mipmapped array (as source argument)

cudaMipmappedArray_t
CUDA mipmapped array.

cudaStream_t
CUDA stream.

cudaSurfaceObject_t
An opaque value that represents a CUDA Surface object.

cudaTextureObject_t
An opaque value that represents a CUDA texture object.

cudaUserObject_t
CUDA user object for graphs.

Unions

cudaLaunchAttributeValue
Launch attributes union; used as value field of *cudaLaunchAttribute* .

6.2.1. Macros

CUDART_CB __stdcall

CUDA_EGL_MAX_PLANES 3
Maximum number of planes per frame.

CUDA_IPC_HANDLE_SIZE 64
CUDA IPC Handle Size.

CU_UUID_HAS_BEEN_DEFINED
CUDA UUID types.

RESOURCE_ABI_BYTES 40

cudaArrayColorAttachment 0x20
Must be set in cudaExternalMemoryGetMappedMipmappedArray if the mipmapped array is used as a color target in a graphics API.

cudaArrayCubemap 0x04
Must be set in cudaMalloc3DArray to create a cubemap CUDA array.

cudaArrayDefault 0x00
Default CUDA array allocation flag.

cudaArrayDeferredMapping 0x80

Must be set in cudaMallocArray, cudaMalloc3DArray or cudaMallocMipmappedArray in order to create a deferred mapping CUDA array or CUDA mipmapped array.

cudaArrayLayered 0x01

Must be set in cudaMalloc3DArray to create a layered CUDA array.

cudaArraySparse 0x40

Must be set in cudaMallocArray, cudaMalloc3DArray or cudaMallocMipmappedArray in order to create a sparse CUDA array or CUDA mipmapped array.

cudaArraySparsePropertiesSingleMipTail 0x1

Indicates that the layered sparse CUDA array or CUDA mipmapped array has a single mip tail region for all layers.

cudaArraySurfaceLoadStore 0x02

Must be set in cudaMallocArray or cudaMalloc3DArray in order to bind surfaces to the CUDA array.

cudaArrayTextureGather 0x08

Must be set in cudaMallocArray or cudaMalloc3DArray in order to perform texture gather operations on the CUDA array.

cudaCpuDeviceId ((int)-1)

Device id that represents the CPU.

cudaDevSmResourceGroupLocalityDomainIdAny (~0u)

Special value for localityDomainId when cudaDevSmResourceSplit is called with cudaDevSmResourceGroupLocalityDomainId. Selects SMs from the union of all available locality domains in the input SM resource. See also: cudaDevSmResourceGroupLocalityDomainId for the set of SMs in a single locality domain.

cudaDeviceBlockingSync 0x04

Device flag - Use blocking synchronization.

Deprecated:

This flag was deprecated as of CUDA 4.0 and replaced with *cudaDeviceScheduleBlockingSync*.

cudaDeviceLmemResizeToMax 0x10

Device flag - Keep local memory allocation after launch.

cudaDeviceMapHost 0x08

Device flag - Support mapped pinned allocations.

cudaDeviceMask 0xff
Device flags mask.

cudaDeviceScheduleAuto 0x00
Device flag - Automatic scheduling.

cudaDeviceScheduleBlockingSync 0x04
Device flag - Use blocking synchronization.

cudaDeviceScheduleMask 0x07
Device schedule flags mask.

cudaDeviceScheduleSpin 0x01
Device flag - Spin default scheduling.

cudaDeviceScheduleYield 0x02
Device flag - Yield default scheduling.

cudaDeviceSyncMemops 0x80
Device flag - Ensure synchronous memory operations on this context will synchronize.

cudaEventBlockingSync 0x01
Event uses blocking synchronization.

cudaEventDefault 0x00
Default event flag.

cudaEventDisableTiming 0x02
Event will not record timing data.

cudaEventInterprocess 0x04
Event is suitable for interprocess use.
cudaEventDisableTiming must be set

cudaEventRecordDefault 0x00
Default event record flag.

cudaEventRecordExternal 0x01
Event is captured in the graph as an external event node when performing stream capture.

cudaEventWaitDefault 0x00
Default event wait flag.

cudaEventWaitExternal 0x01
Event is captured in the graph as an external event node when performing stream capture.

cudaExternalMemoryDedicated 0x1

Indicates that the external memory object is a dedicated resource.

cudaExternalSemaphoreSignalSkipNvSciBufMemSync 0x01

When the /p flags parameter of *cudaExternalSemaphoreSignalParams* contains this flag, it indicates that signaling an external semaphore object should skip performing appropriate memory synchronization operations over all the external memory objects that are imported as *cudaExternalMemoryHandleTypeNvSciBuf*, which otherwise are performed by default to ensure data coherency with other importers of the same NvSciBuf memory objects.

cudaExternalSemaphoreWaitSkipNvSciBufMemSync 0x02

When the /p flags parameter of *cudaExternalSemaphoreWaitParams* contains this flag, it indicates that waiting an external semaphore object should skip performing appropriate memory synchronization operations over all the external memory objects that are imported as *cudaExternalMemoryHandleTypeNvSciBuf*, which otherwise are performed by default to ensure data coherency with other importers of the same NvSciBuf memory objects.

cudaGraphKernelNodePortDefault 0

This port activates when the kernel has finished executing.

cudaGraphKernelNodePortLaunchCompletion 2

This port activates when all blocks of the kernel have begun execution.

See also *cudaLaunchAttributeLaunchCompletionEvent*.

cudaGraphKernelNodePortProgrammatic 1

This port activates when all blocks of the kernel have performed *cudaTriggerProgrammaticLaunchCompletion()* or have terminated.

It must be used with edge type *cudaGraphDependencyTypeProgrammatic*. See also *cudaLaunchAttributeProgrammaticEvent*.

cudaHostAllocDefault 0x00

Default page-locked allocation flag.

cudaHostAllocMapped 0x02

Map allocation into device space.

cudaHostAllocPortable 0x01

Pinned memory accessible by all CUDA contexts.

cudaHostAllocWriteCombined 0x04

Write-combined memory.

cudaHostRegisterDefault 0x00

Default host memory registration flag.

cudaHostRegisterIoMemory 0x04
Memory-mapped I/O space.

cudaHostRegisterMapped 0x02
Map registered memory into device space.

cudaHostRegisterPortable 0x01
Pinned memory accessible by all CUDA contexts.

cudaHostRegisterReadOnly 0x08
Memory-mapped read-only.

cudaInitDeviceFlagsAreValid 0x01
Tell the CUDA runtime that DeviceFlags is being set in cudaInitDevice call.

cudaInvalidDeviceId ((int)-2)
Device id that represents an invalid device.

cudaIpcMemLazyEnablePeerAccess 0x01
Automatically enable peer access between remote devices as needed.

cudaKernelNodeAttrID *cudaLaunchAttributeID*

cudaKernelNodeAttrValue cudaLaunchAttributeValue

cudaKernelNodeAttributeAccessPolicyWindow *cudaLaunchAttributeAccessPolicyWindow*

cudaKernelNodeAttributeClusterDimension *cudaLaunchAttributeClusterDimension*

cudaKernelNodeAttributeClusterSchedulingPolicyPreference
cudaLaunchAttributeClusterSchedulingPolicyPreference

cudaKernelNodeAttributeCooperative *cudaLaunchAttributeCooperative*

cudaKernelNodeAttributeDeviceUpdatableKernelNode
cudaLaunchAttributeDeviceUpdatableKernelNode

cudaKernelNodeAttributeMemSyncDomain *cudaLaunchAttributeMemSyncDomain*

cudaKernelNodeAttributeMemSyncDomainMap *cudaLaunchAttributeMemSyncDomainMap*

cudaKernelNodeAttributeNvlinkUtilCentricScheduling
cudaLaunchAttributeNvlinkUtilCentricScheduling

cudaKernelNodeAttributePreferredSharedMemoryCarveout
cudaLaunchAttributePreferredSharedMemoryCarveout

cudaKernelNodeAttributePriority *cudaLaunchAttributePriority*

cudaMemAttachGlobal 0x01

Memory can be accessed by any stream on any device.

cudaMemAttachHost 0x02

Memory cannot be accessed by any stream on any device.

cudaMemAttachSingle 0x04

Memory can only be accessed by a single stream on the associated device.

cudaMemPoolCreateUsageHwDecompress 0x2

This flag, if set, indicates that the memory will be used as a buffer for hardware accelerated decompression.

cudaNvSciSyncAttrSignal 0x1

When /p flags of *cudaDeviceGetNvSciSyncAttributes* is set to this, it indicates that application need signaler specific NvSciSyncAttr to be filled by *cudaDeviceGetNvSciSyncAttributes*.

cudaNvSciSyncAttrWait 0x2

When /p flags of *cudaDeviceGetNvSciSyncAttributes* is set to this, it indicates that application need waiter specific NvSciSyncAttr to be filled by *cudaDeviceGetNvSciSyncAttributes*.

cudaOccupancyDefault 0x00

Default behavior.

cudaOccupancyDisableCachingOverride 0x01

Assume global caching is enabled and cannot be automatically turned off.

cudaPeerAccessDefault 0x00

Default peer addressing enable flag.

cudaStreamAttrID *cudaLaunchAttributeID*

cudaStreamAttrValue cudaLaunchAttributeValue

cudaStreamAttributeAccessPolicyWindow *cudaLaunchAttributeAccessPolicyWindow*

cudaStreamAttributeMemSyncDomain *cudaLaunchAttributeMemSyncDomain*

cudaStreamAttributeMemSyncDomainMap *cudaLaunchAttributeMemSyncDomainMap*

cudaStreamAttributePriority *cudaLaunchAttributePriority*

cudaStreamAttributeSynchronizationPolicy *cudaLaunchAttributeSynchronizationPolicy*

cudaStreamDefault 0x00

Default stream flag.

cudaStreamLegacy ((*cudaStream_t*)0x1)

Legacy stream handle.

Stream handle that can be passed as a cudaStream_t to use an implicit stream with legacy synchronization behavior.

See details of the synchronization behavior.

cudaStreamNonBlocking 0x01

Stream does not synchronize with stream 0 (the NULL stream)

cudaStreamPerThread ((*cudaStream_t*)0x2)

Per-thread stream handle.

Stream handle that can be passed as a cudaStream_t to use an implicit stream with per-thread synchronization behavior.

See details of the synchronization behavior.

cudaSurfaceType1D 0x01

cudaSurfaceType1DLayered 0xF1

cudaSurfaceType2D 0x02

cudaSurfaceType2DLayered 0xF2

cudaSurfaceType3D 0x03

cudaSurfaceTypeCubemap 0x0C

cudaSurfaceTypeCubemapLayered 0xFC

cudaTextureType1D 0x01

cudaTextureType1DLayered 0xF1

cudaTextureType2D 0x02

cudaTextureType2DLayered 0xF2

cudaTextureType3D 0x03

cudaTextureTypeCubemap 0x0C

cudaTextureTypeCubemapLayered 0xFC

6.2.2. Enumerations

enum **cudaAccessProperty**

Specifies performance hint with *cudaAccessPolicyWindow* for hitProp and missProp members.

Values:

enumerator **cudaAccessPropertyNormal** = 0

Normal cache persistence.

enumerator **cudaAccessPropertyStreaming** = 1

Streaming access is less likely to persit from cache.

enumerator **cudaAccessPropertyPersisting** = 2

Persisting access is more likely to persist in cache.

enum **cudaAsyncNotificationType**

Types of async notification that can occur.

Values:

enumerator **cudaAsyncNotificationTypeOverBudget** = 0x1

Sent when the process has exceeded its device memory budget.

enum **cudaAtomicOperation**

CUDA-valid Atomic Operations.

Values:

enumerator **cudaAtomicOperationIntegerAdd** = 0

enumerator **cudaAtomicOperationIntegerMin** = 1

enumerator **cudaAtomicOperationIntegerMax** = 2

enumerator **cudaAtomicOperationIntegerIncrement** = 3

enumerator **cudaAtomicOperationIntegerDecrement** = 4

enumerator **cudaAtomicOperationAnd** = 5

enumerator **cudaAtomicOperationOr** = 6

enumerator **cudaAtomicOperationXOR** = 7

enumerator **cudaAtomicOperationExchange** = 8

enumerator **cudaAtomicOperationCAS** = 9

enumerator **cudaAtomicOperationFloatAdd** = 10

enumerator **cudaAtomicOperationFloatMin** = 11

enumerator **cudaAtomicOperationFloatMax** = 12

enum **cudaAtomicOperationCapability**

CUDA-valid Atomic Operation capabilities.

Values:

enumerator **cudaAtomicCapabilitySigned** = 1u << 0

enumerator **cudaAtomicCapabilityUnsigned** = 1u << 1

enumerator **cudaAtomicCapabilityReduction** = 1u << 2

enumerator **cudaAtomicCapabilityScalar32** = 1u << 3

enumerator **cudaAtomicCapabilityScalar64** = 1u << 4

enumerator **cudaAtomicCapabilityScalar128** = 1u << 5

enumerator **cudaAtomicCapabilityVector32x4** = 1u << 6

enum **cudaCGScope**

CUDA cooperative group scope.

Values:

enumerator **cudaCGScopeInvalid** = 0

Invalid cooperative group scope.

enumerator **cudaCGScopeGrid** = 1
Scope represented by a grid_group.

enumerator **cudaCGScopeReserved** = 2
Reserved.

enum **cudaChannelFormatKind**
Channel format kind.
Values:

enumerator **cudaChannelFormatKindSigned** = 0
Signed channel format.

enumerator **cudaChannelFormatKindUnsigned** = 1
Unsigned channel format.

enumerator **cudaChannelFormatKindFloat** = 2
Float channel format.

enumerator **cudaChannelFormatKindNone** = 3
No channel format.

enumerator **cudaChannelFormatKindNV12** = 4
Unsigned 8-bit integers, planar 4:2:0 YUV format.

enumerator **cudaChannelFormatKindUnsignedNormalized8X1** = 5
1 channel unsigned 8-bit normalized integer

enumerator **cudaChannelFormatKindUnsignedNormalized8X2** = 6
2 channel unsigned 8-bit normalized integer

enumerator **cudaChannelFormatKindUnsignedNormalized8X4** = 7
4 channel unsigned 8-bit normalized integer

enumerator **cudaChannelFormatKindUnsignedNormalized16X1** = 8
1 channel unsigned 16-bit normalized integer

enumerator **cudaChannelFormatKindUnsignedNormalized16X2** = 9
2 channel unsigned 16-bit normalized integer

enumerator **cudaChannelFormatKindUnsignedNormalized16X4** = 10
4 channel unsigned 16-bit normalized integer

enumerator **cudaChannelFormatKindSignedNormalized8X1** = 11
1 channel signed 8-bit normalized integer

enumerator **cudaChannelFormatKindSignedNormalized8X2** = 12

2 channel signed 8-bit normalized integer

enumerator **cudaChannelFormatKindSignedNormalized8X4** = 13

4 channel signed 8-bit normalized integer

enumerator **cudaChannelFormatKindSignedNormalized16X1** = 14

1 channel signed 16-bit normalized integer

enumerator **cudaChannelFormatKindSignedNormalized16X2** = 15

2 channel signed 16-bit normalized integer

enumerator **cudaChannelFormatKindSignedNormalized16X4** = 16

4 channel signed 16-bit normalized integer

enumerator **cudaChannelFormatKindUnsignedBlockCompressed1** = 17

4 channel unsigned normalized block-compressed (BC1 compression) format

enumerator **cudaChannelFormatKindUnsignedBlockCompressed1SRGB** = 18

4 channel unsigned normalized block-compressed (BC1 compression) format with sRGB encoding

enumerator **cudaChannelFormatKindUnsignedBlockCompressed2** = 19

4 channel unsigned normalized block-compressed (BC2 compression) format

enumerator **cudaChannelFormatKindUnsignedBlockCompressed2SRGB** = 20

4 channel unsigned normalized block-compressed (BC2 compression) format with sRGB encoding

enumerator **cudaChannelFormatKindUnsignedBlockCompressed3** = 21

4 channel unsigned normalized block-compressed (BC3 compression) format

enumerator **cudaChannelFormatKindUnsignedBlockCompressed3SRGB** = 22

4 channel unsigned normalized block-compressed (BC3 compression) format with sRGB encoding

enumerator **cudaChannelFormatKindUnsignedBlockCompressed4** = 23

1 channel unsigned normalized block-compressed (BC4 compression) format

enumerator **cudaChannelFormatKindSignedBlockCompressed4** = 24

1 channel signed normalized block-compressed (BC4 compression) format

enumerator **cudaChannelFormatKindUnsignedBlockCompressed5** = 25

2 channel unsigned normalized block-compressed (BC5 compression) format

enumerator **cudaChannelFormatKindSignedBlockCompressed5** = 26

2 channel signed normalized block-compressed (BC5 compression) format

enumerator **cudaChannelFormatKindUnsignedBlockCompressed6H** = 27

3 channel unsigned half-float block-compressed (BC6H compression) format

enumerator **cudaChannelFormatKindSignedBlockCompressed6H** = 28

3 channel signed half-float block-compressed (BC6H compression) format

enumerator **cudaChannelFormatKindUnsignedBlockCompressed7** = 29

4 channel unsigned normalized block-compressed (BC7 compression) format

enumerator **cudaChannelFormatKindUnsignedBlockCompressed7SRGB** = 30

4 channel unsigned normalized block-compressed (BC7 compression) format with sRGB encoding

enumerator **cudaChannelFormatKindUnsignedNormalized1010102** = 31

4 channel unsigned normalized (10-bit, 10-bit, 10-bit, 2-bit) format

enumerator **cudaChannelFormatKindUnsigned8Packed422** = 32

4 channel unsigned 8-bit packed format, with 4:2:2 sampling

enumerator **cudaChannelFormatKindUnsigned8Packed444** = 33

4 channel unsigned 8-bit packed format, with 4:4:4 sampling

enumerator **cudaChannelFormatKindUnsigned8SemiPlanar420** = 34

3 channel unsigned 8-bit semi-planar format, with 4:2:0 sampling

enumerator **cudaChannelFormatKindUnsigned16SemiPlanar420** = 35

3 channel unsigned 16-bit semi-planar format, with 4:2:0 sampling

enumerator **cudaChannelFormatKindUnsigned8SemiPlanar422** = 36

3 channel unsigned 8-bit semi-planar format, with 4:2:2 sampling

enumerator **cudaChannelFormatKindUnsigned16SemiPlanar422** = 37

3 channel unsigned 16-bit semi-planar format, with 4:2:2 sampling

enumerator **cudaChannelFormatKindUnsigned8SemiPlanar444** = 38

3 channel unsigned 8-bit semi-planar format, with 4:4:4 sampling

enumerator **cudaChannelFormatKindUnsigned16SemiPlanar444** = 39

3 channel unsigned 16-bit semi-planar format, with 4:4:4 sampling

enumerator **cudaChannelFormatKindUnsigned8Planar420** = 40

3 channel unsigned 8-bit planar format, with 4:2:0 sampling

enumerator **cudaChannelFormatKindUnsigned16Planar420** = 41
3 channel unsigned 16-bit planar format, with 4:2:0 sampling

enumerator **cudaChannelFormatKindUnsigned8Planar422** = 42
3 channel unsigned 8-bit planar format, with 4:2:2 sampling

enumerator **cudaChannelFormatKindUnsigned16Planar422** = 43
3 channel unsigned 16-bit planar format, with 4:2:2 sampling

enumerator **cudaChannelFormatKindUnsigned8Planar444** = 44
3 channel unsigned 8-bit planar format, with 4:4:4 sampling

enumerator **cudaChannelFormatKindUnsigned16Planar444** = 45
3 channel unsigned 16-bit planar format, with 4:4:4 sampling

enum **cudaClusterSchedulingPolicy**
Cluster scheduling policies.
These may be passed to *cudaFuncSetAttribute*
Values:

enumerator **cudaClusterSchedulingPolicyDefault** = 0
the default policy

enumerator **cudaClusterSchedulingPolicySpread** = 1
spread the blocks within a cluster to the SMs

enumerator **cudaClusterSchedulingPolicyLoadBalancing** = 2
allow the hardware to load-balance the blocks in a cluster to the SMs

enumerator **cudaClusterSchedulingPolicyRubinDsmemLocality** = 3

enum **cudaComputeMode**
CUDA device compute modes.
Values:

enumerator **cudaComputeModeDefault** = 0
Default compute mode (Multiple threads can use *cudaSetDevice()* with this device)

enumerator **cudaComputeModeExclusive** = 1
Compute-exclusive-thread mode (Only one thread in one process will be able to use *cudaSetDevice()* with this device)

enumerator **cudaComputeModeProhibited** = 2
Compute-prohibited mode (No threads can use *cudaSetDevice()* with this device)

enumerator **cudaComputeModeExclusiveProcess** = 3

Compute-exclusive-process mode (Many threads in one process will be able to use *cudaSetDevice()* with this device)

enum **cudaDevResourceType**

Type of resource.

Values:

enumerator **cudaDevResourceTypeInvalid** = 0

enumerator **cudaDevResourceTypeSm** = 1

Streaming multiprocessors related information.

enumerator **cudaDevResourceTypeWorkqueueConfig** = 1000

Workqueue configuration related information.

enumerator **cudaDevResourceTypeWorkqueue** = 10000

Pre-existing workqueue related information.

enum **cudaDevSmResourceGroup_flags**

Flags for a CUdevSmResource group.

Values:

enumerator **cudaDevSmResourceGroupDefault** = 0

enumerator **cudaDevSmResourceGroupBackfill** = 0x1 << 0

Treats constraints as a hint, ignoring them if necessary to reach the requested smCount.

Lets smCount be a non-multiple of coscheduledSmCount, filling the difference between SM count and already assigned co-scheduled groupings with other SMs. When used with cudaDevSmResourceGroupLocalityDomainId, backfill fills up to the requested smCount using the target locality domain first, then SMs not attributed to any locality domain, then SMs from other locality domains. If no SMs can be found in the requested locality domain, cudaErrorInvalidResourceConfiguration is returned. When used with `cudaDevSmResourceGroupLocalityDomainNonPortableId`, backfill follows the same behavior after first consuming all SMs from the target locality domain.

enumerator **cudaDevSmResourceGroupLocalityDomainId** = 0x1 << 1

The SMs must be located on a specific locality domain, specified by localityDomainId.

enumerator **cudaDevSmResourceGroupLocalityDomainSupplementalSms** = 0x1 << 2

Modifier to `cudaDevSmResourceGroupLocalityDomainId` which allows selection of SMs in the locality domain exceeding the ::cudaDeviceAttr::cudaDevAttrLocalityDomainMultiprocessorCount.

enum **cudaDevSmResourceSplitByCount_flags**

Values:

enumerator **cudaDevSmResourceSplitIgnoreSmCoscheduling** = 0x1

enumerator **cudaDevSmResourceSplitMaxPotentialClusterSize** = 0x2

enum **cudaDevWorkqueueConfigScope**

Sharing scope for workqueues.

Values:

enumerator **cudaDevWorkqueueConfigScopeDeviceCtx** = 0

Use all shared workqueue resources on the device.

Default driver behaviour.

enumerator **cudaDevWorkqueueConfigScopeGreenCtxBalanced** = 1

When possible, use non-overlapping workqueue resources with other balanced green contexts.

enum **cudaDeviceAttr**

CUDA device attributes.

Values:

enumerator **cudaDevAttrMaxThreadsPerBlock** = 1

Maximum number of threads per block.

enumerator **cudaDevAttrMaxBlockDimX** = 2

Maximum block dimension X.

enumerator **cudaDevAttrMaxBlockDimY** = 3

Maximum block dimension Y.

enumerator **cudaDevAttrMaxBlockDimZ** = 4

Maximum block dimension Z.

enumerator **cudaDevAttrMaxGridDimX** = 5

Maximum grid dimension X.

enumerator **cudaDevAttrMaxGridDimY** = 6

Maximum grid dimension Y.

enumerator **cudaDevAttrMaxGridDimZ** = 7

Maximum grid dimension Z.

enumerator **cudaDevAttrMaxSharedMemoryPerBlock** = 8

Maximum shared memory available per block in bytes.

enumerator **cudaDevAttrTotalConstantMemory** = 9
Memory available on device for **constant** variables in a CUDA C kernel in bytes.

enumerator **cudaDevAttrWarpSize** = 10
Warp size in threads.

enumerator **cudaDevAttrMaxPitch** = 11
Maximum pitch in bytes allowed by memory copies.

enumerator **cudaDevAttrMaxRegistersPerBlock** = 12
Maximum number of 32-bit registers available per block.

enumerator **cudaDevAttrClockRate** = 13
Peak clock frequency in kilohertz.

enumerator **cudaDevAttrTextureAlignment** = 14
Alignment requirement for textures.

enumerator **cudaDevAttrGpuOverlap** = 15
Device can possibly copy memory and execute a kernel concurrently.

enumerator **cudaDevAttrMultiProcessorCount** = 16
Number of multiprocessors on device.

enumerator **cudaDevAttrKernelExecTimeout** = 17
Specifies whether there is a run time limit on kernels.

enumerator **cudaDevAttrIntegrated** = 18
Device is integrated with host memory.

enumerator **cudaDevAttrCanMapHostMemory** = 19
Device can map host memory into CUDA address space.

enumerator **cudaDevAttrComputeMode** = 20
Compute mode (See *cudaComputeMode* for details)

enumerator **cudaDevAttrMaxTexture1DWidth** = 21
Maximum 1D texture width.

enumerator **cudaDevAttrMaxTexture2DWidth** = 22
Maximum 2D texture width.

enumerator **cudaDevAttrMaxTexture2DHeight** = 23
Maximum 2D texture height.

enumerator **cudaDevAttrMaxTexture3DWidth** = 24
Maximum 3D texture width.

enumerator **cudaDevAttrMaxTexture3DHeight** = 25
Maximum 3D texture height.

enumerator **cudaDevAttrMaxTexture3DDepth** = 26
Maximum 3D texture depth.

enumerator **cudaDevAttrMaxTexture2DLayeredWidth** = 27
Maximum 2D layered texture width.

enumerator **cudaDevAttrMaxTexture2DLayeredHeight** = 28
Maximum 2D layered texture height.

enumerator **cudaDevAttrMaxTexture2DLayeredLayers** = 29
Maximum layers in a 2D layered texture.

enumerator **cudaDevAttrSurfaceAlignment** = 30
Alignment requirement for surfaces.

enumerator **cudaDevAttrConcurrentKernels** = 31
Device can possibly execute multiple kernels concurrently.

enumerator **cudaDevAttrEccEnabled** = 32
Device has ECC support enabled.

enumerator **cudaDevAttrPciBusId** = 33
PCI bus ID of the device.

enumerator **cudaDevAttrPciDeviceId** = 34
PCI device ID of the device.

enumerator **cudaDevAttrTccDriver** = 35
Device is using TCC driver model.

enumerator **cudaDevAttrMemoryClockRate** = 36
Peak memory clock frequency in kilohertz.

enumerator **cudaDevAttrGlobalMemoryBusWidth** = 37
Global memory bus width in bits.

enumerator **cudaDevAttrL2CacheSize** = 38
Size of L2 cache in bytes.

enumerator **cudaDevAttrMaxThreadsPerMultiProcessor** = 39
Maximum resident threads per multiprocessor.

enumerator **cudaDevAttrAsyncEngineCount** = 40
Number of asynchronous engines.

enumerator **cudaDevAttrUnifiedAddressing** = 41
Device shares a unified address space with the host.

enumerator **cudaDevAttrMaxTexture1DLayeredWidth** = 42
Maximum 1D layered texture width.

enumerator **cudaDevAttrMaxTexture1DLayeredLayers** = 43
Maximum layers in a 1D layered texture.

enumerator **cudaDevAttrMaxTexture2DGatherWidth** = 45
Maximum 2D texture width if cudaArrayTextureGather is set.

enumerator **cudaDevAttrMaxTexture2DGatherHeight** = 46
Maximum 2D texture height if cudaArrayTextureGather is set.

enumerator **cudaDevAttrMaxTexture3DWidthAlt** = 47
Alternate maximum 3D texture width.

enumerator **cudaDevAttrMaxTexture3DHeightAlt** = 48
Alternate maximum 3D texture height.

enumerator **cudaDevAttrMaxTexture3DDepthAlt** = 49
Alternate maximum 3D texture depth.

enumerator **cudaDevAttrPciDomainId** = 50
PCI domain ID of the device.

enumerator **cudaDevAttrTexturePitchAlignment** = 51
Pitch alignment requirement for textures.

enumerator **cudaDevAttrMaxTextureCubemapWidth** = 52
Maximum cubemap texture width/height.

enumerator **cudaDevAttrMaxTextureCubemapLayeredWidth** = 53
Maximum cubemap layered texture width/height.

enumerator **cudaDevAttrMaxTextureCubemapLayeredLayers** = 54
Maximum layers in a cubemap layered texture.

enumerator **cudaDevAttrMaxSurface1DWidth** = 55
Maximum 1D surface width.

enumerator **cudaDevAttrMaxSurface2DWidth** = 56
Maximum 2D surface width.

enumerator **cudaDevAttrMaxSurface2DHeight** = 57
Maximum 2D surface height.

enumerator **cudaDevAttrMaxSurface3DWidth** = 58
Maximum 3D surface width.

enumerator **cudaDevAttrMaxSurface3DHeight** = 59
Maximum 3D surface height.

enumerator **cudaDevAttrMaxSurface3DDepth** = 60
Maximum 3D surface depth.

enumerator **cudaDevAttrMaxSurface1DLayeredWidth** = 61
Maximum 1D layered surface width.

enumerator **cudaDevAttrMaxSurface1DLayeredLayers** = 62
Maximum layers in a 1D layered surface.

enumerator **cudaDevAttrMaxSurface2DLayeredWidth** = 63
Maximum 2D layered surface width.

enumerator **cudaDevAttrMaxSurface2DLayeredHeight** = 64
Maximum 2D layered surface height.

enumerator **cudaDevAttrMaxSurface2DLayeredLayers** = 65
Maximum layers in a 2D layered surface.

enumerator **cudaDevAttrMaxSurfaceCubemapWidth** = 66
Maximum cubemap surface width.

enumerator **cudaDevAttrMaxSurfaceCubemapLayeredWidth** = 67
Maximum cubemap layered surface width.

enumerator **cudaDevAttrMaxSurfaceCubemapLayeredLayers** = 68
Maximum layers in a cubemap layered surface.

enumerator **cudaDevAttrMaxTexture1DLinearWidth** = 69
Maximum 1D linear texture width.

enumerator **cudaDevAttrMaxTexture2DLinearWidth** = 70
Maximum 2D linear texture width.

enumerator **cudaDevAttrMaxTexture2DLinearHeight** = 71
Maximum 2D linear texture height.

enumerator **cudaDevAttrMaxTexture2DLinearPitch** = 72
Maximum 2D linear texture pitch in bytes.

enumerator **cudaDevAttrMaxTexture2DMipmappedWidth** = 73
Maximum mipmapped 2D texture width.

enumerator **cudaDevAttrMaxTexture2DMipmappedHeight** = 74
Maximum mipmapped 2D texture height.

enumerator **cudaDevAttrComputeCapabilityMajor** = 75
Major compute capability version number.

enumerator **cudaDevAttrComputeCapabilityMinor** = 76
Minor compute capability version number.

enumerator **cudaDevAttrMaxTexture1DMipmappedWidth** = 77
Maximum mipmapped 1D texture width.

enumerator **cudaDevAttrStreamPrioritiesSupported** = 78
Device supports stream priorities.

enumerator **cudaDevAttrGlobalL1CacheSupported** = 79
Device supports caching globals in L1.

enumerator **cudaDevAttrLocalL1CacheSupported** = 80
Device supports caching locals in L1.

enumerator **cudaDevAttrMaxSharedMemoryPerMultiprocessor** = 81
Maximum shared memory available per multiprocessor in bytes.

enumerator **cudaDevAttrMaxRegistersPerMultiprocessor** = 82
Maximum number of 32-bit registers available per multiprocessor.

enumerator **cudaDevAttrManagedMemory** = 83
Device can allocate managed memory on this system.

enumerator **cudaDevAttrIsMultiGpuBoard** = 84
Device is on a multi-GPU board.

enumerator **cudaDevAttrMultiGpuBoardGroupID** = 85

Unique identifier for a group of devices on the same multi-GPU board.

enumerator **cudaDevAttrHostNativeAtomicSupported** = 86

Link between the device and the host supports native atomic operations.

enumerator **cudaDevAttrSingleToDoublePrecisionPerfRatio** = 87

Ratio of single precision performance (in floating-point operations per second) to double precision performance.

enumerator **cudaDevAttrPageableMemoryAccess** = 88

Device supports coherently accessing pageable memory without calling cudaHostRegister on it.

enumerator **cudaDevAttrConcurrentManagedAccess** = 89

Device can coherently access managed memory concurrently with the CPU.

enumerator **cudaDevAttrComputePreemptionSupported** = 90

Device supports Compute Preemption.

enumerator **cudaDevAttrCanUseHostPointerForRegisteredMem** = 91

Device can access host registered memory at the same virtual address as the CPU.

enumerator **cudaDevAttrReserved92** = 92

enumerator **cudaDevAttrReserved93** = 93

enumerator **cudaDevAttrReserved94** = 94

enumerator **cudaDevAttrCooperativeLaunch** = 95

Device supports launching cooperative kernels via *cudaLaunchCooperativeKernel*.

enumerator **cudaDevAttrReserved96** = 96

enumerator **cudaDevAttrMaxSharedMemoryPerBlockOptin** = 97

The maximum optin shared memory per block.

This value may vary by chip. See *cudaFuncSetAttribute*

enumerator **cudaDevAttrCanFlushRemoteWrites** = 98

Device supports flushing of outstanding remote writes.

enumerator **cudaDevAttrHostRegisterSupported** = 99

Device supports host memory registration via *cudaHostRegister*.

enumerator **cudaDevAttrPageableMemoryAccessUsesHostPageTables** = 100

Device accesses pageable memory via the host's page tables.

enumerator **cudaDevAttrDirectManagedMemAccessFromHost** = 101

Host can directly access managed memory on the device without migration.

enumerator **cudaDevAttrMaxBlocksPerMultiprocessor** = 106

Maximum number of blocks per multiprocessor.

enumerator **cudaDevAttrMaxPersistingL2CacheSize** = 108

Maximum L2 persisting lines capacity setting in bytes.

enumerator **cudaDevAttrMaxAccessPolicyWindowSize** = 109

Maximum value of *cudaAccessPolicyWindow::num_bytes*.

enumerator **cudaDevAttrReservedSharedMemoryPerBlock** = 111

Shared memory reserved by CUDA driver per block in bytes.

enumerator **cudaDevAttrSparseCudaArraySupported** = 112

Device supports sparse CUDA arrays and sparse CUDA mipmapped arrays.

enumerator **cudaDevAttrHostRegisterReadOnlySupported** = 113

Device supports using the *cudaHostRegister* flag cudaHostRegisterReadOnly to register memory that must be mapped as read-only to the GPU.

enumerator **cudaDevAttrTimelineSemaphoreInteropSupported** = 114

External timeline semaphore interop is supported on the device.

enumerator **cudaDevAttrMemoryPoolsSupported** = 115

Device supports using the *cudaMallocAsync* and ::cudaMemPool family of APIs.

enumerator **cudaDevAttrGPUDirectRDMASupported** = 116

Device supports GPUDirect RDMA APIs, like nvidia_p2p_get_pages (see https://docs.nvidia.com/cuda/gpudirect-rdma for more information)

enumerator **cudaDevAttrGPUDirectRDMAFlushWritesOptions** = 117

The returned attribute shall be interpreted as a bitmask, where the individual bits are listed in the *cudaFlushGPUDirectRDMAWritesOptions* enum.

enumerator **cudaDevAttrGPUDirectRDMAWritesOrdering** = 118

GPUDirect RDMA writes to the device do not need to be flushed for consumers within the scope indicated by the returned attribute.

See *cudaGPUDirectRDMAWritesOrdering* for the numerical values returned here.

enumerator **cudaDevAttrMemoryPoolSupportedHandleTypes** = 119

Handle types supported with mempool based IPC.

enumerator **cudaDevAttrClusterLaunch** = 120
Indicates device supports cluster launch.

enumerator **cudaDevAttrDeferredMappingCudaArraySupported** = 121
Device supports deferred mapping CUDA arrays and CUDA mipmapped arrays.

enumerator **cudaDevAttrReserved122** = 122

enumerator **cudaDevAttrReserved123** = 123

enumerator **cudaDevAttrReserved124** = 124

enumerator **cudaDevAttrIpcEventSupport** = 125
Device supports IPC Events.

enumerator **cudaDevAttrMemSyncDomainCount** = 126
Number of memory synchronization domains the device supports.

enumerator **cudaDevAttrReserved127** = 127

enumerator **cudaDevAttrReserved128** = 128

enumerator **cudaDevAttrReserved129** = 129

enumerator **cudaDevAttrNumaConfig** = 130
NUMA configuration of a device: value is of type *cudaDeviceNumaConfig* enum.

enumerator **cudaDevAttrNumaId** = 131
NUMA node ID of the GPU memory.

enumerator **cudaDevAttrReserved132** = 132

enumerator **cudaDevAttrMpsEnabled** = 133
Contexts created on this device will be shared via MPS.

enumerator **cudaDevAttrHostNumaId** = 134
NUMA ID of the host node closest to the device or -1 when system does not support NUMA.

enumerator **cudaDevAttrD3D12CigSupported** = 135
Device supports CIG with D3D12.

enumerator **cudaDevAttrVulkanCigSupported** = 138
Device supports CIG with Vulkan.

enumerator **cudaDevAttrGpuPciDeviceId** = 139

The combined 16-bit PCI device ID and 16-bit PCI vendor ID.

enumerator **cudaDevAttrGpuPciSubsystemId** = 140

The combined 16-bit PCI subsystem ID and 16-bit PCI subsystem vendor ID.

enumerator **cudaDevAttrReserved141** = 141

enumerator **cudaDevAttrHostNumaMemoryPoolsSupported** = 142

Device supports HOST_NUMA location with the *cudaMallocAsync* and ::cudaMemPool family of APIs.

enumerator **cudaDevAttrHostNumaMultinodeIpcSupported** = 143

Device supports HostNuma location IPC between nodes in a multi-node system.

enumerator **cudaDevAttrHostMemoryPoolsSupported** = 144

Device suports HOST location with the ::cuMemAllocAsync and ::cuMemPool family of APIs.

enumerator **cudaDevAttrReserved145** = 145

enumerator **cudaDevAttrOnlyPartialHostNativeAtomicSupported** = 147

Link between the device and the host supports only some native atomic operations.

enumerator **cudaDevAttrAtomicReductionSupported** = 148

Device supports atomic reduction operations in stream batch memory operations.

enumerator **cudaDevAttrLocalityDomainCount** = 149

Number of locality domains.

enumerator **cudaDevAttrOversizedSharedMemoryPerBlock** = 150

The maximum oversized shared memory per block.

This value may vary by chip. See *cudaFuncSetAttribute*

enumerator **cudaDevAttrCigStreamsSupported** = 151

Device supports CIG streams.

enumerator **cudaDevAttrLocalityDomainMultiprocessorCount** = 157

Number of multiprocessors on each locality domain.

enumerator **cudaDevAttrTensorMapLargeTensorMinSize** = 160

Smallest tensor size in bytes at which TMA descriptors are treated as “large” (BASE_128KB OOB mode).

tensors >= this threshold are eligible for the tensormap.replace.assume_large PTX modifier. Returns INT_MAX on devices without this distinction where assume_large is a no-op.

enumerator **cudaDevAttrMax**

enum **cudaDeviceNumaConfig**

CUDA device NUMA config.

Values:

enumerator **cudaDeviceNumaConfigNone** = 0

The GPU is not a NUMA node.

enumerator **cudaDeviceNumaConfigNumaNode**

The GPU is a NUMA node, cudaDevAttrNumaId contains its NUMA ID.

enum **cudaDeviceP2PAttr**

CUDA device P2P attributes.

Values:

enumerator **cudaDevP2PAttrPerformanceRank** = 1

A relative value indicating the performance of the link between two devices.

enumerator **cudaDevP2PAttrAccessSupported** = 2

Peer access is enabled.

enumerator **cudaDevP2PAttrNativeAtomicSupported** = 3

Native atomic operation over the link supported.

enumerator **cudaDevP2PAttrCudaArrayAccessSupported** = 4

Accessing CUDA arrays over the link supported.

enumerator **cudaDevP2PAttrOnlyPartialNativeAtomicSupported** = 5

Only some CUDA-valid atomic operations over the link are supported.

enum **cudaDriverEntryPointQueryResult**

Enum for status from obtaining driver entry points, used with ::cudaApiGetDriverEntryPoint.

Values:

enumerator **cudaDriverEntryPointSuccess** = 0

Search for symbol found a match.

enumerator **cudaDriverEntryPointSymbolNotFound** = 1

Search for symbol was not found.

enumerator **cudaDriverEntryPointVersionNotSufficent** = 2

Search for symbol was found but version wasn't great enough.

enum **cudaEglColorFormat**

CUDA EGL Color Format - The different planar and multiplanar formats currently supported for CUDA_EGL interops.

Values:

enumerator **cudaEglColorFormatYUV420Planar** = 0

Y, U, V in three surfaces, each in a separate surface, U/V width = 1/2 Y width, U/V height = 1/2 Y height.

enumerator **cudaEglColorFormatYUV420SemiPlanar** = 1

Y, UV in two surfaces (UV as one surface) with VU byte ordering, width, height ratio same as YUV420Planar.

enumerator **cudaEglColorFormatYUV422Planar** = 2

Y, U, V each in a separate surface, U/V width = 1/2 Y width, U/V height = Y height.

enumerator **cudaEglColorFormatYUV422SemiPlanar** = 3

Y, UV in two surfaces with VU byte ordering, width, height ratio same as YUV422Planar.

enumerator **cudaEglColorFormatARGB** = 6

R/G/B/A four channels in one surface with BGRA byte ordering.

enumerator **cudaEglColorFormatRGBA** = 7

R/G/B/A four channels in one surface with ABGR byte ordering.

enumerator **cudaEglColorFormatL** = 8

single luminance channel in one surface.

enumerator **cudaEglColorFormatR** = 9

single color channel in one surface.

enumerator **cudaEglColorFormatYUV444Planar** = 10

Y, U, V in three surfaces, each in a separate surface, U/V width = Y width, U/V height = Y height.

enumerator **cudaEglColorFormatYUV444SemiPlanar** = 11

Y, UV in two surfaces (UV as one surface) with VU byte ordering, width, height ratio same as YUV444Planar.

enumerator **cudaEglColorFormatYUYV422** = 12

Y, U, V in one surface, interleaved as UYVY in one channel.

enumerator **cudaEglColorFormatUYVY422** = 13

Y, U, V in one surface, interleaved as YUYV in one channel.

enumerator **cudaEglColorFormatABGR** = 14

R/G/B/A four channels in one surface with RGBA byte ordering.

enumerator **cudaEglColorFormatBGRA** = 15

R/G/B/A four channels in one surface with ARGB byte ordering.

enumerator **cudaEglColorFormatA** = 16

Alpha color format - one channel in one surface.

enumerator **cudaEglColorFormatRG** = 17

R/G color format - two channels in one surface with GR byte ordering.

enumerator **cudaEglColorFormatAYUV** = 18

Y, U, V, A four channels in one surface, interleaved as VUYA.

enumerator **cudaEglColorFormatYVU444SemiPlanar** = 19

Y, VU in two surfaces (VU as one surface) with UV byte ordering, U/V width = Y width, U/V height = Y height.

enumerator **cudaEglColorFormatYVU422SemiPlanar** = 20

Y, VU in two surfaces (VU as one surface) with UV byte ordering, U/V width = 1/2 Y width, U/V height = Y height.

enumerator **cudaEglColorFormatYVU420SemiPlanar** = 21

Y, VU in two surfaces (VU as one surface) with UV byte ordering, U/V width = 1/2 Y width, U/V height = 1/2 Y height.

enumerator **cudaEglColorFormatY10V10U10_444SemiPlanar** = 22

Y10, V10U10 in two surfaces (VU as one surface) with UV byte ordering, U/V width = Y width, U/V height = Y height.

enumerator **cudaEglColorFormatY10V10U10_420SemiPlanar** = 23

Y10, V10U10 in two surfaces (VU as one surface) with UV byte ordering, U/V width = 1/2 Y width, U/V height = 1/2 Y height.

enumerator **cudaEglColorFormatY12V12U12_444SemiPlanar** = 24

Y12, V12U12 in two surfaces (VU as one surface) with UV byte ordering, U/V width = Y width, U/V height = Y height.

enumerator **cudaEglColorFormatY12V12U12_420SemiPlanar** = 25

Y12, V12U12 in two surfaces (VU as one surface) with UV byte ordering, U/V width = 1/2 Y width, U/V height = 1/2 Y height.

enumerator **cudaEglColorFormatVYUY_ER** = 26

Extended Range Y, U, V in one surface, interleaved as YVYU in one channel.

enumerator **cudaEglColorFormatUYVY_ER** = 27

Extended Range Y, U, V in one surface, interleaved as YUYV in one channel.

enumerator **cudaEglColorFormatYUYV_ER** = 28

Extended Range Y, U, V in one surface, interleaved as UYVY in one channel.

enumerator **cudaEglColorFormatYVYU_ER** = 29

Extended Range Y, U, V in one surface, interleaved as VYUY in one channel.

enumerator **cudaEglColorFormatYUVA_ER** = 31

Extended Range Y, U, V, A four channels in one surface, interleaved as AVUY.

enumerator **cudaEglColorFormatAYUV_ER** = 32

Extended Range Y, U, V, A four channels in one surface, interleaved as VUYA.

enumerator **cudaEglColorFormatYUV444Planar_ER** = 33

Extended Range Y, U, V in three surfaces, U/V width = Y width, U/V height = Y height.

enumerator **cudaEglColorFormatYUV422Planar_ER** = 34

Extended Range Y, U, V in three surfaces, U/V width = 1/2 Y width, U/V height = Y height.

enumerator **cudaEglColorFormatYUV420Planar_ER** = 35

Extended Range Y, U, V in three surfaces, U/V width = 1/2 Y width, U/V height = 1/2 Y height.

enumerator **cudaEglColorFormatYUV444SemiPlanar_ER** = 36

Extended Range Y, UV in two surfaces (UV as one surface) with VU byte ordering, U/V width = Y width, U/V height = Y height.

enumerator **cudaEglColorFormatYUV422SemiPlanar_ER** = 37

Extended Range Y, UV in two surfaces (UV as one surface) with VU byte ordering, U/V width = 1/2 Y width, U/V height = Y height.

enumerator **cudaEglColorFormatYUV420SemiPlanar_ER** = 38

Extended Range Y, UV in two surfaces (UV as one surface) with VU byte ordering, U/V width = 1/2 Y width, U/V height = 1/2 Y height.

enumerator **cudaEglColorFormatYVU444Planar_ER** = 39

Extended Range Y, V, U in three surfaces, U/V width = Y width, U/V height = Y height.

enumerator **cudaEglColorFormatYVU422Planar_ER** = 40

Extended Range Y, V, U in three surfaces, U/V width = 1/2 Y width, U/V height = Y height.

enumerator **cudaEglColorFormatYVU420Planar_ER** = 41

Extended Range Y, V, U in three surfaces, U/V width = 1/2 Y width, U/V height = 1/2 Y height.

enumerator **cudaEglColorFormatYVU444SemiPlanar_ER** = 42

Extended Range Y, VU in two surfaces (VU as one surface) with UV byte ordering, U/V width = Y width, U/V height = Y height.

enumerator **cudaEglColorFormatYVU422SemiPlanar_ER** = 43

Extended Range Y, VU in two surfaces (VU as one surface) with UV byte ordering, U/V width = 1/2 Y width, U/V height = Y height.

enumerator **cudaEglColorFormatYVU420SemiPlanar_ER** = 44

Extended Range Y, VU in two surfaces (VU as one surface) with UV byte ordering, U/V width = 1/2 Y width, U/V height = 1/2 Y height.

enumerator **cudaEglColorFormatBayerRGGB** = 45

Bayer format - one channel in one surface with interleaved RGGB ordering.

enumerator **cudaEglColorFormatBayerBGGR** = 46

Bayer format - one channel in one surface with interleaved BGGR ordering.

enumerator **cudaEglColorFormatBayerGRBG** = 47

Bayer format - one channel in one surface with interleaved GRBG ordering.

enumerator **cudaEglColorFormatBayerGBRG** = 48

Bayer format - one channel in one surface with interleaved GBRG ordering.

enumerator **cudaEglColorFormatBayer10RGGB** = 49

Bayer10 format - one channel in one surface with interleaved RGGB ordering.

Out of 16 bits, 10 bits used 6 bits No-op.

enumerator **cudaEglColorFormatBayer10BGGR** = 50

Bayer10 format - one channel in one surface with interleaved BGGR ordering.

Out of 16 bits, 10 bits used 6 bits No-op.

enumerator **cudaEglColorFormatBayer10GRBG** = 51

Bayer10 format - one channel in one surface with interleaved GRBG ordering.

Out of 16 bits, 10 bits used 6 bits No-op.

enumerator **cudaEglColorFormatBayer10GBRG** = 52

Bayer10 format - one channel in one surface with interleaved GBRG ordering.

Out of 16 bits, 10 bits used 6 bits No-op.

enumerator **cudaEglColorFormatBayer12RGGB** = 53

Bayer12 format - one channel in one surface with interleaved RGGB ordering.

Out of 16 bits, 12 bits used 4 bits No-op.

enumerator **cudaEglColorFormatBayer12BGGR** = 54
Bayer12 format - one channel in one surface with interleaved BGGR ordering.
Out of 16 bits, 12 bits used 4 bits No-op.

enumerator **cudaEglColorFormatBayer12GRBG** = 55
Bayer12 format - one channel in one surface with interleaved GRBG ordering.
Out of 16 bits, 12 bits used 4 bits No-op.

enumerator **cudaEglColorFormatBayer12GBRG** = 56
Bayer12 format - one channel in one surface with interleaved GBRG ordering.
Out of 16 bits, 12 bits used 4 bits No-op.

enumerator **cudaEglColorFormatBayer14RGGB** = 57
Bayer14 format - one channel in one surface with interleaved RGGB ordering.
Out of 16 bits, 14 bits used 2 bits No-op.

enumerator **cudaEglColorFormatBayer14BGGR** = 58
Bayer14 format - one channel in one surface with interleaved BGGR ordering.
Out of 16 bits, 14 bits used 2 bits No-op.

enumerator **cudaEglColorFormatBayer14GRBG** = 59
Bayer14 format - one channel in one surface with interleaved GRBG ordering.
Out of 16 bits, 14 bits used 2 bits No-op.

enumerator **cudaEglColorFormatBayer14GBRG** = 60
Bayer14 format - one channel in one surface with interleaved GBRG ordering.
Out of 16 bits, 14 bits used 2 bits No-op.

enumerator **cudaEglColorFormatBayer20RGGB** = 61
Bayer20 format - one channel in one surface with interleaved RGGB ordering.
Out of 32 bits, 20 bits used 12 bits No-op.

enumerator **cudaEglColorFormatBayer20BGGR** = 62
Bayer20 format - one channel in one surface with interleaved BGGR ordering.
Out of 32 bits, 20 bits used 12 bits No-op.

enumerator **cudaEglColorFormatBayer20GRBG** = 63
Bayer20 format - one channel in one surface with interleaved GRBG ordering.
Out of 32 bits, 20 bits used 12 bits No-op.

enumerator **cudaEglColorFormatBayer20GBRG** = 64
Bayer20 format - one channel in one surface with interleaved GBRG ordering.
Out of 32 bits, 20 bits used 12 bits No-op.

enumerator **cudaEglColorFormatYVU444Planar** = 65

Y, V, U in three surfaces, each in a separate surface, U/V width = Y width, U/V height = Y height.

enumerator **cudaEglColorFormatYVU422Planar** = 66

Y, V, U in three surfaces, each in a separate surface, U/V width = 1/2 Y width, U/V height = Y height.

enumerator **cudaEglColorFormatYVU420Planar** = 67

Y, V, U in three surfaces, each in a separate surface, U/V width = 1/2 Y width, U/V height = 1/2 Y height.

enumerator **cudaEglColorFormatBayerIspRGGB** = 68

Nvidia proprietary Bayer ISP format - one channel in one surface with interleaved RGGB ordering and mapped to opaque integer datatype.

enumerator **cudaEglColorFormatBayerIspBGGR** = 69

Nvidia proprietary Bayer ISP format - one channel in one surface with interleaved BGGR ordering and mapped to opaque integer datatype.

enumerator **cudaEglColorFormatBayerIspGRBG** = 70

Nvidia proprietary Bayer ISP format - one channel in one surface with interleaved GRBG ordering and mapped to opaque integer datatype.

enumerator **cudaEglColorFormatBayerIspGBRG** = 71

Nvidia proprietary Bayer ISP format - one channel in one surface with interleaved GBRG ordering and mapped to opaque integer datatype.

enumerator **cudaEglColorFormatBayerBCCR** = 72

Bayer format - one channel in one surface with interleaved BCCR ordering.

enumerator **cudaEglColorFormatBayerRCCB** = 73

Bayer format - one channel in one surface with interleaved RCCB ordering.

enumerator **cudaEglColorFormatBayerCRBC** = 74

Bayer format - one channel in one surface with interleaved CRBC ordering.

enumerator **cudaEglColorFormatBayerCBRC** = 75

Bayer format - one channel in one surface with interleaved CBRC ordering.

enumerator **cudaEglColorFormatBayer10CCCC** = 76

Bayer10 format - one channel in one surface with interleaved CCCC ordering.

Out of 16 bits, 10 bits used 6 bits No-op.

enumerator **cudaEglColorFormatBayer12BCCR** = 77

Bayer12 format - one channel in one surface with interleaved BCCR ordering.

Out of 16 bits, 12 bits used 4 bits No-op.

enumerator **cudaEglColorFormatBayer12RCCB** = 78

Bayer12 format - one channel in one surface with interleaved RCCB ordering.

Out of 16 bits, 12 bits used 4 bits No-op.

enumerator **cudaEglColorFormatBayer12CRBC** = 79

Bayer12 format - one channel in one surface with interleaved CRBC ordering.

Out of 16 bits, 12 bits used 4 bits No-op.

enumerator **cudaEglColorFormatBayer12CBRC** = 80

Bayer12 format - one channel in one surface with interleaved CBRC ordering.

Out of 16 bits, 12 bits used 4 bits No-op.

enumerator **cudaEglColorFormatBayer12CCCC** = 81

Bayer12 format - one channel in one surface with interleaved CCCC ordering.

Out of 16 bits, 12 bits used 4 bits No-op.

enumerator **cudaEglColorFormatY** = 82

Color format for single Y plane.

enumerator **cudaEglColorFormatYUV420SemiPlanar_2020** = 83

Y, UV in two surfaces (UV as one surface) U/V width = 1/2 Y width, U/V height = 1/2 Y height.

enumerator **cudaEglColorFormatYVU420SemiPlanar_2020** = 84

Y, VU in two surfaces (VU as one surface) U/V width = 1/2 Y width, U/V height = 1/2 Y height.

enumerator **cudaEglColorFormatYUV420Planar_2020** = 85

Y, U, V in three surfaces, each in a separate surface, U/V width = 1/2 Y width, U/V height = 1/2 Y height.

enumerator **cudaEglColorFormatYVU420Planar_2020** = 86

Y, V, U in three surfaces, each in a separate surface, U/V width = 1/2 Y width, U/V height = 1/2 Y height.

enumerator **cudaEglColorFormatYUV420SemiPlanar_709** = 87

Y, UV in two surfaces (UV as one surface) U/V width = 1/2 Y width, U/V height = 1/2 Y height.

enumerator **cudaEglColorFormatYVU420SemiPlanar_709** = 88

Y, VU in two surfaces (VU as one surface) U/V width = 1/2 Y width, U/V height = 1/2 Y height.

enumerator **cudaEglColorFormatYUV420Planar_709** = 89

Y, U, V in three surfaces, each in a separate surface, U/V width = 1/2 Y width, U/V height = 1/2 Y height.

enumerator **cudaEglColorFormatYVU420Planar_709** = 90

Y, V, U in three surfaces, each in a separate surface, U/V width = 1/2 Y width, U/V height = 1/2 Y height.

enumerator **cudaEglColorFormatY10V10U10_420SemiPlanar_709** = 91

Y10, V10U10 in two surfaces (VU as one surface) U/V width = 1/2 Y width, U/V height = 1/2 Y height.

enumerator **cudaEglColorFormatY10V10U10_420SemiPlanar_2020** = 92

Y10, V10U10 in two surfaces (VU as one surface) U/V width = 1/2 Y width, U/V height = 1/2 Y height.

enumerator **cudaEglColorFormatY10V10U10_422SemiPlanar_2020** = 93

Y10, V10U10 in two surfaces (VU as one surface) U/V width = 1/2 Y width, U/V height = Y height.

enumerator **cudaEglColorFormatY10V10U10_422SemiPlanar** = 94

Y10, V10U10 in two surfaces (VU as one surface) U/V width = 1/2 Y width, U/V height = Y height.

enumerator **cudaEglColorFormatY10V10U10_422SemiPlanar_709** = 95

Y10, V10U10 in two surfaces (VU as one surface) U/V width = 1/2 Y width, U/V height = Y height.

enumerator **cudaEglColorFormatY_ER** = 96

Extended Range Color format for single Y plane.

enumerator **cudaEglColorFormatY_709_ER** = 97

Extended Range Color format for single Y plane.

enumerator **cudaEglColorFormatY10_ER** = 98

Extended Range Color format for single Y10 plane.

enumerator **cudaEglColorFormatY10_709_ER** = 99

Extended Range Color format for single Y10 plane.

enumerator **cudaEglColorFormatY12_ER** = 100

Extended Range Color format for single Y12 plane.

enumerator **cudaEglColorFormatY12_709_ER** = 101

Extended Range Color format for single Y12 plane.

enumerator **cudaEglColorFormatYUVA** = 102

Y, U, V, A four channels in one surface, interleaved as AVUY.

enumerator **cudaEglColorFormatYVYU** = 104

Y, U, V in one surface, interleaved as YVYU in one channel.

enumerator **cudaEglColorFormatVYUY** = 105

Y, U, V in one surface, interleaved as VYUY in one channel.

enumerator **cudaEglColorFormatY10V10U10_420SemiPlanar_ER** = 106

Extended Range Y10, V10U10 in two surfaces (VU as one surface) U/V width = 1/2 Y width, U/V height = 1/2 Y height.

enumerator **cudaEglColorFormatY10V10U10_420SemiPlanar_709_ER** = 107

Extended Range Y10, V10U10 in two surfaces (VU as one surface) U/V width = 1/2 Y width, U/V height = 1/2 Y height.

enumerator **cudaEglColorFormatY10V10U10_444SemiPlanar_ER** = 108

Extended Range Y10, V10U10 in two surfaces (VU as one surface) U/V width = Y width, U/V height = Y height.

enumerator **cudaEglColorFormatY10V10U10_444SemiPlanar_709_ER** = 109

Extended Range Y10, V10U10 in two surfaces (VU as one surface) U/V width = Y width, U/V height = Y height.

enumerator **cudaEglColorFormatY12V12U12_420SemiPlanar_ER** = 110

Extended Range Y12, V12U12 in two surfaces (VU as one surface) U/V width = 1/2 Y width, U/V height = 1/2 Y height.

enumerator **cudaEglColorFormatY12V12U12_420SemiPlanar_709_ER** = 111

Extended Range Y12, V12U12 in two surfaces (VU as one surface) U/V width = 1/2 Y width, U/V height = 1/2 Y height.

enumerator **cudaEglColorFormatY12V12U12_444SemiPlanar_ER** = 112

Extended Range Y12, V12U12 in two surfaces (VU as one surface) U/V width = Y width, U/V height = Y height.

enumerator **cudaEglColorFormatY12V12U12_444SemiPlanar_709_ER** = 113

Extended Range Y12, V12U12 in two surfaces (VU as one surface) U/V width = Y width, U/V height = Y height.

enumerator **cudaEglColorFormatUYVY709** = 114

Y, U, V in one surface, interleaved as UYVY in one channel.

enumerator **cudaEglColorFormatUYVY709_ER** = 115

Extended Range Y, U, V in one surface, interleaved as UYVY in one channel.

enumerator **cudaEglColorFormatUYVY2020** = 116

Y, U, V in one surface, interleaved as UYVY in one channel.

enum **cudaEglFrameType**

CUDA EglFrame type - array or pointer.

Values:

enumerator **cudaEglFrameTypeArray** = 0

Frame type CUDA array.

enumerator **cudaEglFrameTypePitch** = 1

Frame type CUDA pointer.

enum **cudaEglResourceLocationFlags**

Resource location flags- sysmem or vidmem.

For CUDA context on iGPU, since video and system memory are equivalent - these flags will not have an effect on the execution.

For CUDA context on dGPU, applications can use the flag *cudaEglResourceLocationFlags* to give a hint about the desired location.

cudaEglResourceLocationSysmem - the frame data is made resident on the system memory to be accessed by CUDA.

cudaEglResourceLocationVidmem - the frame data is made resident on the dedicated video memory to be accessed by CUDA.

There may be an additional latency due to new allocation and data migration, if the frame is produced on a different memory.

Values:

enumerator **cudaEglResourceLocationSysmem** = 0x00

Resource location sysmem.

enumerator **cudaEglResourceLocationVidmem** = 0x01

Resource location vidmem.

enum **cudaError**

impl_private

CUDA error types

Values:

enumerator **cudaSuccess** = 0

The API call returned with no errors.

In the case of query calls, this also means that the operation being queried is complete (see *cudaEventQuery()* and *cudaStreamQuery()*).

enumerator **cudaErrorInvalidValue** = 1

This indicates that one or more of the parameters passed to the API call is not within an acceptable range of values.

enumerator **cudaErrorMemoryAllocation** = 2

The API call failed because it was unable to allocate enough memory or other resources to perform the requested operation.

enumerator **cudaErrorInitializationError** = 3

The API call failed because the CUDA driver and runtime could not be initialized.

enumerator **cudaErrorCudartUnloading** = 4

This indicates that a CUDA Runtime API call cannot be executed because it is being called during process shut down, at a point in time after CUDA driver has been unloaded.

enumerator **cudaErrorProfilerDisabled** = 5

This indicates profiler is not initialized for this run.

This can happen when the application is running with external profiling tools like visual profiler.

enumerator **cudaErrorProfilerNotInitialized** = 6

Deprecated:

This error return is deprecated as of CUDA 5.0. It is no longer an error to attempt to enable/disable the profiling via *cudaProfilerStart* or *cudaProfilerStop* without initialization.

enumerator **cudaErrorProfilerAlreadyStarted** = 7

Deprecated:

This error return is deprecated as of CUDA 5.0. It is no longer an error to call *cudaProfilerStart()* when profiling is already enabled.

enumerator **cudaErrorProfilerAlreadyStopped** = 8

Deprecated:

This error return is deprecated as of CUDA 5.0. It is no longer an error to call *cudaProfilerStop()* when profiling is already disabled.

enumerator **cudaErrorInvalidConfiguration** = 9

This indicates that a kernel launch is requesting resources that can never be satisfied by the current device.

Requesting more shared memory per block than the device supports will trigger this error, as will requesting too many threads or blocks. See *cudaDeviceProp* for more device limitations.

enumerator **cudaErrorVersionTranslation** = 10

This indicates that the driver is newer than the runtime version and returned graph node parameter information that the runtime does not understand and is unable to translate.

enumerator **cudaErrorInvalidPitchValue** = 12

This indicates that one or more of the pitch-related parameters passed to the API call is not within the acceptable range for pitch.

enumerator **cudaErrorInvalidSymbol** = 13

This indicates that the symbol name/identifier passed to the API call is not a valid name or identifier.

enumerator **cudaErrorInvalidHostPointer** = 16

This indicates that at least one host pointer passed to the API call is not a valid host pointer.

Deprecated:

This error return is deprecated as of CUDA 10.1.

enumerator **cudaErrorInvalidDevicePointer** = 17

This indicates that at least one device pointer passed to the API call is not a valid device pointer.

Deprecated:

This error return is deprecated as of CUDA 10.1.

enumerator **cudaErrorInvalidTexture** = 18

This indicates that the texture passed to the API call is not a valid texture.

enumerator **cudaErrorInvalidTextureBinding** = 19

This indicates that the texture binding is not valid.

This occurs if you call ::cudaGetTextureAlignmentOffset() with an unbound texture.

enumerator **cudaErrorInvalidChannelDescriptor** = 20

This indicates that the channel descriptor passed to the API call is not valid.

This occurs if the format is not one of the formats specified by *cudaChannelFormatKind*, or if one of the dimensions is invalid.

enumerator **cudaErrorInvalidMemcpyDirection** = 21

This indicates that the direction of the memcpy passed to the API call is not one of the types specified by *cudaMemcpyKind*.

enumerator **cudaErrorAddressOfConstant** = 22

This indicated that the user has taken the address of a constant variable, which was forbidden up until the CUDA 3.1 release.

Deprecated:

This error return is deprecated as of CUDA 3.1. Variables in constant memory may now have their address taken by the runtime via *cudaGetSymbolAddress()*.

enumerator **cudaErrorTextureFetchFailed** = 23

This indicated that a texture fetch was not able to be performed.

This was previously used for device emulation of texture operations.

Deprecated:

This error return is deprecated as of CUDA 3.1. Device emulation mode was removed with the CUDA 3.1 release.

enumerator **cudaErrorTextureNotBound** = 24

This indicated that a texture was not bound for access.

This was previously used for device emulation of texture operations.

Deprecated:

This error return is deprecated as of CUDA 3.1. Device emulation mode was removed with the CUDA 3.1 release.

enumerator **cudaErrorSynchronizationError** = 25

This indicated that a synchronization operation had failed.

This was previously used for some device emulation functions.

Deprecated:

This error return is deprecated as of CUDA 3.1. Device emulation mode was removed with the CUDA 3.1 release.

enumerator **cudaErrorInvalidFilterSetting** = 26

This indicates that a non-float texture was being accessed with linear filtering.

This is not supported by CUDA.

enumerator **cudaErrorInvalidNormSetting** = 27

This indicates that an attempt was made to read an unsupported data type as a normalized float.

This is not supported by CUDA.

enumerator **cudaErrorMixedDeviceExecution** = 28

Mixing of device and device emulation code was not allowed.

Deprecated:

This error return is deprecated as of CUDA 3.1. Device emulation mode was removed with the CUDA 3.1 release.

enumerator **cudaErrorNotYetImplemented** = 31

This indicates that the API call is not yet implemented.

Production releases of CUDA will never return this error.

Deprecated:

This error return is deprecated as of CUDA 4.1.

enumerator **cudaErrorMemoryValueTooLarge** = 32

This indicated that an emulated device pointer exceeded the 32-bit address range.

Deprecated:

This error return is deprecated as of CUDA 3.1. Device emulation mode was removed with the CUDA 3.1 release.

enumerator **cudaErrorStubLibrary** = 34

This indicates that the CUDA driver that the application has loaded is a stub library.

Applications that run with the stub rather than a real driver loaded will result in CUDA API returning this error.

enumerator **cudaErrorInsufficientDriver** = 35

This indicates that the installed NVIDIA CUDA driver is older than the CUDA runtime library.

This is not a supported configuration. Users should install an updated NVIDIA display driver to allow the application to run.

enumerator **cudaErrorCallRequiresNewerDriver** = 36

This indicates that the API call requires a newer CUDA driver than the one currently installed.

Users should install an updated NVIDIA CUDA driver to allow the API call to succeed.

enumerator **cudaErrorInvalidSurface** = 37

This indicates that the surface passed to the API call is not a valid surface.

enumerator **cudaErrorDuplicateVariableName** = 43

This indicates that multiple global or constant variables (across separate CUDA source files in the application) share the same string name.

enumerator **cudaErrorDuplicateTextureName** = 44

This indicates that multiple textures (across separate CUDA source files in the application) share the same string name.

enumerator **cudaErrorDuplicateSurfaceName** = 45

This indicates that multiple surfaces (across separate CUDA source files in the application) share the same string name.

enumerator **cudaErrorDevicesUnavailable** = 46

This indicates that all CUDA devices are busy or unavailable at the current time.

Devices are often busy/unavailable due to use of *cudaComputeModeProhibited*, *cudaComputeModeExclusiveProcess*, or when long running CUDA kernels have filled up the GPU and are blocking new work from starting. They can also be unavailable due to memory constraints on a device that already has active CUDA work being performed.

enumerator **cudaErrorIncompatibleDriverContext** = 49

This indicates that the current context is not compatible with this the CUDA Runtime.

This can only occur if you are using CUDA Runtime/Driver interoperability and have created an existing Driver context using the driver API. The Driver context may be incompatible either because the Driver context was created using an older version of the API, because the Runtime API call expects a primary driver context and the Driver context is not primary,

or because the Driver context has been destroyed. Please see *Interactions* with the CUDA Driver API" for more information.

enumerator **cudaErrorMissingConfiguration** = 52

The device function being invoked (usually via *cudaLaunchKernel()*) was not previously configured via the ::cudaConfigureCall() function.

enumerator **cudaErrorPriorLaunchFailure** = 53

This indicated that a previous kernel launch failed.

This was previously used for device emulation of kernel launches.

Deprecated:

This error return is deprecated as of CUDA 3.1. Device emulation mode was removed with the CUDA 3.1 release.

enumerator **cudaErrorLaunchMaxDepthExceeded** = 65

This error indicates that a device runtime grid launch did not occur because the depth of the child grid would exceed the maximum supported number of nested grid launches.

enumerator **cudaErrorLaunchFileScopedTex** = 66

This error indicates that a grid launch did not occur because the kernel uses file-scoped textures which are unsupported by the device runtime.

Kernels launched via the device runtime only support textures created with the Texture Object API's.

enumerator **cudaErrorLaunchFileScopedSurf** = 67

This error indicates that a grid launch did not occur because the kernel uses file-scoped surfaces which are unsupported by the device runtime.

Kernels launched via the device runtime only support surfaces created with the Surface Object API's.

enumerator **cudaErrorSyncDepthExceeded** = 68

This error indicates that a call to *cudaDeviceSynchronize* made from the device runtime failed because the call was made at grid depth greater than than either the default (2 levels of grids) or user specified device limit *cudaLimitDevRuntimeSyncDepth*.

To be able to synchronize on launched grids at a greater depth successfully, the maximum nested depth at which *cudaDeviceSynchronize* will be called must be specified with the *cudaLimitDevRuntimeSyncDepth* limit to the *cudaDeviceSetLimit* api before the host-side launch of a kernel using the device runtime. Keep in mind that additional levels of sync depth require the runtime to reserve large amounts of device memory that cannot be used for user allocations. Note that *cudaDeviceSynchronize* made from device runtime is only supported on devices of compute capability < 9.0.

enumerator **cudaErrorLaunchPendingCountExceeded** = 69

This error indicates that a device runtime grid launch failed because the launch would exceed the limit *cudaLimitDevRuntimePendingLaunchCount*.

For this launch to proceed successfully, *cudaDeviceSetLimit* must be called to set the *cudaLimitDevRuntimePendingLaunchCount* to be higher than the upper bound of outstanding

launches that can be issued to the device runtime. Keep in mind that raising the limit of pending device runtime launches will require the runtime to reserve device memory that cannot be used for user allocations.

enumerator **cudaErrorInvalidDeviceFunction** = 98

The requested device function does not exist or is not compiled for the proper device architecture.

enumerator **cudaErrorNoDevice** = 100

This indicates that no CUDA-capable devices were detected by the installed CUDA driver.

enumerator **cudaErrorInvalidDevice** = 101

This indicates that the device ordinal supplied by the user does not correspond to a valid CUDA device or that the action requested is invalid for the specified device.

enumerator **cudaErrorDeviceNotLicensed** = 102

This indicates that the device doesn't have a valid Grid License.

enumerator **cudaErrorSoftwareValidityNotEstablished** = 103

By default, the CUDA runtime may perform a minimal set of self-tests, as well as CUDA driver tests, to establish the validity of both.

Introduced in CUDA 11.2, this error return indicates that at least one of these tests has failed and the validity of either the runtime or the driver could not be established.

enumerator **cudaErrorStartupFailure** = 127

This indicates an internal startup failure in the CUDA runtime.

enumerator **cudaErrorInvalidKernelImage** = 200

This indicates that the device kernel image is invalid.

enumerator **cudaErrorDeviceUninitialized** = 201

This most frequently indicates that there is no context bound to the current thread.

This can also be returned if the context passed to an API call is not a valid handle (such as a context that has had ::cuCtxDestroy() invoked on it). This can also be returned if a user mixes different API versions (i.e. 3010 context with 3020 API calls). See ::cuCtxGetApiVersion() for more details.

enumerator **cudaErrorMapBufferObjectFailed** = 205

This indicates that the buffer object could not be mapped.

enumerator **cudaErrorUnmapBufferObjectFailed** = 206

This indicates that the buffer object could not be unmapped.

enumerator **cudaErrorArrayIsMapped** = 207

This indicates that the specified array is currently mapped and thus cannot be destroyed.

enumerator **cudaErrorAlreadyMapped** = 208

This indicates that the resource is already mapped.

enumerator **cudaErrorNoKernelImageForDevice** = 209

This indicates that there is no kernel image available that is suitable for the device.

This can occur when a user specifies code generation options for a particular CUDA source file that do not include the corresponding device configuration.

enumerator **cudaErrorAlreadyAcquired** = 210

This indicates that a resource has already been acquired.

enumerator **cudaErrorNotMapped** = 211

This indicates that a resource is not mapped.

enumerator **cudaErrorNotMappedAsArray** = 212

This indicates that a mapped resource is not available for access as an array.

enumerator **cudaErrorNotMappedAsPointer** = 213

This indicates that a mapped resource is not available for access as a pointer.

enumerator **cudaErrorECCUncorrectable** = 214

This indicates that an uncorrectable ECC error was detected during execution.

enumerator **cudaErrorUnsupportedLimit** = 215

This indicates that the *cudaLimit* passed to the API call is not supported by the active device.

enumerator **cudaErrorDeviceAlreadyInUse** = 216

This indicates that a call tried to access an exclusive-thread device that is already in use by a different thread.

enumerator **cudaErrorPeerAccessUnsupported** = 217

This error indicates that P2P access is not supported across the given devices.

enumerator **cudaErrorInvalidPtx** = 218

A PTX compilation failed.

The runtime may fall back to compiling PTX if an application does not contain a suitable binary for the current device.

enumerator **cudaErrorInvalidGraphicsContext** = 219

This indicates an error with the OpenGL or DirectX context.

enumerator **cudaErrorNvlinkUncorrectable** = 220

This indicates that an uncorrectable NVLink error was detected during the execution.

enumerator **cudaErrorJitCompilerNotFound** = 221

This indicates that the PTX JIT compiler library was not found.

The JIT Compiler library is used for PTX compilation. The runtime may fall back to compiling PTX if an application does not contain a suitable binary for the current device.

enumerator **cudaErrorUnsupportedPtxVersion** = 222

This indicates that the provided PTX was compiled with an unsupported toolchain.

The most common reason for this, is the PTX was generated by a compiler newer than what is supported by the CUDA driver and PTX JIT compiler.

enumerator **cudaErrorJitCompilationDisabled** = 223

This indicates that the JIT compilation was disabled.

The JIT compilation compiles PTX. The runtime may fall back to compiling PTX if an application does not contain a suitable binary for the current device.

enumerator **cudaErrorUnsupportedExecAffinity** = 224

This indicates that the provided execution affinity is not supported by the device.

enumerator **cudaErrorUnsupportedDevSideSync** = 225

This indicates that the code to be compiled by the PTX JIT contains unsupported call to cudaDeviceSynchronize.

enumerator **cudaErrorContained** = 226

This indicates that an exception occurred on the device that is now contained by the GPU's error containment capability.

Common causes are - a. Certain types of invalid accesses of peer GPU memory over nvlink b. Certain classes of hardware errors This leaves the process in an inconsistent state and any further CUDA work will return the same error. To continue using CUDA, the process must be terminated and relaunched.

enumerator **cudaErrorInsufficientLoaderVersion** = 227

This indicates that loader version is insufficient for Fatbin.

enumerator **cudaErrorInvalidSource** = 300

This indicates that the device kernel source is invalid.

enumerator **cudaErrorFileNotFound** = 301

This indicates that the file specified was not found.

enumerator **cudaErrorSharedObjectSymbolNotFound** = 302

This indicates that a link to a shared object failed to resolve.

enumerator **cudaErrorSharedObjectInitFailed** = 303

This indicates that initialization of a shared object failed.

enumerator **cudaErrorOperatingSystem** = 304

This error indicates that an OS call failed.

enumerator **cudaErrorInvalidResourceHandle** = 400

This indicates that a resource handle passed to the API call was not valid.

Resource handles are opaque types like *cudaStream_t* and *cudaEvent_t*.

enumerator **cudaErrorIllegalState** = 401

This indicates that a resource required by the API call is not in a valid state to perform the requested operation.

enumerator **cudaErrorLossyQuery** = 402

This indicates an attempt was made to introspect an object in a way that would discard semantically important information.

This is either due to the object using funtionality newer than the API version used to introspect it or omission of optional return arguments.

enumerator **cudaErrorSymbolNotFound** = 500

This indicates that a named symbol was not found.

Examples of symbols are global/constant variable names, driver function names, texture names, and surface names.

enumerator **cudaErrorNotReady** = 600

This indicates that asynchronous operations issued previously have not completed yet.

This result is not actually an error, but must be indicated differently than *cudaSuccess* (which indicates completion). Calls that may return this value include *cudaEventQuery()* and *cudaStreamQuery()*.

enumerator **cudaErrorIllegalAddress** = 700

The device encountered a load or store instruction on an invalid memory address.

This leaves the process in an inconsistent state and any further CUDA work will return the same error. To continue using CUDA, the process must be terminated and relaunched.

enumerator **cudaErrorLaunchOutOfResources** = 701

This indicates that a launch did not occur because it did not have appropriate resources.

Although this error is similar to *cudaErrorInvalidConfiguration*, this error usually indicates that the user has attempted to pass too many arguments to the device kernel, or the kernel launch specifies too many threads for the kernel's register count.

enumerator **cudaErrorLaunchTimeout** = 702

This indicates that the device kernel took too long to execute.

This can only occur if timeouts are enabled - see the device attribute cudaDevAttrKernelExecTimeout for more information. This leaves the process in an inconsistent state and any further CUDA work will return the same error. To continue using CUDA, the process must be terminated and relaunched.

enumerator **cudaErrorLaunchIncompatibleTexturing** = 703

This error indicates a kernel launch that uses an incompatible texturing mode.

enumerator **cudaErrorPeerAccessAlreadyEnabled** = 704

This error indicates that a call to *cudaDeviceEnablePeerAccess()* is trying to re-enable peer addressing on from a context which has already had peer addressing enabled.

enumerator **cudaErrorPeerAccessNotEnabled** = 705

This error indicates that *cudaDeviceDisablePeerAccess()* is trying to disable peer addressing which has not been enabled yet via *cudaDeviceEnablePeerAccess()*.

enumerator **cudaErrorSetOnActiveProcess** = 708

This indicates that the user has called *cudaSetValidDevices()*, *cudaSetDeviceFlags()*, *cudaD3D9SetDirect3DDevice()*, *cudaD3D10SetDirect3DDevice*, *cudaD3D11SetDirect3DDevice()*, or *cudaVDPAUSetVDPAUDevice()* after initializing the CUDA runtime by calling non-device management operations (allocating memory and launching kernels are examples of non-device management operations).

This error can also be returned if using runtime/driver interoperability and there is an existing ::CUcontext active on the host thread.

enumerator **cudaErrorContextIsDestroyed** = 709

This error indicates that the context current to the calling thread has been destroyed using ::cuCtxDestroy, or is a primary context which has not yet been initialized.

enumerator **cudaErrorAssert** = 710

An assert triggered in device code during kernel execution.

The device cannot be used again. All existing allocations are invalid. To continue using CUDA, the process must be terminated and relaunched.

enumerator **cudaErrorTooManyPeers** = 711

This error indicates that the hardware resources required to enable peer access have been exhausted for one or more of the devices passed to ::cudaEnablePeerAccess().

enumerator **cudaErrorHostMemoryAlreadyRegistered** = 712

This error indicates that the memory range passed to *cudaHostRegister()* has already been registered.

enumerator **cudaErrorHostMemoryNotRegistered** = 713

This error indicates that the pointer passed to *cudaHostUnregister()* does not correspond to any currently registered memory region.

enumerator **cudaErrorHardwareStackError** = 714

Device encountered an error in the call stack during kernel execution, possibly due to stack corruption or exceeding the stack size limit.

This leaves the process in an inconsistent state and any further CUDA work will return the same error. To continue using CUDA, the process must be terminated and relaunched.

enumerator **cudaErrorIllegalInstruction** = 715

The device encountered an illegal instruction during kernel execution This leaves the process in an inconsistent state and any further CUDA work will return the same error.

To continue using CUDA, the process must be terminated and relaunched.

enumerator **cudaErrorMisalignedAddress** = 716

The device encountered a load or store instruction on a memory address which is not aligned.

This leaves the process in an inconsistent state and any further CUDA work will return the same error. To continue using CUDA, the process must be terminated and relaunched.

enumerator **cudaErrorInvalidAddressSpace** = 717

While executing a kernel, the device encountered an instruction which can only operate on memory locations in certain address spaces (global, shared, or local), but was supplied a memory address not belonging to an allowed address space.

This leaves the process in an inconsistent state and any further CUDA work will return the same error. To continue using CUDA, the process must be terminated and relaunched.

enumerator **cudaErrorInvalidPc** = 718

The device encountered an invalid program counter.

This leaves the process in an inconsistent state and any further CUDA work will return the same error. To continue using CUDA, the process must be terminated and relaunched.

enumerator **cudaErrorLaunchFailure** = 719

An exception occurred on the device while executing a kernel.

Common causes include dereferencing an invalid device pointer and accessing out of bounds shared memory. Less common cases can be system specific - more information about these cases can be found in the system specific user guide. This leaves the process in an inconsistent state and any further CUDA work will return the same error. To continue using CUDA, the process must be terminated and relaunched.

enumerator **cudaErrorCooperativeLaunchTooLarge** = 720

This error indicates that the number of blocks launched per grid for a kernel that was launched via either *cudaLaunchCooperativeKernel* exceeds the maximum number of blocks as allowed by *cudaOccupancyMaxActiveBlocksPerMultiprocessor* or *cudaOccupancyMaxActiveBlocksPerMultiprocessorWithFlags* times the number of multiprocessors as specified by the device attribute *cudaDevAttrMultiProcessorCount*.

enumerator **cudaErrorTensorMemoryLeak** = 721

An exception occurred on the device while exiting a kernel using tensor memory: the tensor memory was not completely deallocated.

This leaves the process in an inconsistent state and any further CUDA work will return the same error. To continue using CUDA, the process must be terminated and relaunched.

enumerator **cudaErrorNotPermitted** = 800

This error indicates the attempted operation is not permitted.

enumerator **cudaErrorNotSupported** = 801

This error indicates the attempted operation is not supported on the current system or device.

enumerator **cudaErrorSystemNotReady** = 802

This error indicates that the system is not yet ready to start any CUDA work.

To continue using CUDA, verify the system configuration is in a valid state and all required driver daemons are actively running. More information about this error can be found in the system specific user guide.

enumerator **cudaErrorSystemDriverMismatch** = 803

This error indicates that there is a mismatch between the versions of the display driver and the CUDA driver.

Refer to the compatibility documentation for supported versions.

enumerator **cudaErrorCompatNotSupportedOnDevice** = 804

This error indicates that the system was upgraded to run with forward compatibility but the visible hardware detected by CUDA does not support this configuration.

Refer to the compatibility documentation for the supported hardware matrix or ensure that only supported hardware is visible during initialization via the CUDA_VISIBLE_DEVICES environment variable.

enumerator **cudaErrorMpsConnectionFailed** = 805

This error indicates that the MPS client failed to connect to the MPS control daemon or the MPS server.

enumerator **cudaErrorMpsRpcFailure** = 806

This error indicates that the remote procedural call between the MPS server and the MPS client failed.

enumerator **cudaErrorMpsServerNotReady** = 807

This error indicates that the MPS server is not ready to accept new MPS client requests.

This error can be returned when the MPS server is in the process of recovering from a fatal failure.

enumerator **cudaErrorMpsMaxClientsReached** = 808

This error indicates that the hardware resources required to create MPS client have been exhausted.

enumerator **cudaErrorMpsMaxConnectionsReached** = 809

This error indicates the the hardware resources required to device connections have been exhausted.

enumerator **cudaErrorMpsClientTerminated** = 810

This error indicates that the MPS client has been terminated by the server.

To continue using CUDA, the process must be terminated and relaunched.

enumerator **cudaErrorCdpNotSupported** = 811

This error indicates, that the program is using CUDA Dynamic Parallelism, but the current configuration, like MPS, does not support it.

enumerator **cudaErrorCdpVersionMismatch** = 812

This error indicates, that the program contains an unsupported interaction between different versions of CUDA Dynamic Parallelism.

enumerator **cudaErrorStreamCaptureUnsupported** = 900

The operation is not permitted when the stream is capturing.

enumerator **cudaErrorStreamCaptureInvalidated** = 901

The current capture sequence on the stream has been invalidated due to a previous error.

enumerator **cudaErrorStreamCaptureMerge** = 902

The operation would have resulted in a merge of two independent capture sequences.

enumerator **cudaErrorStreamCaptureUnmatched** = 903

The capture was not initiated in this stream.

enumerator **cudaErrorStreamCaptureUnjoined** = 904

The capture sequence contains a fork that was not joined to the primary stream.

enumerator **cudaErrorStreamCaptureIsolation** = 905

A dependency would have been created which crosses the capture sequence boundary.

Only implicit in-stream ordering dependencies are allowed to cross the boundary.

enumerator **cudaErrorStreamCaptureImplicit** = 906

The operation would have resulted in a disallowed implicit dependency on a current capture sequence from cudaStreamLegacy.

enumerator **cudaErrorCapturedEvent** = 907

The operation is not permitted on an event which was last recorded in a capturing stream.

enumerator **cudaErrorStreamCaptureWrongThread** = 908

A stream capture sequence not initiated with the cudaStreamCaptureModeRelaxed argument to *cudaStreamBeginCapture* was passed to *cudaStreamEndCapture* in a different thread.

enumerator **cudaErrorTimeout** = 909

This indicates that the wait operation has timed out.

enumerator **cudaErrorGraphExecUpdateFailure** = 910

This error indicates that the graph update was not performed because it included changes which violated constraints specific to instantiated graph update.

enumerator **cudaErrorExternalDevice** = 911

This indicates that an error has occurred in a device outside of GPU.

It can be a synchronous error w.r.t. CUDA API or an asynchronous error from the external device. In case of asynchronous error, it means that if cuda was waiting for an external device's signal before consuming shared data, the external device signaled an error indicating that the data is not valid for consumption. This leaves the process in an inconsistent state and any further CUDA work will return the same error. To continue using CUDA, the process must be terminated and relaunched. In case of synchronous error, it means that one or more external devices have encountered an error and cannot complete the operation.

enumerator **cudaErrorInvalidClusterSize** = 912

This indicates that a kernel launch error has occurred due to cluster misconfiguration.

enumerator **cudaErrorFunctionNotLoaded** = 913

Indiciates a function handle is not loaded when calling an API that requires a loaded function.

enumerator **cudaErrorInvalidResourceType** = 914

This error indicates one or more resources passed in are not valid resource types for the operation.

enumerator **cudaErrorInvalidResourceConfiguration** = 915

This error indicates one or more resources are insufficient or non-applicable for the operation.

enumerator **cudaErrorStreamDetached** = 917

This error indicates that the requested operation is not permitted because the stream is in a detached state.

This can occur if the green context associated with the stream has been destroyed, limiting the stream's operational capabilities.

enumerator **cudaErrorGraphRecaptureFailure** = 918

This error indicates that a graph recapture failed and had to be terminated.

enumerator **cudaErrorFabricNotReady** = 919

This error indicates GPU fabric is not ready within the bounded wait while the fabric manager probe is still in progress.

Applications may retry after a delay; for the initialization wait budget, see environment variables such as CUDA_FABRIC_INIT_TIMEOUT_MS.

enumerator **cudaErrorUnknown** = 999

This indicates that an unknown internal error has occurred.

enumerator **cudaErrorApiFailureBase** = 10000

enum **cudaExternalMemoryHandleType**

External memory handle types.

Values:

enumerator **cudaExternalMemoryHandleTypeOpaqueFd** = 1
Handle is an opaque file descriptor.

enumerator **cudaExternalMemoryHandleTypeOpaqueWin32** = 2
Handle is an opaque shared NT handle.

enumerator **cudaExternalMemoryHandleTypeOpaqueWin32Kmt** = 3
Handle is an opaque, globally shared handle.

enumerator **cudaExternalMemoryHandleTypeD3D12Heap** = 4
Handle is a D3D12 heap object.

enumerator **cudaExternalMemoryHandleTypeD3D12Resource** = 5
Handle is a D3D12 committed resource.

enumerator **cudaExternalMemoryHandleTypeD3D11Resource** = 6
Handle is a shared NT handle to a D3D11 resource.

enumerator **cudaExternalMemoryHandleTypeD3D11ResourceKmt** = 7
Handle is a globally shared handle to a D3D11 resource.

enumerator **cudaExternalMemoryHandleTypeNvSciBuf** = 8
Handle is an NvSciBuf object.

enum **cudaExternalSemaphoreHandleType**
External semaphore handle types.
Values:

enumerator **cudaExternalSemaphoreHandleTypeOpaqueFd** = 1
Handle is an opaque file descriptor.

enumerator **cudaExternalSemaphoreHandleTypeOpaqueWin32** = 2
Handle is an opaque shared NT handle.

enumerator **cudaExternalSemaphoreHandleTypeOpaqueWin32Kmt** = 3
Handle is an opaque, globally shared handle.

enumerator **cudaExternalSemaphoreHandleTypeD3D12Fence** = 4
Handle is a shared NT handle referencing a D3D12 fence object.

enumerator **cudaExternalSemaphoreHandleTypeD3D11Fence** = 5
Handle is a shared NT handle referencing a D3D11 fence object.

enumerator **cudaExternalSemaphoreHandleTypeNvSciSync** = 6
Opaque handle to NvSciSync Object.

enumerator **cudaExternalSemaphoreHandleTypeKeyedMutex** = 7
Handle is a shared NT handle referencing a D3D11 keyed mutex object.

enumerator **cudaExternalSemaphoreHandleTypeKeyedMutexKmt** = 8
Handle is a shared KMT handle referencing a D3D11 keyed mutex object.

enumerator **cudaExternalSemaphoreHandleTypeTimelineSemaphoreFd** = 9
Handle is an opaque handle file descriptor referencing a timeline semaphore.

enumerator **cudaExternalSemaphoreHandleTypeTimelineSemaphoreWin32** = 10
Handle is an opaque handle file descriptor referencing a timeline semaphore.

enum **cudaFabricOpStatusInfo**
Fabric operation status info.
Values:

enumerator **cudaFabricOpStatusInfoSuccess** = 0x0

enumerator **cudaFabricOpStatusInfoLast** = *cudaFabricOpStatusInfoSuccess*

enumerator **cudaFabricOpStatusInfoMax** = 0x7FFFFFFF

enum **cudaFabricOpStatusSource**
Fabric operation status source.
Values:

enumerator **cudaFabricOpStatusSourceMbarrierV1** = 0x0
1B-aligned 1B-wide status from an mbarrier.layout::v1

enumerator **cudaFabricOpStatusSourceMax** = 0x7FFFFFFF

enum **cudaFlushGPUDirectRDMAWritesOptions**
CUDA GPUDirect RDMA flush writes APIs supported on the device.
Values:

enumerator **cudaFlushGPUDirectRDMAWritesOptionHost** = 1 << 0
cudaDeviceFlushGPUDirectRDMAWrites() and its CUDA Driver API counterpart are supported on the device.

enumerator **cudaFlushGPUDirectRDMAWritesOptionMemOps** = 1 << 1

The ::CU_STREAM_WAIT_VALUE_FLUSH flag and the ::CU_STREAM_MEM_OP_FLUSH_REMOTE_WRITES MemOp are supported on the CUDA device.

enum **cudaFlushGPUDirectRDMAWritesScope**

CUDA GPUDirect RDMA flush writes scopes.

Values:

enumerator **cudaFlushGPUDirectRDMAWritesToOwner** = 100

Blocks until remote writes are visible to the CUDA device context owning the data.

enumerator **cudaFlushGPUDirectRDMAWritesToAllDevices** = 200

Blocks until remote writes are visible to all CUDA device contexts.

enum **cudaFlushGPUDirectRDMAWritesTarget**

CUDA GPUDirect RDMA flush writes targets.

Values:

enumerator **cudaFlushGPUDirectRDMAWritesTargetCurrentDevice**

Sets the target for *cudaDeviceFlushGPUDirectRDMAWrites()* to the currently active CUDA device context.

enum **cudaFuncAttribute**

CUDA function attributes that can be set using *cudaFuncSetAttribute*.

Values:

enumerator **cudaFuncAttributeMaxDynamicSharedMemorySize** = 8

Maximum dynamic shared memory size.

enumerator **cudaFuncAttributePreferredSharedMemoryCarveout** = 9

Preferred shared memory-L1 cache split.

enumerator **cudaFuncAttributeClusterDimMustBeSet** = 10

Indicator to enforce valid cluster dimension specification on kernel launch.

enumerator **cudaFuncAttributeRequiredClusterWidth** = 11

Required cluster width.

enumerator **cudaFuncAttributeRequiredClusterHeight** = 12

Required cluster height.

enumerator **cudaFuncAttributeRequiredClusterDepth** = 13

Required cluster depth.

enumerator **cudaFuncAttributeNonPortableClusterSizeAllowed** = 14

Whether non-portable cluster scheduling policy is supported.

enumerator **cudaFuncAttributeClusterSchedulingPolicyPreference** = 15

Required cluster scheduling policy preference.

enumerator **cudaFuncAttributeSharedMemoryMode** = 16

Setting that controls a kernel's use of non-portable or oversized shared memory configurations.

enumerator **cudaFuncAttributeMax**

enum **cudaFuncCache**

CUDA function cache configurations.

Values:

enumerator **cudaFuncCachePreferNone** = 0

Default function cache configuration, no preference.

enumerator **cudaFuncCachePreferShared** = 1

Prefer larger shared memory and smaller L1 cache

enumerator **cudaFuncCachePreferL1** = 2

Prefer larger L1 cache and smaller shared memory.

enumerator **cudaFuncCachePreferEqual** = 3

Prefer equal size L1 cache and shared memory.

enum **cudaGPUDirectRDMAWritesOrdering**

CUDA GPUDirect RDMA flush writes ordering features of the device.

Values:

enumerator **cudaGPUDirectRDMAWritesOrderingNone** = 0

The device does not natively support ordering of GPUDirect RDMA writes.

::cudaFlushGPUDirectRDMAWrites() can be leveraged if supported.

enumerator **cudaGPUDirectRDMAWritesOrderingOwner** = 100

Natively, the device can consistently consume GPUDirect RDMA writes, although other CUDA devices may not.

enumerator **cudaGPUDirectRDMAWritesOrderingAllDevices** = 200

Any CUDA device in the system can consistently consume GPUDirect RDMA writes to this device.

enum **cudaGetDriverEntryPointFlags**

Flags to specify search options to be used with *cudaGetDriverEntryPoint* For more details see ::cuGetProcAddress.

Values:

enumerator **cudaEnableDefault** = 0x0

Default search mode for driver symbols.

enumerator **cudaEnableLegacyStream** = 0x1

Search for legacy versions of driver symbols.

enumerator **cudaEnablePerThreadDefaultStream** = 0x2

Search for per-thread versions of driver symbols.

enum **cudaGraphChildGraphNodeOwnership**

Child graph node ownership.

Values:

enumerator **cudaGraphChildGraphOwnershipClone** = 0

Default behavior for a child graph node.

Child graph is cloned into the parent and memory allocation/free nodes can't be present in the child graph.

enumerator **cudaGraphChildGraphOwnershipMove** = 1

The child graph is moved to the parent.

The handle to the child graph is owned by the parent and will be destroyed when the parent is destroyed.

The following restrictions apply to child graphs after they have been moved: Cannot be independently instantiated or destroyed; Cannot be added as a child graph of a separate parent graph; Cannot be used as an argument to cudaGraphExecUpdate; Cannot have additional memory allocation or free nodes added.

enumerator **cudaGraphChildGraphOwnershipInvalid** = -1

Invalid ownership flag.

Set when params are queried to prevent accidentally reusing the driver-owned graph object

enum **cudaGraphConditionalHandleFlags**

Values:

enumerator **cudaGraphCondAssignDefault** = 1

Apply default handle value when graph is launched.

enum **cudaGraphConditionalNodeType**

CUDA conditional node types.

Values:

enumerator **cudaGraphCondTypeIf** = 0

Conditional 'if/else' Node.

Body[0] executed if condition is non-zero. If `size` == 2, an optional ELSE graph is created and this is executed if the condition is zero.

enumerator **cudaGraphCondTypeWhile** = 1

Conditional 'while' Node.

Body executed repeatedly while condition value is non-zero.

enumerator **cudaGraphCondTypeSwitch** = 2

Conditional 'switch' Node.

Body[n] is executed once, where 'n' is the value of the condition. If the condition does not match a body index, no body is launched.

enum **cudaGraphDebugDotFlags**

CUDA Graph debug write options.

Values:

enumerator **cudaGraphDebugDotFlagsVerbose** = 1 << 0

Output all debug data as if every debug flag is enabled.

enumerator **cudaGraphDebugDotFlagsKernelNodeParams** = 1 << 2

Adds *cudaKernelNodeParams* to output.

enumerator **cudaGraphDebugDotFlagsMemcpyNodeParams** = 1 << 3

Adds *cudaMemcpy3DParms* to output.

enumerator **cudaGraphDebugDotFlagsMemsetNodeParams** = 1 << 4

Adds *cudaMemsetParams* to output.

enumerator **cudaGraphDebugDotFlagsHostNodeParams** = 1 << 5

Adds *cudaHostNodeParams* to output.

enumerator **cudaGraphDebugDotFlagsEventNodeParams** = 1 << 6

Adds cudaEvent_t handle from record and wait nodes to output.

enumerator **cudaGraphDebugDotFlagsExtSemasSignalNodeParams** = 1 << 7

Adds *cudaExternalSemaphoreSignalNodeParams* values to output.

enumerator **cudaGraphDebugDotFlagsExtSemasWaitNodeParams** = 1 << 8

Adds *cudaExternalSemaphoreWaitNodeParams* to output.

enumerator **cudaGraphDebugDotFlagsKernelNodeAttributes** = 1 << 9

Adds cudaKernelNodeAttrID values to output.

enumerator **cudaGraphDebugDotFlagsHandles** = 1 << 10

Adds node handles and every kernel function handle to output.

enumerator **cudaGraphDebugDotFlagsConditionalNodeParams** = 1 << 15

Adds *cudaConditionalNodeParams* to output.

enumerator **cudaGraphDebugDotFlagsMemcpyBatchNodeParams** = 1 << 16

Adds batch memcpy node parameters to the output.

enum **cudaGraphDependencyType**

Type annotations that can be applied to graph edges as part of *cudaGraphEdgeData*.

Values:

enumerator **cudaGraphDependencyTypeDefault** = 0

This is an ordinary dependency.

enumerator **cudaGraphDependencyTypeProgrammatic** = 1

This dependency type allows the downstream node to use *cudaGridDependencySynchronize()*.

It may only be used between kernel nodes, and must be used with either the *cudaGraphKernelNodePortProgrammatic* or *cudaGraphKernelNodePortLaunchCompletion* outgoing port.

enum **cudaGraphExecUpdateResult**

CUDA Graph Update error types.

Values:

enumerator **cudaGraphExecUpdateSuccess** = 0x0

The update succeeded.

enumerator **cudaGraphExecUpdateError** = 0x1

The update failed for an unexpected reason which is described in the return value of the function.

enumerator **cudaGraphExecUpdateErrorTopologyChanged** = 0x2

The update failed because the topology changed.

enumerator **cudaGraphExecUpdateErrorNodeTypeChanged** = 0x3

The update failed because a node type changed.

enumerator **cudaGraphExecUpdateErrorFunctionChanged** = 0x4

The update failed because the function of a kernel node changed (CUDA driver < 11.2)

enumerator **cudaGraphExecUpdateErrorParametersChanged** = 0x5

The update failed because the parameters changed in a way that is not supported.

enumerator **cudaGraphExecUpdateErrorNotSupported** = 0x6

The update failed because something about the node is not supported.

enumerator **cudaGraphExecUpdateErrorUnsupportedFunctionChange** = 0x7

The update failed because the function of a kernel node changed in an unsupported way.

enumerator **cudaGraphExecUpdateErrorAttributesChanged** = 0x8

The update failed because the node attributes changed in a way that is not supported.

enum **cudaGraphInstantiateFlags**

Flags for instantiating a graph.

Values:

enumerator **cudaGraphInstantiateFlagAutoFreeOnLaunch** = 1

Automatically free memory allocated in a graph before relaunching.

enumerator **cudaGraphInstantiateFlagUpload** = 2

Automatically upload the graph after instantiation.

Only supported by *cudaGraphInstantiateWithParams* . The upload will be performed using the

stream provided in `instantiateParams`.

enumerator **cudaGraphInstantiateFlagDeviceLaunch** = 4

Instantiate the graph to be launchable from the device.

This flag can only

be used on platforms which support unified addressing. This flag cannot be

used in conjunction with cudaGraphInstantiateFlagAutoFreeOnLaunch.

enumerator **cudaGraphInstantiateFlagUseNodePriority** = 8

Run the graph using the per-node priority attributes rather than the priority of the stream it is launched into.

enum **cudaGraphInstantiateResult**

Graph instantiation results.

Values:

enumerator **cudaGraphInstantiateSuccess** = 0

Instantiation succeeded.

enumerator **cudaGraphInstantiateError** = 1

Instantiation failed for an unexpected reason which is described in the return value of the function.

enumerator **cudaGraphInstantiateInvalidStructure** = 2

Instantiation failed due to invalid structure, such as cycles.

enumerator **cudaGraphInstantiateNodeOperationNotSupported** = 3

Instantiation for device launch failed because the graph contained an unsupported operation.

enumerator **cudaGraphInstantiateMultipleDevicesNotSupported** = 4

Instantiation for device launch failed due to the nodes belonging to different contexts.

enumerator **cudaGraphInstantiateConditionalHandleUnused** = 5

One or more conditional handles are not associated with conditional nodes.

enum **cudaGraphKernelNodeField**

Specifies the field to update when performing multiple node updates from the device.

Values:

enumerator **cudaGraphKernelNodeFieldInvalid** = 0

Invalid field.

enumerator **cudaGraphKernelNodeFieldGridDim**

Grid dimension update.

enumerator **cudaGraphKernelNodeFieldParam**

Kernel parameter update.

enumerator **cudaGraphKernelNodeFieldEnabled**

Node enable/disable.

enum **cudaGraphMemAttributeType**

Graph memory attributes.

Values:

enumerator **cudaGraphMemAttrUsedMemCurrent** = 0x0

(value type = cuuint64_t) Amount of memory, in bytes, currently associated with graphs.

enumerator **cudaGraphMemAttrUsedMemHigh** = 0x1

(value type = cuuint64_t) High watermark of memory, in bytes, associated with graphs since the last time it was reset.

High watermark can only be reset to zero.

enumerator **cudaGraphMemAttrReservedMemCurrent** = 0x2

(value type = cuuint64_t) Amount of memory, in bytes, currently allocated for use by the CUDA graphs asynchronous allocator.

enumerator **cudaGraphMemAttrReservedMemHigh** = 0x3

(value type = cuuint64_t) High watermark of memory, in bytes, currently allocated for use by the CUDA graphs asynchronous allocator.

enum **cudaGraphNodeType**

CUDA Graph node types.

Values:

enumerator **cudaGraphNodeTypeKernel** = 0x00

GPU kernel node.

enumerator **cudaGraphNodeTypeMemcpy** = 0x01

Memcpy node.

enumerator **cudaGraphNodeTypeMemset** = 0x02

Memset node.

enumerator **cudaGraphNodeTypeHost** = 0x03

Host (executable) node.

enumerator **cudaGraphNodeTypeGraph** = 0x04

Node which executes an embedded graph.

enumerator **cudaGraphNodeTypeEmpty** = 0x05

Empty (no-op) node.

enumerator **cudaGraphNodeTypeWaitEvent** = 0x06

External event wait node.

enumerator **cudaGraphNodeTypeEventRecord** = 0x07

External event record node.

enumerator **cudaGraphNodeTypeExtSemaphoreSignal** = 0x08

External semaphore signal node.

enumerator **cudaGraphNodeTypeExtSemaphoreWait** = 0x09

External semaphore wait node.

enumerator **cudaGraphNodeTypeMemAlloc** = 0x0a

Memory allocation node.

enumerator **cudaGraphNodeTypeMemFree** = 0x0b

Memory free node.

enumerator **cudaGraphNodeTypeConditional** = 0x0d

Conditional node.

```
                                    May be used to implement a conditional
↪execution path or loop
                                    inside of a graph. The graph(s) contained
↪within the body of the conditional node
                                    can be selectively executed or iterated
↪upon based on the value of a conditional
                                    variable.

                                    Handles must be created in advance of
↪creating the node
                                    using ::cudaGraphConditionalHandleCreate.

                                    The following restrictions apply to graphs
↪which contain conditional nodes:
                                      The graph cannot be used in a child node.
                                      Only one instantiation of the graph may
↪exist at any point in time.
                                      The graph cannot be cloned.

                                    To set the control value, supply a default
↪value when creating the handle and/or
                                    call ::cudaGraphSetConditional from device
↪code.
```

enumerator **cudaGraphNodeTypeMemcpyBatch** = 0x0e

Batch Memcpy Node.

enumerator **cudaGraphNodeTypeReserved16** = 0x10

Reserved.

enumerator **cudaGraphNodeTypeExternalEngine** = 0x11

Reserved for External Engine node.

enumerator **cudaGraphNodeTypeCount**

enum **cudaGraphRecaptureStatus**

Possible recapture statuses that can be returned to the user callback.

Values:

enumerator **cudaGraphRecaptureEligibleForUpdate** = 0

Node is eligible for update in an instantiated graph.

enumerator **cudaGraphRecaptureIneligibleForUpdate** = 1

Parameter changes in the node cannot be applied to an instantiated graph.

enumerator **cudaGraphRecaptureError** = 2

Error while attempting to recapture the node.

The recapture will be ended regardless of the return value from the callback.

enum **cudaGraphicsCubeFace**

CUDA graphics interop array indices for cube maps.

Values:

enumerator **cudaGraphicsCubeFacePositiveX** = 0x00

Positive X face of cubemap.

enumerator **cudaGraphicsCubeFaceNegativeX** = 0x01

Negative X face of cubemap.

enumerator **cudaGraphicsCubeFacePositiveY** = 0x02

Positive Y face of cubemap.

enumerator **cudaGraphicsCubeFaceNegativeY** = 0x03

Negative Y face of cubemap.

enumerator **cudaGraphicsCubeFacePositiveZ** = 0x04

Positive Z face of cubemap.

enumerator **cudaGraphicsCubeFaceNegativeZ** = 0x05

Negative Z face of cubemap.

enum **cudaGraphicsMapFlags**

CUDA graphics interop map flags.

Values:

enumerator **cudaGraphicsMapFlagsNone** = 0

Default; Assume resource can be read/written.

enumerator **cudaGraphicsMapFlagsReadOnly** = 1

CUDA will not write to this resource.

enumerator **cudaGraphicsMapFlagsWriteDiscard** = 2

CUDA will only write to and will not read from this resource.

enum **cudaGraphicsRegisterFlags**

CUDA graphics interop register flags.

Values:

enumerator **cudaGraphicsRegisterFlagsNone** = 0
Default.

enumerator **cudaGraphicsRegisterFlagsReadOnly** = 1
CUDA will not write to this resource.

enumerator **cudaGraphicsRegisterFlagsWriteDiscard** = 2
CUDA will only write to and will not read from this resource.

enumerator **cudaGraphicsRegisterFlagsSurfaceLoadStore** = 4
CUDA will bind this resource to a surface reference.

enumerator **cudaGraphicsRegisterFlagsTextureGather** = 8
CUDA will perform texture gather operations on this resource.

enum **cudaHostTaskSyncMode**
Flags for host task sync mode.
Values:

enumerator **cudaHostTaskBlocking** = 0x0

enumerator **cudaHostTaskSpinWait** = 0x1

enum **cudaJitOption**
Online compiler and linker options.
Values:

enumerator **cudaJitMaxRegisters** = 0
Max number of registers that a thread may use.

Option type: unsigned int

Applies to: compiler only

enumerator **cudaJitThreadsPerBlock** = 1

IN: Specifies minimum number of threads per block to target compilation for

OUT: Returns the number of threads the compiler actually targeted.

This restricts the resource utilization of the compiler (e.g. max registers) such that a block with the given number of threads should be able to launch based on register limitations. Note, this option does not currently take into account any other resource limitations, such as shared memory utilization.

Option type: unsigned int

Applies to: compiler only

enumerator **cudaJitWallTime** = 2

Overwrites the option value with the total wall clock time, in milliseconds, spent in the compiler and linker

Option type: float

Applies to: compiler and linker.

enumerator **cudaJitInfoLogBuffer** = 3

Pointer to a buffer in which to print any log messages that are informational in nature (the buffer size is specified via option *cudaJitInfoLogBufferSizeBytes*)

Option type: char *

Applies to: compiler and linker.

enumerator **cudaJitInfoLogBufferSizeBytes** = 4

IN: Log buffer size in bytes.

Log messages will be capped at this size (including null terminator)

OUT: Amount of log buffer filled with messages

Option type: unsigned int

Applies to: compiler and linker

enumerator **cudaJitErrorLogBuffer** = 5

Pointer to a buffer in which to print any log messages that reflect errors (the buffer size is specified via option *cudaJitErrorLogBufferSizeBytes*)

Option type: char *

Applies to: compiler and linker.

enumerator **cudaJitErrorLogBufferSizeBytes** = 6

IN: Log buffer size in bytes.

Log messages will be capped at this size (including null terminator)

OUT: Amount of log buffer filled with messages

Option type: unsigned int

Applies to: compiler and linker

enumerator **cudaJitOptimizationLevel** = 7

Level of optimizations to apply to generated code (0 - 4), with 4 being the default and highest level of optimizations.

Option type: unsigned int

Applies to: compiler only

enumerator **cudaJitFallbackStrategy** = 10

Specifies choice of fallback strategy if matching cubin is not found.

Choice is based on supplied *cudaJit_Fallback*. Option type: unsigned int for enumerated type *cudaJit_Fallback* Applies to: compiler only

enumerator **cudaJitGenerateDebugInfo** = 11

Specifies whether to create debug information in output (-g) (0: false, default)

Option type: int

Applies to: compiler and linker.

enumerator **cudaJitLogVerbose** = 12

Generate verbose log messages (0: false, default)

Option type: int

Applies to: compiler and linker.

enumerator **cudaJitGenerateLineInfo** = 13

Generate line number information (-lineinfo) (0: false, default)

Option type: int

Applies to: compiler only.

enumerator **cudaJitCacheMode** = 14

Specifies whether to enable caching explicitly (-dlcm)

Choice is based on supplied *cudaJit_CacheMode*.

Option type: unsigned int for enumerated type *cudaJit_CacheMode* Applies to: compiler only

enumerator **cudaJitPositionIndependentCode** = 30

Generate position independent code (0: false)

Option type: int

Applies to: compiler only.

enumerator **cudaJitMinCtaPerSm** = 31

This option hints to the JIT compiler the minimum number of CTAs from the kernel's grid to be mapped to a SM.

This option is ignored when used together with *cudaJitMaxRegisters* or *cudaJitThreadsPerBlock*. Optimizations based on this option need *cudaJitMaxThreadsPerBlock* to be specified as well. For kernels already using PTX directive .minnctapersm, this option will be ignored by default. Use *cudaJitOverrideDirectiveValues* to let this option take precedence over the PTX directive. Option type: unsigned int

Applies to: compiler only

enumerator **cudaJitMaxThreadsPerBlock** = 32

Maximum number threads in a thread block, computed as the product of the maximum extent specifed for each dimension of the block.

This limit is guaranteed not to be exeeded in any invocation of the kernel. Exceeding the the maximum number of threads results in runtime error or kernel launch failure. For kernels already using PTX directive .maxntid, this option will be ignored by default. Use *cudaJitOverrideDirectiveValues* to let this option take precedence over the PTX directive. Option type: int

Applies to: compiler only

enumerator **cudaJitOverrideDirectiveValues** = 33

This option lets the values specified using *cudaJitMaxRegisters*, *cudaJitThreadsPerBlock*, *cudaJitMaxThreadsPerBlock* and *cudaJitMinCtaPerSm* take precedence over any PTX directives.

(0: Disable, default; 1: Enable) Option type: int

Applies to: compiler only

enum **cudaJit_CacheMode**

Caching modes for dlcm.

Values:

enumerator **cudaJitCacheOptionNone** = 0

Compile with no -dlcm flag specified.

enumerator **cudaJitCacheOptionCG**

Compile with L1 cache disabled.

enumerator **cudaJitCacheOptionCA**

Compile with L1 cache enabled.

enum **cudaJit_Fallback**

Cubin matching fallback strategies.

Values:

enumerator **cudaPreferPtx** = 0

Prefer to compile ptx if exact binary match not found.

enumerator **cudaPreferBinary**

Prefer to fall back to compatible binary code if exact match not found.

enum **cudaKernelFunctionType**

CUDA Kernel Function Handle Type.

Values:

enumerator **cudaKernelFunctionTypeUnspecified** = 0x00
CUDA will attempt to deduce the type of the function handle.

enumerator **cudaKernelFunctionTypeDeviceEntry** = 0x01
Function handle is a device-entry function pointer(i.e. **global** function pointer)

enumerator **cudaKernelFunctionTypeKernel** = 0x02
Function handle is a cudaKernel_t.

enumerator **cudaKernelFunctionTypeFunction** = 0x03
Function handle is a cudaFunction_t.

enum **cudaLaunchAttributeID**
Launch attributes enum; used as id field of *cudaLaunchAttribute*.
Values:

enumerator **cudaLaunchAttributeIgnore** = 0
Ignored entry, for convenient composition.

enumerator **cudaLaunchAttributeAccessPolicyWindow** = 1
Valid for streams, graph nodes, launches.
See cudaLaunchAttributeValue::accessPolicyWindow.

enumerator **cudaLaunchAttributeCooperative** = 2
Valid for graph nodes, launches.
See cudaLaunchAttributeValue::cooperative.

enumerator **cudaLaunchAttributeSynchronizationPolicy** = 3
Valid for streams.
See cudaLaunchAttributeValue::syncPolicy.

enumerator **cudaLaunchAttributeClusterDimension** = 4
Valid for graph nodes, launches.
See cudaLaunchAttributeValue::clusterDim.

enumerator **cudaLaunchAttributeClusterSchedulingPolicyPreference** = 5
Valid for graph nodes, launches.
See cudaLaunchAttributeValue::clusterSchedulingPolicyPreference.

enumerator **cudaLaunchAttributeProgrammaticStreamSerialization** = 6
Valid for launches.

Setting cudaLaunchAttributeValue::programmaticStreamSerializationAllowed to non-0 signals that the kernel will use programmatic means to resolve its stream dependency, so that the CUDA runtime should opportunistically allow the grid's execution to overlap with the previous kernel in the stream, if that kernel requests the overlap. The dependent launches can choose to wait on the dependency using the programmatic sync (*cudaGridDependencySynchronize()* or equivalent PTX instructions).

enumerator **cudaLaunchAttributeProgrammaticEvent** = 7

Valid for launches.

Set cudaLaunchAttributeValue::programmaticEvent to record the event. Event recorded through this launch attribute is guaranteed to only trigger after all block in the associated kernel trigger the event. A block can trigger the event programmatically in a future CUDA release. A trigger can also be inserted at the beginning of each block's execution if triggerAtBlockStart is set to non-0. The dependent launches can choose to wait on the dependency using the programmatic sync (*cudaGridDependencySynchronize()* or equivalent PTX instructions). Note that dependents (including the CPU thread calling *cudaEventSynchronize()*) are not guaranteed to observe the release precisely when it is released. For example, *cudaEventSynchronize()* may only observe the event trigger long after the associated kernel has completed. This recording type is primarily meant for establishing programmatic dependency between device tasks. Note also this type of dependency allows, but does not guarantee, concurrent execution of tasks.

The event supplied must not be an interprocess or interop event. The event must disable timing (i.e. must be created with the *cudaEventDisableTiming* flag set).

enumerator **cudaLaunchAttributePriority** = 8

Valid for streams, graph nodes, launches.

See cudaLaunchAttributeValue::priority.

enumerator **cudaLaunchAttributeMemSyncDomainMap** = 9

Valid for streams, graph nodes, launches.

See cudaLaunchAttributeValue::memSyncDomainMap.

enumerator **cudaLaunchAttributeMemSyncDomain** = 10

Valid for streams, graph nodes, launches.

See cudaLaunchAttributeValue::memSyncDomain.

enumerator **cudaLaunchAttributePreferredClusterDimension** = 11

Valid for graph nodes and launches.

Set cudaLaunchAttributeValue::preferredClusterDim to allow the kernel launch to specify a preferred substitute cluster dimension. Blocks may be grouped according to either the dimensions specified with this attribute (grouped into a "preferred substitute cluster"), or the one specified with *cudaLaunchAttributeClusterDimension* attribute (grouped into a "regular cluster"). The cluster dimensions of a "preferred substitute cluster" shall be an integer multiple greater than zero of the regular cluster dimensions. The device will attempt - on a best-effort basis - to group thread blocks into preferred clusters over grouping them into regular clusters. When it deems necessary (primarily when the device temporarily runs out of physical resources to launch the larger preferred clusters), the device may switch to launch the regular clusters instead to attempt to utilize as much of the physical device resources as possible.

Each type of cluster will have its enumeration / coordinate setup as if the grid consists solely of its type of cluster. For example, if the preferred substitute cluster dimensions double the regular cluster dimensions, there might be simultaneously a regular cluster indexed at (1,0,0), and a preferred cluster indexed at (1,0,0). In this example, the preferred substitute cluster (1,0,0) replaces regular clusters (2,0,0) and (3,0,0) and groups their blocks.

This attribute will only take effect when a regular cluster dimension has been specified. The preferred substitute cluster dimension must be an integer multiple greater than zero of the regular cluster dimension and must divide the grid. It must also be no more than `maxBlocksPerCluster`, if it is set in the kernel's `__launch_bounds__`. Otherwise it must be less than the maximum value the driver can support. Otherwise, setting this attribute to a value physically unable to fit on any particular device is permitted.

enumerator **cudaLaunchAttributeLaunchCompletionEvent** = 12

Valid for launches.

Set cudaLaunchAttributeValue::launchCompletionEvent to record the event.

Nominally, the event is triggered once all blocks of the kernel have begun execution. Currently this is a best effort. If a kernel B has a launch completion dependency on a kernel A, B may wait until A is complete. Alternatively, blocks of B may begin before all blocks of A have begun, for example if B can claim execution resources unavailable to A (e.g. they run on different GPUs) or if B is a higher priority than A. Exercise caution if such an ordering inversion could lead to deadlock.

A launch completion event is nominally similar to a programmatic event with `triggerAtBlockStart` set except that it is not visible to *cudaGridDependencySynchronize()* and can be used with compute capability less than 9.0.

The event supplied must not be an interprocess or interop event. The event must disable timing (i.e. must be created with the *cudaEventDisableTiming* flag set).

enumerator **cudaLaunchAttributeDeviceUpdatableKernelNode** = 13

Valid for graph nodes, launches.

This attribute is graphs-only, and passing it to a launch in a non-capturing stream will result in an error.

:cudaLaunchAttributeValue::deviceUpdatableKernelNode::deviceUpdatable can only be set to 0 or 1. Setting the field to 1 indicates that the corresponding kernel node should be device-updatable. On success, a handle will be returned via ::cudaLaunchAttributeValue::deviceUpdatableKernelNode::devNode which can be passed to the various device-side update functions to update the node's kernel parameters from within another kernel. For more information on the types of device updates that can be made, as well as the relevant limitations thereof, see *cudaGraphKernelNodeUpdatesApply* .

Nodes which are device-updatable have additional restrictions compared to regular kernel nodes. Firstly, device-updatable nodes cannot be removed from their graph via *cudaGraphDestroyNode*. Additionally, once opted-in to this functionality, a node cannot opt out, and any attempt to set the deviceUpdatable attribute to 0 will result in an error. Device-updatable kernel nodes also cannot have their attributes copied to/from another kernel node via *cudaGraphKernelNodeCopyAttributes*. Graphs containing one or more device-updatable nodes also do not allow multiple instantiation, and neither the graph nor its instantiated version can be passed to *cudaGraphExecUpdate* .

If a graph contains device-updatable nodes and updates those nodes from the device from within the graph, the graph must be uploaded with ::cuGraphUpload before it is launched. For such a graph, if host-side executable graph updates are made to the device-updatable nodes, the graph must be uploaded before it is launched again.

enumerator **cudaLaunchAttributePreferredSharedMemoryCarveout** = 14

Valid for launches.

On devices where the L1 cache and shared memory use the same hardware resources, setting cudaLaunchAttributeValue::sharedMemCarveout to a percentage between 0-100 signals sets the shared memory carveout preference in percent of the total shared memory for that kernel launch. This attribute takes precedence over *cudaFuncAttributePreferredSharedMemoryCarveout*. This is only a hint, and the driver can choose a different configuration if required for the launch.

enumerator **cudaLaunchAttributeNvlinkUtilCentricScheduling** = 16

Valid for streams, graph nodes, launches.

This attribute is a hint to the CUDA runtime that the launch should attempt to make the kernel maximize its NVLINK utilization.

When possible to honor this hint, CUDA will assume each block in the grid launch will carry out an even amount of NVLINK traffic, and make a best-effort attempt to adjust the kernel launch based on that assumption.

This attribute is a hint only. CUDA makes no functional or performance guarantee. Its applicability can be affected by many different factors, including driver version (i.e. CUDA doesn't guarantee the performance characteristics will be maintained between driver versions or a driver update could alter or regress previously observed perf characteristics.) It also doesn't guarantee a successful result, i.e. applying the attribute may not improve the performance of either the targeted kernel or the encapsulating application.

Valid values for cudaLaunchAttributeValue::nvlinkUtilCentricScheduling are 0 (disabled) and 1 (enabled).

enumerator **cudaLaunchAttributePortableClusterSizeMode** = 17

Valid for graph nodes, launches.

This indicates whether the kernel launch is allowed to use a non-portable cluster size. Valid values for cudaLaunchAttributeValue::portableClusterSizeMode are values for *cudaLaunchAttributePortableClusterMode* Any other value will return *cudaErrorInvalidValue*

enumerator **cudaLaunchAttributeSharedMemoryMode** = 18

Valid for graph nodes, launches.

This indicates that the kernel launch is allowed to use a non-portable shared memory mode.

enum **cudaLaunchAttributePortableClusterMode**

Enum for defining applicability of portable cluster size, used with *cudaLaunchKernelEx*.

Values:

enumerator **cudaLaunchPortableClusterModeDefault** = 0

The default to use for allowing non-portable cluster size on launch - uses current function attribute for *cudaFuncAttributeNonPortableClusterSizeAllowed*.

enumerator **cudaLaunchPortableClusterModeRequirePortable** = 1

Specifies that the cluster size requested must be a portable size.

enumerator **cudaLaunchPortableClusterModeAllowNonPortable** = 2

Specifies that the cluster size requested may be a non-portable size.

enum **cudaLaunchMemSyncDomain**

Memory Synchronization Domain.

A kernel can be launched in a specified memory synchronization domain that affects all memory operations issued by that kernel. A memory barrier issued in one domain will only order memory operations in that domain, thus eliminating latency increase from memory barriers ordering unrelated traffic.

By default, kernels are launched in domain 0. Kernel launched with *cudaLaunchMemSyncDomainRemote* will have a different domain ID. User may also alter the domain ID with *cudaLaunchMemSyncDomainMap* for a specific stream / graph node / kernel launch. See *cudaLaunchAttributeMemSyncDomain*, *cudaStreamSetAttribute*, *cudaLaunchKernelEx*, *cudaGraphKernelNodeSetAttribute*.

Memory operations done in kernels launched in different domains are considered system-scope distanced. In other words, a GPU scoped memory synchronization is not sufficient for memory order to be observed by kernels in another memory synchronization domain even if they are on the same GPU.

Values:

enumerator **cudaLaunchMemSyncDomainDefault** = 0

Launch kernels in the default domain.

enumerator **cudaLaunchMemSyncDomainRemote** = 1

Launch kernels in the remote domain.

enum **cudaLibraryOption**

Library options to be specified with *cudaLibraryLoadData()* or *cudaLibraryLoadFromFile()*

Values:

enumerator **cudaLibraryHostUniversalFunctionAndDataTable** = 0

enumerator **cudaLibraryBinaryIsPreserved** = 1

Specifes that the argument `code` passed to *cudaLibraryLoadData()* will be preserved.

Specifying this option will let the driver know that `code` can be accessed at any point until *cudaLibraryUnload()*. The default behavior is for the driver to allocate and maintain its own copy of `code`. Note that this is only a memory usage optimization hint and the driver can choose to ignore it if required. Specifying this option with *cudaLibraryLoadFromFile()* is invalid and will return *cudaErrorInvalidValue*.

enum **cudaLimit**

CUDA Limits.

Values:

enumerator **cudaLimitStackSize** = 0x00

GPU thread stack size.

enumerator **cudaLimitPrintfFifoSize** = 0x01

GPU printf FIFO size.

enumerator **cudaLimitMallocHeapSize** = 0x02

GPU malloc heap size.

enumerator **cudaLimitDevRuntimeSyncDepth** = 0x03

GPU device runtime synchronize depth.

enumerator **cudaLimitDevRuntimePendingLaunchCount** = 0x04

GPU device runtime pending launch count.

enumerator **cudaLimitMaxL2FetchGranularity** = 0x05

A value between 0 and 128 that indicates the maximum fetch granularity of L2 (in Bytes).

This is a hint

enumerator **cudaLimitPersistingL2CacheSize** = 0x06

A size in bytes for L2 persisting lines cache size.

enumerator **cudaLimitPerBlockMemorySize** = 0x0A

Per-block memory size.

enum **cudaLogLevel**

Values:

enumerator **cudaLogLevelError** = 0

enumerator **cudaLogLevelWarning** = 1

enum **cudaMemAccessFlags**

Specifies the memory protection flags for mapping.

Values:

enumerator **cudaMemAccessFlagsProtNone** = 0

Default, make the address range not accessible.

enumerator **cudaMemAccessFlagsProtRead** = 1
Make the address range read accessible.

enumerator **cudaMemAccessFlagsProtReadWrite** = 3
Make the address range read-write accessible.

enum **cudaMemAllocationHandleType**
Flags for specifying particular handle types.
Values:

enumerator **cudaMemHandleTypeNone** = 0x0
Does not allow any export mechanism.
>

enumerator **cudaMemHandleTypePosixFileDescriptor** = 0x1
Allows a file descriptor to be used for exporting.
Permitted only on POSIX systems. (int)

enumerator **cudaMemHandleTypeWin32** = 0x2
Allows a Win32 NT handle to be used for exporting.
(HANDLE)

enumerator **cudaMemHandleTypeWin32Kmt** = 0x4
Allows a Win32 KMT handle to be used for exporting.
(D3DKMT_HANDLE)

enumerator **cudaMemHandleTypeFabric** = 0x8
Allows a fabric handle to be used for exporting.
(*cudaMemFabricHandle_t*)

enum **cudaMemAllocationType**
Defines the allocation types available.
Values:

enumerator **cudaMemAllocationTypeInvalid** = 0x0

enumerator **cudaMemAllocationTypePinned** = 0x1
This allocation type is 'pinned', i.e.
cannot migrate from its current location while the application is actively using it

enumerator **cudaMemAllocationTypeManaged** = 0x2
This allocation type is managed memory.

enumerator **cudaMemAllocationTypeMax** = 0x7FFFFFFF

enum **cudaMemLocationType**

Specifies the type of location.

Values:

enumerator **cudaMemLocationTypeInvalid** = 0

enumerator **cudaMemLocationTypeNone** = 0

Location is unspecified.

This is used when creating a managed memory pool to indicate no preferred location for the pool

enumerator **cudaMemLocationTypeDevice** = 1

Location is a device location, thus id is a device ordinal.

enumerator **cudaMemLocationTypeHost** = 2

Location is host, id is ignored.

enumerator **cudaMemLocationTypeHostNuma** = 3

Location is a host NUMA node, thus id is a host NUMA node id.

enumerator **cudaMemLocationTypeHostNumaCurrent** = 4

Location is the host NUMA node closest to the current thread's CPU, id is ignored.

enumerator **cudaMemLocationTypeInvisible** = 5

Location is not visible but device is accessible, id is always cudaInvalidDeviceId.

enumerator **cudaMemLocationTypeDeviceLocalityDomain** = 6

Location is a portion of device memory, specified by the locality domain ID.

enum **cudaMemPoolAttr**

CUDA memory pool attributes.

Values:

enumerator **cudaMemPoolReuseFollowEventDependencies** = 0x1

(value type = int) Allow cuMemAllocAsync to use memory asynchronously freed in another streams as long as a stream ordering dependency of the allocating stream on the free action exists.

Cuda events and null stream interactions can create the required stream ordered dependencies. (default enabled)

enumerator **cudaMemPoolReuseAllowOpportunistic** = 0x2

(value type = int) Allow reuse of already completed frees when there is no dependency between the free and allocation.

(default enabled)

enumerator **cudaMemPoolReuseAllowInternalDependencies** = 0x3

(value type = int) Allow cuMemAllocAsync to insert new stream dependencies in order to establish the stream ordering required to reuse a piece of memory released by cuFreeAsync (default enabled).

enumerator **cudaMemPoolAttrReleaseThreshold** = 0x4

(value type = cuuint64_t) Amount of reserved memory in bytes to hold onto before trying to release memory back to the OS.

When more than the release threshold bytes of memory are held by the memory pool, the allocator will try to release memory back to the OS on the next call to stream, event or context synchronize. (default 0)

enumerator **cudaMemPoolAttrReservedMemCurrent** = 0x5

(value type = cuuint64_t) Amount of backing memory currently allocated for the mempool.

enumerator **cudaMemPoolAttrReservedMemHigh** = 0x6

(value type = cuuint64_t) High watermark of backing memory allocated for the mempool since the last time it was reset.

High watermark can only be reset to zero.

enumerator **cudaMemPoolAttrUsedMemCurrent** = 0x7

(value type = cuuint64_t) Amount of memory from the pool that is currently in use by the application.

enumerator **cudaMemPoolAttrUsedMemHigh** = 0x8

(value type = cuuint64_t) High watermark of the amount of memory from the pool that was in use by the application since the last time it was reset.

High watermark can only be reset to zero.

enumerator **cudaMemPoolAttrAllocationType** = 0x9

(value type = cudaMemAllocationType) The allocation type of the mempool

enumerator **cudaMemPoolAttrExportHandleTypes** = 0xA

(value type = cudaMemAllocationHandleType) Available export handle types for the mempool.

For imported pools this value is always cudaMemHandleTypeNone as an imported pool cannot be re-exported

enumerator **cudaMemPoolAttrLocationId** = 0xB

(value type = int) The location id for the mempool.

If the location type for this pool is cudaMemLocationTypeInvisible then ID will be cudaInvalidDeviceId

enumerator **cudaMemPoolAttrLocationType** = 0xC

(value type = cudaMemLocationType) The location type for the mempool.

For imported memory pools where the device is not directly visible to the importing process or pools imported via fabric handles across nodes this will be cudaMemLocationTypeInvisible

enumerator **cudaMemPoolAttrMaxPoolSize** = 0xD

(value type = cuuint64_t) Maximum size of the pool in bytes, this value may be higher than what was initially passed to cudaMemPoolCreate due to alignment requirements.

A value of 0 indicates no maximum size. For cudaMemAllocationTypeManaged and IPC imported pools this value will be system dependent.

enumerator **cudaMemPoolAttrHwDecompressEnabled** = 0xE

(value type = int) Indicates whether the pool has hardware compresssion enabled

enumerator **cudaMemPoolAttrLocalityDomainId** = 0xF

(value type = int) The locality domain id for the mempool, if the mempool is localized to a locality domain.

A value of -1 indicates that the mempool is not localized to a locality domain.

enum **cudaMemRangeAttribute**

CUDA range attributes.

Values:

enumerator **cudaMemRangeAttributeReadMostly** = 1

Whether the range will mostly be read and only occassionally be written to.

enumerator **cudaMemRangeAttributePreferredLocation** = 2

The preferred location of the range.

enumerator **cudaMemRangeAttributeAccessedBy** = 3

Memory range has *cudaMemAdviseSetAccessedBy* set for specified device.

enumerator **cudaMemRangeAttributeLastPrefetchLocation** = 4

The last location to which the range was prefetched.

enumerator **cudaMemRangeAttributePreferredLocationType** = 5

The preferred location type of the range.

enumerator **cudaMemRangeAttributePreferredLocationId** = 6

The preferred location id of the range.

enumerator **cudaMemRangeAttributeLastPrefetchLocationType** = 7

The last location type to which the range was prefetched.

enumerator **cudaMemRangeAttributeLastPrefetchLocationId** = 8

The last location id to which the range was prefetched.

enum **cudaMemcpy3DOperandType**

These flags allow applications to convey the operand type for individual copies specified in *cudaMemcpy3DBatchAsync*.

Values:

enumerator **cudaMemcpyOperandTypePointer** = 0x1

Memcpy operand is a valid pointer.

enumerator **cudaMemcpyOperandTypeArray** = 0x2

Memcpy operand is a CUarray.

enumerator **cudaMemcpyOperandTypeMax** = 0x7FFFFFFF

enum **cudaMemcpyFlags**

Flags to specify for copies within a batch.

For more details see *cudaMemcpyBatchAsync*.

Values:

enumerator **cudaMemcpyFlagDefault** = 0x0

enumerator **cudaMemcpyFlagPreferOverlapWithCompute** = 0x1

Hint to the driver to try and overlap the copy with compute work on the SMs.

enum **cudaMemcpyKind**

CUDA memory copy types.

Values:

enumerator **cudaMemcpyHostToHost** = 0

Host -> Host.

enumerator **cudaMemcpyHostToDevice** = 1

Host -> Device.

enumerator **cudaMemcpyDeviceToHost** = 2

Device -> Host.

enumerator **cudaMemcpyDeviceToDevice** = 3

Device -> Device.

enumerator **cudaMemcpyDefault** = 4

Direction of the transfer is inferred from the pointer values.

Requires unified virtual addressing

enum **cudaMemcpySrcAccessOrder**

Values:

enumerator **cudaMemcpySrcAccessOrderInvalid** = 0x0

Default invalid.

enumerator **cudaMemcpySrcAccessOrderStream** = 0x1

Indicates that access to the source pointer must be in stream order.

enumerator **cudaMemcpySrcAccessOrderDuringApiCall** = 0x2

Indicates that access to the source pointer can be out of stream order and all accesses must be complete before the API call returns.

This flag is suited for ephemeral sources (ex., stack variables) when it's known that no prior operations in the stream can be accessing the memory and also that the lifetime of the memory is limited to the scope that the source variable was declared in. Specifying this flag allows the driver to optimize the copy and removes the need for the user to synchronize the stream after the API call.

enumerator **cudaMemcpySrcAccessOrderAny** = 0x3

Indicates that access to the source pointer can be out of stream order and the accesses can happen even after the API call returns.

This flag is suited for host pointers allocated outside CUDA (ex., via malloc) when it's known that no prior operations in the stream can be accessing the memory. Specifying this flag allows the driver to optimize the copy on certain platforms.

enumerator **cudaMemcpySrcAccessOrderMax** = 0x7FFFFFFF

enum **cudaMemoryAdvise**

CUDA Memory Advise values.

Values:

enumerator **cudaMemAdviseSetReadMostly** = 1

Data will mostly be read and only occassionally be written to.

enumerator **cudaMemAdviseUnsetReadMostly** = 2

Undo the effect of *cudaMemAdviseSetReadMostly*.

enumerator **cudaMemAdviseSetPreferredLocation** = 3

Set the preferred location for the data as the specified device.

enumerator **cudaMemAdviseUnsetPreferredLocation** = 4

Clear the preferred location for the data.

enumerator **cudaMemAdviseSetAccessedBy** = 5

Data will be accessed by the specified device, so prevent page faults as much as possible.

enumerator **cudaMemAdviseUnsetAccessedBy** = 6

Let the Unified Memory subsystem decide on the page faulting policy for the specified device.

enum **cudaMemoryType**

CUDA memory types.

Values:

enumerator **cudaMemoryTypeUnregistered** = 0

Unregistered memory.

enumerator **cudaMemoryTypeHost** = 1

Host memory.

enumerator **cudaMemoryTypeDevice** = 2

Device memory.

enumerator **cudaMemoryTypeManaged** = 3

Managed memory.

enum **cudaResourceType**

CUDA resource types.

Values:

enumerator **cudaResourceTypeArray** = 0x00

Array resource.

enumerator **cudaResourceTypeMipmappedArray** = 0x01

Mipmapped array resource.

enumerator **cudaResourceTypeLinear** = 0x02

Linear resource.

enumerator **cudaResourceTypePitch2D** = 0x03

Pitch 2D resource.

enum **cudaResourceViewFormat**

CUDA texture resource view formats.

Values:

enumerator **cudaResViewFormatNone** = 0x00
No resource view format (use underlying resource format)

enumerator **cudaResViewFormatUnsignedChar1** = 0x01
1 channel unsigned 8-bit integers

enumerator **cudaResViewFormatUnsignedChar2** = 0x02
2 channel unsigned 8-bit integers

enumerator **cudaResViewFormatUnsignedChar4** = 0x03
4 channel unsigned 8-bit integers

enumerator **cudaResViewFormatSignedChar1** = 0x04
1 channel signed 8-bit integers

enumerator **cudaResViewFormatSignedChar2** = 0x05
2 channel signed 8-bit integers

enumerator **cudaResViewFormatSignedChar4** = 0x06
4 channel signed 8-bit integers

enumerator **cudaResViewFormatUnsignedShort1** = 0x07
1 channel unsigned 16-bit integers

enumerator **cudaResViewFormatUnsignedShort2** = 0x08
2 channel unsigned 16-bit integers

enumerator **cudaResViewFormatUnsignedShort4** = 0x09
4 channel unsigned 16-bit integers

enumerator **cudaResViewFormatSignedShort1** = 0x0a
1 channel signed 16-bit integers

enumerator **cudaResViewFormatSignedShort2** = 0x0b
2 channel signed 16-bit integers

enumerator **cudaResViewFormatSignedShort4** = 0x0c
4 channel signed 16-bit integers

enumerator **cudaResViewFormatUnsignedInt1** = 0x0d
1 channel unsigned 32-bit integers

enumerator **cudaResViewFormatUnsignedInt2** = 0x0e
2 channel unsigned 32-bit integers

enumerator **cudaResViewFormatUnsignedInt4** = 0x0f
4 channel unsigned 32-bit integers

enumerator **cudaResViewFormatSignedInt1** = 0x10
1 channel signed 32-bit integers

enumerator **cudaResViewFormatSignedInt2** = 0x11
2 channel signed 32-bit integers

enumerator **cudaResViewFormatSignedInt4** = 0x12
4 channel signed 32-bit integers

enumerator **cudaResViewFormatHalf1** = 0x13
1 channel 16-bit floating point

enumerator **cudaResViewFormatHalf2** = 0x14
2 channel 16-bit floating point

enumerator **cudaResViewFormatHalf4** = 0x15
4 channel 16-bit floating point

enumerator **cudaResViewFormatFloat1** = 0x16
1 channel 32-bit floating point

enumerator **cudaResViewFormatFloat2** = 0x17
2 channel 32-bit floating point

enumerator **cudaResViewFormatFloat4** = 0x18
4 channel 32-bit floating point

enumerator **cudaResViewFormatUnsignedBlockCompressed1** = 0x19
Block compressed 1.

enumerator **cudaResViewFormatUnsignedBlockCompressed2** = 0x1a
Block compressed 2.

enumerator **cudaResViewFormatUnsignedBlockCompressed3** = 0x1b
Block compressed 3.

enumerator **cudaResViewFormatUnsignedBlockCompressed4** = 0x1c
Block compressed 4 unsigned.

enumerator **cudaResViewFormatSignedBlockCompressed4** = 0x1d
Block compressed 4 signed.

enumerator **cudaResViewFormatUnsignedBlockCompressed5** = 0x1e
Block compressed 5 unsigned.

enumerator **cudaResViewFormatSignedBlockCompressed5** = 0x1f
Block compressed 5 signed.

enumerator **cudaResViewFormatUnsignedBlockCompressed6H** = 0x20
Block compressed 6 unsigned half-float.

enumerator **cudaResViewFormatSignedBlockCompressed6H** = 0x21
Block compressed 6 signed half-float.

enumerator **cudaResViewFormatUnsignedBlockCompressed7** = 0x22
Block compressed 7.

enum **cudaSharedCarveout**
Shared memory carveout configurations.
These may be passed to cudaFuncSetAttribute
Values:

enumerator **cudaSharedmemCarveoutDefault** = -1
No preference for shared memory or L1 (default)

enumerator **cudaSharedmemCarveoutMaxShared** = 100
Prefer maximum available shared memory, minimum L1 cache.

enumerator **cudaSharedmemCarveoutMaxL1** = 0
Prefer maximum available L1 cache, minimum shared memory.

enum **cudaSharedMemConfig**
CUDA shared memory configuration.

Deprecated:

Values:

enumerator **cudaSharedMemBankSizeDefault** = 0

enumerator **cudaSharedMemBankSizeFourByte** = 1

enumerator **cudaSharedMemBankSizeEightByte** = 2

enum **cudaSharedMemoryMode**

Shared memory related attributes for use with ::cuLaunchKernelEx.

Values:

enumerator **cudaSharedMemoryModeDefault** = 0

The default to use for allowing non-portable shared memory size on launch - uses current function attributes for *cudaFuncAttributeMaxDynamicSharedMemorySize*.

enumerator **cudaSharedMemoryModeRequirePortable** = 1

Specifies that the shared memory size requested must be a portable size within *cudaDevAttrMaxSharedMemoryPerBlock*.

enumerator **cudaSharedMemoryModeAllowNonPortable** = 2

Specifies that the shared memory size requested may be a non-portable size up to *cudaDevAttrMaxSharedMemoryPerBlockOptin*.

enumerator **cudaSharedMemoryModeAllowOversizedSharedMemory** = 3

Specifies that oversized shared memory configurations may be used (with the limitation of only 8kB L1 cache)

enumerator **cudaSharedMemoryModePreferOversizedSharedMemory** = 4

Specifies that oversized shared memory configurations may be used (with the limitation of only 8kB L1 cache), and prefer an oversized shared memory configuration.

enum **cudaStreamCaptureMode**

Possible modes for stream capture thread interactions.

For more details see *cudaStreamBeginCapture* and *cudaThreadExchangeStreamCaptureMode*

Values:

enumerator **cudaStreamCaptureModeGlobal** = 0

enumerator **cudaStreamCaptureModeThreadLocal** = 1

enumerator **cudaStreamCaptureModeRelaxed** = 2

enum **cudaStreamCaptureStatus**

Possible stream capture statuses returned by *cudaStreamIsCapturing*.

Values:

enumerator **cudaStreamCaptureStatusNone** = 0

Stream is not capturing.

enumerator **cudaStreamCaptureStatusActive** = 1

Stream is actively capturing.

enumerator **cudaStreamCaptureStatusInvalidated** = 2
Stream is part of a capture sequence that has been invalidated, but not terminated.

enum **cudaStreamUpdateCaptureDependenciesFlags**
Flags for *cudaStreamUpdateCaptureDependencies*.
Values:

enumerator **cudaStreamAddCaptureDependencies** = 0x0
Add new nodes to the dependency set.

enumerator **cudaStreamSetCaptureDependencies** = 0x1
Replace the dependency set with the new nodes.

enum **cudaSurfaceBoundaryMode**
CUDA Surface boundary modes.
Values:

enumerator **cudaBoundaryModeZero** = 0
Zero boundary mode.

enumerator **cudaBoundaryModeClamp** = 1
Clamp boundary mode.

enumerator **cudaBoundaryModeTrap** = 2
Trap boundary mode.

enum **cudaSurfaceFormatMode**
CUDA Surface format modes.
Values:

enumerator **cudaFormatModeForced** = 0
Forced format mode.

enumerator **cudaFormatModeAuto** = 1
Auto format mode.

enum **cudaSynchronizationPolicy**
Values:

enumerator **cudaSyncPolicyAuto** = 1

enumerator **cudaSyncPolicySpin** = 2

enumerator **cudaSyncPolicyYield** = 3

enumerator **cudaSyncPolicyBlockingSync** = 4

enum **cudaTextureAddressMode**

CUDA texture address modes.

Values:

enumerator **cudaAddressModeWrap** = 0

Wrapping address mode.

enumerator **cudaAddressModeClamp** = 1

Clamp to edge address mode.

enumerator **cudaAddressModeMirror** = 2

Mirror address mode.

enumerator **cudaAddressModeBorder** = 3

Border address mode.

enum **cudaTextureFilterMode**

CUDA texture filter modes.

Values:

enumerator **cudaFilterModePoint** = 0

Point filter mode.

enumerator **cudaFilterModeLinear** = 1

Linear filter mode.

enum **cudaTextureReadMode**

CUDA texture read modes.

Values:

enumerator **cudaReadModeElementType** = 0

Read texture as specified element type.

enumerator **cudaReadModeNormalizedFloat** = 1

Read texture as normalized float.

enum **cudaUserObjectFlags**

Flags for user objects for graphs.

Values:

enumerator **cudaUserObjectNoDestructorSync** = 0x1

Indicates the destructor execution is not synchronized by any CUDA handle.

enum **cudaUserObjectRetainFlags**

Flags for retaining user object references for graphs.

Values:

enumerator **cudaGraphUserObjectMove** = 0x1

Transfer references from the caller rather than creating new references.

6.2.3. Typedefs

typedef const struct cudaArray ***cudaArray_const_t**

CUDA array (as source copy argument)

typedef struct cudaArray ***cudaArray_t**

CUDA array.

typedef void (***cudaAsyncCallback**)(*cudaAsyncNotificationInfo_t**, void*, *cudaAsyncCallbackHandle_t*)

typedef struct cudaAsyncCallbackEntry ***cudaAsyncCallbackHandle_t**

CUDA async callback handle.

typedef struct CUdevResourceDesc_st ***cudaDevResourceDesc_t**

An opaque descriptor handle.

The descriptor encapsulates multiple created and configured resources. Created via ::cudaDeviceResourceGenerateDesc

typedef struct CUeglStreamConnection_st ***cudaEglStreamConnection**

CUDA EGLSream Connection.

typedef enum *cudaError* **cudaError_t**

CUDA Error types.

typedef struct CUevent_st ***cudaEvent_t**

CUDA event types.

typedef struct cudaExecutionContext_st ***cudaExecutionContext_t**

An opaque handle to a CUDA execution context.

It represents an execution context created via CUDA Runtime APIs such as *cudaGreenCtxCreate*.

typedef struct CUexternalMemory_st ***cudaExternalMemory_t**

CUDA external memory.

typedef struct CUexternalSemaphore_st ***cudaExternalSemaphore_t**

CUDA external semaphore.

typedef struct CUfunc_st ***cudaFunction_t**

CUDA function.

typedef unsigned long long **cudaGraphConditionalHandle**

CUDA handle for conditional graph nodes.

typedef struct CUgraphDeviceUpdatableNode_st ***cudaGraphDeviceNode_t**

CUDA device node handle for device-side node update.

typedef struct CUgraphExec_st ***cudaGraphExec_t**

CUDA executable (launchable) graph.

typedef struct CUgraphNode_st ***cudaGraphNode_t**

CUDA graph node.

typedef struct CUgraph_st ***cudaGraph_t**

CUDA graph.

typedef struct cudaGraphicsResource ***cudaGraphicsResource_t**

CUDA graphics resource types.

void(__stdcall * cudaHostFn_t)(void *userData)

CUDA host function.

Param userData
Argument value passed to the function

typedef struct CUkern_st ***cudaKernel_t**

CUDA kernel.

typedef struct CUlib_st ***cudaLibrary_t**

CUDA library.

typedef unsigned int **cudaLogIterator**

typedef struct CUlogsCallbackEntry_st ***cudaLogsCallbackHandle**

typedef struct CUmemPoolHandle_st ***cudaMemPool_t**

CUDA memory pool.

typedef const struct cudaMipmappedArray ***cudaMipmappedArray_const_t**

CUDA mipmapped array (as source argument)

typedef struct cudaMipmappedArray ***cudaMipmappedArray_t**

CUDA mipmapped array.

typedef struct CUstream_st ***cudaStream_t**

CUDA stream.

typedef unsigned long long **cudaSurfaceObject_t**

An opaque value that represents a CUDA Surface object.

typedef unsigned long long **cudaTextureObject_t**

An opaque value that represents a CUDA texture object.

typedef struct CUuserObject_st ***cudaUserObject_t**

CUDA user object for graphs.

6.3. Device Management

impl_private

This section describes the device management functions of the CUDA runtime application programming interface.

Functions

__host__ cudaError_t *cudaChooseDevice*(int *device, const struct cudaDeviceProp *prop)
Select compute-device which best matches criteria.

__host__ cudaError_t *cudaDeviceFlushGPUDirectRDMAWrites*(enum cudaFlushGPUDirectRDMAWritesTarget target, enum cudaFlushGPUDirectRDMAWritesScope scope)
Blocks until remote writes are visible to the specified scope.

__host__ cudaError_t *cudaDeviceGetAttribute*(int *value, enum cudaDeviceAttr attr, int device)
Returns information about the device.

__host__ cudaError_t *cudaDeviceGetByPCIBusId*(int *device, const char *pciBusId)
Returns a handle to a compute device.

__host__ cudaError_t *cudaDeviceGetCacheConfig*(enum cudaFuncCache *pCacheConfig)
Returns the preferred cache configuration for the current device.

__host__ cudaError_t *cudaDeviceGetDefaultMemPool*(cudaMemPool_t *memPool, int device)
Returns the default mempool of a device.

__host__ cudaError_t *cudaDeviceGetHostAtomicCapabilities*(unsigned int *capabilities, const enum cudaAtomicOperation *operations, unsigned int count, int device)
Queries details about atomic operations supported between the device and host.

__host__ cudaError_t *cudaDeviceGetLimit*(size_t *pValue, enum cudaLimit limit)
Return resource limits.

__host__ cudaError_t *cudaDeviceGetMemPool*(cudaMemPool_t *memPool, int device)
Gets the current mempool for a device.

__host__ cudaError_t *cudaDeviceGetNvSciSyncAttributes*(void *nvSciSyncAttrList, int device, int flags)
Return NvSciSync attributes that this device can support.

__host__ cudaError_t *cudaDeviceGetP2PAtomicCapabilities*(unsigned int *capabilities, const enum cudaAtomicOperation *operations, unsigned int count, int srcDevice, int dstDevice)
Queries details about atomic operations supported between two devices.

__host__ cudaError_t *cudaDeviceGetP2PAttribute*(int *value, enum cudaDeviceP2PAttr attr, int srcDevice, int dstDevice)
Queries attributes of the link between two devices.

__host__ cudaError_t *cudaDeviceGetPCIBusId*(char *pciBusId, int len, int device)
Returns a PCI Bus Id string for the device.

__host__ cudaError_t *cudaDeviceGetStreamPriorityRange*(int *leastPriority, int *greatestPriority)
Returns numerical values that correspond to the least and greatest stream priorities.

__host__ cudaError_t *cudaDeviceGetTexture1DLinearMaxWidth*(size_t *maxWidthInElements, const struct cudaChannelFormatDesc *fmtDesc, int device)
Returns the maximum number of elements allocatable in a 1D linear texture for a given element size.

__host__ cudaError_t *cudaDeviceRegisterAsyncNotification*(int device, cudaAsyncCallback callbackFunc, void *userData, cudaAsyncCallbackHandle_t *callback)
Registers a callback function to receive async notifications.

__host__ cudaError_t *cudaDeviceReset*(void)
Destroy all allocations and reset all state on the current device in the current process.

__host__ cudaError_t *cudaDeviceSetCacheConfig*(enum cudaFuncCache cacheConfig)
Sets the preferred cache configuration for the current device.

__host__ cudaError_t *cudaDeviceSetLimit*(enum cudaLimit limit, size_t value)
Set resource limits.

__host__ cudaError_t *cudaDeviceSetMemPool*(int device, cudaMemPool_t memPool)
Sets the current memory pool of a device.

__host__ cudaError_t *cudaDeviceSynchronize*(void)
Wait for compute device to finish.

__host__ cudaError_t *cudaDeviceUnregisterAsyncNotification*(int device, cudaAsyncCallbackHandle_t callback)
Unregisters an async notification callback.

__host__ cudaError_t *cudaGetDevice*(int *device)
Returns which device is currently being used.

__host__ cudaError_t *cudaGetDeviceCount*(int *count)
Returns the number of compute-capable devices.

__host__ cudaError_t *cudaGetDeviceFlags*(unsigned int *flags)
Gets the flags for the current device.

__host__ cudaError_t *cudaGetDeviceProperties*(struct cudaDeviceProp *prop, int device)
Returns information about the compute-device.

__host__ cudaError_t *cudaInitDevice*(int device, unsigned int deviceFlags, unsigned int flags)
Initialize device to be used for GPU executions.

__host__ cudaError_t *cudaIpcCloseMemHandle*(void *devPtr)
Attempts to close memory mapped with cudaIpcOpenMemHandle.

__host__ cudaError_t *cudaIpcGetEventHandle*(cudaIpcEventHandle_t *handle, cudaEvent_t event)
Gets an interprocess handle for a previously allocated event.

__host__ cudaError_t *cudaIpcGetMemHandle*(cudaIpcMemHandle_t *handle, void *devPtr)
Gets an interprocess memory handle for an existing device memory allocation.

__host__ cudaError_t *cudaIpcOpenEventHandle*(cudaEvent_t *event, cudaIpcEventHandle_t handle)
Opens an interprocess event handle for use in the current process.

__host__ cudaError_t *cudaIpcOpenMemHandle*(void **devPtr, cudaIpcMemHandle_t handle, unsigned int flags)
Opens an interprocess memory handle exported from another process and returns a device pointer usable in the local process.

__host__ cudaError_t *cudaSetDevice*(int device)
Set device to be used for GPU executions.

__host__ cudaError_t *cudaSetDeviceFlags*(unsigned int flags)
Sets flags to be used for device executions.

__host__ cudaError_t *cudaSetValidDevices*(int *device_arr, int len)
Set a list of devices that can be used for CUDA.

6.3.1. Functions

__host__ *cudaError_t* **cudaChooseDevice**(int *device, const struct *cudaDeviceProp* *prop)
Select compute-device which best matches criteria.

Returns in `*device` the device which has properties that best match `*prop`.

See also:

cudaGetDeviceCount, *cudaGetDevice*, *cudaSetDevice*, *cudaGetDeviceProperties*, *cudaInitDevice*

Note: Note that this function may also return error codes from previous, asynchronous launches.

Note: Note that this function may also return *cudaErrorInitializationError*, *cudaErrorInsufficientDriver* or *cudaErrorNoDevice* if this call tries to initialize internal CUDA RT state.

Note: Note that as specified by *cudaStreamAddCallback* no CUDA function may be called from callback. *cudaErrorNotPermitted* may, but is not guaranteed to, be returned as a diagnostic in such case.

Parameters

- **device** – - Device with best match
- **prop** – - Desired device properties

Returns
cudaSuccess, *cudaErrorInvalidValue*

__host__ *cudaError_t* **cudaDeviceFlushGPUDirectRDMAWrites**(enum *cudaFlushGPUDirectRDMAWritesTarget* target, enum *cudaFlushGPUDirectRDMAWritesScope* scope)

Blocks until remote writes are visible to the specified scope.

Blocks until remote writes to the target context via mappings created through GPUDirect RDMA APIs, like nvidia_p2p_get_pages (see https://docs.nvidia.com/cuda/gpudirect-rdma for more information), are visible to the specified scope.

If the scope equals or lies within the scope indicated by *cudaDevAttrGPUDirectRDMAWritesOrdering*, the call will be a no-op and can be safely omitted for performance. This can be determined by comparing the numerical values between the two enums, with smaller scopes having smaller values.

Users may query support for this API via *cudaDevAttrGPUDirectRDMAFlushWritesOptions*.

See also:

::cuFlushGPUDirectRDMAWrites

Note: Note that this function may also return error codes from previous, asynchronous launches.

Note: Note that this function may also return *cudaErrorInitializationError*, *cudaErrorInsufficientDriver* or *cudaErrorNoDevice* if this call tries to initialize internal CUDA RT state.

Note: Note that as specified by *cudaStreamAddCallback* no CUDA function may be called from callback. *cudaErrorNotPermitted* may, but is not guaranteed to, be returned as a diagnostic in such case.

Parameters

- **target** – - The target of the operation, see cudaFlushGPUDirectRDMAWritesTarget
- **scope** – - The scope of the operation, see cudaFlushGPUDirectRDMAWritesScope

Returns

cudaSuccess, *cudaErrorNotSupported*,

__host__ *cudaError_t* **cudaDeviceGetAttribute**(int *value, enum *cudaDeviceAttr* attr, int device)

Returns information about the device.

Returns in `*value` the integer value of the attribute `attr` on device `device`.

See also:

cudaGetDeviceCount, *cudaGetDevice*, *cudaSetDevice*, *cudaChooseDevice*, *cudaGetDeviceProperties*, *cudaInitDevice*, ::cuDeviceGetAttribute

Note: Note that this function may also return error codes from previous, asynchronous launches.

Note: Note that this function may also return *cudaErrorInitializationError*, *cudaErrorInsufficientDriver* or *cudaErrorNoDevice* if this call tries to initialize internal CUDA RT state.

Note: Note that as specified by *cudaStreamAddCallback* no CUDA function may be called from callback. *cudaErrorNotPermitted* may, but is not guaranteed to, be returned as a diagnostic in such case.

Parameters

- **value** – - Returned device attribute value
- **attr** – - Device attribute to query
- **device** – - Device number to query

Returns

cudaSuccess, *cudaErrorInvalidDevice*, *cudaErrorInvalidValue*

__host__ *cudaError_t* **cudaDeviceGetByPCIBusId**(int *device, const char *pciBusId)

Returns a handle to a compute device.

Returns in `*device` a device ordinal given a PCI bus ID string.

where `domain`, `bus`, `device`, and `function` are all hexadecimal values

See also:

cudaDeviceGetPCIBusId, ::cuDeviceGetByPCIBusId

Note: Note that this function may also return error codes from previous, asynchronous launches.

Note: Note that this function may also return *cudaErrorInitializationError*, *cudaErrorInsufficientDriver* or *cudaErrorNoDevice* if this call tries to initialize internal CUDA RT state.

Note: Note that as specified by *cudaStreamAddCallback* no CUDA function may be called from callback. *cudaErrorNotPermitted* may, but is not guaranteed to, be returned as a diagnostic in such case.

Parameters

- **device** – - Returned device ordinal

- **pciBusId** – - String in one of the following forms:

Returns

cudaSuccess, *cudaErrorInvalidValue*, *cudaErrorInvalidDevice*

__host__ *cudaError_t* **cudaDeviceGetCacheConfig**(enum *cudaFuncCache* *pCacheConfig)

Returns the preferred cache configuration for the current device.

On devices where the L1 cache and shared memory use the same hardware resources, this returns through `pCacheConfig` the preferred cache configuration for the current device. This is only a preference. The runtime will use the requested configuration if possible, but it is free to choose a different configuration if required to execute functions.

This will return a `pCacheConfig` of *cudaFuncCachePreferNone* on devices where the size of the L1 cache and shared memory are fixed.

The supported cache configurations are:

- *cudaFuncCachePreferNone*: no preference for shared memory or L1 (default)
- *cudaFuncCachePreferShared*: prefer larger shared memory and smaller L1 cache
- *cudaFuncCachePreferL1*: prefer larger L1 cache and smaller shared memory
- *cudaFuncCachePreferEqual*: prefer equal size L1 cache and shared memory

See also:

cudaDeviceSetCacheConfig, *cudaFuncSetCacheConfig (C API)*, *cudaFuncSetCacheConfig (C++ API)*, ::cuCtxGetCacheConfig

---

Note: Note that this function may also return error codes from previous, asynchronous launches.

Note: Note that this function may also return *cudaErrorInitializationError*, *cudaErrorInsufficientDriver* or *cudaErrorNoDevice* if this call tries to initialize internal CUDA RT state.

Note: Note that as specified by *cudaStreamAddCallback* no CUDA function may be called from callback. *cudaErrorNotPermitted* may, but is not guaranteed to, be returned as a diagnostic in such case.

Parameters

pCacheConfig – - Returned cache configuration

Returns

cudaSuccess

__host__ *cudaError_t* **cudaDeviceGetDefaultMemPool**(*cudaMemPool_t* *memPool, int device)

Returns the default mempool of a device.

The default mempool of a device contains device memory from that device.

See also:

::cuDeviceGetDefaultMemPool, *cudaMallocAsync*, *cudaMemPoolTrimTo*, *cudaMemPoolGetAttribute*, *cudaDeviceSetMemPool*, *cudaMemPoolSetAttribute*, *cudaMemPoolSetAccess*

Note: Note that this function may also return error codes from previous, asynchronous launches.

Note: Note that this function may also return *cudaErrorInitializationError*, *cudaErrorInsufficientDriver* or *cudaErrorNoDevice* if this call tries to initialize internal CUDA RT state.

Note: Note that as specified by *cudaStreamAddCallback* no CUDA function may be called from callback. *cudaErrorNotPermitted* may, but is not guaranteed to, be returned as a diagnostic in such case.

Returns

cudaSuccess, *cudaErrorInvalidDevice*, *cudaErrorInvalidValue* *cudaErrorNotSupported*

__host__ *cudaError_t* **cudaDeviceGetHostAtomicCapabilities**(unsigned int *capabilities, const enum *cudaAtomicOperation* *operations, unsigned int count, int device)

Queries details about atomic operations supported between the device and host.

Returns in `*capabilities` the details about requested atomic `*operations` over the the link between `dev` and the host. The allocated size of `*operations` and `*capabilities` must be `count`.

For each *cudaAtomicOperation* in `*operations`, the corresponding result in `*capabilities` will be a bitmask indicating which of *cudaAtomicOperationCapability* the link supports natively.

Returns *cudaErrorInvalidDevice* if `dev` is not valid.

Returns *cudaErrorInvalidValue* if `*capabilities` or `*operations` is NULL, if `count` is 0, or if any of `*operations` is not valid.

See also:

cudaDeviceGetAttribute, *cudaDeviceGetP2PAtomicCapabilities*, ::cuDeviceGeHostAtomicCapabilities

---

Note: Note that this function may also return error codes from previous, asynchronous launches.

Parameters

- **capabilities** – - Returned capability details of each requested operation
- **operations** – - Requested operations
- **count** – - Count of requested operations and size of capabilities
- **dev** – - Device handle

Returns
cudaSuccess, *cudaErrorInvalidDevice*, *cudaErrorInvalidValue*

__host__ *cudaError_t* **cudaDeviceGetLimit**(size_t *pValue, enum *cudaLimit* limit)

Return resource limits.

Returns in `*pValue` the current size of `limit`. The following *cudaLimit* values are supported.

- *cudaLimitStackSize* is the stack size in bytes of each GPU thread.
- *cudaLimitPrintfFifoSize* is the size in bytes of the shared FIFO used by the ::printf() device system call.
- *cudaLimitMallocHeapSize* is the size in bytes of the heap used by the ::malloc() and ::free() device system calls.
- *cudaLimitDevRuntimeSyncDepth* is the maximum grid depth at which a thread can isssue the device runtime call *cudaDeviceSynchronize()* to wait on child grid launches to complete. This functionality is removed for devices of compute capability >= 9.0, and hence will return error *cudaErrorUnsupportedLimit* on such devices.
- *cudaLimitDevRuntimePendingLaunchCount* is the maximum number of outstanding device runtime launches.
- *cudaLimitMaxL2FetchGranularity* is the L2 cache fetch granularity.
- *cudaLimitPersistingL2CacheSize* is the persisting L2 cache size in bytes.

See also:

cudaDeviceSetLimit, ::cuCtxGetLimit

Note: Note that this function may also return error codes from previous, asynchronous launches.

Note: Note that this function may also return *cudaErrorInitializationError*, *cudaErrorInsufficientDriver* or *cudaErrorNoDevice* if this call tries to initialize internal CUDA RT state.

Note: Note that as specified by *cudaStreamAddCallback* no CUDA function may be called from callback. *cudaErrorNotPermitted* may, but is not guaranteed to, be returned as a diagnostic in such case.

Parameters

- **limit** – - Limit to query
- **pValue** – - Returned size of the limit

Returns
cudaSuccess, *cudaErrorUnsupportedLimit*, *cudaErrorInvalidValue*

__host__ *cudaError_t* **cudaDeviceGetMemPool**(*cudaMemPool_t* *memPool, int device)

Gets the current mempool for a device.

Returns the last pool provided to *cudaDeviceSetMemPool* for this device or the device's default memory pool if *cudaDeviceSetMemPool* has never been called. By default the current mempool is the default mempool for a device, otherwise the returned pool must have been set with ::cuDeviceSetMemPool or *cudaDeviceSetMemPool*.

See also:

::cuDeviceGetMemPool, *cudaDeviceGetDefaultMemPool*, *cudaDeviceSetMemPool*

Note: Note that this function may also return error codes from previous, asynchronous launches.

Note: Note that this function may also return *cudaErrorInitializationError*, *cudaErrorInsufficientDriver* or *cudaErrorNoDevice* if this call tries to initialize internal CUDA RT state.

Note: Note that as specified by *cudaStreamAddCallback* no CUDA function may be called from callback. *cudaErrorNotPermitted* may, but is not guaranteed to, be returned as a diagnostic in such case.

Returns

cudaSuccess, *cudaErrorInvalidValue* *cudaErrorNotSupported*

__host__ *cudaError_t* **cudaDeviceGetNvSciSyncAttributes**(void *nvSciSyncAttrList, int device, int flags)

Return NvSciSync attributes that this device can support.

Returns in `nvSciSyncAttrList`, the properties of NvSciSync that this CUDA device, `dev` can support. The returned `nvSciSyncAttrList` can be used to create an NvSciSync that matches this device's capabilities.

If NvSciSyncAttrKey_RequiredPerm field in `nvSciSyncAttrList` is already set this API will return *cudaErrorInvalidValue*.

The applications should set `nvSciSyncAttrList` to a valid NvSciSyncAttrList failing which this API will return ::cudaErrorInvalidHandle.

The `flags` controls how applications intends to use the NvSciSync created from the `nvSciSyncAttrList`. The valid flags are:

- *cudaNvSciSyncAttrSignal*, specifies that the applications intends to signal an NvSciSync on this CUDA device.
- *cudaNvSciSyncAttrWait*, specifies that the applications intends to wait on an NvSciSync on this CUDA device.

At least one of these flags must be set, failing which the API returns *cudaErrorInvalidValue*. Both the flags are orthogonal to one another: a developer may set both these flags that allows to set both wait and signal specific attributes in the same `nvSciSyncAttrList`.

Note that this API updates the input `nvSciSyncAttrList` with values equivalent to the following public attribute key-values: NvSciSyncAttrKey_RequiredPerm is set to

- NvSciSyncAccessPerm_SignalOnly if *cudaNvSciSyncAttrSignal* is set in `flags`.
- NvSciSyncAccessPerm_WaitOnly if *cudaNvSciSyncAttrWait* is set in `flags`.
- NvSciSyncAccessPerm_WaitSignal if both *cudaNvSciSyncAttrWait* and *cudaNvSciSyncAttrSignal* are set in `flags`. NvSciSyncAttrKey_PrimitiveInfo is set to
- NvSciSyncAttrValPrimitiveType_SysmemSemaphore on any valid `device`.
- NvSciSyncAttrValPrimitiveType_Syncpoint if `device` is a Tegra device.
- NvSciSyncAttrValPrimitiveType_SysmemSemaphorePayload64b if `device` is GA10X+. NvSciSyncAttrKey_GpuId is set to the same UUID that is returned in *cudaDeviceProp.uuid* from ::cudaDeviceGetProperties for this `device`.

cudaSuccess, *cudaErrorDeviceUninitialized*, *cudaErrorInvalidValue*, ::cudaErrorInvalidHandle, *cudaErrorInvalidDevice*, *cudaErrorNotSupported*, *cudaErrorMemoryAllocation*

See also:

cudaImportExternalSemaphore, *cudaDestroyExternalSemaphore*, *cudaSignalExternalSemaphoresAsync*, *cudaWaitExternalSemaphoresAsync*

Parameters

- **nvSciSyncAttrList** – - Return NvSciSync attributes supported.
- **device** – - Valid Cuda Device to get NvSciSync attributes for.
- **flags** – - flags describing NvSciSync usage.

Returns

__host__ *cudaError_t* **cudaDeviceGetP2PAtomicCapabilities**(unsigned int *capabilities, const enum *cudaAtomicOperation* *operations, unsigned int count, int srcDevice, int dstDevice)

Queries details about atomic operations supported between two devices.

Returns in `*capabilities` the details about requested atomic `*operations` over the the link between `srcDevice` and `dstDevice`. The allocated size of `*operations` and `*capabilities` must be `count`.

For each *cudaAtomicOperation* in `*operations`, the corresponding result in `*capabilities` will be a bitmask indicating which of *cudaAtomicOperationCapability* the link supports natively.

Returns *cudaErrorInvalidDevice* if `srcDevice` or `dstDevice` are not valid or if they represent the same device.

Returns *cudaErrorInvalidValue* if `*capabilities` or `*operations` is NULL, if `count` is 0, or if any of `*operations` is not valid.

See also:

cudaDeviceGetP2PAttribute, ::cuDeviceGetP2PAttribute, ::cuDeviceGetP2PAtomicCapabilities

Note: Note that this function may also return error codes from previous, asynchronous launches.

Parameters

- **capabilities** – - Returned capability details of each requested operation
- **operations** – - Requested operations
- **count** – - Count of requested operations and size of capabilities
- **srcDevice** – - The source device of the target link
- **dstDevice** – - The destination device of the target link

Returns

cudaSuccess, *cudaErrorInvalidDevice*, *cudaErrorInvalidValue*

__host__ *cudaError_t* **cudaDeviceGetP2PAttribute**(int *value, enum *cudaDeviceP2PAttr* attr, int srcDevice, int dstDevice)

Queries attributes of the link between two devices.

Returns in `*value` the value of the requested attribute `attrib` of the link between `srcDevice` and `dstDevice`. The supported attributes are:

- *cudaDevP2PAttrPerformanceRank*: A relative value indicating the performance of the link between two devices. Lower value means better performance (0 being the value used for most performant link).
- *cudaDevP2PAttrAccessSupported*: 1 if peer access is enabled.
- *cudaDevP2PAttrNativeAtomicSupported*: 1 if all native atomic operations over the link are supported.
- *cudaDevP2PAttrCudaArrayAccessSupported*: 1 if accessing CUDA arrays over the link is supported.
- *cudaDevP2PAttrOnlyPartialNativeAtomicSupported*: 1 if some CUDA-valid atomic operations over the link are supported. Information about specific operations can be retrieved with *cudaDeviceGetP2PAtomicCapabilities*.

Returns *cudaErrorInvalidDevice* if `srcDevice` or `dstDevice` are not valid or if they represent the same device.

Returns *cudaErrorInvalidValue* if `attrib` is not valid or if `value` is a null pointer.

See also:

cudaDeviceEnablePeerAccess, *cudaDeviceDisablePeerAccess*, *cudaDeviceCanAccessPeer*, ::cuDeviceGetP2PAttribute *cudaDeviceGetP2PAtomicCapabilities*

Note: Note that this function may also return error codes from previous, asynchronous launches.

Note: Note that this function may also return *cudaErrorInitializationError*, *cudaErrorInsufficientDriver* or *cudaErrorNoDevice* if this call tries to initialize internal CUDA RT state.

Note: Note that as specified by *cudaStreamAddCallback* no CUDA function may be called from callback. *cudaErrorNotPermitted* may, but is not guaranteed to, be returned as a diagnostic in such case.

Parameters

- **value** – - Returned value of the requested attribute
- **attrib** – - The requested attribute of the link between `srcDevice` and `dstDevice`.
- **srcDevice** – - The source device of the target link.
- **dstDevice** – - The destination device of the target link.

Returns

cudaSuccess, *cudaErrorInvalidDevice*, *cudaErrorInvalidValue*

__host__ *cudaError_t* **cudaDeviceGetPCIBusId**(char *pciBusId, int len, int device)

Returns a PCI Bus Id string for the device.

Returns an ASCII string identifying the device `dev` in the NULL-terminated string pointed to by `pciBusId`. `len` specifies the maximum length of the string that may be returned.

where `domain`, `bus`, `device`, and `function` are all hexadecimal values. pciBusId should be large enough to store 13 characters including the NULL-terminator.

See also:

cudaDeviceGetByPCIBusId, ::cuDeviceGetPCIBusId

Note: Note that this function may also return error codes from previous, asynchronous launches.

Note: Note that this function may also return *cudaErrorInitializationError*, *cudaErrorInsufficientDriver* or *cudaErrorNoDevice* if this call tries to initialize internal CUDA RT state.

Note: Note that as specified by *cudaStreamAddCallback* no CUDA function may be called from callback. *cudaErrorNotPermitted* may, but is not guaranteed to, be returned as a diagnostic in such case.

Parameters

- **pciBusId** – - Returned identifier string for the device in the following format
- **len** – - Maximum length of string to store in `name`
- **device** – - Device to get identifier string for

Returns

cudaSuccess, *cudaErrorInvalidValue*, *cudaErrorInvalidDevice*

__host__ *cudaError_t* **cudaDeviceGetStreamPriorityRange**(int *leastPriority, int *greatestPriority)

Returns numerical values that correspond to the least and greatest stream priorities.

Returns in `*leastPriority` and `*greatestPriority` the numerical values that correspond to the least and greatest stream priorities respectively. Stream priorities follow a convention where lower numbers imply greater priorities. The range of meaningful stream priorities is given by [`*greatestPriority`, `*leastPriority`]. If the user attempts to create a stream with a priority value that is outside the the meaningful range as specified by this API, the priority is automatically clamped down or up to either `*leastPriority` or `*greatestPriority` respectively. See *cudaStreamCreateWithPriority* for details on creating a priority stream. A NULL may be passed in for `*leastPriority` or `*greatestPriority` if the value is not desired.

This function will return '0' in both `*leastPriority` and `*greatestPriority` if the current context's device does not support stream priorities (see *cudaDeviceGetAttribute*).

See also:

cudaStreamCreateWithPriority, *cudaStreamGetPriority*, ::cuCtxGetStreamPriorityRange

---

Note: Note that this function may also return error codes from previous, asynchronous launches.

Note: Note that this function may also return *cudaErrorInitializationError*, *cudaErrorInsufficientDriver* or *cudaErrorNoDevice* if this call tries to initialize internal CUDA RT state.

Note: Note that as specified by *cudaStreamAddCallback* no CUDA function may be called from callback. *cudaErrorNotPermitted* may, but is not guaranteed to, be returned as a diagnostic in such case.

Parameters

- **leastPriority** – - Pointer to an int in which the numerical value for least stream priority is returned
- **greatestPriority** – - Pointer to an int in which the numerical value for greatest stream priority is returned

Returns

cudaSuccess

__host__ *cudaError_t* **cudaDeviceGetTexture1DLinearMaxWidth**(size_t *maxWidthInElements, const struct *cudaChannelFormatDesc* *fmtDesc, int device)

Returns the maximum number of elements allocatable in a 1D linear texture for a given element size.

Returns in `maxWidthInElements` the maximum number of elements allocatable in a 1D linear texture for given format descriptor `fmtDesc`.

See also:

::cuDeviceGetTexture1DLinearMaxWidth

Note: Note that this function may also return error codes from previous, asynchronous launches.

Note: Note that this function may also return *cudaErrorInitializationError*, *cudaErrorInsufficientDriver* or *cudaErrorNoDevice* if this call tries to initialize internal CUDA RT state.

Note: Note that as specified by *cudaStreamAddCallback* no CUDA function may be called from callback. *cudaErrorNotPermitted* may, but is not guaranteed to, be returned as a diagnostic in such case.

Parameters

- **`maxWidthInElements`** – - Returns maximum number of texture elements allocatable for given `fmtDesc`.
- **`fmtDesc`** – - Texture format description.

Returns

cudaSuccess, *cudaErrorUnsupportedLimit*, *cudaErrorInvalidValue*

__host__ *cudaError_t* **`cudaDeviceRegisterAsyncNotification`**(int device, *cudaAsyncCallback* callbackFunc, void *userData, *cudaAsyncCallbackHandle_t* *callback)

Registers a callback function to receive async notifications.

Registers `callbackFunc` to receive async notifications.

The `userData` parameter is passed to the callback function at async notification time. Likewise, `callback` is also passed to the callback function to distinguish between multiple registered callbacks.

The callback function being registered should be designed to return quickly (~10ms). Any long running tasks should be queued for execution on an application thread.

Callbacks may not call cudaDeviceRegisterAsyncNotification or cudaDeviceUnregisterAsyncNotification. Doing so will result in *cudaErrorNotPermitted*. Async notification callbacks execute in an undefined order and may be serialized.

Returns in `*callback` a handle representing the registered callback instance.

See also:

cudaDeviceUnregisterAsyncNotification

Note: Note that this function may also return error codes from previous, asynchronous launches.

Parameters

- **device** – - The device on which to register the callback
- **callbackFunc** – - The function to register as a callback
- **userData** – - A generic pointer to user data. This is passed into the callback function.
- **callback** – - A handle representing the registered callback instance

Returns

cudaSuccess *cudaErrorNotSupported* *cudaErrorInvalidDevice* *cudaErrorInvalidValue* *cudaErrorNotPermitted* *cudaErrorUnknown*

__host__ *cudaError_t* **cudaDeviceReset**(void)

Destroy all allocations and reset all state on the current device in the current process.

Explicitly destroys and cleans up all resources associated with the current device in the current process. It is the caller's responsibility to ensure that the resources are not accessed or passed in subsequent API calls and doing so will result in undefined behavior. These resources include CUDA types *cudaStream_t*, *cudaEvent_t*, *cudaArray_t*, *cudaMipmappedArray_t*, *cudaPitchedPtr*, *cudaTextureObject_t*, *cudaSurfaceObject_t*, ::textureReference, ::surfaceReference, *cudaExternalMemory_t*, *cudaExternalSemaphore_t* and *cudaGraphicsResource_t*. These resources also include memory allocations by *cudaMalloc*, *cudaMallocHost*, *cudaMallocManaged* and *cudaMallocPitch*. Any subsequent API call to this device will reinitialize the device.

Note that this function will reset the device immediately. It is the caller's responsibility to ensure that the device is not being accessed by any other host threads from the process when this function is called.

See also:

cudaDeviceSynchronize

Note: *cudaDeviceReset()* will not destroy memory allocations by *cudaMallocAsync()* and *cudaMallocFromPoolAsync()*. These memory allocations need to be destroyed explicitly.

Note: If a non-primary ::CUcontext is current to the thread, *cudaDeviceReset()* will destroy only the internal CUDA RT state for that ::CUcontext.

Note: Note that this function may also return error codes from previous, asynchronous launches.

Note: Note that this function may also return *cudaErrorInitializationError*, *cudaErrorInsufficientDriver* or *cudaErrorNoDevice* if this call tries to initialize internal CUDA RT state.

Note: Note that as specified by *cudaStreamAddCallback* no CUDA function may be called from callback. *cudaErrorNotPermitted* may, but is not guaranteed to, be returned as a diagnostic in such case.

Returns

cudaSuccess

__host__ *cudaError_t* **cudaDeviceSetCacheConfig**(enum *cudaFuncCache* cacheConfig)

Sets the preferred cache configuration for the current device.

On devices where the L1 cache and shared memory use the same hardware resources, this sets through `cacheConfig` the preferred cache configuration for the current device. This is only a preference. The runtime will use the requested configuration if possible, but it is free to choose a different configuration if required to execute the function. Any function preference set via *cudaFuncSetCacheConfig (C API)* or *cudaFuncSetCacheConfig (C++ API)* will be preferred over this device-wide setting. Setting the device-wide cache configuration to *cudaFuncCachePreferNone* will cause subsequent kernel launches to prefer to not change the cache configuration unless required to launch the kernel.

This setting does nothing on devices where the size of the L1 cache and shared memory are fixed.

Launching a kernel with a different preference than the most recent preference setting may insert a device-side synchronization point.

The supported cache configurations are:

- *cudaFuncCachePreferNone*: no preference for shared memory or L1 (default)
- *cudaFuncCachePreferShared*: prefer larger shared memory and smaller L1 cache
- *cudaFuncCachePreferL1*: prefer larger L1 cache and smaller shared memory
- *cudaFuncCachePreferEqual*: prefer equal size L1 cache and shared memory

See also:

cudaDeviceGetCacheConfig, *cudaFuncSetCacheConfig (C API)*, *cudaFuncSetCacheConfig (C++ API)*, ::cuCtxSetCacheConfig

Note: Note that this function may also return error codes from previous, asynchronous launches.

Note: Note that this function may also return *cudaErrorInitializationError*, *cudaErrorInsufficientDriver* or *cudaErrorNoDevice* if this call tries to initialize internal CUDA RT state.

Note: Note that as specified by *cudaStreamAddCallback* no CUDA function may be called from callback. *cudaErrorNotPermitted* may, but is not guaranteed to, be returned as a diagnostic in such case.

Parameters

cacheConfig – - Requested cache configuration

Returns
cudaSuccess

__host__ *cudaError_t* **cudaDeviceSetLimit**(enum *cudaLimit* limit, size_t value)

Set resource limits.

Setting `limit` to `value` is a request by the application to update the current limit maintained by the device. The driver is free to modify the requested value to meet h/w requirements (this could be clamping to minimum or maximum values, rounding up to nearest element size, etc). The application can use *cudaDeviceGetLimit()* to find out exactly what the limit has been set to.

Setting each *cudaLimit* has its own specific restrictions, so each is discussed here.

- *cudaLimitStackSize* controls the stack size in bytes of each GPU thread.
- *cudaLimitPrintfFifoSize* controls the size in bytes of the shared FIFO used by the ::printf() device system call. Setting *cudaLimitPrintfFifoSize* must not be performed after launching any kernel that uses the ::printf() device system call - in such case *cudaErrorInvalidValue* will be returned.
- *cudaLimitMallocHeapSize* controls the size in bytes of the heap used by the ::malloc() and ::free() device system calls. Setting *cudaLimitMallocHeapSize* must not be performed after launching any kernel that uses the ::malloc() or ::free() device system calls - in such case *cudaErrorInvalidValue* will be returned.
- *cudaLimitDevRuntimeSyncDepth* controls the maximum nesting depth of a grid at which a thread can safely call *cudaDeviceSynchronize()*. Setting this limit must be performed before any launch of a kernel that uses the device runtime and calls *cudaDeviceSynchronize()* above the default sync depth, two levels of grids. Calls to *cudaDeviceSynchronize()* will fail with error code *cudaErrorSyncDepthExceeded* if the limitation is violated. This limit can be set smaller than the default or up the maximum launch depth of 24. When setting this limit, keep in mind that additional levels of sync depth require the runtime to reserve large amounts of device memory which can no longer be used for user allocations. If these reservations of device memory fail, *cudaDeviceSetLimit* will return *cudaErrorMemoryAllocation*, and the limit can be reset to a lower value. This limit is only applicable to devices of compute capability < 9.0. Attempting to set this limit on devices of other compute capability will results in error *cudaErrorUnsupportedLimit* being returned.
- *cudaLimitDevRuntimePendingLaunchCount* controls the maximum number of outstanding device runtime launches that can be made from the current device. A grid is outstanding from the point of launch up until the grid is known to have been completed. Device runtime launches which violate this limitation fail and return *cudaErrorLaunchPendingCountExceeded* when *cudaGetLastError()* is called after launch. If more pending launches than the default (2048 launches) are needed for a module using the device runtime, this limit can be increased. Keep in mind that being able to sustain additional pending launches will require the runtime to reserve larger amounts of device memory upfront which can no longer be used for allocations. If these reservations fail, *cudaDeviceSetLimit* will return *cudaErrorMemoryAllocation*, and the limit can be reset to a lower value. This limit is only applicable to devices of compute capability 3.5 and higher. Attempting to set this limit on devices of compute capability less than 3.5 will result in the error *cudaErrorUnsupportedLimit* being returned.
- *cudaLimitMaxL2FetchGranularity* controls the L2 cache fetch granularity. Values can range from 0B to 128B. This is purely a performance hint and it can be ignored or clamped depending on the platform.

- *cudaLimitPersistingL2CacheSize* controls size in bytes available for persisting L2 cache. This is purely a performance hint and it can be ignored or clamped depending on the platform.

See also:

cudaDeviceGetLimit, ::cuCtxSetLimit

Note: Note that this function may also return error codes from previous, asynchronous launches.

Note: Note that this function may also return *cudaErrorInitializationError*, *cudaErrorInsufficientDriver* or *cudaErrorNoDevice* if this call tries to initialize internal CUDA RT state.

Note: Note that as specified by *cudaStreamAddCallback* no CUDA function may be called from callback. *cudaErrorNotPermitted* may, but is not guaranteed to, be returned as a diagnostic in such case.

Parameters

- **`limit`** – - Limit to set
- **`value`** – - Size of limit

Returns

cudaSuccess, *cudaErrorUnsupportedLimit*, *cudaErrorInvalidValue*, *cudaErrorMemoryAllocation*

__host__ *cudaError_t* **`cudaDeviceSetMemPool`**(int device, *cudaMemPool_t* memPool)

Sets the current memory pool of a device.

The memory pool must be local to the specified device. Unless a mempool is specified in the *cudaMallocAsync* call, *cudaMallocAsync* allocates from the current mempool of the provided stream's device. By default, a device's current memory pool is its default memory pool.

See also:

::cuDeviceSetMemPool, *cudaDeviceGetMemPool*, *cudaDeviceGetDefaultMemPool*, *cudaMemPoolCreate*, *cudaMemPoolDestroy*, *cudaMallocFromPoolAsync*

Note: Use *cudaMallocFromPoolAsync* to specify asynchronous allocations from a device different than the one the stream runs on.

Note: Note that this function may also return error codes from previous, asynchronous launches.

Note: Note that as specified by *cudaStreamAddCallback* no CUDA function may be called from callback. *cudaErrorNotPermitted* may, but is not guaranteed to, be returned as a diagnostic in

such case.

Returns

cudaSuccess, *cudaErrorInvalidValue* *cudaErrorInvalidDevice* *cudaErrorNotSupported*

__host__ *cudaError_t* **cudaDeviceSynchronize**(void)

Wait for compute device to finish.

Blocks until the device has completed all preceding requested tasks. *cudaDeviceSynchronize()* returns an error if one of the preceding tasks has failed. If the *cudaDeviceScheduleBlockingSync* flag was set for this device, the host thread will block until the device has finished its work.

See also:

cudaDeviceReset, ::cuCtxSynchronize

Note: Use of cudaDeviceSynchronize in device code was deprecated in CUDA 11.6 and removed for compute_90+ compilation. For compute capability < 9.0, compile-time opt-in by specifying -D CUDA_FORCE_CDP1_IF_SUPPORTED is required to continue using *cudaDeviceSynchronize()* in device code for now. Note that this is different from host-side cudaDeviceSynchronize, which is still supported.

Note: Note that this function may also return error codes from previous, asynchronous launches.

Note: Note that this function may also return *cudaErrorInitializationError*, *cudaErrorInsufficientDriver* or *cudaErrorNoDevice* if this call tries to initialize internal CUDA RT state.

Note: Note that as specified by *cudaStreamAddCallback* no CUDA function may be called from callback. *cudaErrorNotPermitted* may, but is not guaranteed to, be returned as a diagnostic in such case.

Returns

cudaSuccess, *cudaErrorStreamCaptureUnsupported*

__host__ *cudaError_t* **cudaDeviceUnregisterAsyncNotification**(int device,
cudaAsyncCallbackHandle_t callback)

Unregisters an async notification callback.

Unregisters `callback` so that the corresponding callback function will stop receiving async notifications.

See also:

cudaDeviceRegisterAsyncNotification

Note: Note that this function may also return error codes from previous, asynchronous launches.

Parameters

- **device** – - The device from which to remove `callback`.
- **callback** – - The callback instance to unregister from receiving async notifications.

Returns

cudaSuccess cudaErrorNotSupported cudaErrorInvalidDevice cudaErrorInvalidValue cudaErrorNotPermitted cudaErrorUnknown

__host__ *cudaError_t* **cudaGetDevice**(int *device)

Returns which device is currently being used.

Returns in `*device` the current device for the calling host thread.

See also:

cudaGetDeviceCount, *cudaSetDevice*, *cudaGetDeviceProperties*, *cudaChooseDevice*, ::cuCtxGetCurrent

Note: Note that this function may also return error codes from previous, asynchronous launches.

Note: Note that this function may also return *cudaErrorInitializationError*, *cudaErrorInsufficientDriver* or *cudaErrorNoDevice* if this call tries to initialize internal CUDA RT state.

Note: Note that as specified by *cudaStreamAddCallback* no CUDA function may be called from callback. *cudaErrorNotPermitted* may, but is not guaranteed to, be returned as a diagnostic in such case.

Parameters

device – - Returns the device on which the active host thread executes the device code.

Returns

cudaSuccess, *cudaErrorInvalidValue*, ::cudaErrorDeviceUnavailable,

__host__ *cudaError_t* **cudaGetDeviceCount**(int *count)

Returns the number of compute-capable devices.

Returns in `*count` the number of devices with compute capability greater or equal to 2.0 that are available for execution.

See also:

cudaGetDevice, *cudaSetDevice*, *cudaGetDeviceProperties*, *cudaChooseDevice*, *cudaInitDevice*, ::cuDeviceGetCount

---

Note: Note that this function may also return error codes from previous, asynchronous launches.

Note: Note that this function may also return *cudaErrorInitializationError*, *cudaErrorInsufficientDriver* or *cudaErrorNoDevice* if this call tries to initialize internal CUDA RT state.

Note: Note that as specified by *cudaStreamAddCallback* no CUDA function may be called from callback. *cudaErrorNotPermitted* may, but is not guaranteed to, be returned as a diagnostic in such case.

Parameters

count – - Returns the number of devices with compute capability greater or equal to 2.0

Returns

cudaSuccess

__host__ *cudaError_t* **cudaGetDeviceFlags**(unsigned int *flags)

Gets the flags for the current device.

Returns in `flags` the flags for the current device. If there is a current device for the calling thread, the flags for the device are returned. If there is no current device, the flags for the first device are returned, which may be the default flags. Compare to the behavior of *cudaSetDeviceFlags*.

Typically, the flags returned should match the behavior that will be seen if the calling thread uses a device after this call, without any change to the flags or current device inbetween by this or another thread. Note that if the device is not initialized, it is possible for another thread to change the flags for the current device before it is initialized. Additionally, when using exclusive mode, if this thread has not requested a specific device, it may use a device other than the first device, contrary to the assumption made by this function.

If a context has been created via the driver API and is current to the calling thread, the flags for that context are always returned.

Flags returned by this function may specifically include *cudaDeviceMapHost* even though it is not accepted by *cudaSetDeviceFlags* because it is implicit in runtime API flags. The reason for this is that the current context may have been created via the driver API in which case the flag is not implicit and may be unset.

See also:

cudaGetDevice, *cudaGetDeviceProperties*, *cudaSetDevice*, *cudaSetDeviceFlags*, *cudaInitDevice*, ::cuCtxGetFlags, ::cuDevicePrimaryCtxGetState

---

Note: Note that this function may also return error codes from previous, asynchronous launches.

Note: Note that this function may also return *cudaErrorInitializationError*, *cudaErrorInsufficientDriver* or *cudaErrorNoDevice* if this call tries to initialize internal CUDA RT state.

Note: Note that as specified by *cudaStreamAddCallback* no CUDA function may be called from callback. *cudaErrorNotPermitted* may, but is not guaranteed to, be returned as a diagnostic in such case.

Parameters

flags – - Pointer to store the device flags

Returns

cudaSuccess, *cudaErrorInvalidDevice*

__host__ *cudaError_t* **cudaGetDeviceProperties**(struct *cudaDeviceProp* *prop, int device)

Returns information about the compute-device.

Returns in `*prop` the properties of device `dev`.

See also:

cudaGetDeviceCount, *cudaGetDevice*, *cudaSetDevice*, *cudaChooseDevice*, *cudaDeviceGetAttribute*, *cudaInitDevice*, ::cuDeviceGetAttribute, ::cuDeviceGetName

Note: Note that this function may also return error codes from previous, asynchronous launches.

Note: Note that this function may also return *cudaErrorInitializationError*, *cudaErrorInsufficientDriver* or *cudaErrorNoDevice* if this call tries to initialize internal CUDA RT state.

Note: Note that as specified by *cudaStreamAddCallback* no CUDA function may be called from callback. *cudaErrorNotPermitted* may, but is not guaranteed to, be returned as a diagnostic in such case.

Parameters

- **prop** – - Properties for the specified device
- **device** – - Device number to get properties for

Returns

cudaSuccess, *cudaErrorInvalidDevice*

__host__ *cudaError_t* **cudaInitDevice**(int device, unsigned int deviceFlags, unsigned int flags)

Initialize device to be used for GPU executions.

This function will initialize the CUDA Runtime structures and primary context on `device` when called, but the context will not be made current to `device`.

When *cudaInitDeviceFlagsAreValid* is set in `flags`, deviceFlags are applied to the requested device. The values of deviceFlags match those of the flags parameters in *cudaSetDeviceFlags*. The effect may be verified by *cudaGetDeviceFlags*.

This function will return an error if the device is in *cudaComputeModeExclusiveProcess* and is occupied by another process or if the device is in *cudaComputeModeProhibited*.

See also:

cudaGetDeviceCount, *cudaGetDevice*, *cudaGetDeviceProperties*, *cudaChooseDevice*, *cudaSetDevice* ::cuCtxSetCurrent

---

Note: Note that this function may also return error codes from previous, asynchronous launches.

Note: Note that this function may also return *cudaErrorInitializationError*, *cudaErrorInsufficientDriver* or *cudaErrorNoDevice* if this call tries to initialize internal CUDA RT state.

Note: Note that as specified by *cudaStreamAddCallback* no CUDA function may be called from callback. *cudaErrorNotPermitted* may, but is not guaranteed to, be returned as a diagnostic in such case.

Parameters

- **`device`** – - Device on which the runtime will initialize itself.
- **`deviceFlags`** – - Parameters for device operation.
- **`flags`** – - Flags for controlling the device initialization.

Returns

cudaSuccess, *cudaErrorInvalidDevice*,

__host__ *cudaError_t* **`cudaIpcCloseMemHandle`**(void *devPtr)

Attempts to close memory mapped with cudaIpcOpenMemHandle.

Decrements the reference count of the memory returnd by *cudaIpcOpenMemHandle* by 1. When the reference count reaches 0, this API unmaps the memory. The original allocation in the exporting process as well as imported mappings in other processes will be unaffected.

Any resources used to enable peer access will be freed if this is the last mapping using them.

IPC functionality is restricted to devices with support for unified addressing on Linux and Windows operating systems. IPC functionality on Windows is supported for compatibility purposes but not recommended as it comes with performance cost. Users can test their device for IPC functionality by calling *cudaDeviceGetAttribute* with *cudaDevAttrIpcEventSupport*

See also:

cudaMalloc, *cudaFree*, *cudaIpcGetEventHandle*, *cudaIpcOpenEventHandle*, *cudaIpcGetMemHandle*, *cudaIpcOpenMemHandle*, ::cuIpcCloseMemHandle

Note: Note that this function may also return *cudaErrorInitializationError*, *cudaErrorInsufficientDriver* or *cudaErrorNoDevice* if this call tries to initialize internal CUDA RT state.

Note: Note that as specified by *cudaStreamAddCallback* no CUDA function may be called from callback. *cudaErrorNotPermitted* may, but is not guaranteed to, be returned as a diagnostic in such case.

Parameters

devPtr – - Device pointer returned by *cudaIpcOpenMemHandle*

Returns

cudaSuccess, *cudaErrorMapBufferObjectFailed*, *cudaErrorNotSupported*, *cudaErrorInvalidValue*

__host__ *cudaError_t* **cudaIpcGetEventHandle**(*cudaIpcEventHandle_t* *handle, *cudaEvent_t* event)

Gets an interprocess handle for a previously allocated event.

Takes as input a previously allocated event. This event must have been created with the *cudaEventInterprocess* and *cudaEventDisableTiming* flags set. This opaque handle may be copied into other processes and opened with *cudaIpcOpenEventHandle* to allow efficient hardware synchronization between GPU work in different processes.

After the event has been been opened in the importing process, *cudaEventRecord*, *cudaEventSynchronize*, *cudaStreamWaitEvent* and *cudaEventQuery* may be used in either process. Performing operations on the imported event after the exported event has been freed with *cudaEventDestroy* will result in undefined behavior.

IPC functionality is restricted to devices with support for unified addressing on Linux and Windows operating systems. IPC functionality on Windows is supported for compatibility purposes but not recommended as it comes with performance cost. Users can test their device for IPC functionality by calling *cudaDeviceGetAttribute* with *cudaDevAttrIpcEventSupport*

See also:

cudaEventCreate, *cudaEventDestroy*, *cudaEventSynchronize*, *cudaEventQuery*, *cudaStreamWaitEvent*, *cudaIpcOpenEventHandle*, *cudaIpcGetMemHandle*, *cudaIpcOpenMemHandle*, *cudaIpcCloseMemHandle*, ::cuIpcGetEventHandle

Note: Note that this function may also return *cudaErrorInitializationError*, *cudaErrorInsufficientDriver* or *cudaErrorNoDevice* if this call tries to initialize internal CUDA RT state.

Note: Note that as specified by *cudaStreamAddCallback* no CUDA function may be called from callback. *cudaErrorNotPermitted* may, but is not guaranteed to, be returned as a diagnostic in such case.

Parameters

- **handle** – - Pointer to a user allocated cudaIpcEventHandle in which to return the opaque event handle
- **event** – - Event allocated with *cudaEventInterprocess* and *cudaEventDisableTiming* flags.

Returns

cudaSuccess, *cudaErrorInvalidResourceHandle*, *cudaErrorMemoryAllocation*, *cudaErrorMapBufferObjectFailed*, *cudaErrorNotSupported*, *cudaErrorInvalidValue*

__host__ *cudaError_t* **cudaIpcGetMemHandle**(*cudaIpcMemHandle_t* *handle, void *devPtr)

Gets an interprocess memory handle for an existing device memory allocation.

Takes a pointer to the base of an existing device memory allocation created with *cudaMalloc* and exports it for use in another process. This is a lightweight operation and may be called multiple times on an allocation without adverse effects.

If a region of memory is freed with *cudaFree* and a subsequent call to *cudaMalloc* returns memory with the same device address, *cudaIpcGetMemHandle* will return a unique handle for the new memory.

IPC functionality is restricted to devices with support for unified addressing on Linux and Windows operating systems. IPC functionality on Windows is supported for compatibility purposes but not recommended as it comes with performance cost. Users can test their device for IPC functionality by calling *cudaDeviceGetAttribute* with *cudaDevAttrIpcEventSupport*

See also:

cudaMalloc, *cudaFree*, *cudaIpcGetEventHandle*, *cudaIpcOpenEventHandle*, *cudaIpcOpenMemHandle*, *cudaIpcCloseMemHandle*, ::cuIpcGetMemHandle

---

Note: Note that this function may also return *cudaErrorInitializationError*, *cudaErrorInsufficientDriver* or *cudaErrorNoDevice* if this call tries to initialize internal CUDA RT state.

Note: Note that as specified by *cudaStreamAddCallback* no CUDA function may be called from callback. *cudaErrorNotPermitted* may, but is not guaranteed to, be returned as a diagnostic in such case.

Parameters

- **handle** – - Pointer to user allocated ::cudaIpcMemHandle to return the handle in.
- **devPtr** – - Base pointer to previously allocated device memory

Returns

cudaSuccess, *cudaErrorMemoryAllocation*, *cudaErrorMapBufferObjectFailed*, *cudaErrorNotSupported*, *cudaErrorInvalidValue*

__host__ *cudaError_t* **cudaIpcOpenEventHandle**(*cudaEvent_t* *event, *cudaIpcEventHandle_t* handle)

Opens an interprocess event handle for use in the current process.

Opens an interprocess event handle exported from another process with *cudaIpcGetEventHandle*. This function returns a *cudaEvent_t* that behaves like a locally created event with the *cudaEventDisableTiming* flag specified. This event must be freed with *cudaEventDestroy*.

Performing operations on the imported event after the exported event has been freed with *cudaEventDestroy* will result in undefined behavior.

IPC functionality is restricted to devices with support for unified addressing on Linux and Windows operating systems. IPC functionality on Windows is supported for compatibility purposes but not recommended as it comes with performance cost. Users can test their device for IPC functionality by calling *cudaDeviceGetAttribute* with *cudaDevAttrIpcEventSupport*

See also:

cudaEventCreate, *cudaEventDestroy*, *cudaEventSynchronize*, *cudaEventQuery*, *cudaStreamWaitEvent*, *cudaIpcGetEventHandle*, *cudaIpcGetMemHandle*, *cudaIpcOpenMemHandle*, *cudaIpcCloseMemHandle*, ::cuIpcOpenEventHandle

---

Note: Note that this function may also return *cudaErrorInitializationError*, *cudaErrorInsufficientDriver* or *cudaErrorNoDevice* if this call tries to initialize internal CUDA RT state.

Note: Note that as specified by *cudaStreamAddCallback* no CUDA function may be called from callback. *cudaErrorNotPermitted* may, but is not guaranteed to, be returned as a diagnostic in such case.

Parameters

- **event** – - Returns the imported event
- **handle** – - Interprocess handle to open

Returns

cudaSuccess, *cudaErrorMapBufferObjectFailed*, *cudaErrorNotSupported*, *cudaErrorInvalidValue*, *cudaErrorDeviceUninitialized*

__host__ *cudaError_t* **cudaIpcOpenMemHandle**(void **devPtr, *cudaIpcMemHandle_t* handle, unsigned int flags)

Opens an interprocess memory handle exported from another process and returns a device pointer usable in the local process.

Maps memory exported from another process with *cudaIpcGetMemHandle* into the current device address space. For contexts on different devices *cudaIpcOpenMemHandle* can attempt to enable peer access between the devices as if the user called *cudaDeviceEnablePeerAccess*. This behavior is controlled by the *cudaIpcMemLazyEnablePeerAccess* flag. *cudaDeviceCanAccessPeer* can determine if a mapping is possible.

cudaIpcOpenMemHandle can open handles to devices that may not be visible in the process calling the API.

Contexts that may open ::cudaIpcMemHandles are restricted in the following way. ::cudaIpcMemHandles from each device in a given process may only be opened by one context per device per other process.

If the memory handle has already been opened by the current context, the reference count on the handle is incremented by 1 and the existing device pointer is returned.

Memory returned from *cudaIpcOpenMemHandle* must be freed with *cudaIpcCloseMemHandle*.

Calling *cudaFree* on an exported memory region before calling *cudaIpcCloseMemHandle* in the importing context will result in undefined behavior.

IPC functionality is restricted to devices with support for unified addressing on Linux and Windows operating systems. IPC functionality on Windows is supported for compatibility purposes but not recommended as it comes with performance cost. Users can test their device for IPC functionality by calling *cudaDeviceGetAttribute* with *cudaDevAttrIpcEventSupport*

See also:

cudaMalloc, *cudaFree*, *cudaIpcGetEventHandle*, *cudaIpcOpenEventHandle*, *cudaIpcGetMemHandle*, *cudaIpcCloseMemHandle*, *cudaDeviceEnablePeerAccess*, *cudaDeviceCanAccessPeer*, ::cudaIpcOpenMemHandle

---

Note: Note that this function may also return *cudaErrorInitializationError*, *cudaErrorInsufficientDriver* or *cudaErrorNoDevice* if this call tries to initialize internal CUDA RT state.

Note: Note that as specified by *cudaStreamAddCallback* no CUDA function may be called from callback. *cudaErrorNotPermitted* may, but is not guaranteed to, be returned as a diagnostic in such case.

Note: No guarantees are made about the address returned in `*devPtr` .

In particular, multiple processes may not receive the same address for the same `handle`.

Parameters

- **`devPtr`** – - Returned device pointer
- **`handle`** – - ::cudaIpcMemHandle to open
- **`flags`** – - Flags for this operation. Must be specified as *cudaIpcMemLazyEnablePeerAccess*

Returns

cudaSuccess, *cudaErrorMapBufferObjectFailed*, *cudaErrorInvalidResourceHandle*, *cudaErrorDeviceUninitialized*, *cudaErrorTooManyPeers*, *cudaErrorNotSupported*, *cudaErrorInvalidValue*

__host__ *cudaError_t* **`cudaSetDevice`**(int device)

Set device to be used for GPU executions.

Sets `device` as the current device for the calling host thread. Valid device id's are 0 to (*cudaGetDeviceCount()* - 1).

Any device memory subsequently allocated from this host thread using *cudaMalloc()*, *cudaMallocPitch()* or *cudaMallocArray()* will be physically resident on `device`. Any host memory allocated from this host thread using *cudaMallocHost()* or *cudaHostAlloc()* or *cudaHostRegister()* will have

its lifetime associated with `device`. Any streams or events created from this host thread will be associated with `device`. Any kernels launched from this host thread using the <<<>>> operator or *cudaLaunchKernel()* will be executed on `device` when using the NULL or per-thread streams (otherwise, they will be executed on the device associated with the stream).

This call may be made from any host thread, to any device, and at any time. This function will do no synchronization with the previous or new device, and should only take significant time when it initializes the runtime's context state. This call will bind the primary context of the specified device to the calling thread and all the subsequent memory allocations, stream and event creations will be associated with the primary context. This function will also immediately initialize the runtime state on the primary context, and the context will be current on `device` immediately. This function will return an error if the device is in *cudaComputeModeExclusiveProcess* and is occupied by another process or if the device is in *cudaComputeModeProhibited*.

It is not required to call *cudaInitDevice* before using this function.

See also:

cudaGetDeviceCount, *cudaGetDevice*, *cudaGetDeviceProperties*, *cudaChooseDevice*, *cudaInitDevice*, ::cuCtxSetCurrent

---

Note: Note that this function may also return error codes from previous, asynchronous launches.

Note: Note that this function may also return *cudaErrorInitializationError*, *cudaErrorInsufficientDriver* or *cudaErrorNoDevice* if this call tries to initialize internal CUDA RT state.

Note: Note that as specified by *cudaStreamAddCallback* no CUDA function may be called from callback. *cudaErrorNotPermitted* may, but is not guaranteed to, be returned as a diagnostic in such case.

Parameters

`device` – - Device on which the active host thread should execute the device code.

Returns

cudaSuccess, *cudaErrorInvalidDevice*, ::cudaErrorDeviceUnavailable,

__host__ *cudaError_t* **`cudaSetDeviceFlags`**(unsigned int flags)

Sets flags to be used for device executions.

Records `flags` as the flags for the current device. If the current device has been set and that device has already been initialized, the previous flags are overwritten. If the current device has not been initialized, it is initialized with the provided flags. If no device has been made current to the calling thread, a default device is selected and initialized with the provided flags.

The three LSBs of the `flags` parameter can be used to control how the CPU thread interacts with the OS scheduler when waiting for results from the device.

- *cudaDeviceScheduleAuto*: The default value if the `flags` parameter is zero, uses a heuristic based on the number of active CUDA contexts in the process C and the number of logical processors in the system P. If C > P, then CUDA will yield to other OS threads when waiting for the device, otherwise CUDA will not yield while waiting for results and actively spin on the processor. Additionally, on Tegra devices, *cudaDeviceScheduleAuto* uses a heuristic based

on the power profile of the platform and may choose *cudaDeviceScheduleBlockingSync* for low-powered devices.

- *cudaDeviceScheduleSpin*: Instruct CUDA to actively spin when waiting for results from the device. This can decrease latency when waiting for the device, but may lower the performance of CPU threads if they are performing work in parallel with the CUDA thread.
- *cudaDeviceScheduleYield*: Instruct CUDA to yield its thread when waiting for results from the device. This can increase latency when waiting for the device, but can increase the performance of CPU threads performing work in parallel with the device.
- *cudaDeviceScheduleBlockingSync*: Instruct CUDA to block the CPU thread on a synchronization primitive when waiting for the device to finish work.
- *cudaDeviceBlockingSync*: Instruct CUDA to block the CPU thread on a synchronization primitive when waiting for the device to finish work. *Deprecated:* This flag was deprecated as of CUDA 4.0 and replaced with *cudaDeviceScheduleBlockingSync*.
- *cudaDeviceMapHost*: This flag enables allocating pinned host memory that is accessible to the device. It is implicit for the runtime but may be absent if a context is created using the driver API. If this flag is not set, *cudaHostGetDevicePointer()* will always return a failure code.
- *cudaDeviceLmemResizeToMax*: Instruct CUDA to not reduce local memory after resizing local memory for a kernel. This can prevent thrashing by local memory allocations when launching many kernels with high local memory usage at the cost of potentially increased memory usage. *Deprecated:* This flag is deprecated and the behavior enabled by this flag is now the default and cannot be disabled.
- *cudaDeviceSyncMemops*: Ensures that synchronous memory operations initiated on this context will always synchronize. See further documentation in the section titled “API Synchronization behavior” to learn more about cases when synchronous memory operations can exhibit asynchronous behavior.

See also:

cudaGetDeviceFlags, *cudaGetDeviceCount*, *cudaGetDevice*, *cudaGetDeviceProperties*, *cudaSetDevice*, *cudaSetValidDevices*, *cudaInitDevice*, *cudaChooseDevice*, ::cuDevicePrimaryCtxSetFlags

---

Note: Note that this function may also return error codes from previous, asynchronous launches.

Note: Note that this function may also return *cudaErrorInitializationError*, *cudaErrorInsufficientDriver* or *cudaErrorNoDevice* if this call tries to initialize internal CUDA RT state.

Note: Note that as specified by *cudaStreamAddCallback* no CUDA function may be called from callback. *cudaErrorNotPermitted* may, but is not guaranteed to, be returned as a diagnostic in such case.

Parameters

`flags` – - Parameters for device operation

Returns

cudaSuccess, *cudaErrorInvalidValue*

__host__ *cudaError_t* **cudaSetValidDevices**(int *device_arr, int len)

Set a list of devices that can be used for CUDA.

Sets a list of devices for CUDA execution in priority order using `device_arr`. The parameter `len` specifies the number of elements in the list. CUDA will try devices from the list sequentially until it finds one that works. If this function is not called, or if it is called with a `len` of 0, then CUDA will go back to its default behavior of trying devices sequentially from a default list containing all of the available CUDA devices in the system. If a specified device ID in the list does not exist, this function will return *cudaErrorInvalidDevice*. If `len` is not 0 and `device_arr` is NULL or if `len` exceeds the number of devices in the system, then *cudaErrorInvalidValue* is returned.

See also:

cudaGetDeviceCount, *cudaSetDevice*, *cudaGetDeviceProperties*, *cudaSetDeviceFlags*, *cudaChooseDevice*

Note: Note that this function may also return error codes from previous, asynchronous launches.

Note: Note that this function may also return *cudaErrorInitializationError*, *cudaErrorInsufficientDriver* or *cudaErrorNoDevice* if this call tries to initialize internal CUDA RT state.

Note: Note that as specified by *cudaStreamAddCallback* no CUDA function may be called from callback. *cudaErrorNotPermitted* may, but is not guaranteed to, be returned as a diagnostic in such case.

Parameters

- **device_arr** – - List of devices to try
- **len** – - Number of devices in specified list

Returns

cudaSuccess, *cudaErrorInvalidValue*, *cudaErrorInvalidDevice*

6.4. Device Management [DEPRECATED]

This section describes the deprecated device management functions of the CUDA runtime application programming interface.

Functions

__host__ cudaError_t *cudaDeviceGetSharedMemConfig*(enum cudaSharedMemConfig *pConfig)
Returns the shared memory configuration for the current device.

__host__ cudaError_t *cudaDeviceSetSharedMemConfig*(enum cudaSharedMemConfig config)
Sets the shared memory configuration for the current device.

6.4.1. Functions

__host__ *cudaError_t* **cudaDeviceGetSharedMemConfig**(enum *cudaSharedMemConfig* *pConfig)
Returns the shared memory configuration for the current device.

Deprecated:

This function will return in `pConfig` the current size of shared memory banks on the current device. On devices with configurable shared memory banks, *cudaDeviceSetSharedMemConfig* can be used to change this setting, so that all subsequent kernel launches will by default use the new bank size. When *cudaDeviceGetSharedMemConfig* is called on devices without configurable shared memory, it will return the fixed bank size of the hardware.

The returned bank configurations can be either:

- cudaSharedMemBankSizeFourByte - shared memory bank width is four bytes.
- cudaSharedMemBankSizeEightByte - shared memory bank width is eight bytes.

See also:

cudaDeviceSetCacheConfig, *cudaDeviceGetCacheConfig*, *cudaDeviceSetSharedMemConfig*, *cudaFuncSetCacheConfig*, ::cuCtxGetSharedMemConfig

---

Note: Note that this function may also return error codes from previous, asynchronous launches.

Note: Note that this function may also return *cudaErrorInitializationError*, *cudaErrorInsufficientDriver* or *cudaErrorNoDevice* if this call tries to initialize internal CUDA RT state.

Note: Note that as specified by *cudaStreamAddCallback* no CUDA function may be called from callback. *cudaErrorNotPermitted* may, but is not guaranteed to, be returned as a diagnostic in such case.

Parameters
pConfig – - Returned cache configuration

Returns
cudaSuccess, *cudaErrorInvalidValue*

__host__ *cudaError_t* **cudaDeviceSetSharedMemConfig**(enum *cudaSharedMemConfig* config)

Sets the shared memory configuration for the current device.

Deprecated:

On devices with configurable shared memory banks, this function will set the shared memory bank size which is used for all subsequent kernel launches. Any per-function setting of shared memory set via *cudaFuncSetSharedMemConfig* will override the device wide setting.

Changing the shared memory configuration between launches may introduce a device side synchronization point.

Changing the shared memory bank size will not increase shared memory usage or affect occupancy of kernels, but may have major effects on performance. Larger bank sizes will allow for greater potential bandwidth to shared memory, but will change what kinds of accesses to shared memory will result in bank conflicts.

This function will do nothing on devices with fixed shared memory bank size.

The supported bank configurations are:

- cudaSharedMemBankSizeDefault: set bank width the device default (currently, four bytes)
- cudaSharedMemBankSizeFourByte: set shared memory bank width to be four bytes natively.
- cudaSharedMemBankSizeEightByte: set shared memory bank width to be eight bytes natively.

See also:

cudaDeviceSetCacheConfig, *cudaDeviceGetCacheConfig*, *cudaDeviceGetSharedMemConfig*, *cudaFuncSetCacheConfig*, ::cuCtxSetSharedMemConfig

---

Note: Note that this function may also return error codes from previous, asynchronous launches.

Note: Note that this function may also return *cudaErrorInitializationError*, *cudaErrorInsufficientDriver* or *cudaErrorNoDevice* if this call tries to initialize internal CUDA RT state.

Note: Note that as specified by *cudaStreamAddCallback* no CUDA function may be called from callback. *cudaErrorNotPermitted* may, but is not guaranteed to, be returned as a diagnostic in such case.

Parameters

config – - Requested cache configuration

Returns

cudaSuccess, *cudaErrorInvalidValue*

6.5. Direct3D 10 Interoperability

impl_private

impl_private

This section describes the Direct3D 10 interoperability functions of the CUDA runtime application programming interface. Note that mapping of Direct3D 10 resources is performed with the graphics API agnostic, resource mapping interface described in *Graphics Interopability*.

Enumerations

cudaD3D10DeviceList
CUDA devices corresponding to a D3D10 device.

Functions

__host__ cudaError_t *cudaD3D10GetDevice*(int *device, IDXGIAdapter *pAdapter)
Gets the device number for an adapter.

__host__ cudaError_t *cudaD3D10GetDevices*(unsigned int *pCudaDeviceCount, int *pCudaDevices, unsigned int cudaDeviceCount, ID3D10Device *pD3D10Device, enum cudaD3D10DeviceList deviceList)
Gets the CUDA devices corresponding to a Direct3D 10 device.

__host__ cudaError_t *cudaGraphicsD3D10RegisterResource*(struct cudaGraphicsResource **resource, ID3D10Resource *pD3DResource, unsigned int flags)
Registers a Direct3D 10 resource for access by CUDA.

6.5.1. Enumerations

enum **cudaD3D10DeviceList**
CUDA devices corresponding to a D3D10 device.

Values:

enumerator **cudaD3D10DeviceListAll** = 1
The CUDA devices for all GPUs used by a D3D10 device.

enumerator **cudaD3D10DeviceListCurrentFrame** = 2
The CUDA devices for the GPUs used by a D3D10 device in its currently rendering frame.

enumerator **cudaD3D10DeviceListNextFrame** = 3
The CUDA devices for the GPUs to be used by a D3D10 device in the next frame

6.5.2. Functions

__host__ *cudaError_t* **cudaD3D10GetDevice**(int *device, IDXGIAdapter *pAdapter)

Gets the device number for an adapter.

Returns in `*device` the CUDA-compatible device corresponding to the adapter `pAdapter` obtained from ::IDXGIFactory::EnumAdapters. This call will succeed only if a device on adapter `pAdapter` is CUDA-compatible.

See also:

cudaGraphicsD3D10RegisterResource, ::cuD3D10GetDevice

Note: Note that this function may also return error codes from previous, asynchronous launches.

Parameters

- **device** – - Returns the device corresponding to pAdapter
- **pAdapter** – - D3D10 adapter to get device for

Returns

cudaSuccess, *cudaErrorInvalidValue*, *cudaErrorUnknown*

__host__ *cudaError_t* **cudaD3D10GetDevices**(unsigned int *pCudaDeviceCount, int *pCudaDevices, unsigned int cudaDeviceCount, ID3D10Device *pD3D10Device, enum *cudaD3D10DeviceList* deviceList)

Gets the CUDA devices corresponding to a Direct3D 10 device.

Returns in `*pCudaDeviceCount` the number of CUDA-compatible devices corresponding to the Direct3D 10 device `pD3D10Device`. Also returns in `*pCudaDevices` at most `cudaDeviceCount` of the the CUDA-compatible devices corresponding to the Direct3D 10 device `pD3D10Device`.

If any of the GPUs being used to render `pDevice` are not CUDA capable then the call will return *cudaErrorNoDevice*.

See also:

cudaGraphicsUnregisterResource, *cudaGraphicsMapResources*, *cudaGraphicsSubResourceGetMappedArray*, *cudaGraphicsResourceGetMappedPointer*, ::cuD3D10GetDevices

---

Note: Note that this function may also return error codes from previous, asynchronous launches.

Parameters

- **pCudaDeviceCount** – - Returned number of CUDA devices corresponding to `pD3D10Device`
- **pCudaDevices** – - Returned CUDA devices corresponding to `pD3D10Device`

- **cudaDeviceCount** – - The size of the output device array pCudaDevices
- **pD3D10Device** – - Direct3D 10 device to query for CUDA devices
- **deviceList** – - The set of devices to return. This set may be *cudaD3D10DeviceListAll* for all devices, *cudaD3D10DeviceListCurrentFrame* for the devices used to render the current frame (in SLI), or *cudaD3D10DeviceListNextFrame* for the devices used to render the next frame (in SLI).

Returns
cudaSuccess, *cudaErrorNoDevice*, *cudaErrorUnknown*

__host__ *cudaError_t* **cudaGraphicsD3D10RegisterResource**(struct cudaGraphicsResource **resource, ID3D10Resource *pD3DResource, unsigned int flags)

Registers a Direct3D 10 resource for access by CUDA.

Registers the Direct3D 10 resource pD3DResource for access by CUDA.

If this call is successful, then the application will be able to map and unmap this resource until it is unregistered through *cudaGraphicsUnregisterResource()*. Also on success, this call will increase the internal reference count on pD3DResource. This reference count will be decremented when this resource is unregistered through *cudaGraphicsUnregisterResource()*.

This call potentially has a high-overhead and should not be called every frame in interactive applications.

The type of pD3DResource must be one of the following.

- ::ID3D10Buffer: may be accessed via a device pointer
- ::ID3D10Texture1D: individual subresources of the texture may be accessed via arrays
- ::ID3D10Texture2D: individual subresources of the texture may be accessed via arrays
- ::ID3D10Texture3D: individual subresources of the texture may be accessed via arrays

The flags argument may be used to specify additional parameters at register time. The valid values for this parameter are

- *cudaGraphicsRegisterFlagsNone*: Specifies no hints about how this resource will be used.
- *cudaGraphicsRegisterFlagsSurfaceLoadStore*: Specifies that CUDA will bind this resource to a surface reference.
- *cudaGraphicsRegisterFlagsTextureGather*: Specifies that CUDA will perform texture gather operations on this resource.

Not all Direct3D resources of the above types may be used for interoperability with CUDA. The following are some limitations.

- The primary rendertarget may not be registered with CUDA.
- Textures which are not of a format which is 1, 2, or 4 channels of 8, 16, or 32-bit integer or floating-point data cannot be shared.
- Surfaces of depth or stencil formats cannot be shared.

A complete list of supported DXGI formats is as follows. For compactness the notation A_{B,C,D} represents A_B, A_C, and A_D.

- DXGI_FORMAT_A8_UNORM
- DXGI_FORMAT_B8G8R8A8_UNORM
- DXGI_FORMAT_B8G8R8X8_UNORM
- DXGI_FORMAT_R16_FLOAT
- DXGI_FORMAT_R16G16B16A16_{FLOAT,SINT,SNORM,UINT,UNORM}
- DXGI_FORMAT_R16G16_{FLOAT,SINT,SNORM,UINT,UNORM}
- DXGI_FORMAT_R16_{SINT,SNORM,UINT,UNORM}
- DXGI_FORMAT_R32_FLOAT
- DXGI_FORMAT_R32G32B32A32_{FLOAT,SINT,UINT}
- DXGI_FORMAT_R32G32_{FLOAT,SINT,UINT}
- DXGI_FORMAT_R32_{SINT,UINT}
- DXGI_FORMAT_R8G8B8A8_{SINT,SNORM,UINT,UNORM,UNORM_SRGB}
- DXGI_FORMAT_R8G8_{SINT,SNORM,UINT,UNORM}
- DXGI_FORMAT_R8_{SINT,SNORM,UINT,UNORM}

If `pD3DResource` is of incorrect type or is already registered, then *cudaErrorInvalidResourceHandle* is returned. If `pD3DResource` cannot be registered, then *cudaErrorUnknown* is returned.

See also:

cudaGraphicsUnregisterResource, *cudaGraphicsMapResources*, *cudaGraphicsSubResourceGetMappedArray*, *cudaGraphicsResourceGetMappedPointer*, ::cuGraphicsD3D10RegisterResource

---

Note: Note that this function may also return error codes from previous, asynchronous launches.

Parameters

- **`resource`** – - Pointer to returned resource handle
- **`pD3DResource`** – - Direct3D resource to register
- **`flags`** – - Parameters for resource registration

Returns

cudaSuccess, *cudaErrorInvalidDevice*, *cudaErrorInvalidValue*, *cudaErrorInvalidResourceHandle*, *cudaErrorUnknown*

6.6. Direct3D 10 Interoperability [DEPRECATED]

This section describes deprecated Direct3D 10 interoperability functions.

Enumerations

cudaD3D10MapFlags
CUDA D3D10 Map Flags.

cudaD3D10RegisterFlags
CUDA D3D10 Register Flags.

Functions

__host__ cudaError_t *cudaD3D10GetDirect3DDevice*(**ID3D10Device **ppD3D10Device**)
Gets the Direct3D device against which the current CUDA context was created.

__host__ cudaError_t *cudaD3D10MapResources*(**int count, ID3D10Resource **ppResources**)
Maps Direct3D Resources for access by CUDA.

__host__ cudaError_t *cudaD3D10RegisterResource*(**ID3D10Resource *pResource, unsigned int flags**)
Registers a Direct3D 10 resource for access by CUDA.

__host__ cudaError_t *cudaD3D10ResourceGetMappedArray*(**cudaArray **ppArray, ID3D10Resource *pResource, unsigned int subResource**)
Gets an array through which to access a subresource of a Direct3D resource which has been mapped for access by CUDA.

__host__ cudaError_t *cudaD3D10ResourceGetMappedPitch*(**size_t *pPitch, size_t *pPitchSlice, ID3D10Resource *pResource, unsigned int subResource**)
Gets the pitch of a subresource of a Direct3D resource which has been mapped for access by CUDA.

__host__ cudaError_t *cudaD3D10ResourceGetMappedPointer*(**void **pPointer, ID3D10Resource *pResource, unsigned int subResource**)
Gets a pointer through which to access a subresource of a Direct3D resource which has been mapped for access by CUDA.

__host__ cudaError_t *cudaD3D10ResourceGetMappedSize*(**size_t *pSize, ID3D10Resource *pResource, unsigned int subResource**)
Gets the size of a subresource of a Direct3D resource which has been mapped for access by CUDA.

__host__ cudaError_t *cudaD3D10ResourceGetSurfaceDimensions*(**size_t *pWidth, size_t *pHeight, size_t *pDepth, ID3D10Resource *pResource, unsigned int subResource**)
Gets the dimensions of a registered Direct3D surface.

__host__ cudaError_t *cudaD3D10ResourceSetMapFlags*(**ID3D10Resource *pResource, unsigned int flags**)
Set usage flags for mapping a Direct3D resource.

__host__ cudaError_t *cudaD3D10SetDirect3DDevice*(**ID3D10Device *pD3D10Device, int device=-1**)
Sets the Direct3D 10 device to use for interoperability with a CUDA device.

__host__ cudaError_t *cudaD3D10UnmapResources*(**int count, ID3D10Resource **ppResources**)
Unmaps Direct3D resources.

__host__ cudaError_t *cudaD3D10UnregisterResource*(ID3D10Resource *pResource)
Unregisters a Direct3D resource.

6.6.1. Enumerations

enum **cudaD3D10MapFlags**
CUDA D3D10 Map Flags.
Values:

enumerator **cudaD3D10MapFlagsNone** = 0
Default; Assume resource can be read/written.

enumerator **cudaD3D10MapFlagsReadOnly** = 1
CUDA kernels will not write to this resource.

enumerator **cudaD3D10MapFlagsWriteDiscard** = 2
CUDA kernels will only write to and will not read from this resource.

enum **cudaD3D10RegisterFlags**
CUDA D3D10 Register Flags.
Values:

enumerator **cudaD3D10RegisterFlagsNone** = 0
Default; Resource can be accessed through a void*.

enumerator **cudaD3D10RegisterFlagsArray** = 1
Resource can be accessed through a CUarray*.

6.6.2. Functions

__host__ *cudaError_t* **cudaD3D10GetDirect3DDevice**(ID3D10Device **ppD3D10Device)
Gets the Direct3D device against which the current CUDA context was created.

Deprecated:
This function is deprecated as of CUDA 5.0.

This function is deprecated and should no longer be used. It is no longer necessary to associate a CUDA device with a D3D10 device in order to achieve maximum interoperability performance.

See also:
cudaD3D10SetDirect3DDevice

Note: Note that this function may also return error codes from previous, asynchronous launches.

Parameters

ppD3D10Device – - Returns the Direct3D device for this thread

Returns

cudaSuccess, *cudaErrorUnknown*

__host__ *cudaError_t* **cudaD3D10MapResources**(int count, ID3D10Resource **ppResources)

Maps Direct3D Resources for access by CUDA.

Deprecated:

This function is deprecated as of CUDA 3.0.

Maps the `count` Direct3D resources in `ppResources` for access by CUDA.

The resources in `ppResources` may be accessed in CUDA kernels until they are unmapped. Direct3D should not access any resources while they are mapped by CUDA. If an application does so, the results are undefined.

This function provides the synchronization guarantee that any Direct3D calls issued before *cudaD3D10MapResources()* will complete before any CUDA kernels issued after *cudaD3D10MapResources()* begin.

If any of `ppResources` have not been registered for use with CUDA or if `ppResources` contains any duplicate entries then *cudaErrorInvalidResourceHandle* is returned. If any of `ppResources` are presently mapped for access by CUDA then *cudaErrorUnknown* is returned.

See also:

cudaGraphicsMapResources

Note: Note that this function may also return error codes from previous, asynchronous launches.

Parameters

- **count** – - Number of resources to map for CUDA
- **ppResources** – - Resources to map for CUDA

Returns

cudaSuccess, *cudaErrorInvalidResourceHandle*, *cudaErrorUnknown*

__host__ *cudaError_t* **cudaD3D10RegisterResource**(ID3D10Resource *pResource, unsigned int flags)

Registers a Direct3D 10 resource for access by CUDA.

Deprecated:

This function is deprecated as of CUDA 3.0.

Registers the Direct3D resource `pResource` for access by CUDA.

If this call is successful, then the application will be able to map and unmap this resource until it is unregistered through *cudaD3D10UnregisterResource()*. Also on success, this call will increase the internal reference count on `pResource`. This reference count will be decremented when this resource is unregistered through *cudaD3D10UnregisterResource()*.

This call potentially has a high-overhead and should not be called every frame in interactive applications.

The type of `pResource` must be one of the following:

- ::ID3D10Buffer: Cannot be used with `flags` set to `cudaD3D10RegisterFlagsArray`.
- ::ID3D10Texture1D: No restrictions.
- ::ID3D10Texture2D: No restrictions.
- ::ID3D10Texture3D: No restrictions.

The `flags` argument specifies the mechanism through which CUDA will access the Direct3D resource. The following values are allowed.

- *cudaD3D10RegisterFlagsNone*: Specifies that CUDA will access this resource through a `void*`. The pointer, size, and pitch for each subresource of this resource may be queried through *cudaD3D10ResourceGetMappedPointer()*, *cudaD3D10ResourceGetMappedSize()*, and *cudaD3D10ResourceGetMappedPitch()* respectively. This option is valid for all resource types.
- *cudaD3D10RegisterFlagsArray*: Specifies that CUDA will access this resource through a `CUarray` queried on a sub-resource basis through *cudaD3D10ResourceGetMappedArray()*. This option is only valid for resources of type ::ID3D10Texture1D, ::ID3D10Texture2D, and ::ID3D10Texture3D.

Not all Direct3D resources of the above types may be used for interoperability with CUDA. The following are some limitations.

- The primary rendertarget may not be registered with CUDA.
- Resources allocated as shared may not be registered with CUDA.
- Textures which are not of a format which is 1, 2, or 4 channels of 8, 16, or 32-bit integer or floating-point data cannot be shared.
- Surfaces of depth or stencil formats cannot be shared.

If Direct3D interoperability is not initialized on this context then *cudaErrorInvalidDevice* is returned. If `pResource` is of incorrect type or is already registered then *cudaErrorInvalidResourceHandle* is returned. If `pResource` cannot be registered then *cudaErrorUnknown* is returned.

See also:

cudaGraphicsD3D10RegisterResource

Note: Note that this function may also return error codes from previous, asynchronous launches.

Parameters

- **pResource** – - Resource to register
- **flags** – - Parameters for resource registration

Returns

cudaSuccess, *cudaErrorInvalidDevice*, *cudaErrorInvalidValue*, *cudaErrorInvalidResourceHandle*, *cudaErrorUnknown*

__host__ *cudaError_t* **cudaD3D10ResourceGetMappedArray**(cudaArray **ppArray, ID3D10Resource *pResource, unsigned int subResource)

Gets an array through which to access a subresource of a Direct3D resource which has been mapped for access by CUDA.

Deprecated:

This function is deprecated as of CUDA 3.0.

Returns in *ppArray an array through which the subresource of the mapped Direct3D resource pResource which corresponds to subResource may be accessed. The value set in ppArray may change every time that pResource is mapped.

If pResource is not registered, then *cudaErrorInvalidResourceHandle* is returned. If pResource was not registered with usage flags *cudaD3D10RegisterFlagsArray*, then *cudaErrorInvalidResourceHandle* is returned. If pResource is not mapped then *cudaErrorUnknown* is returned.

For usage requirements of the subResource parameter, see *cudaD3D10ResourceGetMappedPointer()*.

See also:

cudaGraphicsSubResourceGetMappedArray

Note: Note that this function may also return error codes from previous, asynchronous launches.

Parameters

- **ppArray** – - Returned array corresponding to subresource
- **pResource** – - Mapped resource to access
- **subResource** – - Subresource of pResource to access

Returns

cudaSuccess, *cudaErrorInvalidValue*, *cudaErrorInvalidResourceHandle*, *cudaErrorUnknown*

__host__ *cudaError_t* **cudaD3D10ResourceGetMappedPitch**(size_t *pPitch, size_t *pPitchSlice, ID3D10Resource *pResource, unsigned int subResource)

Gets the pitch of a subresource of a Direct3D resource which has been mapped for access by CUDA.

Deprecated:

This function is deprecated as of CUDA 3.0.

Returns in `*pPitch` and `*pPitchSlice` the pitch and Z-slice pitch of the subresource of the mapped Direct3D resource `pResource`, which corresponds to `subResource`. The values set in `pPitch` and `pPitchSlice` may change every time that `pResource` is mapped.

The pitch and Z-slice pitch values may be used to compute the location of a sample on a surface as follows.

For a 2D surface, the byte offset of the sample at position **x**, **y** from the base pointer of the surface is:

y * **pitch** + (**bytes per pixel**) * **x**

For a 3D surface, the byte offset of the sample at position **x**, **y**, **z** from the base pointer of the surface is:

z* **slicePitch** + **y** * **pitch** + (**bytes per pixel**) * **x**

Both parameters `pPitch` and `pPitchSlice` are optional and may be set to NULL.

If `pResource` is not of type ::ID3D10Texture1D, ::ID3D10Texture2D, or ::ID3D10Texture3D, or if `pResource` has not been registered for use with CUDA, then *cudaErrorInvalidResourceHandle* is returned. If `pResource` was not registered with usage flags *cudaD3D10RegisterFlagsNone*, then *cudaErrorInvalidResourceHandle* is returned. If `pResource` is not mapped for access by CUDA then *cudaErrorUnknown* is returned.

For usage requirements of the `subResource` parameter see *cudaD3D10ResourceGetMappedPointer()*.

See also:

cudaGraphicsSubResourceGetMappedArray

Note: Note that this function may also return error codes from previous, asynchronous launches.

Parameters

- **`pPitch`** – - Returned pitch of subresource
- **`pPitchSlice`** – - Returned Z-slice pitch of subresource
- **`pResource`** – - Mapped resource to access
- **`subResource`** – - Subresource of pResource to access

Returns

cudaSuccess, *cudaErrorInvalidValue*, *cudaErrorInvalidResourceHandle*, *cudaErrorUnknown*

__host__ *cudaError_t* **cudaD3D10ResourceGetMappedPointer**(void **pPointer, ID3D10Resource *pResource, unsigned int subResource)

Gets a pointer through which to access a subresource of a Direct3D resource which has been mapped for access by CUDA.

Deprecated:

This function is deprecated as of CUDA 3.0.

Returns in `*pPointer` the base pointer of the subresource of the mapped Direct3D resource `pResource` which corresponds to `subResource`. The value set in `pPointer` may change every time that `pResource` is mapped.

If `pResource` is not registered, then *cudaErrorInvalidResourceHandle* is returned. If `pResource` was not registered with usage flags *cudaD3D9RegisterFlagsNone*, then *cudaErrorInvalidResourceHandle* is returned. If `pResource` is not mapped then *cudaErrorUnknown* is returned.

If `pResource` is of type ::ID3D10Buffer then `subResource` must be 0. If `pResource` is of any other type, then the value of `subResource` must come from the subresource calculation in ::D3D10CalcSubResource().

See also:

cudaGraphicsResourceGetMappedPointer

Note: Note that this function may also return error codes from previous, asynchronous launches.

Parameters

- **pPointer** – - Returned pointer corresponding to subresource
- **pResource** – - Mapped resource to access
- **subResource** – - Subresource of pResource to access

Returns

cudaSuccess, *cudaErrorInvalidValue*, *cudaErrorInvalidResourceHandle*, *cudaErrorUnknown*

__host__ *cudaError_t* **cudaD3D10ResourceGetMappedSize**(size_t *pSize, ID3D10Resource *pResource, unsigned int subResource)

Gets the size of a subresource of a Direct3D resource which has been mapped for access by CUDA.

Deprecated:

This function is deprecated as of CUDA 3.0.

Returns in `*pSize` the size of the subresource of the mapped Direct3D resource `pResource` which corresponds to `subResource`. The value set in `pSize` may change every time that `pResource` is mapped.

If pResource has not been registered for use with CUDA then ::cudaErrorInvalidHandle is returned. If pResource was not registered with usage flags *cudaD3D10RegisterFlagsNone*, then *cudaErrorInvalidResourceHandle* is returned. If pResource is not mapped for access by CUDA then *cudaErrorUnknown* is returned.

For usage requirements of the subResource parameter see *cudaD3D10ResourceGetMappedPointer()*.

See also:

cudaGraphicsResourceGetMappedPointer

Note: Note that this function may also return error codes from previous, asynchronous launches.

Parameters

- **pSize** – - Returned size of subresource
- **pResource** – - Mapped resource to access
- **subResource** – - Subresource of pResource to access

Returns

cudaSuccess, *cudaErrorInvalidValue*, *cudaErrorInvalidResourceHandle*, *cudaErrorUnknown*

__host__ *cudaError_t* **cudaD3D10ResourceGetSurfaceDimensions**(size_t *pWidth, size_t *pHeight, size_t *pDepth, ID3D10Resource *pResource, unsigned int subResource)

Gets the dimensions of a registered Direct3D surface.

Deprecated:

This function is deprecated as of CUDA 3.0.

Returns in *pWidth, *pHeight, and *pDepth the dimensions of the subresource of the mapped Direct3D resource pResource which corresponds to subResource.

Since anti-aliased surfaces may have multiple samples per pixel, it is possible that the dimensions of a resource will be an integer factor larger than the dimensions reported by the Direct3D runtime.

The parameters pWidth, pHeight, and pDepth are optional. For 2D surfaces, the value returned in *pDepth will be 0.

If pResource is not of type ::ID3D10Texture1D, ::ID3D10Texture2D, or ::ID3D10Texture3D, or if pResource has not been registered for use with CUDA, then ::cudaErrorInvalidHandle is returned.

For usage requirements of subResource parameters see *cudaD3D10ResourceGetMappedPointer()*.

See also:

cudaGraphicsSubResourceGetMappedArray

Note: Note that this function may also return error codes from previous, asynchronous launches.

Parameters

- **pWidth** – - Returned width of surface
- **pHeight** – - Returned height of surface
- **pDepth** – - Returned depth of surface
- **pResource** – - Registered resource to access
- **subResource** – - Subresource of pResource to access

Returns

cudaSuccess, *cudaErrorInvalidValue*, *cudaErrorInvalidResourceHandle*,

__host__ *cudaError_t* **cudaD3D10ResourceSetMapFlags**(ID3D10Resource *pResource, unsigned int flags)

Set usage flags for mapping a Direct3D resource.

Deprecated:

This function is deprecated as of CUDA 3.0.

Set usage flags for mapping the Direct3D resource `pResource`.

Changes to flags will take effect the next time `pResource` is mapped. The `flags` argument may be any of the following:

- *cudaD3D10MapFlagsNone*: Specifies no hints about how this resource will be used. It is therefore assumed that this resource will be read from and written to by CUDA kernels. This is the default value.
- *cudaD3D10MapFlagsReadOnly*: Specifies that CUDA kernels which access this resource will not write to this resource.
- *cudaD3D10MapFlagsWriteDiscard*: Specifies that CUDA kernels which access this resource will not read from this resource and will write over the entire contents of the resource, so none of the data previously stored in the resource will be preserved.

If `pResource` has not been registered for use with CUDA then ::cudaErrorInvalidHandle is returned. If `pResource` is presently mapped for access by CUDA then *cudaErrorUnknown* is returned.

See also:

cudaGraphicsResourceSetMapFlags

Note: Note that this function may also return error codes from previous, asynchronous launches.

Parameters

- **pResource** – - Registered resource to set flags for
- **flags** – - Parameters for resource mapping

Returns

cudaSuccess, *cudaErrorInvalidValue*, *cudaErrorInvalidResourceHandle*, *cudaErrorUnknown*,

__host__ *cudaError_t* **cudaD3D10SetDirect3DDevice**(ID3D10Device *pD3D10Device, int device = -1)

Sets the Direct3D 10 device to use for interoperability with a CUDA device.

Deprecated:

This function is deprecated as of CUDA 5.0.

This function is deprecated and should no longer be used. It is no longer necessary to associate a CUDA device with a D3D10 device in order to achieve maximum interoperability performance.

This function will immediately initialize the primary context on `device` if needed.

See also:

cudaD3D10GetDevice, *cudaGraphicsD3D10RegisterResource*, *cudaDeviceReset*

Note: Note that this function may also return error codes from previous, asynchronous launches.

Parameters

- **pD3D10Device** – - Direct3D device to use for interoperability
- **device** – - The CUDA device to use. This device must be among the devices returned when querying *cudaD3D10DeviceListAll* from *cudaD3D10GetDevices*, may be set to -1 to automatically select an appropriate CUDA device.

Returns

cudaSuccess, *cudaErrorInitializationError*, *cudaErrorInvalidValue*, *cudaErrorSetOnActiveProcess*

__host__ *cudaError_t* **cudaD3D10UnmapResources**(int count, ID3D10Resource **ppResources)

Unmaps Direct3D resources.

Deprecated:

This function is deprecated as of CUDA 3.0.

Unmaps the `count` Direct3D resource in `ppResources`.

This function provides the synchronization guarantee that any CUDA kernels issued before *cudaD3D10UnmapResources()* will complete before any Direct3D calls issued after *cudaD3D10UnmapResources()* begin.

If any of `ppResources` have not been registered for use with CUDA or if `ppResources` contains any duplicate entries, then *cudaErrorInvalidResourceHandle* is returned. If any of `ppResources` are not presently mapped for access by CUDA then *cudaErrorUnknown* is returned.

See also:

cudaGraphicsUnmapResources

Note: Note that this function may also return error codes from previous, asynchronous launches.

Parameters

- **count** – - Number of resources to unmap for CUDA
- **ppResources** – - Resources to unmap for CUDA

Returns

cudaSuccess, *cudaErrorInvalidResourceHandle*, *cudaErrorUnknown*

__host__ *cudaError_t* **cudaD3D10UnregisterResource**(ID3D10Resource *pResource)

Unregisters a Direct3D resource.

Deprecated:

This function is deprecated as of CUDA 3.0.

Unregisters the Direct3D resource `resource` so it is not accessible by CUDA unless registered again.

If `pResource` is not registered, then *cudaErrorInvalidResourceHandle* is returned.

See also:

cudaGraphicsUnregisterResource

Note: Note that this function may also return error codes from previous, asynchronous launches.

Parameters

pResource – - Resource to unregister

Returns

cudaSuccess, *cudaErrorInvalidResourceHandle*, *cudaErrorUnknown*

6.7. Direct3D 11 Interoperability

impl_private

impl_private

This section describes the Direct3D 11 interoperability functions of the CUDA runtime application programming interface. Note that mapping of Direct3D 11 resources is performed with the graphics API agnostic, resource mapping interface described in *Graphics Interopability*.

Enumerations

cudaD3D11DeviceList
CUDA devices corresponding to a D3D11 device.

Functions

__host__ cudaError_t *cudaD3D11GetDevice*(int *device, IDXGIAdapter *pAdapter)
Gets the device number for an adapter.

__host__ cudaError_t *cudaD3D11GetDevices*(unsigned int *pCudaDeviceCount, int *pCudaDevices, unsigned int cudaDeviceCount, ID3D11Device *pD3D11Device, enum cudaD3D11DeviceList deviceList)
Gets the CUDA devices corresponding to a Direct3D 11 device.

__host__ cudaError_t *cudaGraphicsD3D11RegisterResource*(struct cudaGraphicsResource **resource, ID3D11Resource *pD3DResource, unsigned int flags)
Register a Direct3D 11 resource for access by CUDA.

6.7.1. Enumerations

enum **cudaD3D11DeviceList**
CUDA devices corresponding to a D3D11 device.

Values:

enumerator **cudaD3D11DeviceListAll** = 1
The CUDA devices for all GPUs used by a D3D11 device.

enumerator **cudaD3D11DeviceListCurrentFrame** = 2
The CUDA devices for the GPUs used by a D3D11 device in its currently rendering frame.

enumerator **cudaD3D11DeviceListNextFrame** = 3
The CUDA devices for the GPUs to be used by a D3D11 device in the next frame

6.7.2. Functions

__host__ *cudaError_t* **cudaD3D11GetDevice**(int *device, IDXGIAdapter *pAdapter)
Gets the device number for an adapter.

Returns in `*device` the CUDA-compatible device corresponding to the adapter `pAdapter` obtained from ::IDXGIFactory::EnumAdapters. This call will succeed only if a device on adapter `pAdapter` is CUDA-compatible.

See also:

cudaGraphicsUnregisterResource, *cudaGraphicsMapResources*, *cudaGraphicsSubResourceGetMappedArray*, *cudaGraphicsResourceGetMappedPointer*, ::cuD3D11GetDevice

Note: Note that this function may also return error codes from previous, asynchronous launches.

Parameters

- **device** – - Returns the device corresponding to pAdapter
- **pAdapter** – - D3D11 adapter to get device for

Returns

cudaSuccess, *cudaErrorInvalidValue*, *cudaErrorUnknown*

__host__ *cudaError_t* **cudaD3D11GetDevices**(unsigned int *pCudaDeviceCount, int *pCudaDevices, unsigned int cudaDeviceCount, ID3D11Device *pD3D11Device, enum *cudaD3D11DeviceList* deviceList)

Gets the CUDA devices corresponding to a Direct3D 11 device.

Returns in `*pCudaDeviceCount` the number of CUDA-compatible devices corresponding to the Direct3D 11 device `pD3D11Device`. Also returns in `*pCudaDevices` at most `cudaDeviceCount` of the the CUDA-compatible devices corresponding to the Direct3D 11 device `pD3D11Device`.

If any of the GPUs being used to render `pDevice` are not CUDA capable then the call will return *cudaErrorNoDevice*.

See also:

cudaGraphicsUnregisterResource, *cudaGraphicsMapResources*, *cudaGraphicsSubResourceGetMappedArray*, *cudaGraphicsResourceGetMappedPointer*, ::cuD3D11GetDevices

Note: Note that this function may also return error codes from previous, asynchronous launches.

Parameters

- **pCudaDeviceCount** – - Returned number of CUDA devices corresponding to `pD3D11Device`
- **pCudaDevices** – - Returned CUDA devices corresponding to `pD3D11Device`
- **cudaDeviceCount** – - The size of the output device array `pCudaDevices`
- **pD3D11Device** – - Direct3D 11 device to query for CUDA devices
- **deviceList** – - The set of devices to return. This set may be *cudaD3D11DeviceListAll* for all devices, *cudaD3D11DeviceListCurrentFrame* for the devices used to render the current frame (in SLI), or *cudaD3D11DeviceListNextFrame* for the devices used to render the next frame (in SLI).

Returns

cudaSuccess, *cudaErrorNoDevice*, *cudaErrorUnknown*

__host__ *cudaError_t* **cudaGraphicsD3D11RegisterResource**(struct cudaGraphicsResource **resource, ID3D11Resource *pD3DResource, unsigned int flags)

Register a Direct3D 11 resource for access by CUDA.

Registers the Direct3D 11 resource `pD3DResource` for access by CUDA.

If this call is successful, then the application will be able to map and unmap this resource until it is unregistered through *cudaGraphicsUnregisterResource()*. Also on success, this call will increase the internal reference count on `pD3DResource`. This reference count will be decremented when this resource is unregistered through *cudaGraphicsUnregisterResource()*.

This call potentially has a high-overhead and should not be called every frame in interactive applications.

The type of `pD3DResource` must be one of the following.

- ::ID3D11Buffer: may be accessed via a device pointer
- ::ID3D11Texture1D: individual subresources of the texture may be accessed via arrays
- ::ID3D11Texture2D: individual subresources of the texture may be accessed via arrays
- ::ID3D11Texture3D: individual subresources of the texture may be accessed via arrays

The `flags` argument may be used to specify additional parameters at register time. The valid values for this parameter are

- *cudaGraphicsRegisterFlagsNone*: Specifies no hints about how this resource will be used.
- *cudaGraphicsRegisterFlagsSurfaceLoadStore*: Specifies that CUDA will bind this resource to a surface reference.
- *cudaGraphicsRegisterFlagsTextureGather*: Specifies that CUDA will perform texture gather operations on this resource.

Not all Direct3D resources of the above types may be used for interoperability with CUDA. The following are some limitations.

- The primary rendertarget may not be registered with CUDA.
- Textures which are not of a format which is 1, 2, or 4 channels of 8, 16, or 32-bit integer or floating-point data cannot be shared.
- Surfaces of depth or stencil formats cannot be shared.

A complete list of supported DXGI formats is as follows. For compactness the notation A_{B,C,D} represents A_B, A_C, and A_D.

- DXGI_FORMAT_A8_UNORM
- DXGI_FORMAT_B8G8R8A8_UNORM
- DXGI_FORMAT_B8G8R8X8_UNORM
- DXGI_FORMAT_R16_FLOAT
- DXGI_FORMAT_R16G16B16A16_{FLOAT,SINT,SNORM,UINT,UNORM}
- DXGI_FORMAT_R16G16_{FLOAT,SINT,SNORM,UINT,UNORM}

- DXGI_FORMAT_R16_{SINT,SNORM,UINT,UNORM}
- DXGI_FORMAT_R32_FLOAT
- DXGI_FORMAT_R32G32B32A32_{FLOAT,SINT,UINT}
- DXGI_FORMAT_R32G32_{FLOAT,SINT,UINT}
- DXGI_FORMAT_R32_{SINT,UINT}
- DXGI_FORMAT_R8G8B8A8_{SINT,SNORM,UINT,UNORM,UNORM_SRGB}
- DXGI_FORMAT_R8G8_{SINT,SNORM,UINT,UNORM}
- DXGI_FORMAT_R8_{SINT,SNORM,UINT,UNORM}

If `pD3DResource` is of incorrect type or is already registered, then *cudaErrorInvalidResourceHandle* is returned. If `pD3DResource` cannot be registered, then *cudaErrorUnknown* is returned.

See also:

cudaGraphicsUnregisterResource, *cudaGraphicsMapResources*, *cudaGraphicsSubResourceGetMappedArray*, *cudaGraphicsResourceGetMappedPointer*, ::cuGraphicsD3D11RegisterResource

Note: Note that this function may also return error codes from previous, asynchronous launches.

Parameters

- **`resource`** – - Pointer to returned resource handle
- **`pD3DResource`** – - Direct3D resource to register
- **`flags`** – - Parameters for resource registration

Returns

cudaSuccess, *cudaErrorInvalidDevice*, *cudaErrorInvalidValue*, *cudaErrorInvalidResourceHandle*, *cudaErrorUnknown*

6.8. Direct3D 11 Interoperability [DEPRECATED]

This section describes deprecated Direct3D 11 interoperability functions.

Functions

__host__ cudaError_t *cudaD3D11GetDirect3DDevice*(ID3D11Device **ppD3D11Device)
Gets the Direct3D device against which the current CUDA context was created.

__host__ cudaError_t *cudaD3D11SetDirect3DDevice*(ID3D11Device *pD3D11Device, int device=-1)
Sets the Direct3D 11 device to use for interoperability with a CUDA device.

6.8.1. Functions

__host__ *cudaError_t* **cudaD3D11GetDirect3DDevice**(ID3D11Device **ppD3D11Device)

Gets the Direct3D device against which the current CUDA context was created.

Deprecated:

This function is deprecated as of CUDA 5.0.

This function is deprecated and should no longer be used. It is no longer necessary to associate a CUDA device with a D3D11 device in order to achieve maximum interoperability performance.

See also:

cudaD3D11SetDirect3DDevice

Note: Note that this function may also return error codes from previous, asynchronous launches.

Parameters

ppD3D11Device – - Returns the Direct3D device for this thread

Returns

cudaSuccess, *cudaErrorUnknown*

__host__ *cudaError_t* **cudaD3D11SetDirect3DDevice**(ID3D11Device *pD3D11Device, int device = -1)

Sets the Direct3D 11 device to use for interoperability with a CUDA device.

Deprecated:

This function is deprecated as of CUDA 5.0.

This function is deprecated and should no longer be used. It is no longer necessary to associate a CUDA device with a D3D11 device in order to achieve maximum interoperability performance.

This function will immediately initialize the primary context on `device` if needed.

See also:

cudaD3D11GetDevice, *cudaGraphicsD3D11RegisterResource*, *cudaDeviceReset*

Note: Note that this function may also return error codes from previous, asynchronous launches.

Parameters

- **pD3D11Device** – - Direct3D device to use for interoperability
- **device** – - The CUDA device to use. This device must be among the devices returned when querying *cudaD3D11DeviceListAll* from *cudaD3D11GetDevices*, may be set to -1 to automatically select an appropriate CUDA device.

Returns
cudaSuccess, *cudaErrorInitializationError*, *cudaErrorInvalidValue*, *cudaErrorSetOnActiveProcess*

6.9. Direct3D 9 Interoperability

impl_private

impl_private

This section describes the Direct3D 9 interoperability functions of the CUDA runtime application programming interface. Note that mapping of Direct3D 9 resources is performed with the graphics API agnostic, resource mapping interface described in *Graphics Interopability*.

Enumerations

cudaD3D9DeviceList
CUDA devices corresponding to a D3D9 device.

Functions

__host__ cudaError_t *cudaD3D9GetDevice*(int *device, const char *pszAdapterName)
Gets the device number for an adapter.

__host__ cudaError_t *cudaD3D9GetDevices*(unsigned int *pCudaDeviceCount, int *pCudaDevices, unsigned int cudaDeviceCount, IDirect3DDevice9 *pD3D9Device, enum cudaD3D9DeviceList deviceList)
Gets the CUDA devices corresponding to a Direct3D 9 device.

__host__ cudaError_t *cudaD3D9GetDirect3DDevice*(IDirect3DDevice9 **ppD3D9Device)
Gets the Direct3D device against which the current CUDA context was created.

__host__ cudaError_t *cudaD3D9SetDirect3DDevice*(IDirect3DDevice9 *pD3D9Device, int device=-1)
Sets the Direct3D 9 device to use for interoperability with a CUDA device.

__host__ cudaError_t *cudaGraphicsD3D9RegisterResource*(struct cudaGraphicsResource **resource, IDirect3DResource9 *pD3DResource, unsigned int flags)
Register a Direct3D 9 resource for access by CUDA.

6.9.1. Enumerations

enum **cudaD3D9DeviceList**

CUDA devices corresponding to a D3D9 device.

Values:

enumerator **cudaD3D9DeviceListAll** = 1

The CUDA devices for all GPUs used by a D3D9 device.

enumerator **cudaD3D9DeviceListCurrentFrame** = 2

The CUDA devices for the GPUs used by a D3D9 device in its currently rendering frame.

enumerator **cudaD3D9DeviceListNextFrame** = 3

The CUDA devices for the GPUs to be used by a D3D9 device in the next frame

6.9.2. Functions

__host__ *cudaError_t* **cudaD3D9GetDevice**(int *device, const char *pszAdapterName)

Gets the device number for an adapter.

Returns in `*device` the CUDA-compatible device corresponding to the adapter name `pszAdapterName` obtained from ::EnumDisplayDevices or ::IDirect3D9::GetAdapterIdentifier(). If no device on the adapter with name `pszAdapterName` is CUDA-compatible then the call will fail.

See also:

cudaD3D9SetDirect3DDevice, *cudaGraphicsD3D9RegisterResource*, ::cuD3D9GetDevice

Note: Note that this function may also return error codes from previous, asynchronous launches.

Parameters

- **device** – - Returns the device corresponding to pszAdapterName
- **pszAdapterName** – - D3D9 adapter to get device for

Returns

cudaSuccess, *cudaErrorInvalidValue*, *cudaErrorUnknown*

__host__ *cudaError_t* **cudaD3D9GetDevices**(unsigned int *pCudaDeviceCount, int *pCudaDevices, unsigned int cudaDeviceCount, IDirect3DDevice9 *pD3D9Device, enum *cudaD3D9DeviceList* deviceList)

Gets the CUDA devices corresponding to a Direct3D 9 device.

Returns in `*pCudaDeviceCount` the number of CUDA-compatible devices corresponding to the Direct3D 9 device `pD3D9Device`. Also returns in `*pCudaDevices` at most `cudaDeviceCount` of the the CUDA-compatible devices corresponding to the Direct3D 9 device `pD3D9Device`.

If any of the GPUs being used to render `pDevice` are not CUDA capable then the call will return *cudaErrorNoDevice*.

See also:

cudaGraphicsUnregisterResource, *cudaGraphicsMapResources*, *cudaGraphicsSubResourceGetMappedArray*, *cudaGraphicsResourceGetMappedPointer*, ::cuD3D9GetDevices

Note: Note that this function may also return error codes from previous, asynchronous launches.

Parameters

- **pCudaDeviceCount** – - Returned number of CUDA devices corresponding to pD3D9Device
- **pCudaDevices** – - Returned CUDA devices corresponding to pD3D9Device
- **cudaDeviceCount** – - The size of the output device array pCudaDevices
- **pD3D9Device** – - Direct3D 9 device to query for CUDA devices
- **deviceList** – - The set of devices to return. This set may be *cudaD3D9DeviceListAll* for all devices, *cudaD3D9DeviceListCurrentFrame* for the devices used to render the current frame (in SLI), or *cudaD3D9DeviceListNextFrame* for the devices used to render the next frame (in SLI).

Returns

cudaSuccess, *cudaErrorNoDevice*, *cudaErrorUnknown*

__host__ *cudaError_t* **cudaD3D9GetDirect3DDevice**(IDirect3DDevice9 **ppD3D9Device)

Gets the Direct3D device against which the current CUDA context was created.

Returns in *ppD3D9Device the Direct3D device against which this CUDA context was created in *cudaD3D9SetDirect3DDevice()*.

See also:

cudaD3D9SetDirect3DDevice, ::cuD3D9GetDirect3DDevice

Note: Note that this function may also return error codes from previous, asynchronous launches.

Parameters

ppD3D9Device – - Returns the Direct3D device for this thread

Returns

cudaSuccess, *cudaErrorInvalidGraphicsContext*, *cudaErrorUnknown*

__host__ *cudaError_t* **cudaD3D9SetDirect3DDevice**(IDirect3DDevice9 *pD3D9Device, int device = -1)

Sets the Direct3D 9 device to use for interoperability with a CUDA device.

Records pD3D9Device as the Direct3D 9 device to use for Direct3D 9 interoperability with the CUDA device device and sets device as the current device for the calling host thread.

This function will immediately initialize the primary context on device if needed.

If device has already been initialized then this call will fail with the error *cudaErrorSetOnActiveProcess*. In this case it is necessary to reset device using *cudaDeviceReset()* before Direct3D 9 interoperability on device may be enabled.

Successfully initializing CUDA interoperability with pD3D9Device will increase the internal reference count on pD3D9Device. This reference count will be decremented when device is reset using *cudaDeviceReset()*.

Note that this function is never required for correct functionality. Use of this function will result in accelerated interoperability only when the operating system is Windows Vista or Windows 7, and

the device pD3DDdevice is not an IDirect3DDevice9Ex. In all other cirumstances, this function is not necessary.

See also:

cudaD3D9GetDevice, *cudaGraphicsD3D9RegisterResource*, *cudaDeviceReset*

Note: Note that this function may also return error codes from previous, asynchronous launches.

Parameters

- **pD3D9Device** – - Direct3D device to use for this thread
- **device** – - The CUDA device to use. This device must be among the devices returned when querying *cudaD3D9DeviceListAll* from *cudaD3D9GetDevices*, may be set to -1 to automatically select an appropriate CUDA device.

Returns

cudaSuccess, *cudaErrorInitializationError*, *cudaErrorInvalidValue*, *cudaErrorSetOnActiveProcess*

__host__ *cudaError_t* **cudaGraphicsD3D9RegisterResource**(struct cudaGraphicsResource **resource, IDirect3DResource9 *pD3DResource, unsigned int flags)

Register a Direct3D 9 resource for access by CUDA.

Registers the Direct3D 9 resource pD3DResource for access by CUDA.

If this call is successful then the application will be able to map and unmap this resource until it is unregistered through *cudaGraphicsUnregisterResource()*. Also on success, this call will increase the internal reference count on pD3DResource. This reference count will be decremented when this resource is unregistered through *cudaGraphicsUnregisterResource()*.

This call potentially has a high-overhead and should not be called every frame in interactive applications.

The type of pD3DResource must be one of the following.

- ::IDirect3DVertexBuffer9: may be accessed through a device pointer
- ::IDirect3DIndexBuffer9: may be accessed through a device pointer
- ::IDirect3DSurface9: may be accessed through an array. Only stand-alone objects of type ::IDirect3DSurface9 may be explicitly shared. In particular, individual mipmap levels and faces of cube maps may not be registered directly. To access individual surfaces associated with a texture, one must register the base texture object.
- ::IDirect3DBaseTexture9: individual surfaces on this texture may be accessed through an array.

The flags argument may be used to specify additional parameters at register time. The valid values for this parameter are

- *cudaGraphicsRegisterFlagsNone*: Specifies no hints about how this resource will be used.

- *cudaGraphicsRegisterFlagsSurfaceLoadStore*: Specifies that CUDA will bind this resource to a surface reference.
- *cudaGraphicsRegisterFlagsTextureGather*: Specifies that CUDA will perform texture gather operations on this resource.

Not all Direct3D resources of the above types may be used for interoperability with CUDA. The following are some limitations.

- The primary rendertarget may not be registered with CUDA.
- Resources allocated as shared may not be registered with CUDA.
- Textures which are not of a format which is 1, 2, or 4 channels of 8, 16, or 32-bit integer or floating-point data cannot be shared.
- Surfaces of depth or stencil formats cannot be shared.

A complete list of supported formats is as follows:

- D3DFMT_L8
- D3DFMT_L16
- D3DFMT_A8R8G8B8
- D3DFMT_X8R8G8B8
- D3DFMT_G16R16
- D3DFMT_A8B8G8R8
- D3DFMT_A8
- D3DFMT_A8L8
- D3DFMT_Q8W8V8U8
- D3DFMT_V16U16
- D3DFMT_A16B16G16R16F
- D3DFMT_A16B16G16R16
- D3DFMT_R32F
- D3DFMT_G16R16F
- D3DFMT_A32B32G32R32F
- D3DFMT_G32R32F
- D3DFMT_R16F

If `pD3DResource` is of incorrect type or is already registered, then *cudaErrorInvalidResourceHandle* is returned. If `pD3DResource` cannot be registered, then *cudaErrorUnknown* is returned.

See also:

cudaD3D9SetDirect3DDevice, *cudaGraphicsUnregisterResource*, *cudaGraphicsMapResources*, *cudaGraphicsSubResourceGetMappedArray*, *cudaGraphicsResourceGetMappedPointer*, ::cudaGraphicsD3D9RegisterResource

Note: Note that this function may also return error codes from previous, asynchronous launches.

Parameters

- **resource** – - Pointer to returned resource handle
- **pD3DResource** – - Direct3D resource to register
- **flags** – - Parameters for resource registration

Returns

cudaSuccess, *cudaErrorInvalidDevice*, *cudaErrorInvalidValue*, *cudaErrorInvalidResourceHandle*, *cudaErrorUnknown*

6.10. Direct3D 9 Interoperability [DEPRECATED]

This section describes deprecated Direct3D 9 interoperability functions.

Enumerations

cudaD3D9MapFlags
CUDA D3D9 Map Flags.

cudaD3D9RegisterFlags
CUDA D3D9 Register Flags.

Functions

__host__ cudaError_t *cudaD3D9Begin*(IDirect3DDevice9 *pDevice)

__host__ cudaError_t *cudaD3D9End*(void)

__host__ cudaError_t *cudaD3D9MapResources*(int count, IDirect3DResource9 **ppResources)
Map Direct3D resources for access by CUDA.

__host__ cudaError_t *cudaD3D9MapVertexBuffer*(void **dptr, IDirect3DVertexBuffer9 *pVB)

__host__ cudaError_t *cudaD3D9RegisterResource*(IDirect3DResource9 *pResource, unsigned int flags)
Registers a Direct3D resource for access by CUDA.

__host__ cudaError_t *cudaD3D9RegisterVertexBuffer*(IDirect3DVertexBuffer9 *pVB)

__host__ cudaError_t *cudaD3D9ResourceGetMappedArray*(cudaArray **ppArray, IDirect3DResource9 *pResource, unsigned int face, unsigned int level)
Get an array through which to access a subresource of a Direct3D resource which has been mapped for access by CUDA.

__host__ cudaError_t *cudaD3D9ResourceGetMappedPitch*(size_t *pPitch, size_t *pPitchSlice, IDirect3DResource9 *pResource, unsigned int face, unsigned int level)
Get the pitch of a subresource of a Direct3D resource which has been mapped for access by CUDA.

__host__ cudaError_t *cudaD3D9ResourceGetMappedPointer*(void **pPointer, IDirect3DResource9 *pResource, unsigned int face, unsigned int level)
Get a pointer through which to access a subresource of a Direct3D resource which has been mapped for access by CUDA.

__host__ cudaError_t *cudaD3D9ResourceGetMappedSize*(size_t *pSize, IDirect3DResource9 *pResource, unsigned int face, unsigned int level)
Get the size of a subresource of a Direct3D resource which has been mapped for access by CUDA.

__host__ cudaError_t *cudaD3D9ResourceGetSurfaceDimensions*(size_t *pWidth, size_t *pHeight, size_t *pDepth, IDirect3DResource9 *pResource, unsigned int face, unsigned int level)
Get the dimensions of a registered Direct3D surface.

__host__ cudaError_t *cudaD3D9ResourceSetMapFlags*(IDirect3DResource9 *pResource, unsigned int flags)
Set usage flags for mapping a Direct3D resource.

__host__ cudaError_t *cudaD3D9UnmapResources*(int count, IDirect3DResource9 **ppResources)
Unmap Direct3D resources for access by CUDA.

__host__ cudaError_t *cudaD3D9UnmapVertexBuffer*(IDirect3DVertexBuffer9 *pVB)

__host__ cudaError_t *cudaD3D9UnregisterResource*(IDirect3DResource9 *pResource)
Unregisters a Direct3D resource for access by CUDA.

__host__ cudaError_t *cudaD3D9UnregisterVertexBuffer*(IDirect3DVertexBuffer9 *pVB)

6.10.1. Enumerations

enum **cudaD3D9MapFlags**

CUDA D3D9 Map Flags.

Values:

enumerator **cudaD3D9MapFlagsNone** = 0

Default; Assume resource can be read/written.

enumerator **cudaD3D9MapFlagsReadOnly** = 1

CUDA kernels will not write to this resource.

enumerator **cudaD3D9MapFlagsWriteDiscard** = 2

CUDA kernels will only write to and will not read from this resource.

enum **cudaD3D9RegisterFlags**

CUDA D3D9 Register Flags.

Values:

enumerator **cudaD3D9RegisterFlagsNone** = 0

Default; Resource can be accessed througa void*.

enumerator **cudaD3D9RegisterFlagsArray** = 1

Resource can be accessed through a CUarray*.

6.10.2. Functions

__host__ *cudaError_t* **cudaD3D9Begin**(IDirect3DDevice9 *pDevice)

__host__ *cudaError_t* **cudaD3D9End**(void)

__host__ *cudaError_t* **cudaD3D9MapResources**(int count, IDirect3DResource9 **ppResources)

Map Direct3D resources for access by CUDA.

Deprecated:

This function is deprecated as of CUDA 3.0.

Maps the `count` Direct3D resources in `ppResources` for access by CUDA.

The resources in `ppResources` may be accessed in CUDA kernels until they are unmapped. Direct3D should not access any resources while they are mapped by CUDA. If an application does so, the results are undefined.

This function provides the synchronization guarantee that any Direct3D calls issued before *cudaD3D9MapResources()* will complete before any CUDA kernels issued after *cudaD3D9MapResources()* begin.

If any of `ppResources` have not been registered for use with CUDA or if `ppResources` contains any duplicate entries then *cudaErrorInvalidResourceHandle* is returned. If any of `ppResources` are presently mapped for access by CUDA then *cudaErrorUnknown* is returned.

See also:

cudaGraphicsMapResources

Note: Note that this function may also return error codes from previous, asynchronous launches.

Parameters

- **count** – - Number of resources to map for CUDA
- **ppResources** – - Resources to map for CUDA

Returns

cudaSuccess, *cudaErrorInvalidResourceHandle*, *cudaErrorUnknown*

__host__ *cudaError_t* **cudaD3D9MapVertexBuffer**(void **dptr, IDirect3DVertexBuffer9 *pVB)

__host__ *cudaError_t* **cudaD3D9RegisterResource**(IDirect3DResource9 *pResource, unsigned int flags)

Registers a Direct3D resource for access by CUDA.

Deprecated:

This function is deprecated as of CUDA 3.0.

Registers the Direct3D resource `pResource` for access by CUDA.

If this call is successful, then the application will be able to map and unmap this resource until it is unregistered through *cudaD3D9UnregisterResource()*. Also on success, this call will increase the internal reference count on `pResource`. This reference count will be decremented when this resource is unregistered through *cudaD3D9UnregisterResource()*.

This call potentially has a high-overhead and should not be called every frame in interactive applications.

The type of `pResource` must be one of the following.

- ::IDirect3DVertexBuffer9: No notes.
- ::IDirect3DIndexBuffer9: No notes.
- ::IDirect3DSurface9: Only stand-alone objects of type ::IDirect3DSurface9 may be explicitly shared. In particular, individual mipmap levels and faces of cube maps may not be registered directly. To access individual surfaces associated with a texture, one must register the base texture object.
- ::IDirect3DBaseTexture9: When a texture is registered, all surfaces associated with all mipmap levels of all faces of the texture will be accessible to CUDA.

The `flags` argument specifies the mechanism through which CUDA will access the Direct3D resource. The following value is allowed:

- *cudaD3D9RegisterFlagsNone*: Specifies that CUDA will access this resource through a `void*`. The pointer, size, and pitch for each subresource of this resource may be queried through *cudaD3D9ResourceGetMappedPointer()*, *cudaD3D9ResourceGetMappedSize()*, and *cudaD3D9ResourceGetMappedPitch()* respectively. This option is valid for all resource types.

Not all Direct3D resources of the above types may be used for interoperability with CUDA. The following are some limitations:

- The primary rendertarget may not be registered with CUDA.
- Resources allocated as shared may not be registered with CUDA.
- Any resources allocated in ::D3DPOOL_SYSTEMMEM or ::D3DPOOL_MANAGED may not be registered with CUDA.
- Textures which are not of a format which is 1, 2, or 4 channels of 8, 16, or 32-bit integer or floating-point data cannot be shared.
- Surfaces of depth or stencil formats cannot be shared.

If Direct3D interoperability is not initialized on this context, then *cudaErrorInvalidDevice* is returned. If `pResource` is of incorrect type (e.g, is a non-stand-alone ::IDirect3DSurface9) or is

already registered, then *cudaErrorInvalidResourceHandle* is returned. If `pResource` cannot be registered then *cudaErrorUnknown* is returned.

See also:

cudaGraphicsD3D9RegisterResource

Note: Note that this function may also return error codes from previous, asynchronous launches.

Parameters

- **`pResource`** – - Resource to register
- **`flags`** – - Parameters for resource registration

Returns

cudaSuccess, *cudaErrorInvalidValue*, *cudaErrorInvalidResourceHandle*, *cudaErrorUnknown*

__host__ *cudaError_t* **`cudaD3D9RegisterVertexBuffer`**(IDirect3DVertexBuffer9 *pVB)

__host__ *cudaError_t* **`cudaD3D9ResourceGetMappedArray`**(cudaArray **ppArray,
IDirect3DResource9 *pResource,
unsigned int face, unsigned int level)

Get an array through which to access a subresource of a Direct3D resource which has been mapped for access by CUDA.

Deprecated:

This function is deprecated as of CUDA 3.0.

Returns in `*pArray` an array through which the subresource of the mapped Direct3D resource `pResource`, which corresponds to `face` and `level` may be accessed. The value set in `pArray` may change every time that `pResource` is mapped.

If `pResource` is not registered then *cudaErrorInvalidResourceHandle* is returned. If `pResource` was not registered with usage flags *cudaD3D9RegisterFlagsArray*, then *cudaErrorInvalidResourceHandle* is returned. If `pResource` is not mapped, then *cudaErrorUnknown* is returned.

For usage requirements of `face` and `level` parameters, see *cudaD3D9ResourceGetMappedPointer()*.

See also:

cudaGraphicsSubResourceGetMappedArray

Note: Note that this function may also return error codes from previous, asynchronous launches.

Parameters

- **`ppArray`** – - Returned array corresponding to subresource

- **pResource** – - Mapped resource to access
- **face** – - Face of resource to access
- **level** – - Level of resource to access

Returns

cudaSuccess, *cudaErrorInvalidResourceHandle*, *cudaErrorUnknown*

__host__ *cudaError_t* **cudaD3D9ResourceGetMappedPitch**(size_t *pPitch, size_t *pPitchSlice, IDirect3DResource9 *pResource, unsigned int face, unsigned int level)

Get the pitch of a subresource of a Direct3D resource which has been mapped for access by CUDA.

Deprecated:

This function is deprecated as of CUDA 3.0.

Returns in `*pPitch` and `*pPitchSlice` the pitch and Z-slice pitch of the subresource of the mapped Direct3D resource `pResource`, which corresponds to `face` and `level`. The values set in `pPitch` and `pPitchSlice` may change every time that `pResource` is mapped.

The pitch and Z-slice pitch values may be used to compute the location of a sample on a surface as follows.

For a 2D surface, the byte offset of the sample at position **x**, **y** from the base pointer of the surface is:

y * **pitch** + (**bytes per pixel**) * **x**

For a 3D surface, the byte offset of the sample at position **x**, **y**, **z** from the base pointer of the surface is:

z* **slicePitch** + **y** * **pitch** + (**bytes per pixel**) * **x**

Both parameters `pPitch` and `pPitchSlice` are optional and may be set to NULL.

If `pResource` is not of type ::IDirect3DBaseTexture9 or one of its sub-types or if `pResource` has not been registered for use with CUDA, then *cudaErrorInvalidResourceHandle* is returned. If `pResource` was not registered with usage flags *cudaD3D9RegisterFlagsNone*, then *cudaErrorInvalidResourceHandle* is returned. If `pResource` is not mapped for access by CUDA then *cudaErrorUnknown* is returned.

For usage requirements of `face` and `level` parameters, see *cudaD3D9ResourceGetMappedPointer()*.

See also:

cudaGraphicsResourceGetMappedPointer

Note: Note that this function may also return error codes from previous, asynchronous launches.

Parameters

- **pPitch** – - Returned pitch of subresource

- **pPitchSlice** – - Returned Z-slice pitch of subresource
- **pResource** – - Mapped resource to access
- **face** – - Face of resource to access
- **level** – - Level of resource to access

Returns

cudaSuccess, *cudaErrorInvalidValue*, *cudaErrorInvalidResourceHandle*, *cudaErrorUnknown*

__host__ *cudaError_t* **cudaD3D9ResourceGetMappedPointer**(void **pPointer, IDirect3DResource9 *pResource, unsigned int face, unsigned int level)

Get a pointer through which to access a subresource of a Direct3D resource which has been mapped for access by CUDA.

Deprecated:

This function is deprecated as of CUDA 3.0.

Returns in `*pPointer` the base pointer of the subresource of the mapped Direct3D resource `pResource`, which corresponds to `face` and `level`. The value set in `pPointer` may change every time that `pResource` is mapped.

If `pResource` is not registered, then *cudaErrorInvalidResourceHandle* is returned. If `pResource` was not registered with usage flags *cudaD3D9RegisterFlagsNone*, then *cudaErrorInvalidResourceHandle* is returned. If `pResource` is not mapped, then *cudaErrorUnknown* is returned.

If `pResource` is of type ::IDirect3DCubeTexture9, then `face` must one of the values enumerated by type ::D3DCUBEMAP_FACES. For all other types, `face` must be 0. If `face` is invalid, then *cudaErrorInvalidValue* is returned.

If `pResource` is of type ::IDirect3DBaseTexture9, then `level` must correspond to a valid mipmap level. Only mipmap level 0 is supported for now. For all other types `level` must be 0. If `level` is invalid, then *cudaErrorInvalidValue* is returned.

See also:

cudaGraphicsResourceGetMappedPointer

Note: Note that this function may also return error codes from previous, asynchronous launches.

Parameters

- **pPointer** – - Returned pointer corresponding to subresource
- **pResource** – - Mapped resource to access
- **face** – - Face of resource to access
- **level** – - Level of resource to access

Returns

cudaSuccess, *cudaErrorInvalidValue*, *cudaErrorInvalidResourceHandle*, *cudaErrorUnknown*

__host__ *cudaError_t* **cudaD3D9ResourceGetMappedSize**(size_t *pSize, IDirect3DResource9 *pResource, unsigned int face, unsigned int level)

Get the size of a subresource of a Direct3D resource which has been mapped for access by CUDA.

Deprecated:

This function is deprecated as of CUDA 3.0.

Returns in `*pSize` the size of the subresource of the mapped Direct3D resource `pResource`, which corresponds to `face` and `level`. The value set in `pSize` may change every time that `pResource` is mapped.

If `pResource` has not been registered for use with CUDA then *cudaErrorInvalidResourceHandle* is returned. If `pResource` was not registered with usage flags *cudaD3D9RegisterFlagsNone*, then *cudaErrorInvalidResourceHandle* is returned. If `pResource` is not mapped for access by CUDA then *cudaErrorUnknown* is returned.

For usage requirements of `face` and `level` parameters, see *cudaD3D9ResourceGetMappedPointer()*.

See also:

cudaGraphicsResourceGetMappedPointer

Note: Note that this function may also return error codes from previous, asynchronous launches.

Parameters

- **pSize** – - Returned size of subresource
- **pResource** – - Mapped resource to access
- **face** – - Face of resource to access
- **level** – - Level of resource to access

Returns

cudaSuccess, *cudaErrorInvalidValue*, *cudaErrorInvalidResourceHandle*, *cudaErrorUnknown*

__host__ *cudaError_t* **cudaD3D9ResourceGetSurfaceDimensions**(size_t *pWidth, size_t *pHeight, size_t *pDepth, IDirect3DResource9 *pResource, unsigned int face, unsigned int level)

Get the dimensions of a registered Direct3D surface.

Deprecated:

This function is deprecated as of CUDA 3.0.

Returns in `*pWidth`, `*pHeight`, and `*pDepth` the dimensions of the subresource of the mapped Direct3D resource `pResource` which corresponds to `face` and `level`.

Since anti-aliased surfaces may have multiple samples per pixel, it is possible that the dimensions of a resource will be an integer factor larger than the dimensions reported by the Direct3D runtime.

The parameters `pWidth`, `pHeight`, and `pDepth` are optional. For 2D surfaces, the value returned in `*pDepth` will be 0.

If `pResource` is not of type ::IDirect3DBaseTexture9 or ::IDirect3DSurface9 or if `pResource` has not been registered for use with CUDA, then *cudaErrorInvalidResourceHandle* is returned.

For usage requirements of `face` and `level` parameters, see *cudaD3D9ResourceGetMappedPointer*.

See also:

cudaGraphicsSubResourceGetMappedArray

Note: Note that this function may also return error codes from previous, asynchronous launches.

Parameters

- **pWidth** – - Returned width of surface
- **pHeight** – - Returned height of surface
- **pDepth** – - Returned depth of surface
- **pResource** – - Registered resource to access
- **face** – - Face of resource to access
- **level** – - Level of resource to access

Returns

cudaSuccess, *cudaErrorInvalidValue*, *cudaErrorInvalidResourceHandle*,

__host__ *cudaError_t* **cudaD3D9ResourceSetMapFlags**(IDirect3DResource9 *pResource, unsigned int flags)

Set usage flags for mapping a Direct3D resource.

Deprecated:

This function is deprecated as of CUDA 3.0.

Set flags for mapping the Direct3D resource `pResource`.

Changes to flags will take effect the next time `pResource` is mapped. The `flags` argument may be any of the following:

- *cudaD3D9MapFlagsNone*: Specifies no hints about how this resource will be used. It is therefore assumed that this resource will be read from and written to by CUDA kernels. This is the default value.
- *cudaD3D9MapFlagsReadOnly*: Specifies that CUDA kernels which access this resource will not write to this resource.

- *cudaD3D9MapFlagsWriteDiscard*: Specifies that CUDA kernels which access this resource will not read from this resource and will write over the entire contents of the resource, so none of the data previously stored in the resource will be preserved.

If `pResource` has not been registered for use with CUDA, then *cudaErrorInvalidResourceHandle* is returned. If `pResource` is presently mapped for access by CUDA, then *cudaErrorUnknown* is returned.

See also:

::cudaInteropResourceSetMapFlags

Note: Note that this function may also return error codes from previous, asynchronous launches.

Parameters

- **pResource** – - Registered resource to set flags for
- **flags** – - Parameters for resource mapping

Returns

cudaSuccess, *cudaErrorInvalidValue*, *cudaErrorInvalidResourceHandle*, *cudaErrorUnknown*

__host__ *cudaError_t* **cudaD3D9UnmapResources**(int count, IDirect3DResource9 **ppResources)

Unmap Direct3D resources for access by CUDA.

Deprecated:

This function is deprecated as of CUDA 3.0.

Unmaps the `count` Direct3D resources in `ppResources`.

This function provides the synchronization guarantee that any CUDA kernels issued before *cudaD3D9UnmapResources()* will complete before any Direct3D calls issued after *cudaD3D9UnmapResources()* begin.

If any of `ppResources` have not been registered for use with CUDA or if `ppResources` contains any duplicate entries, then *cudaErrorInvalidResourceHandle* is returned. If any of `ppResources` are not presently mapped for access by CUDA then *cudaErrorUnknown* is returned.

See also:

cudaGraphicsUnmapResources

Note: Note that this function may also return error codes from previous, asynchronous launches.

Parameters

- **count** – - Number of resources to unmap for CUDA
- **ppResources** – - Resources to unmap for CUDA

Returns
cudaSuccess, *cudaErrorInvalidResourceHandle*, *cudaErrorUnknown*

__host__ *cudaError_t* **cudaD3D9UnmapVertexBuffer**(IDirect3DVertexBuffer9 *pVB)

__host__ *cudaError_t* **cudaD3D9UnregisterResource**(IDirect3DResource9 *pResource)
Unregisters a Direct3D resource for access by CUDA.

Deprecated:
This function is deprecated as of CUDA 3.0.

Unregisters the Direct3D resource `pResource` so it is not accessible by CUDA unless registered again.

If `pResource` is not registered, then *cudaErrorInvalidResourceHandle* is returned.

See also:

cudaGraphicsUnregisterResource

Note: Note that this function may also return error codes from previous, asynchronous launches.

Parameters
pResource – - Resource to unregister

Returns
cudaSuccess, *cudaErrorInvalidResourceHandle*, *cudaErrorUnknown*

__host__ *cudaError_t* **cudaD3D9UnregisterVertexBuffer**(IDirect3DVertexBuffer9 *pVB)

6.11. Driver Entry Point Access

This section describes the driver entry point access functions of CUDA runtime application programming interface.

Functions

__host__ cudaError_t *cudaGetDriverEntryPoint*(const char *symbol, void **funcPtr, unsigned long long flags, enum cudaDriverEntryPointQueryResult *driverStatus=NULL)
Returns the requested driver API function pointer.

__host__ cudaError_t *cudaGetDriverEntryPointByVersion*(const char *symbol, void **funcPtr, unsigned int cudaVersion, unsigned long long flags, enum cudaDriverEntryPointQueryResult *driverStatus=NULL)
Returns the requested driver API function pointer by CUDA version.

6.11.1. Functions

__host__ *cudaError_t* **cudaGetDriverEntryPoint**(const char *symbol, void **funcPtr, unsigned long long flags, enum *cudaDriverEntryPointQueryResult* *driverStatus = NULL)

Returns the requested driver API function pointer.

Deprecated:

This function is deprecated as of CUDA 13.0

Returns in `**funcPtr` the address of the CUDA driver function for the requested flags.

For a requested driver symbol, if the CUDA version in which the driver symbol was introduced is less than or equal to the CUDA runtime version, the API will return the function pointer to the corresponding versioned driver function.

The pointer returned by the API should be cast to a function pointer matching the requested driver function's definition in the API header file. The function pointer typedef can be picked up from the corresponding typedefs header file. For example, cudaTypedefs.h consists of function pointer typedefs for driver APIs defined in cuda.h.

The API will return *cudaSuccess* and set the returned `funcPtr` if the requested driver function is valid and supported on the platform.

The API will return *cudaSuccess* and set the returned `funcPtr` to NULL if the requested driver function is not supported on the platform, no ABI compatible driver function exists for the CUDA runtime version or if the driver symbol is invalid.

It will also set the optional `driverStatus` to one of the values in *cudaDriverEntryPointQueryResult* with the following meanings:

- *cudaDriverEntryPointSuccess* - The requested symbol was succesfully found based on input arguments and `pfn` is valid
- *cudaDriverEntryPointSymbolNotFound* - The requested symbol was not found
- *cudaDriverEntryPointVersionNotSufficent* - The requested symbol was found but is not supported by the current runtime version (CUDART_VERSION)

The requested flags can be:

- *cudaEnableDefault*: This is the default mode. This is equivalent to *cudaEnablePerThreadDefaultStream* if the code is compiled with —default-stream per-thread compilation flag or the macro CUDA_API_PER_THREAD_DEFAULT_STREAM is defined; *cudaEnableLegacyStream* otherwise.
- *cudaEnableLegacyStream*: This will enable the search for all driver symbols that match the requested driver symbol name except the corresponding per-thread versions.
- *cudaEnablePerThreadDefaultStream*: This will enable the search for all driver symbols that match the requested driver symbol name including the per-thread versions. If a per-thread version is not found, the API will return the legacy version of the driver function.

See also:

::cuGetProcAddress

Note: This API is deprecated and *cudaGetDriverEntryPointByVersion* (with a hardcoded ::cudaVersion) should be used instead.

Note: Version mixing among CUDA-defined types and driver API versions is strongly discouraged and doing so can result in an undefined behavior. See Rules for version mixing.

Note: Note that this function may also return *cudaErrorInitializationError*, *cudaErrorInsufficientDriver* or *cudaErrorNoDevice* if this call tries to initialize internal CUDA RT state.

Note: Note that as specified by *cudaStreamAddCallback* no CUDA function may be called from callback. *cudaErrorNotPermitted* may, but is not guaranteed to, be returned as a diagnostic in such case.

Parameters

- **symbol** – - The base name of the driver API function to look for. As an example, for the driver API ::cuMemAlloc_v2, `symbol` would be cuMemAlloc. Note that the API will use the CUDA runtime version to return the address to the most recent ABI compatible driver symbol, ::cuMemAlloc or ::cuMemAlloc_v2.
- **funcPtr** – - Location to return the function pointer to the requested driver function
- **flags** – - Flags to specify search options.
- **driverStatus** – - Optional location to store the status of finding the symbol from the driver. See *cudaDriverEntryPointQueryResult* for possible values.

Returns

cudaSuccess, *cudaErrorInvalidValue*, *cudaErrorNotSupported*

__host__ *cudaError_t* **cudaGetDriverEntryPointByVersion**(const char *symbol, void **funcPtr, unsigned int cudaVersion, unsigned long long flags, enum *cudaDriverEntryPointQueryResult* *driverStatus = NULL)

Returns the requested driver API function pointer by CUDA version.

Returns in `**funcPtr` the address of the CUDA driver function for the requested flags and CUDA driver version.

The CUDA version is specified as (1000 * major + 10 * minor), so CUDA 11.2 should be specified as 11020. For a requested driver symbol, if the specified CUDA version is greater than or equal to the CUDA version in which the driver symbol was introduced, this API will return the function pointer to the corresponding versioned function. If the specified CUDA version is greater than the driver version, the API will return *cudaErrorInvalidValue*.

The pointer returned by the API should be cast to a function pointer matching the requested driver function's definition in the API header file. The function pointer typedef can be picked up

from the corresponding typedefs header file. For example, cudaTypedefs.h consists of function pointer typedefs for driver APIs defined in cuda.h.

For the case where the CUDA version requested is greater than the CUDA Toolkit installed, there may not be an appropriate function pointer typedef in the corresponding header file and may need a custom typedef to match the driver function signature returned. This can be done by getting the typedefs from a later toolkit or creating appropriately matching custom function typedefs.

The API will return *cudaSuccess* and set the returned `funcPtr` if the requested driver function is valid and supported on the platform.

The API will return *cudaSuccess* and set the returned `funcPtr` to NULL if the requested driver function is not supported on the platform, no ABI compatible driver function exists for the requested version or if the driver symbol is invalid.

It will also set the optional `driverStatus` to one of the values in *cudaDriverEntryPointQueryResult* with the following meanings:

- *cudaDriverEntryPointSuccess* - The requested symbol was succesfully found based on input arguments and `pfn` is valid
- *cudaDriverEntryPointSymbolNotFound* - The requested symbol was not found
- *cudaDriverEntryPointVersionNotSufficent* - The requested symbol was found but is not supported by the specified version `cudaVersion`

The requested flags can be:

- *cudaEnableDefault*: This is the default mode. This is equivalent to *cudaEnablePerThreadDefaultStream* if the code is compiled with —default-stream per-thread compilation flag or the macro CUDA_API_PER_THREAD_DEFAULT_STREAM is defined; *cudaEnableLegacyStream* otherwise.
- *cudaEnableLegacyStream*: This will enable the search for all driver symbols that match the requested driver symbol name except the corresponding per-thread versions.
- *cudaEnablePerThreadDefaultStream*: This will enable the search for all driver symbols that match the requested driver symbol name including the per-thread versions. If a per-thread version is not found, the API will return the legacy version of the driver function.

See also:

::cuGetProcAddress

Note: Version mixing among CUDA-defined types and driver API versions is strongly discouraged and doing so can result in an undefined behavior. See Rules for version mixing.

Note: Note that this function may also return *cudaErrorInitializationError*, *cudaErrorInsufficientDriver* or *cudaErrorNoDevice* if this call tries to initialize internal CUDA RT state.

Note: Note that as specified by *cudaStreamAddCallback* no CUDA function may be called from callback. *cudaErrorNotPermitted* may, but is not guaranteed to, be returned as a diagnostic in such case.

Parameters

- **symbol** – - The base name of the driver API function to look for. As an example, for the driver API ::cuMemAlloc_v2, `symbol` would be cuMemAlloc.
- **funcPtr** – - Location to return the function pointer to the requested driver function
- **cudaVersion** – - The CUDA version to look for the requested driver symbol.
- **flags** – - Flags to specify search options.
- **driverStatus** – - Optional location to store the status of finding the symbol from the driver. See *cudaDriverEntryPointQueryResult* for possible values.

Returns

cudaSuccess, *cudaErrorInvalidValue*, *cudaErrorNotSupported*

6.12. EGL Interoperability

This section describes the EGL interoperability functions of the CUDA runtime application programming interface.

Functions

__host__ cudaError_t *cudaEGLStreamConsumerAcquireFrame*(cudaEglStreamConnection *conn, cudaGraphicsResource_t *pCudaResource, cudaStream_t *pStream, unsigned int timeout)
Acquire an image frame from the EGLStream with CUDA as a consumer.

__host__ cudaError_t *cudaEGLStreamConsumerConnect*(cudaEglStreamConnection *conn, EGLStreamKHR eglStream)
Connect CUDA to EGLStream as a consumer.

__host__ cudaError_t *cudaEGLStreamConsumerConnectWithFlags*(cudaEglStreamConnection *conn, EGLStreamKHR eglStream, unsigned int flags)
Connect CUDA to EGLStream as a consumer with given flags.

__host__ cudaError_t *cudaEGLStreamConsumerDisconnect*(cudaEglStreamConnection *conn)
Disconnect CUDA as a consumer to EGLStream .

__host__ cudaError_t *cudaEGLStreamConsumerReleaseFrame*(cudaEglStreamConnection *conn, cudaGraphicsResource_t pCudaResource, cudaStream_t *pStream)
Releases the last frame acquired from the EGLStream.

__host__ cudaError_t *cudaEGLStreamProducerConnect*(cudaEglStreamConnection *conn, EGLStreamKHR eglStream, EGLint width, EGLint height)
Connect CUDA to EGLStream as a producer.

__host__ cudaError_t *cudaEGLStreamProducerDisconnect*(cudaEglStreamConnection *conn)
Disconnect CUDA as a producer to EGLStream .

__host__ cudaError_t *cudaEGLStreamProducerPresentFrame*(cudaEglStreamConnection *conn, cudaEglFrame eglframe, cudaStream_t *pStream)
Present a CUDA eglFrame to the EGLStream with CUDA as a producer.

__host__ cudaError_t *cudaEGLStreamProducerReturnFrame*(cudaEglStreamConnection *conn, cudaEglFrame *eglframe, cudaStream_t *pStream)
Return the CUDA eglFrame to the EGLStream last released by the consumer.

__host__ cudaError_t *cudaEventCreateFromEGLSync*(cudaEvent_t *phEvent, EGLSyncKHR eglSync, unsigned int flags)
Creates an event from EGLSync object.

__host__ cudaError_t *cudaGraphicsEGLRegisterImage*(struct cudaGraphicsResource **pCudaResource, EGLImageKHR image, unsigned int flags)
Registers an EGL image.

__host__ cudaError_t *cudaGraphicsResourceGetMappedEglFrame*(cudaEglFrame *eglFrame, cudaGraphicsResource_t resource, unsigned int index, unsigned int mipLevel)
Get an eglFrame through which to access a registered EGL graphics resource.

6.12.1. Functions

__host__ *cudaError_t* **cudaEGLStreamConsumerAcquireFrame**(*cudaEglStreamConnection* *conn, *cudaGraphicsResource_t* *pCudaResource, *cudaStream_t* *pStream, unsigned int timeout)

Acquire an image frame from the EGLStream with CUDA as a consumer.

Acquire an image frame from EGLStreamKHR. *cudaGraphicsResourceGetMappedEglFrame* can be called on `pCudaResource` to get *cudaEglFrame*.

See also:

cudaEGLStreamConsumerConnect, *cudaEGLStreamConsumerDisconnect*, *cudaEGLStreamConsumerReleaseFrame*, ::cuEGLStreamConsumerAcquireFrame

Parameters

- **conn** – - Connection on which to acquire
- **pCudaResource** – - CUDA resource on which the EGLStream frame will be mapped for use.
- **pStream** – - CUDA stream for synchronization and any data migrations implied by *cudaEglResourceLocationFlags*.
- **timeout** – - Desired timeout in usec.

Returns
cudaSuccess, *cudaErrorInvalidValue*, *cudaErrorUnknown*, *cudaErrorLaunchTimeout*

__host__ *cudaError_t* **cudaEGLStreamConsumerConnect**(*cudaEglStreamConnection* *conn, EGLStreamKHR eglStream)

Connect CUDA to EGLStream as a consumer.

Connect CUDA as a consumer to EGLStreamKHR specified by `eglStream`.

The EGLStreamKHR is an EGL object that transfers a sequence of image frames from one API to another.

See also:

cudaEGLStreamConsumerDisconnect, *cudaEGLStreamConsumerAcquireFrame*, *cudaEGLStreamConsumerReleaseFrame*, ::cuEGLStreamConsumerConnect

Parameters

- **conn** – - Pointer to the returned connection handle
- **eglStream** – - EGLStreamKHR handle

Returns
cudaSuccess, *cudaErrorInvalidValue*, *cudaErrorUnknown*

__host__ *cudaError_t* **cudaEGLStreamConsumerConnectWithFlags**(*cudaEglStreamConnection* *conn, EGLStreamKHR eglStream, unsigned int flags)

Connect CUDA to EGLStream as a consumer with given flags.

Connect CUDA as a consumer to EGLStreamKHR specified by `stream` with specified `flags` defined by *cudaEglResourceLocationFlags*.

The flags specify whether the consumer wants to access frames from system memory or video memory. Default is *cudaEglResourceLocationVidmem*.

See also:

cudaEGLStreamConsumerDisconnect, *cudaEGLStreamConsumerAcquireFrame*, *cudaEGLStreamConsumerReleaseFrame*, ::cuEGLStreamConsumerConnectWithFlags

Parameters

- **conn** – - Pointer to the returned connection handle
- **eglStream** – - EGLStreamKHR handle
- **flags** – - Flags denote intended location - system or video.

Returns
cudaSuccess, *cudaErrorInvalidValue*, *cudaErrorUnknown*

__host__ *cudaError_t* **cudaEGLStreamConsumerDisconnect**(*cudaEglStreamConnection* *conn)

Disconnect CUDA as a consumer to EGLStream .

Disconnect CUDA as a consumer to EGLStreamKHR.

See also:

cudaEGLStreamConsumerConnect, *cudaEGLStreamConsumerAcquireFrame*, *cudaEGLStreamConsumerReleaseFrame*, ::cuEGLStreamConsumerDisconnect

Parameters
conn – - Conection to disconnect.

Returns
cudaSuccess, *cudaErrorInvalidValue*, *cudaErrorUnknown*

__host__ *cudaError_t* **cudaEGLStreamConsumerReleaseFrame**(*cudaEglStreamConnection* *conn, *cudaGraphicsResource_t* pCudaResource, *cudaStream_t* *pStream)

Releases the last frame acquired from the EGLStream.

Release the acquired image frame specified by `pCudaResource` to EGLStreamKHR.

See also:

cudaEGLStreamConsumerConnect, *cudaEGLStreamConsumerDisconnect*, *cudaEGLStreamConsumerAcquireFrame*, ::cuEGLStreamConsumerReleaseFrame

Parameters

- **conn** – - Connection on which to release
- **pCudaResource** – - CUDA resource whose corresponding frame is to be released
- **pStream** – - CUDA stream on which release will be done.

Returns
cudaSuccess, *cudaErrorInvalidValue*, *cudaErrorUnknown*

__host__ *cudaError_t* **cudaEGLStreamProducerConnect**(*cudaEglStreamConnection* *conn, EGLStreamKHR eglStream, EGLint width, EGLint height)

Connect CUDA to EGLStream as a producer.

Connect CUDA as a producer to EGLStreamKHR specified by `stream`.

The EGLStreamKHR is an EGL object that transfers a sequence of image frames from one API to another.

See also:

cudaEGLStreamProducerDisconnect, *cudaEGLStreamProducerPresentFrame*, *cudaEGLStreamProducerReturnFrame*, ::cuEGLStreamProducerConnect

Parameters

- **conn** – - Pointer to the returned connection handle
- **eglStream** – - EGLStreamKHR handle
- **width** – - width of the image to be submitted to the stream
- **height** – - height of the image to be submitted to the stream

Returns
cudaSuccess, *cudaErrorInvalidValue*, *cudaErrorUnknown*

__host__ *cudaError_t* **cudaEGLStreamProducerDisconnect**(*cudaEglStreamConnection* *conn)

Disconnect CUDA as a producer to EGLStream .

Disconnect CUDA as a producer to EGLStreamKHR.

See also:

cudaEGLStreamProducerConnect, *cudaEGLStreamProducerPresentFrame*, *cudaEGLStreamProducerReturnFrame*, ::cuEGLStreamProducerDisconnect

Parameters

conn – - Conection to disconnect.

Returns

cudaSuccess, *cudaErrorInvalidValue*, *cudaErrorUnknown*

__host__ *cudaError_t* **cudaEGLStreamProducerPresentFrame**(*cudaEglStreamConnection* *conn, *cudaEglFrame* eglframe, *cudaStream_t* *pStream)

Present a CUDA eglFrame to the EGLStream with CUDA as a producer.

The *cudaEglFrame* is defined as:

```
typedef struct cudaEglFrame_st {
    union {
        cudaArray_t            pArray[CUDA_EGL_MAX_PLANES];
        struct cudaPitchedPtr  pPitch[CUDA_EGL_MAX_PLANES];
    } frame;
    cudaEglPlaneDesc planeDesc[CUDA_EGL_MAX_PLANES];
    unsigned int planeCount;
    cudaEglFrameType frameType;
    cudaEglColorFormat eglColorFormat;
} cudaEglFrame;
```

For *cudaEglFrame* of type *cudaEglFrameTypePitch*, the application may present sub-region of a memory allocation. In that case, *cudaPitchedPtr::ptr* will specify the start address of the sub-region in the allocation and *cudaEglPlaneDesc* will specify the dimensions of the sub-region.

See also:

cudaEGLStreamProducerConnect, *cudaEGLStreamProducerDisconnect*, *cudaEGLStreamProducerReturnFrame*, ::cuEGLStreamProducerPresentFrame

Parameters

- **conn** – - Connection on which to present the CUDA array
- **eglframe** – - CUDA Eglstream Proucer Frame handle to be sent to the consumer over EglStream.
- **pStream** – - CUDA stream on which to present the frame.

Returns

cudaSuccess, *cudaErrorInvalidValue*, *cudaErrorUnknown*

__host__ *cudaError_t* **cudaEGLStreamProducerReturnFrame**(*cudaEglStreamConnection* *conn, *cudaEglFrame* *eglframe, *cudaStream_t* *pStream)

Return the CUDA eglFrame to the EGLStream last released by the consumer.

This API can potentially return cudaErrorLaunchTimeout if the consumer has not returned a frame to EGL stream. If timeout is returned the application can retry.

See also:

cudaEGLStreamProducerConnect, *cudaEGLStreamProducerDisconnect*, *cudaEGLStreamProducerPresentFrame*, ::cuEGLStreamProducerReturnFrame

Parameters

- **conn** – - Connection on which to present the CUDA array
- **eglframe** – - CUDA Eglstream Proucer Frame handle returned from the consumer over EglStream.
- **pStream** – - CUDA stream on which to return the frame.

Returns

cudaSuccess, *cudaErrorLaunchTimeout*, *cudaErrorInvalidValue*, *cudaErrorUnknown*

__host__ *cudaError_t* **cudaEventCreateFromEGLSync**(*cudaEvent_t* *phEvent, EGLSyncKHR eglSync, unsigned int flags)

Creates an event from EGLSync object.

Creates an event *phEvent from an EGLSyncKHR eglSync with the flages specified via `flags`. Valid flags include:

- *cudaEventDefault*: Default event creation flag.
- *cudaEventBlockingSync*: Specifies that the created event should use blocking synchronization. A CPU thread that uses *cudaEventSynchronize()* to wait on an event created with this flag will block until the event has actually been completed.

cudaEventRecord and TimingData are not supported for events created from EGLSync.

The EGLSyncKHR is an opaque handle to an EGL sync object. typedef void* EGLSyncKHR

See also:

cudaEventQuery, *cudaEventSynchronize*, *cudaEventDestroy*

Parameters

- **phEvent** – - Returns newly created event
- **eglSync** – - Opaque handle to EGLSync object
- **flags** – - Event creation flags

Returns

cudaSuccess, *cudaErrorInitializationError*, *cudaErrorInvalidValue*, *cudaErrorLaunchFailure*, *cudaErrorMemoryAllocation*

__host__ *cudaError_t* **cudaGraphicsEGLRegisterImage**(struct cudaGraphicsResource **pCudaResource, EGLImageKHR image, unsigned int flags)

Registers an EGL image.

Registers the EGLImageKHR specified by `image` for access by CUDA. A handle to the registered object is returned as `pCudaResource`. Additional Mapping/Unmapping is not required for the registered resource and *cudaGraphicsResourceGetMappedEglFrame* can be directly called on the `pCudaResource`.

The application will be responsible for synchronizing access to shared objects. The application must ensure that any pending operation which access the objects have completed before passing control to CUDA. This may be accomplished by issuing and waiting for glFinish command on all GLcontexts (for OpenGL and likewise for other APIs). The application will be also responsible for ensuring that any pending operation on the registered CUDA resource has completed prior to executing subsequent commands in other APIs accesing the same memory objects. This can be accomplished by calling cuCtxSynchronize or cuEventSynchronize (preferably).

The surface's intended usage is specified using `flags`, as follows:

- *cudaGraphicsRegisterFlagsNone*: Specifies no hints about how this resource will be used. It is therefore assumed that this resource will be read from and written to by CUDA. This is the default value.
- *cudaGraphicsRegisterFlagsReadOnly*: Specifies that CUDA will not write to this resource.
- *cudaGraphicsRegisterFlagsWriteDiscard*: Specifies that CUDA will not read from this resource and will write over the entire contents of the resource, so none of the data previously stored in the resource will be preserved.

The EGLImageKHR is an object which can be used to create EGLImage target resource. It is defined as a void pointer. typedef void* EGLImageKHR

See also:

cudaGraphicsUnregisterResource, *cudaGraphicsResourceGetMappedEglFrame*, ::cuGraphicsEGLRegisterImage

Parameters

- **pCudaResource** – - Pointer to the returned object handle
- **image** – - An EGLImageKHR image which can be used to create target resource.
- **flags** – - Map flags

Returns

cudaSuccess, *cudaErrorInvalidResourceHandle*, *cudaErrorInvalidValue*, *cudaErrorUnknown*

__host__ *cudaError_t* **cudaGraphicsResourceGetMappedEglFrame**(*cudaEglFrame* *eglFrame, *cudaGraphicsResource_t* resource, unsigned int index, unsigned int mipLevel)

Get an eglFrame through which to access a registered EGL graphics resource.

Returns in *eglFrame an eglFrame pointer through which the registered graphics resource resource may be accessed. This API can only be called for EGL graphics resources.

The *cudaEglFrame* is defined as

```
typedef struct cudaEglFrame_st {
    union {
        cudaArray_t             pArray[CUDA_EGL_MAX_PLANES];
        struct cudaPitchedPtr   pPitch[CUDA_EGL_MAX_PLANES];
    } frame;
    cudaEglPlaneDesc planeDesc[CUDA_EGL_MAX_PLANES];
    unsigned int planeCount;
    cudaEglFrameType frameType;
    cudaEglColorFormat eglColorFormat;
} cudaEglFrame;
```

See also:

cudaGraphicsSubResourceGetMappedArray, *cudaGraphicsResourceGetMappedPointer*, ::cuGraphicsResourceGetMappedEglFrame

---

Note: Note that in case of multiplanar *eglFrame, pitch of only first plane (unsigned int *cudaEglPlaneDesc::pitch*) is to be considered by the application.

Parameters

- **eglFrame** – - Returned eglFrame.
- **resource** – - Registered resource to access.
- **index** – - Index for cubemap surfaces.
- **mipLevel** – - Mipmap level for the subresource to access.

Returns

cudaSuccess, *cudaErrorInvalidValue*, *cudaErrorUnknown*

6.13. Error Handling

This section describes the error handling functions of the CUDA runtime application programming interface.

Functions

__host__ const char * *cudaGetErrorName*(cudaError_t error)
Returns the string representation of an error code enum name.

__host__ const char * *cudaGetErrorString*(cudaError_t error)
Returns the description string for an error code.

__host__ cudaError_t *cudaGetLastError*(void)
Returns the last error from a runtime call.

__host__ cudaError_t *cudaPeekAtLastError*(void)
Returns the last error from a runtime call.

6.13.1. Functions

__host__ const char ***cudaGetErrorName**(*cudaError_t* error)

Returns the string representation of an error code enum name.

Returns a string containing the name of an error code in the enum. If the error code is not recognized, "unrecognized error code" is returned.

See also:

cudaGetErrorString, *cudaGetLastError*, *cudaPeekAtLastError*, *cudaError*, ::cuGetErrorName

Parameters
error – - Error code to convert to string

Returns
char* pointer to a NULL-terminated string

__host__ const char ***cudaGetErrorString**(*cudaError_t* error)

Returns the description string for an error code.

Returns the description string for an error code. If the error code is not recognized, "unrecognized error code" is returned.

See also:

cudaGetErrorName, *cudaGetLastError*, *cudaPeekAtLastError*, *cudaError*, ::cuGetErrorString

Parameters
error – - Error code to convert to string

Returns
char* pointer to a NULL-terminated string

__host__ *cudaError_t* **cudaGetLastError**(void)

Returns the last error from a runtime call.

Returns the last error that has been produced by any of the runtime calls in the same instance of the CUDA Runtime library in the host thread and resets it to *cudaSuccess*.

Note: Multiple instances of the CUDA Runtime library can be present in an application when using a library that statically links the CUDA Runtime.

See also:

cudaPeekAtLastError, *cudaGetErrorName*, *cudaGetErrorString*, *cudaError*

Note: Note that this function may also return error codes from previous, asynchronous launches.

Note: Note that this function may also return *cudaErrorInitializationError*, *cudaErrorInsufficientDriver* or *cudaErrorNoDevice* if this call tries to initialize internal CUDA RT state.

Note: Note that as specified by *cudaStreamAddCallback* no CUDA function may be called from callback. *cudaErrorNotPermitted* may, but is not guaranteed to, be returned as a diagnostic in such case.

Returns

cudaSuccess, *cudaErrorMissingConfiguration*, *cudaErrorMemoryAllocation*, *cudaErrorInitializationError*, *cudaErrorLaunchFailure*, *cudaErrorLaunchTimeout*, *cudaErrorLaunchOutOfResources*, *cudaErrorInvalidDeviceFunction*, *cudaErrorInvalidConfiguration*, *cudaErrorInvalidDevice*, *cudaErrorInvalidValue*, *cudaErrorInvalidPitchValue*, *cudaErrorInvalidSymbol*, *cudaErrorUnmapBufferObjectFailed*, *cudaErrorInvalidDevicePointer*, *cudaErrorInvalidTexture*, *cudaErrorInvalidTextureBinding*, *cudaErrorInvalidChannelDescriptor*, *cudaErrorInvalidMemcpyDirection*, *cudaErrorInvalidFilterSetting*, *cudaErrorInvalidNormSetting*, *cudaErrorUnknown*, *cudaErrorInvalidResourceHandle*, *cudaErrorInsufficientDriver*, *cudaErrorNoDevice*, *cudaErrorSetOnActiveProcess*, *cudaErrorStartupFailure*, *cudaErrorInvalidPtx*, *cudaErrorUnsupportedPtxVersion*, *cudaErrorNoKernelImageForDevice*, *cudaErrorJitCompilerNotFound*, *cudaErrorJitCompilationDisabled*

__host__ *cudaError_t* **cudaPeekAtLastError**(void)

Returns the last error from a runtime call.

Returns the last error that has been produced by any of the runtime calls in the same instance of the CUDA Runtime library in the host thread. This call does not reset the error to *cudaSuccess* like *cudaGetLastError()*.

Note: Multiple instances of the CUDA Runtime library can be present in an application when using a library that statically links the CUDA Runtime.

See also:

cudaGetLastError, *cudaGetErrorName*, *cudaGetErrorString*, *cudaError*

Note: Note that this function may also return error codes from previous, asynchronous launches.

Note: Note that this function may also return *cudaErrorInitializationError*, *cudaErrorInsufficientDriver* or *cudaErrorNoDevice* if this call tries to initialize internal CUDA RT state.

Note: Note that as specified by *cudaStreamAddCallback* no CUDA function may be called from callback. *cudaErrorNotPermitted* may, but is not guaranteed to, be returned as a diagnostic in such case.

Returns

cudaSuccess, *cudaErrorMissingConfiguration*, *cudaErrorMemoryAllocation*, *cudaErrorInitializationError*, *cudaErrorLaunchFailure*, *cudaErrorLaunchTimeout*, *cudaErrorLaunchOutOfResources*, *cudaErrorInvalidDeviceFunction*, *cudaErrorInvalidConfiguration*, *cudaErrorInvalidDevice*, *cudaErrorInvalidValue*, *cudaErrorInvalidPitchValue*, *cudaErrorInvalidSymbol*, *cudaErrorUnmapBufferObjectFailed*, *cudaErrorInvalidDevicePointer*, *cudaErrorInvalidTexture*, *cudaErrorInvalidTextureBinding*, *cudaErrorInvalidChannelDescriptor*, *cudaErrorInvalidMemcpyDirection*, *cudaErrorInvalidFilterSetting*, *cudaErrorInvalidNormSetting*, *cudaErrorUnknown*, *cudaErrorInvalidResourceHandle*, *cudaErrorInsufficientDriver*, *cudaErrorNoDevice*, *cudaErrorSetOnActiveProcess*, *cudaErrorStartupFailure*, *cudaErrorInvalidPtx*, *cudaErrorUnsupportedPtxVersion*, *cudaErrorNoKernelImageForDevice*, *cudaErrorJitCompilerNotFound*, *cudaErrorJitCompilationDisabled*

6.14. Error Log Management Functions

This section describes the error log management functions of the CUDA runtime application programming interface.

The Error Log Management interface will operate on both the CUDA Driver and CUDA Runtime.

Functions

__host__ cudaError_t *cudaLogsCurrent*(cudaLogIterator *iterator_out, unsigned int flags)
Sets log iterator to point to the end of log buffer, where the next message would be written.

__host__ cudaError_t *cudaLogsDumpToFile*(cudaLogIterator *iterator, const char *pathToFile, unsigned int flags)
Dump accumulated driver logs into a file.

__host__ cudaError_t *cudaLogsDumpToMemory*(cudaLogIterator *iterator, char *buffer, size_t *size, unsigned int flags)
Dump accumulated driver logs into a buffer.

__host__ cudaError_t *cudaLogsRegisterCallback*(cudaLogsCallback_t callbackFunc, void *userData, cudaLogsCallbackHandle *callback_out)
Register a callback function to receive error log messages.

__host__ cudaError_t *cudaLogsUnregisterCallback*(cudaLogsCallbackHandle callback)
Unregister a log message callback.

Typedefs

cudaLogsCallback_t
Type of public error reporting callback functions.

6.14.1. Functions

__host__ *cudaError_t* **cudaLogsCurrent**(*cudaLogIterator* *iterator_out, unsigned int flags)
Sets log iterator to point to the end of log buffer, where the next message would be written.

Parameters

- **iterator_out** – - Location to store an iterator to the current tail of the logs
- **flags** – - Reserved for future use, must be 0

Returns
cudaSuccess, *cudaErrorInvalidValue*,

__host__ *cudaError_t* **cudaLogsDumpToFile**(*cudaLogIterator* *iterator, const char *pathToFile, unsigned int flags)

Dump accumulated driver logs into a file.

Logs generated by the driver are stored in an internal buffer and can be copied out using this API. This API dumps all driver logs starting from `iterator` into `pathToFile` provided.

Note: `iterator` is auto-advancing. Dumping logs will update the value of `iterator` to receive the next generated log.

Note: The driver reserves limited memory for storing logs. The oldest logs may be overwritten and become unrecoverable. An indication will appear in the destination outupt if the logs have been truncated. Call dump after each failed API to mitigate this risk.

Parameters

- **iterator** – - Optional auto-advancing iterator specifying the starting log to read. NULL value dumps all logs.
- **pathToFile** – - Path to output file for dumping logs
- **flags** – - Reserved for future use, must be 0

Returns
cudaSuccess, *cudaErrorInvalidValue*,

__host__ *cudaError_t* **cudaLogsDumpToMemory**(*cudaLogIterator* *iterator, char *buffer, size_t *size, unsigned int flags)

Dump accumulated driver logs into a buffer.

Logs generated by the driver are stored in an internal buffer and can be copied out using this API. This API dumps driver logs from `iterator` into `buffer` up to the size specified in `*size`.

The driver will always null terminate the buffer but there will not be a null character between log entries, only a newline \n. The driver will then return the actual number of bytes written in `*size`, excluding the null terminator. If there are no messages to dump, `*size` will be set to 0 and the function will return ::CUDA_SUCCESS. If the provided `buffer` is not large enough to hold any messages, `*size` will be set to 0 and the function will return ::CUDA_ERROR_INVALID_VALUE.

Note: `iterator` is auto-advancing. Dumping logs will update the value of `iterator` to receive the next generated log.

Note: The driver reserves limited memory for storing logs. The maximum size of the buffer is 25600 bytes. The oldest logs may be overwritten and become unrecoverable. An indication will appear in the destination outupt if the logs have been truncated. Call dump after each failed API to mitigate this risk.

Note: If the provided value in `*size` is not large enough to hold all buffered messages, a message will be added at the head of the buffer indicating this. The driver then computes the number of messages it is able to store in `buffer` and writes it out. The final message in `buffer` will always be the most recent log message as of when the API is called.

Parameters

- **`iterator`** – - Optional auto-advancing iterator specifying the starting log to read. NULL value dumps all logs.
- **`buffer`** – - Pointer to dump logs
- **`size`** – - See description
- **`flags`** – - Reserved for future use, must be 0

Returns

cudaSuccess, *cudaErrorInvalidValue*,

__host__ *cudaError_t* **`cudaLogsRegisterCallback`**(cudaLogsCallback_t callbackFunc, void *userData, *cudaLogsCallbackHandle* *callback_out)

Register a callback function to receive error log messages.

Parameters

- **`callbackFunc`** – - The function to register as a callback
- **`userData`** – - A generic pointer to user data. This is passed into the callback function.
- **`callback_out`** – - Optional location to store the callback handle after it is registered

Returns

cudaSuccess, *cudaErrorInvalidValue*,

__host__ *cudaError_t* **cudaLogsUnregisterCallback**(*cudaLogsCallbackHandle* callback)

Unregister a log message callback.

Parameters

callback – - The callback instance to unregister from receiving log messages

Returns

cudaSuccess, *cudaErrorInvalidValue*,

6.14.2. Typedefs

void(__stdcall * cudaLogsCallback_t)(void *data, cudaLogLevel logLevel, char *message, size_t length)

Type of public error reporting callback functions.

Param data
User parameter provided at registration

Param logLevel
Severity level of the log message

Param message
Error log message being reported

Param length
Length of the message in bytes

6.15. Event Management

This section describes the event management functions of the CUDA runtime application programming interface.

Functions

__host__ cudaError_t *cudaEventCreate*(cudaEvent_t *event)
Creates an event object.

__host__ cudaError_t *cudaEventCreateWithFlags*(cudaEvent_t *event, unsigned int flags)
Creates an event object with the specified flags.

__host__ cudaError_t *cudaEventDestroy*(cudaEvent_t event)
Destroys an event object.

__host__ cudaError_t *cudaEventElapsedTime*(float *ms, cudaEvent_t start, cudaEvent_t end)
Computes the elapsed time between events.

__host__ cudaError_t *cudaEventQuery*(cudaEvent_t event)
Queries an event's status.

__host__ cudaError_t *cudaEventRecord*(cudaEvent_t event, cudaStream_t stream=0)
Records an event.

__host__ cudaError_t *cudaEventRecordWithFlags*(cudaEvent_t event, cudaStream_t stream=0, unsigned int flags=0)
Records an event.

__host__ cudaError_t *cudaEventSynchronize*(cudaEvent_t event)
Waits for an event to complete.

6.15.1. Functions

__host__ *cudaError_t* **cudaEventCreate**(*cudaEvent_t* *event)

Creates an event object.

Creates an event object for the current device using *cudaEventDefault*.

See also:

cudaEventCreate (C++ API), *cudaEventCreateWithFlags*, *cudaEventRecord*, *cudaEventQuery*, *cudaEventSynchronize*, *cudaEventDestroy*, *cudaEventElapsedTime*, *cudaStreamWaitEvent*, ::cuEventCreate

Note: Note that this function may also return error codes from previous, asynchronous launches.

Note: Note that this function may also return *cudaErrorInitializationError*, *cudaErrorInsufficientDriver* or *cudaErrorNoDevice* if this call tries to initialize internal CUDA RT state.

Note: Note that as specified by *cudaStreamAddCallback* no CUDA function may be called from callback. *cudaErrorNotPermitted* may, but is not guaranteed to, be returned as a diagnostic in such case.

Parameters
event – - Newly created event

Returns
cudaSuccess, *cudaErrorInvalidValue*, *cudaErrorLaunchFailure*, *cudaErrorMemoryAllocation*

__host__ *cudaError_t* **cudaEventCreateWithFlags**(*cudaEvent_t* *event, unsigned int flags)

Creates an event object with the specified flags.

Creates an event object for the current device with the specified flags. Valid flags include:

- *cudaEventDefault*: Default event creation flag.
- *cudaEventBlockingSync*: Specifies that event should use blocking synchronization. A host thread that uses *cudaEventSynchronize()* to wait on an event created with this flag will block until the event actually completes.

- *cudaEventDisableTiming*: Specifies that the created event does not need to record timing data. Events created with this flag specified and the *cudaEventBlockingSync* flag not specified will provide the best performance when used with *cudaStreamWaitEvent()* and *cudaEventQuery()*.
- *cudaEventInterprocess*: Specifies that the created event may be used as an interprocess event by *cudaIpcGetEventHandle()*. *cudaEventInterprocess* must be specified along with *cudaEventDisableTiming*.

See also:

cudaEventCreate (C API), *cudaEventSynchronize*, *cudaEventDestroy*, *cudaEventElapsedTime*, *cudaStreamWaitEvent*, ::cuEventCreate

---

Note: Note that this function may also return error codes from previous, asynchronous launches.

Note: Note that this function may also return *cudaErrorInitializationError*, *cudaErrorInsufficientDriver* or *cudaErrorNoDevice* if this call tries to initialize internal CUDA RT state.

Note: Note that as specified by *cudaStreamAddCallback* no CUDA function may be called from callback. *cudaErrorNotPermitted* may, but is not guaranteed to, be returned as a diagnostic in such case.

Parameters

- **`event`** – - Newly created event
- **`flags`** – - Flags for new event

Returns

cudaSuccess, *cudaErrorInvalidValue*, *cudaErrorLaunchFailure*, *cudaErrorMemoryAllocation*

__host__ *cudaError_t* **`cudaEventDestroy`**(*cudaEvent_t* event)

Destroys an event object.

Destroys the event specified by `event`.

An event may be destroyed before it is complete (i.e., while *cudaEventQuery()* would return *cudaErrorNotReady*). In this case, the call does not block on completion of the event, and any associated resources will automatically be released asynchronously at completion.

See also:

cudaEventCreate (C API), *cudaEventCreateWithFlags*, *cudaEventQuery*, *cudaEventSynchronize*, *cudaEventRecord*, *cudaEventElapsedTime*, ::cuEventDestroy

---

Note: Note that this function may also return error codes from previous, asynchronous launches.

Note: Note that this function may also return *cudaErrorInitializationError*, *cudaErrorInsufficientDriver* or *cudaErrorNoDevice* if this call tries to initialize internal CUDA RT state.

Note: Note that as specified by *cudaStreamAddCallback* no CUDA function may be called from callback. *cudaErrorNotPermitted* may, but is not guaranteed to, be returned as a diagnostic in such case.

Note: Use of the handle after this call is undefined behavior.

Note: Returns *cudaErrorInvalidResourceHandle* in the event of being passed NULL as the input event.

Parameters

event – - Event to destroy

Returns

cudaSuccess, *cudaErrorInvalidValue*, *cudaErrorInvalidResourceHandle*, *cudaErrorLaunchFailure*

__host__ *cudaError_t* **cudaEventElapsedTime**(float *ms, *cudaEvent_t* start, *cudaEvent_t* end)

Computes the elapsed time between events.

Computes the elapsed time between two events (in milliseconds with a resolution of around 0.5 microseconds). Note this API is not guaranteed to return the latest errors for pending work. As such this API is intended to serve as a elapsed time calculation only and polling for completion on the events to be compared should be done with *cudaEventQuery* instead.

If either event was last recorded in a non-NULL stream, the resulting time may be greater than expected (even if both used the same stream handle). This happens because the *cudaEventRecord()* operation takes place asynchronously and there is no guarantee that the measured latency is actually just between the two events. Any number of other different stream operations could execute in between the two measured events, thus altering the timing in a significant way.

If *cudaEventRecord()* has not been called on either event, then *cudaErrorInvalidResourceHandle* is returned. If *cudaEventRecord()* has been called on both events but one or both of them has not yet been completed (that is, *cudaEventQuery()* would return *cudaErrorNotReady* on at least one of the events), *cudaErrorNotReady* is returned. If either event was created with the *cudaEventDisableTiming* flag, then this function will return *cudaErrorInvalidResourceHandle*.

See also:

cudaEventCreate (C API), *cudaEventCreateWithFlags*, *cudaEventQuery*, *cudaEventSynchronize*, *cudaEventDestroy*, *cudaEventRecord*, ::cuEventElapsedTime

Note: Note that this function may also return error codes from previous, asynchronous launches.

Note: Note that this function may also return *cudaErrorInitializationError*, *cudaErrorInsufficientDriver* or *cudaErrorNoDevice* if this call tries to initialize internal CUDA RT state.

Note: Note that as specified by *cudaStreamAddCallback* no CUDA function may be called from callback. *cudaErrorNotPermitted* may, but is not guaranteed to, be returned as a diagnostic in such case.

Note: Returns *cudaErrorInvalidResourceHandle* in the event of being passed NULL as the input event.

Parameters

- **ms** – - Time between `start` and `end` in ms
- **start** – - Starting event
- **end** – - Ending event

Returns

cudaSuccess, *cudaErrorNotReady*, *cudaErrorInvalidValue*, *cudaErrorInvalidResourceHandle*, *cudaErrorLaunchFailure*, *cudaErrorUnknown*

__host__ *cudaError_t* **cudaEventQuery**(*cudaEvent_t* event)

Queries an event's status.

Queries the status of all work currently captured by `event`. See *cudaEventRecord()* for details on what is captured by an event.

Returns *cudaSuccess* if all captured work has been completed, or *cudaErrorNotReady* if any captured work is incomplete.

For the purposes of Unified Memory, a return value of *cudaSuccess* is equivalent to having called *cudaEventSynchronize()*.

See also:

cudaEventCreate (C API), *cudaEventCreateWithFlags*, *cudaEventRecord*, *cudaEventSynchronize*, *cudaEventDestroy*, *cudaEventElapsedTime*, ::cuEventQuery

Note: Note that this function may also return error codes from previous, asynchronous launches.

Note: Note that this function may also return *cudaErrorInitializationError*, *cudaErrorInsufficientDriver* or *cudaErrorNoDevice* if this call tries to initialize internal CUDA RT state.

Note: Note that as specified by *cudaStreamAddCallback* no CUDA function may be called from callback. *cudaErrorNotPermitted* may, but is not guaranteed to, be returned as a diagnostic in

such case.

Note: Returns *cudaErrorInvalidResourceHandle* in the event of being passed NULL as the input event.

Parameters

event – - Event to query

Returns

cudaSuccess, *cudaErrorNotReady*, *cudaErrorInvalidValue*, *cudaErrorInvalidResourceHandle*, *cudaErrorLaunchFailure*

__host__ *cudaError_t* **cudaEventRecord**(*cudaEvent_t* event, *cudaStream_t* stream = 0)

Records an event.

Captures in `event` the contents of `stream` at the time of this call. `event` and `stream` must be on the same CUDA context. Calls such as *cudaEventQuery()* or *cudaStreamWaitEvent()* will then examine or wait for completion of the work that was captured. Uses of `stream` after this call do not modify `event`. See note on default stream behavior for what is captured in the default case.

cudaEventRecord() can be called multiple times on the same event and will overwrite the previously captured state. Other APIs such as *cudaStreamWaitEvent()* use the most recently captured state at the time of the API call, and are not affected by later calls to *cudaEventRecord()*. Before the first call to *cudaEventRecord()*, an event represents an empty set of work, so for example *cudaEventQuery()* would return *cudaSuccess*.

See also:

cudaEventCreate (C API), *cudaEventCreateWithFlags*, *cudaEventQuery*, *cudaEventSynchronize*, *cudaEventDestroy*, *cudaEventElapsedTime*, *cudaStreamWaitEvent*, *cudaEventRecordWithFlags*, ::cuEventRecord

---

Note: This function uses standard default stream semantics.

Note: Note that this function may also return error codes from previous, asynchronous launches.

Note: Note that this function may also return *cudaErrorInitializationError*, *cudaErrorInsufficientDriver* or *cudaErrorNoDevice* if this call tries to initialize internal CUDA RT state.

Note: Note that as specified by *cudaStreamAddCallback* no CUDA function may be called from callback. *cudaErrorNotPermitted* may, but is not guaranteed to, be returned as a diagnostic in such case.

Note: Returns *cudaErrorInvalidResourceHandle* in the event of being passed NULL as the input event.

Parameters

- **event** – - Event to record
- **stream** – - Stream in which to record event

Returns

cudaSuccess, *cudaErrorInvalidValue*, *cudaErrorInvalidResourceHandle*, *cudaErrorLaunchFailure*

__host__ *cudaError_t* **cudaEventRecordWithFlags**(*cudaEvent_t* event, *cudaStream_t* stream = 0, unsigned int flags = 0)

Records an event.

Captures in `event` the contents of `stream` at the time of this call. `event` and `stream` must be on the same CUDA context. Calls such as *cudaEventQuery()* or *cudaStreamWaitEvent()* will then examine or wait for completion of the work that was captured. Uses of `stream` after this call do not modify `event`. See note on default stream behavior for what is captured in the default case.

cudaEventRecordWithFlags() can be called multiple times on the same event and will overwrite the previously captured state. Other APIs such as *cudaStreamWaitEvent()* use the most recently captured state at the time of the API call, and are not affected by later calls to *cudaEventRecordWithFlags()*. Before the first call to *cudaEventRecordWithFlags()*, an event represents an empty set of work, so for example *cudaEventQuery()* would return *cudaSuccess*.

flags include:

- *cudaEventRecordDefault*: Default event creation flag.
- *cudaEventRecordExternal*: Event is captured in the graph as an external event node when performing stream capture.

See also:

cudaEventCreate (C API), *cudaEventCreateWithFlags*, *cudaEventQuery*, *cudaEventSynchronize*, *cudaEventDestroy*, *cudaEventElapsedTime*, *cudaStreamWaitEvent*, *cudaEventRecord*, ::cuEventRecord,

---

Note: This function uses standard default stream semantics.

Note: Note that this function may also return error codes from previous, asynchronous launches.

Note: Note that this function may also return *cudaErrorInitializationError*, *cudaErrorInsufficientDriver* or *cudaErrorNoDevice* if this call tries to initialize internal CUDA RT state.

Note: Note that as specified by *cudaStreamAddCallback* no CUDA function may be called from callback. *cudaErrorNotPermitted* may, but is not guaranteed to, be returned as a diagnostic in such case.

Note: Returns *cudaErrorInvalidResourceHandle* in the event of being passed NULL as the input event.

Parameters

- **event** – - Event to record
- **stream** – - Stream in which to record event
- **flags** – - Parameters for the operation(See above)

Returns

cudaSuccess, *cudaErrorInvalidValue*, *cudaErrorInvalidResourceHandle*, *cudaErrorLaunchFailure*

__host__ *cudaError_t* **cudaEventSynchronize**(*cudaEvent_t* event)

Waits for an event to complete.

Waits until the completion of all work currently captured in `event`. See *cudaEventRecord()* for details on what is captured by an event.

Waiting for an event that was created with the *cudaEventBlockingSync* flag will cause the calling CPU thread to block until the event has been completed by the device. If the *cudaEventBlockingSync* flag has not been set, then the CPU thread will busy-wait until the event has been completed by the device.

See also:

cudaEventCreate (C API), *cudaEventCreateWithFlags*, *cudaEventRecord*, *cudaEventQuery*, *cudaEventDestroy*, *cudaEventElapsedTime*, ::cuEventSynchronize

Note: Note that this function may also return error codes from previous, asynchronous launches.

Note: Note that this function may also return *cudaErrorInitializationError*, *cudaErrorInsufficientDriver* or *cudaErrorNoDevice* if this call tries to initialize internal CUDA RT state.

Note: Note that as specified by *cudaStreamAddCallback* no CUDA function may be called from callback. *cudaErrorNotPermitted* may, but is not guaranteed to, be returned as a diagnostic in such case.

Note: Returns *cudaErrorInvalidResourceHandle* in the event of being passed NULL as the input event.

Parameters

event – - Event to wait for

Returns

cudaSuccess, *cudaErrorInvalidValue*, *cudaErrorInvalidResourceHandle*, *cudaErrorLaunchFailure*

6.16. Execution Context Management

This section describes the execution context management functions of the CUDA runtime application programming interface.

6.16.1. Overview

A CUDA execution context *cudaExecutionContext_t* serves as an abstraction for the contexts exposed by the CUDA Runtime, specifically green contexts and the primary context, and provides a unified programming model and API interface for contexts in the Runtime.

There are two primary ways today to obtain an execution context:

- *cudaDeviceGetExecutionCtx*: Returns the execution context that corresponds to the primary context of the specified device.
- *cudaGreenCtxCreate*: Creates a green context with the specified resources and returns an execution context.

Once you have an execution context at hand, you can perform context-level operations via the CUDA Runtime APIs. This includes:

- Submitting work via streams created with *cudaExecutionCtxStreamCreate*.
- Querying context via *cudaExecutionCtxGetDevResource*, *cudaExecutionCtxGetDevice*, etc.
- Synchronizing and tracking context-level operations via *cudaExecutionCtxSynchronize*, *cudaExecutionCtxRecordEvent*, *cudaExecutionCtxWaitEvent*.
- Performing context-level graph node operations via *cudaGraphAddNode* by specifying the context in `nodeParams`. Note that individual node creation APIs, such as *cudaGraphAddKernelNode*, do not support specifying an execution context.

Note: The above APIs take in an explicit cudaExecutionContext_t handle and ignores the context that is current to the calling thread. This enables explicit context-based programming without relying on thread-local state. If no context is specified, the APIs return *cudaErrorInvalidValue*.

Note: Developers should treat *cudaExecutionContext_t* as an opaque handle and avoid assumptions about its underlying representation. The CUDA Runtime does not provide a way to convert this handle into driver-level contexts, such as ::CUcontext or ::CUgreenCtx.

6.16.2. Lifetime of CUDA Resources

The lifetime of CUDA resources (memory, streams, events, modules, etc) is not tied to the lifetime of the execution context. Their lifetime is tied to the device against which they were created. As such, usage of *cudaDeviceReset()* should be avoided to persist the lifetime of these resources.

6.16.3. APIs Operating on Current Context

The CUDA runtime does not provide a way to set an execution context as current. Since, the majority of the runtime APIs operate on the current context, we document below how the developer can work with these APIs.

6.16.3.1 APIs Operating on Device Resources

To work with these APIs (for example, *cudaMalloc*, *cudaEventCreate*, etc), developers are expected to call *cudaSetDevice()* prior to invoking them. Doing so does not impact functional correctness as these APIs operate on resources that are device-wide. If users have a context handle at hand, they can get the device handle from the context handle using *cudaExecutionCtxGetDevice()*.

6.16.3.2 APIs Operating on Context Resources

These APIs (for example, *cudaLaunchKernel*, *cudaMemcpyAsync*, *cudaMemsetAsync*, etc) take in a stream and resources are inferred from the context bound to the stream at creation. See *cudaExecutionCtxStreamCreate* for more details. Developers are expected to use the stream-based APIs for context awareness and always pass an explicit stream handle to ensure context-awareness, and avoid reliance on the default NULL stream, which implicitly binds to the current context.

6.16.4. Green Contexts

Green contexts are a lightweight alternative to traditional contexts, that can be used to select a subset of device resources. This allows the developer to, for example, select SMs from distinct spatial partitions of the GPU and target them via CUDA stream operations, kernel launches, etc.

Here are the broad initial steps to follow to get started:

- (1) Start with an initial set of resources. For SM resources, they can be fetched via *cudaDeviceGetDevResource*. In case of workqueues, a new configuration can be used or an existing one queried via the *cudaDeviceGetDevResource* API.
- (2) Modify these resources by either partitioning them (in case of SMs) or changing the configuration (in case of workqueues). To partition SMs, we recommend *cudaDevSmResourceSplit*. Changing the workqueue configuration can be done directly in place.
- (3) Finalize the specification of resources by creating a descriptor via *cudaDevResourceGenerateDesc*.
- (4) Create a green context via *cudaGreenCtxCreate*. This provisions the resource, such as workqueues (until this step it was only a configuration specification).
- (5) Create a stream via *cudaExecutionCtxStreamCreate*, and use it throughout your application.

SMs

There are two possible partition operations - with *cudaDevSmResourceSplitByCount* the partitions created have to follow default SM count granularity requirements, so it will often be rounded up and aligned to a default value. On the other hand, *cudaDevSmResourceSplit* is explicit and allows for creation of non-equal groups. It will not round up automatically - instead it is the developer's responsibility to query and set the correct values. These requirements can be queried with *cudaDeviceGetDevResource* to determine the alignment granularity (sm.smCoscheduledAlignment). A general guideline on the default values for each compute architecture:

- On Compute Architecture 7.X, 8.X, and all Tegra SoC:
  - The smCount must be a multiple of 2.
  - The alignment (and default value of coscheduledSmCount) is 2.
- On Compute Architecture 9.0+:
  - The smCount must be a multiple of 8, or coscheduledSmCount if provided.
  - The alignment (and default value of coscheduledSmCount) is 8. While the maximum value for coscheduled SM count is 32 on all Compute Architecture 9.0+, it's recommended to follow cluster size requirements. The portable cluster size and the max cluster size should be used in order to benefit from this co-scheduling.

Workqueues

For `cudaDevResourceTypeWorkqueueConfig`, the resource specifies the expected maximum number of concurrent stream-ordered workloads via the `wqConcurrencyLimit` field. The `sharingScope` field determines how workqueue resources are shared:

- `cudaDevWorkqueueConfigScopeDeviceCtx:` Use all shared workqueue resources across all contexts (default driver behavior).
- `cudaDevWorkqueueConfigScopeGreenCtxBalanced:` When possible, use non-overlapping workqueue resources with other balanced green contexts.

The maximum concurrency limit depends on ::CUDA_DEVICE_MAX_CONNECTIONS and can be queried from the device via *cudaDeviceGetDevResource*. Configurations may exceed this concurrency limit, but the driver will not guarantee that work submission remains non-overlapping.

For `cudaDevResourceTypeWorkqueue`, the resource represents a pre-existing workqueue that can be retrieved from existing execution contexts. This allows reusing workqueue resources across different execution contexts.

On Concurrency

Even if the green contexts have disjoint SM partitions, it is not guaranteed that the kernels launched in them will run concurrently or have forward progress guarantees. This is due to other resources that could cause a dependency. Using a combination of disjoint SMs and `cudaDevWorkqueueConfigScopeGreenCtxBalanced` workqueue configurations can provide the best chance of avoiding interference. More resources will be added in the future to provide stronger guarantees.

Additionally, there are two known scenarios, where its possible for the workload to run on more SMs than was provisioned (but never less).

- On Volta+ MPS: When `CUDA_MPS_ACTIVE_THREAD_PERCENTAGE` is used, the set of SMs that are used for running kernels can be scaled up to the value of SMs used for the MPS client.
- On Compute Architecture 9.x: When a module with dynamic parallelism (CDP) is loaded, all future kernels running under green contexts may use and share an additional set of 2 SMs.

Memory Copy Operations

Green context restrictions apply to memory copy operations only when the copy is performed using a green context. For cross-device copies, green context restrictions may not be applied.

Functions

__host__ cudaError_t *cudaDevResourceGenerateDesc*(cudaDevResourceDesc_t *phDesc, cudaDevResource *resources, unsigned int nbResources)
Generate a resource descriptor.

__host__ cudaError_t *cudaDevSmResourceSplit*(cudaDevResource *result, unsigned int nbGroups, const cudaDevResource *input, cudaDevResource *remainder, unsigned int flags, cudaDevSmResourceGroupParams *groupParams)
Splits a `cudaDevResourceTypeSm` resource into structured groups.

__host__ cudaError_t *cudaDevSmResourceSplitByCount*(cudaDevResource *result, unsigned int *nbGroups, const cudaDevResource *input, cudaDevResource *remaining, unsigned int flags, unsigned int minCount)
Splits `cudaDevResourceTypeSm` resources.

__host__ cudaError_t *cudaDeviceGetDevResource*(int device, cudaDevResource *resource, enum cudaDevResourceType type)
Get device resources.

__host__ cudaError_t *cudaDeviceGetExecutionCtx*(cudaExecutionContext_t *ctx, int device)
Returns the execution context for a device.

__host__ cudaError_t *cudaExecutionCtxDestroy*(cudaExecutionContext_t ctx)
Destroy a execution context.

__host__ cudaError_t *cudaExecutionCtxGetDevResource*(cudaExecutionContext_t ctx, cudaDevResource *resource, enum cudaDevResourceType type)
Get context resources.

__host__ cudaError_t *cudaExecutionCtxGetDevice*(int *device, cudaExecutionContext_t ctx)
Returns the device handle for the execution context.

__host__ cudaError_t *cudaExecutionCtxGetId*(cudaExecutionContext_t ctx, unsigned long long *ctxId)
Returns the unique Id associated with the execution context supplied.

__host__ cudaError_t *cudaExecutionCtxRecordEvent*(cudaExecutionContext_t ctx, cudaEvent_t event)
Records an event for the specified execution context.

__host__ cudaError_t *cudaExecutionCtxStreamCreate*(cudaStream_t *phStream, cudaExecutionContext_t ctx, unsigned int flags, int priority)
Creates a stream and initializes it for the given execution context.

__host__ cudaError_t *cudaExecutionCtxSynchronize*(cudaExecutionContext_t ctx)
Block for the specified execution context's tasks to complete.

__host__ cudaError_t *cudaExecutionCtxWaitEvent*(cudaExecutionContext_t ctx, cudaEvent_t event)
Make an execution context wait on an event.

__host__ cudaError_t *cudaGreenCtxCreate*(cudaExecutionContext_t *phCtx, cudaDevResourceDesc_t desc, int device, unsigned int flags)
Creates a green context with a specified set of resources.

__host__ cudaError_t *cudaStreamGetDevResource*(cudaStream_t hStream, cudaDevResource *resource, enum cudaDevResourceType type)
Get stream resources.

6.16.5. Functions

__host__ *cudaError_t* **cudaDevResourceGenerateDesc**(*cudaDevResourceDesc_t* *phDesc, *cudaDevResource* *resources, unsigned int nbResources)

Generate a resource descriptor.

Generates a single resource descriptor with the set of resources specified in `resources`. The generated resource descriptor is necessary for the creation of green contexts via the *cudaGreenCtxCreate* API. Resources of the same type can be passed in, provided they meet the requirements as noted below.

A successful API call must have:

- A valid output pointer for the phDesc descriptor as well as a valid array of `resources` pointers, with the array size passed in `nbResources`. If multiple resources are provided in `resources`, the device they came from must be the same, otherwise *cudaErrorInvalidResourceConfiguration* is returned. If multiple resources are provided in `resources` and they are of type *cudaDevResourceTypeSm*, they must be outputs (whether `result` or `remaining`) from the same split API instance and have the same smCoscheduledAlignment values, otherwise *cudaErrorInvalidResourceConfiguration* is returned.

Note: The API is not supported on 32-bit platforms.

See also:

::cuDevResourceGenerateDesc, *cudaDeviceGetDevResource*, *cudaExecutionCtxGetDevResource*, *cudaDevSmResourceSplit*, *cudaGreenCtxCreate*

Note: Note that as specified by *cudaStreamAddCallback* no CUDA function may be called from callback. *cudaErrorNotPermitted* may, but is not guaranteed to, be returned as a diagnostic in such case.

Parameters

- **phDesc** – - Output descriptor
- **resources** – - Array of resources to be included in the descriptor
- **nbResources** – - Number of resources passed in `resources`

Returns

cudaSuccess, *cudaErrorInvalidValue*, *cudaErrorNotPermitted*, *cudaErrorInvalidResourceType*, *cudaErrorInvalidResourceConfiguration*, *cudaErrorNotSupported*, ::cudaErrorOutOfMemory, *cudaErrorCudartUnloading*, *cudaErrorInitializationError*

__host__ *cudaError_t* **cudaDevSmResourceSplit**(*cudaDevResource* *result, unsigned int nbGroups, const *cudaDevResource* *input, *cudaDevResource* *remainder, unsigned int flags, *cudaDevSmResourceGroupParams* *groupParams)

Splits a cudaDevResourceTypeSm resource into structured groups.

This API will split a resource of *cudaDevResourceTypeSm* into nbGroups structured device resource groups (the result array), as well as an optional remainder, according to a set of requirements specified in the groupParams array. The term "structured" is a trait that specifies the result has SMs that are co-scheduled together. This co-scheduling can be specified via the coscheduledSmCount field of the groupParams structure, while the smCount will specify how many SMs are required in total for that result. The remainder is always "unstructured", it does not have any set guarantees with respect to co-scheduling and those properties will need to either be queried via the occupancy set of APIs or further split into structured groups by this API.

The API has a discovery mode for use cases where it is difficult to know ahead of time what the SM count should be. Discovery happens when the smCount field of a given groupParams array entry is set to 0 - the smCount will be filled in by the API with the derived SM count according to the provided groupParams fields and constraints. Discovery can be used with both a valid result array and with a NULL result pointer value. The latter is useful in situations where the smCount will end up being zero, which is an invalid value to create a result entry with, but allowed for discovery purposes when the result is NULL.

The groupParams array is evaluated from index 0 to nbGroups - 1. For each index in the groupParams array, the API will evaluate which SMs may be a good fit based on constraints and assign those SMs to result. This evaluation order is important to consider when using discovery mode, as it helps discover the remaining SMs.

For a valid call:

- result should point to a *cudaDevResource* array of size nbGroups, or alternatively, may be NULL, if the developer wishes for only the groupParams entries to be updated
- input should be a valid *cudaDevResourceTypeSm* resource that originates from querying the execution context, or device.
- The remainder group may be NULL.
- There are no API flags at this time, so the value passed in should be 0.
- A *cudaDevSmResourceGroupParams* array of size nbGroups. Each entry must be zero-initialized.
  - smCount: must be either 0 or in the range of [2,inputSmCount] where inputSmCount is the amount of SMs the input resource has. smCount must be a multiple of 2, as well as a multiple of coscheduledSmCount. When assigning SMs to a group (and if results are expected by having the result parameter set), smCount cannot end up with 0 or a value less than coscheduledSmCount otherwise *cudaErrorInvalidResourceConfiguration* will be returned.
  - coscheduledSmCount: allows grouping SMs together in order to be able to launch clusters on Compute Architecture 9.0+. The default value may be queried from the device's *cudaDevResourceTypeSm* resource (8 on Compute Architecture 9.0+ and 2 otherwise). The maximum is 32 on Compute Architecture 9.0+ and 2 otherwise.
  - preferredCoscheduledSmCount: Attempts to merge coscheduledSmCount groups into larger groups, in order to make use of preferredClusterDimensions on Compute Architecture 10.0+. The default value is set to coscheduledSmCount.

- `flags:`
  - `cudaDevSmResourceGroupBackfill:` Treats constraints as a hint, ignoring them if necessary to reach the requested `smCount`. Lets `smCount` be a non-multiple of `coscheduledSmCount`, filling the difference between SM count and already assigned co-scheduled groupings with other SMs. This lets any resulting group behave similar to the `remainder` group for example. When used with `cudaDevSmResourceGroupLocalityDomainId`, backfill fills up to the requested `smCount` using the target locality domain first, then SMs not attributed to any locality domain, then SMs from other locality domains. If no SMs can be found in the requested locality domain, *cudaErrorInvalidResourceConfiguration* is returned. When used with `cudaDevSmResourceGroupLocalityDomainSupplementalSms`, backfill follows the same behavior after first consuming all SMs from the target locality domain.
  - `cudaDevSmResourceGroupLocalityDomainId:` Specifies that the SM partition should be localized to the specified `localityDomainId`.
  - `cudaDevSmResourceGroupLocalityDomainSupplementalSms:` Modifier to `cudaDevSmResourceGroupLocalityDomainId` which allows selection of SMs in the locality domain exceeding the ::cudaDeviceAttr::cudaDevAttrLocalityDomainMultiprocessorCount.
- `localityDomainId:` Specifies the locality domain that the partitioned SMs must be located on. Only valid when `cudaDevSmResourceGroupLocalityDomainId` is set in flags. The special value `cudaDevSmResourceGroupLocalityDomainIdAny` specifies that the partitioned SMs must be selected from the union of all available locality domains in the input SM resource.

Example params and their effect:

A groupParams array element is defined in the following order:

```
{ .smCount, .coscheduledSmCount, .preferredCoscheduledSmCount, .flags, \/\* .
↪reserved \*\/ }
```

```
// Example 1
// Will discover how many SMs there are, that are co-scheduled in groups of
↪smCoscheduledAlignment.
// The rest is placed in the optional remainder.
cudaDevSmResourceGroupParams params { 0, 0, 0, 0 };
```

```
// Example 2
// Assuming the device has 10+ SMs, the result will have 10 SMs that are co-
↪scheduled in groups of 2 SMs.
// The rest is placed in the optional remainder.
cudaDevSmResourceGroupParams params { 10, 2, 0, 0};
// Setting the coscheduledSmCount to 2 guarantees that we can always have a valid
↪result
// as long as the SM count is less than or equal to the input resource SM count.
```

```
// Example 3
// A single piece is split-off, but instead of assigning the rest to the remainder,
↪a second group contains everything else
// This assumes the device has 10+ SMs (8 of which are coscheduled in groups of 4),
// otherwise the second group could end up with 0 SMs, which is not allowed.
cudaDevSmResourceGroupParams params[] = {
```

(continues on next page)

(continued from previous page)

```
    {8, 4, 0, 0},
    {0, 2, 0, cudaDevSmResourceGroupBackfill},
};
```

The difference between a catch-all param group as the last entry and the remainder is in two aspects:

- The remainder may be NULL / _TYPE_INVALID (if there are no SMs remaining), while a result group must always be valid.
- The remainder does not have a structure, while the result group will always need to adhere to a structure of coscheduledSmCount (even if its just 2), and therefore must always have enough coscheduled SMs to cover that requirement (even with the `cudaDevSmResourceGroupBackfill` flag enabled).

Splitting an input into N groups, can be accomplished by repeatedly splitting off 1 group and re-splitting the remainder (a bisect operation). However, it's recommended to accomplish this with a single call wherever possible.

See also:

::cuDevSmResourceSplit, *cudaDeviceGetDevResource*, *cudaExecutionCtxGetDevResource*, *cudaDevResourceGenerateDesc*

Note: Note that as specified by *cudaStreamAddCallback* no CUDA function may be called from callback. *cudaErrorNotPermitted* may, but is not guaranteed to, be returned as a diagnostic in such case.

Parameters

- **`result`** – - Output array of *`cudaDevResource`* resources. Can be NULL, alongside an smCount of 0, for discovery purpose.
- **`nbGroups`** – - Specifies the number of groups in `result` and `groupParams`
- **`input`** – - Input SM resource to be split. Must be a valid `cudaDevResourceTypeSm` resource.
- **`remainder`** – - If splitting the input resource leaves any SMs, the remainder is placed in here.
- **`flags`** – - Flags specifying how the API should behave. The value should be 0 for now.
- **`groupParams`** – - Description of how the SMs should be split and assigned to the corresponding result entry.

Returns

cudaSuccess, *cudaErrorInvalidValue*, *cudaErrorNotPermitted*, *cudaErrorInvalidResourceType*, *cudaErrorInvalidResourceConfiguration*, *cudaErrorNotSupported*, *cudaErrorCudartUnloading*, *cudaErrorInitializationError*

__host__ *cudaError_t* **cudaDevSmResourceSplitByCount**(*cudaDevResource* *result, unsigned int *nbGroups, const *cudaDevResource* *input, *cudaDevResource* *remaining, unsigned int flags, unsigned int minCount)

Splits `cudaDevResourceTypeSm` resources.

Splits `cudaDevResourceTypeSm` resources into `nbGroups`, adhering to the minimum SM count specified in `minCount` and the usage flags in `flags`. If `result` is NULL, the API simulates a split and provides the amount of groups that would be created in `nbGroups`. Otherwise, `nbGroups` must point to the amount of elements in `result` and on return, the API will overwrite `nbGroups` with the amount actually created. The groups are written to the array in `result`. `nbGroups` can be less than the total amount if a smaller number of groups is needed.

This API is used to spatially partition the input resource. The input resource needs to come from one of *cudaDeviceGetDevResource*, or *cudaExecutionCtxGetDevResource*. A limitation of the API is that the output results cannot be split again without first creating a descriptor and a green context with that descriptor.

When creating the groups, the API will take into account the performance and functional characteristics of the input resource, and guarantee a split that will create a disjoint set of symmetrical partitions. This may lead to fewer groups created than purely dividing the total SM count by the `minCount` due to cluster requirements or alignment and granularity requirements for the minCount. These requirements can be queried with *cudaDeviceGetDevResource*, or *cudaExecutionCtxGetDevResource* for *cudaDevResourceTypeSm*, using the `minSmPartitionSize` and `smCoscheduledAlignment` fields to determine minimum partition size and alignment granularity, respectively.

The `remainder` set does not have the same functional or performance guarantees as the groups in `result`. Its use should be carefully planned and future partitions of the `remainder` set are discouraged.

The following flags are supported:

- `cudaDevSmResourceSplitIgnoreSmCoscheduling` : Lower the minimum SM count and alignment, and treat each SM independent of its hierarchy. This allows more fine grained partitions but at the cost of advanced features (such as large clusters on compute capability 9.0+).
- `cudaDevSmResourceSplitMaxPotentialClusterSize` : Compute Capability 9.0+ only. Attempt to create groups that may allow for maximally sized thread clusters. This can be queried post green context creation using *cudaOccupancyMaxPotentialClusterSize*.

A successful API call must either have:

- A valid array of `result` pointers of size passed in `nbGroups`, with `input` of type `cudaDevResourceTypeSm`. Value of `minCount` must be between 0 and the SM count specified in `input`. `remaining` may be NULL.
- NULL passed in for `result`, with a valid integer pointer in `nbGroups` and `input` of type `cudaDevResourceTypeSm`. Value of `minCount` must be between 0 and the SM count specified in `input`. `remaining` may be NULL. This queries the number of groups that would be created by the API.

Note: The API is not supported on 32-bit platforms.

See also:

::cuDevSmResourceSplitByCount, *cudaDeviceGetDevResource*, *cudaExecutionCtxGetDevResource*, *cudaDevResourceGenerateDesc*

Note: Note that as specified by *cudaStreamAddCallback* no CUDA function may be called from callback. *cudaErrorNotPermitted* may, but is not guaranteed to, be returned as a diagnostic in such case.

Parameters

- **result** – - Output array of *cudaDevResource* resources. Can be NULL to query the number of groups.
- **nbGroups** – - This is a pointer, specifying the number of groups that would be or should be created as described below.
- **input** – - Input SM resource to be split. Must be a valid *cudaDevSmResource* resource.
- **remaining** – - If the input resource cannot be cleanly split among `nbGroups`, the remaining is placed in here. Can be ommitted (NULL) if the user does not need the remaining set.
- **flags** – - Flags specifying how these partitions are used or which constraints to abide by when splitting the input. Zero is valid for default behavior.
- **minCount** – - Minimum number of SMs required

Returns

cudaSuccess, *cudaErrorInvalidValue*, *cudaErrorNotPermitted*, *cudaErrorInvalidResourceType*, *cudaErrorInvalidResourceConfiguration*, *cudaErrorNotSupported*, *cudaErrorCudartUnloading*, *cudaErrorInitializationError*

__host__ *cudaError_t* **cudaDeviceGetDevResource**(int device, *cudaDevResource* *resource, enum *cudaDevResourceType* type)

Get device resources.

Get the `type` resources available to the `device`. This may often be the starting point for further partitioning or configuring of resources.

Note: The API is not supported on 32-bit platforms.

See also:

::cuDeviceGetDevResource, *cudaExecutionCtxGetDevResource*, *cudaDevSmResourceSplit*, *cudaDevResourceGenerateDesc*

Note: Note that as specified by *cudaStreamAddCallback* no CUDA function may be called from callback. *cudaErrorNotPermitted* may, but is not guaranteed to, be returned as a diagnostic in such case.

Parameters

- **device** – - Device to get resource for
- **resource** – - Output pointer to a *cudaDevResource* structure
- **type** – - Type of resource to retrieve

Returns

cudaSuccess, *cudaErrorInvalidValue*, *cudaErrorNotPermitted*, *cudaErrorInvalidDevice*, *cudaErrorInvalidResourceType*, *cudaErrorNotSupported*, *cudaErrorCudartUnloading*, *cudaErrorInitializationError*

__host__ *cudaError_t* **cudaDeviceGetExecutionCtx**(*cudaExecutionContext_t* *ctx, int device)

Returns the execution context for a device.

Returns in `ctx` the execution context for the specified device. This is the device's primary context. The returned context can then be passed to APIs that take in a cudaExecutionContext_t enabling explicit context-based programming without relying on thread-local state.

Passing the returned execution context to *cudaExecutionCtxDestroy()* is not allowed and will result in undefined behavior.

See also:

cudaExecutionCtxGetDevice, *cudaExecutionCtxGetId*

Parameters

- **ctx** – - Returns the device execution context
- **device** – - Device to get the execution context for

Returns

cudaSuccess, *cudaErrorInvalidValue*, *cudaErrorInvalidDevice*

__host__ *cudaError_t* **cudaExecutionCtxDestroy**(*cudaExecutionContext_t* ctx)

Destroy a execution context.

Destroys the specified execution context `ctx`. It is the responsibility of the caller to ensure that no API call issues using `ctx` while *cudaExecutionCtxDestroy()* is executing or subsequently.

If `ctx` is a green context, any resources provisioned for it (that were initially available via the resource descriptor) are released as well.

The API does not destroy streams created via *cudaExecutionCtxStreamCreate*. Users are expected to destroy these streams explicitly using *cudaStreamDestroy* to avoid resource leaks. Once the execution context is destroyed, any subsequent API calls involving these streams will return *cudaErrorStreamDetached* with the exception of the following APIs:

- *cudaStreamDestroy*. Note this is only supported on CUDA drivers 13.1 and above.

Additionally, the API will invalidate all active captures on these streams.

Passing in a `ctx` that was not explicitly created via CUDA Runtime APIs is not allowed and will result in undefined behavior.

See also:

cudaGreenCtxCreate

Note: Note that as specified by *cudaStreamAddCallback* no CUDA function may be called from callback. *cudaErrorNotPermitted* may, but is not guaranteed to, be returned as a diagnostic in such case.

Note: The context parameter is required and the API ignores the context that is current to the calling thread. This enables explicit context-based programming without relying on thread-local state. If no context is specified, the API will return *cudaErrorInvalidValue*.

Parameters

ctx – - Execution context to destroy (required parameter, see note below)

Returns

cudaSuccess, *cudaErrorInvalidValue*, *cudaErrorNotPermitted*, *cudaErrorCudartUnloading*, *cudaErrorInitializationError*

__host__ *cudaError_t* **cudaExecutionCtxGetDevResource**(*cudaExecutionContext_t* ctx, *cudaDevResource* *resource, enum *cudaDevResourceType* type)

Get context resources.

Get the `type` resources available to context represented by `ctx`.

Note: The API is not supported on 32-bit platforms.

See also:

cudaDeviceGetDevResource, *cudaDevSmResourceSplit*, *cudaDevResourceGenerateDesc*, *cudaGreenCtxCreate*

Note: Note that this function may also return error codes from previous, asynchronous launches.

Note: Note that as specified by *cudaStreamAddCallback* no CUDA function may be called from callback. *cudaErrorNotPermitted* may, but is not guaranteed to, be returned as a diagnostic in such case.

Note: The context parameter is required and the API ignores the context that is current to the calling thread. This enables explicit context-based programming without relying on thread-local state. If no context is specified, the API will return *cudaErrorInvalidValue*.

Parameters

- **ctx** – - Execution context to get resource for (required parameter, see note below)
- **resource** – - Output pointer to a *cudaDevResource* structure
- **type** – - Type of resource to retrieve

Returns

cudaSuccess, *cudaErrorInvalidValue*, *cudaErrorNotSupported*, *cudaErrorNotPermitted*, *cudaErrorCudartUnloading*, *cudaErrorInitializationError*

__host__ *cudaError_t* **cudaExecutionCtxGetDevice**(int *device, *cudaExecutionContext_t* ctx)

Returns the device handle for the execution context.

Returns in `*device` the handle of the specified execution context's device. The execution context should not be NULL.

See also:

cudaGreenCtxCreate, *cudaExecutionCtxDestroy*, ::cuCtxGetDevice

---

Note: Note that this function may also return error codes from previous, asynchronous launches.

Note: Note that as specified by *cudaStreamAddCallback* no CUDA function may be called from callback. *cudaErrorNotPermitted* may, but is not guaranteed to, be returned as a diagnostic in such case.

Note: The context parameter is required and the API ignores the context that is current to the calling thread. This enables explicit context-based programming without relying on thread-local state. If no context is specified, the API will return *cudaErrorInvalidValue*.

Parameters

- **device** – - Returned device handle for the specified execution context
- **ctx** – - Execution context for which to obtain the device (required parameter, see note below)

Returns

cudaSuccess, *cudaErrorCudartUnloading*, *cudaErrorInitializationError*, *cudaErrorInvalidValue*, *cudaErrorNotPermitted*

__host__ *cudaError_t* **cudaExecutionCtxGetId**(*cudaExecutionContext_t* ctx, unsigned long long *ctxId)

Returns the unique Id associated with the execution context supplied.

Returns in `ctxId` the unique Id which is associated with a given context. The Id is unique for the life of the program for this instance of CUDA. The execution context should not be NULL.

See also:

cudaGreenCtxCreate, *cudaExecutionCtxDestroy*, *cudaExecutionCtxGetDevice*, ::cuCtxGetId

---

Note: Note that this function may also return error codes from previous, asynchronous launches.

Note: Note that as specified by *cudaStreamAddCallback* no CUDA function may be called from callback. *cudaErrorNotPermitted* may, but is not guaranteed to, be returned as a diagnostic in such case.

Note: The context parameter is required and the API ignores the context that is current to the calling thread. This enables explicit context-based programming without relying on thread-local state. If no context is specified, the API will return *cudaErrorInvalidValue*.

Parameters

- **ctx** – - Context for which to obtain the Id (required parameter, see note below)
- **ctxId** – - Pointer to store the Id of the context

Returns

cudaSuccess, *cudaErrorCudartUnloading*, *cudaErrorInitializationError*, *cudaErrorInvalidValue*, *cudaErrorNotPermitted*

__host__ *cudaError_t* **cudaExecutionCtxRecordEvent**(*cudaExecutionContext_t* ctx, *cudaEvent_t* event)

Records an event for the specified execution context.

Captures in `event` all the activities of the execution context `ctx` at the time of this call. `event` and `ctx` must be from the same CUDA device, otherwise ::cudaErrorInvalidHandle will be returned. Calls such as *cudaEventQuery()* or *cudaExecutionCtxWaitEvent()* will then examine or wait for completion of the work that was captured. Uses of `ctx` after this call do not modify `event`. If the execution context passed to `ctx` is the device (primary) context obtained via *cudaDeviceGetExecutionCtx()*, `event` will capture all the activities of the green contexts created on the device as well.

See also:

cudaEventRecord, *cudaExecutionCtxWaitEvent*, ::cuCtxRecordEvent, ::cuGreenCtxRecordEvent

Note: The API will return *cudaErrorStreamCaptureUnsupported* if the specified execution context `ctx` has a stream in the capture mode. In such a case, the call will invalidate all the conflicting captures.

Note: Note that this function may also return error codes from previous, asynchronous launches.

Note: Note that as specified by *cudaStreamAddCallback* no CUDA function may be called from callback. *cudaErrorNotPermitted* may, but is not guaranteed to, be returned as a diagnostic in such case.

Note: The context parameter is required and the API ignores the context that is current to the calling thread. This enables explicit context-based programming without relying on thread-local

state. If no context is specified, the API will return *cudaErrorInvalidValue*.

Parameters

- **ctx** – - Execution context to record event for (required parameter, see note below)
- **event** – - Event to record

Returns

cudaSuccess, *cudaErrorCudartUnloading*, *cudaErrorInitializationError*, ::cudaErrorInvalidHandle, *cudaErrorStreamCaptureUnsupported*

__host__ *cudaError_t* **cudaExecutionCtxStreamCreate**(*cudaStream_t* *phStream, *cudaExecutionContext_t* ctx, unsigned int flags, int priority)

Creates a stream and initializes it for the given execution context.

The API creates a CUDA stream with the specified `flags` and `priority`, initializing it with resources as defined at the time of creating the specified `ctx`. Additionally, the API also enables work submitted to to the stream to be tracked under `ctx`.

The supported values for `flags` are:

- *cudaStreamDefault*: Default stream creation flag. This would be *cudaStreamNonBlocking* for streams created on a green context.
- *cudaStreamNonBlocking*: Specifies that work running in the created stream may run concurrently with work in stream 0 (the NULL stream), and that the created stream should perform no implicit synchronization with stream 0

Specifying `priority` affects the scheduling priority of work in the stream. Priorities provide a hint to preferentially run work with higher priority when possible, but do not preempt already-running work or provide any other functional guarantee on execution order. `priority` follows a convention where lower numbers represent higher priorities. '0' represents default priority. The range of meaningful numerical priorities can be queried using *cudaDeviceGetStreamPriorityRange*. If the specified priority is outside the numerical range returned by *cudaDeviceGetStreamPriorityRange*, it will automatically be clamped to the lowest or the highest number in the range.

See also:

cudaStreamDestroy, *cudaGreenCtxCreate*, *cudaDeviceGetStreamPriorityRange*, *cudaStreamGetFlags*, *cudaStreamGetPriority*, *cudaStreamGetDevice*, *cudaStreamGetDevResource*, *cudaLaunchKernel*, *cudaEventRecord*, *cudaStreamWaitEvent*, *cudaStreamQuery*, *cudaStreamSynchronize*, *cudaStreamAddCallback*

Note: Note that this function may also return error codes from previous, asynchronous launches.

Note: Note that as specified by *cudaStreamAddCallback* no CUDA function may be called from callback. *cudaErrorNotPermitted* may, but is not guaranteed to, be returned as a diagnostic in such case.

Note: The context parameter is required and the API ignores the context that is current to the calling thread. This enables explicit context-based programming without relying on thread-local state. If no context is specified, the API will return *cudaErrorInvalidValue*.

Note: In the current implementation, only compute kernels launched in priority streams are affected by the stream's priority. Stream priorities have no effect on host-to-device and device-to-host memory operations.

Parameters

- **phStream** – - Returned stream handle
- **ctx** – - Execution context to initialize the stream with (required parameter, see note below)
- **flags** – - Flags for stream creation
- **priority** – - Stream priority

Returns

cudaSuccess, *cudaErrorInvalidValue*, *cudaErrorNotPermitted*, ::cudaErrorOutOfMemory, *cudaErrorCudartUnloading*, *cudaErrorInitializationError*

__host__ *cudaError_t* **cudaExecutionCtxSynchronize**(*cudaExecutionContext_t* ctx)

Block for the specified execution context's tasks to complete.

Blocks until the specified execution context has completed all preceding requested tasks. If the specified execution context is the device (primary) context obtained via *cudaDeviceGetExecutionCtx*, green contexts that have been created on the device will also be synchronized.

The API returns an error if one of the preceding tasks failed.

See also:

cudaGreenCtxCreate, *cudaExecutionCtxDestroy*, *cudaDeviceSynchronize*, ::cuCtxSynchronize_v2

Note: Note that this function may also return error codes from previous, asynchronous launches.

Note: Note that as specified by *cudaStreamAddCallback* no CUDA function may be called from callback. *cudaErrorNotPermitted* may, but is not guaranteed to, be returned as a diagnostic in such case.

Note: The context parameter is required and the API ignores the context that is current to the calling thread. This enables explicit context-based programming without relying on thread-local state. If no context is specified, the API will return *cudaErrorInvalidValue*.

Parameters

ctx – - Execution context to synchronize (required parameter, see note below)

Returns

cudaSuccess, *cudaErrorCudartUnloading*, *cudaErrorDeviceUninitialized*, *cudaErrorInvalidValue*

__host__ *cudaError_t* **cudaExecutionCtxWaitEvent**(*cudaExecutionContext_t* ctx, *cudaEvent_t* event)

Make an execution context wait on an event.

Makes all future work submitted to execution context `ctx` wait for all work captured in `event`. The synchronization will be performed on the device and will not block the calling CPU thread. See *cudaExecutionCtxRecordEvent()* for details on what is captured by an event. If the execution context passed to `ctx` is the device (primary) context obtained via *cudaDeviceGetExecutionCtx()*, all green contexts created on the device will wait for `event` as well.

See also:

cudaExecutionCtxRecordEvent, *cudaStreamWaitEvent*, ::cuCtxWaitEvent, ::cuGreenCtxWaitEvent

---

Note: `event` may be from a different execution context or device than `ctx`.

Note: The API will return *cudaErrorStreamCaptureUnsupported* and invalidate the capture if the specified event `event` is part of an ongoing capture sequence or if the specified execution context `ctx` has a stream in the capture mode.

Note: Note that this function may also return error codes from previous, asynchronous launches.

Note: Note that as specified by *cudaStreamAddCallback* no CUDA function may be called from callback. *cudaErrorNotPermitted* may, but is not guaranteed to, be returned as a diagnostic in such case.

Note: The context parameter is required and the API ignores the context that is current to the calling thread. This enables explicit context-based programming without relying on thread-local state. If no context is specified, the API will return *cudaErrorInvalidValue*.

Parameters

- **ctx** – - Execution context to wait for (required parameter, see note below)
- **event** – - Event to wait on

Returns

cudaSuccess, *cudaErrorCudartUnloading*, *cudaErrorInitializationError*, ::cudaErrorInvalidHandle, *cudaErrorStreamCaptureUnsupported*

__host__ *cudaError_t* **cudaGreenCtxCreate**(*cudaExecutionContext_t* *phCtx, *cudaDevResourceDesc_t* desc, int device, unsigned int flags)

Creates a green context with a specified set of resources.

This API creates a green context with the resources specified in the descriptor `desc` and returns it in the handle represented by `phCtx`.

This API retains the device's primary context for the lifetime of the green context. The primary context will be released when the green context is destroyed. To avoid the overhead of repeated initialization and teardown, it is recommended to explicitly initialize the device's primary context ahead of time using *cudaInitDevice*. This ensures that the primary context remains initialized throughout the program's lifetime, minimizing overhead during green context creation and destruction.

The API does not create a default stream for the green context. Developers are expected to create streams explicitly using *cudaExecutionCtxStreamCreate* to submit work to the green context.

Note: The API is not supported on 32-bit platforms.

See also:

cudaDeviceGetDevResource, *cudaDevSmResourceSplit*, *cudaDevResourceGenerateDesc*, *cudaExecutionCtxGetDevResource*, *cudaExecutionCtxDestroy*, *cudaInitDevice*, *cudaExecutionCtxStreamCreate*

Note: Note that as specified by *cudaStreamAddCallback* no CUDA function may be called from callback. *cudaErrorNotPermitted* may, but is not guaranteed to, be returned as a diagnostic in such case.

Parameters

- **phCtx** – - Pointer for the output handle to the green context
- **desc** – - Descriptor generated via *cudaDevResourceGenerateDesc* which contains the set of resources to be used
- **device** – - Device on which to create the green context.
- **flags** – - Green context creation flags. Must be 0, currently reserved for future use.

Returns

cudaSuccess, *cudaErrorInvalidValue*, *cudaErrorInvalidDevice*, *cudaErrorNotPermitted*, *cudaErrorNotSupported*, ::cudaErrorOutOfMemory, *cudaErrorCudartUnloading*, *cudaErrorInitializationError*

__host__ *cudaError_t* **cudaStreamGetDevResource**(*cudaStream_t* hStream, *cudaDevResource* *resource, enum *cudaDevResourceType* type)

Get stream resources.

Get the `type` resources available to the `hStream` and store them in `resource`.

Note: The API will return *cudaErrorInvalidResourceType* is `type` is `cudaDevResourceTypeWorkqueueConfig` or `cudaDevResourceTypeWorkqueue`.

See also:

cudaGreenCtxCreate, *cudaExecutionCtxStreamCreate*, *cudaStreamCreate*, *cudaDevSmResourceSplit*, *cudaDevResourceGenerateDesc*, ::cuStreamGetDevResource

---

Note: Note that this function may also return error codes from previous, asynchronous launches.

Note: Note that as specified by *cudaStreamAddCallback* no CUDA function may be called from callback. *cudaErrorNotPermitted* may, but is not guaranteed to, be returned as a diagnostic in such case.

Parameters

- **hStream** – - Stream to get resource for
- **resource** – - Output pointer to a *cudaDevResource* structure
- **type** – - Type of resource to retrieve

Returns

cudaSuccess, *cudaErrorCudartUnloading*, *cudaErrorInitializationError*, *cudaErrorDeviceUninitialized*, *cudaErrorInvalidResourceType*, *cudaErrorInvalidValue*, ::cudaErrorInvalidHandle, *cudaErrorNotPermitted*, *cudaErrorCallRequiresNewerDriver*,

6.17. Execution Control

This section describes the execution control functions of the CUDA runtime application programming interface.

Some functions have overloaded C++ API template versions documented separately in the *C++ API Routines* module.

Functions

__host__ cudaError_t *cudaFuncGetAttributes*(struct cudaFuncAttributes *attr, const void *func)
Find out attributes for a given function.

__host__ cudaError_t *cudaFuncGetName*(const char **name, const void *func)
Returns the function name for a device entry function pointer.

__host__ cudaError_t *cudaFuncGetParamCount*(const void *func, size_t *paramCount)
Returns the number of parameters used by the function.

__host__ cudaError_t *cudaFuncGetParamInfo*(const void *func, size_t paramIndex, size_t *paramOffset, size_t *paramSize)
Returns the offset and size of a kernel parameter in the device-side parameter layout.

__host__ cudaError_t *cudaFuncSetAttribute*(const void *func, enum cudaFuncAttribute attr, int value)
Set attributes for a given function.

__host__ cudaError_t *cudaFuncSetCacheConfig*(const void *func, enum cudaFuncCache cacheConfig)
Sets the preferred cache configuration for a device function.

__device__ void * *cudaGetParameterBuffer*(size_t alignment, size_t size)
Obtains a parameter buffer.

__device__ void *cudaGridDependencySynchronize*(void)
Programmatic grid dependency synchronization.

__host__ cudaError_t *cudaLaunchCooperativeKernel*(const void *func, dim3 gridDim, dim3 blockDim, void **args, size_t sharedMem, cudaStream_t stream)
Launches a device function where thread blocks can cooperate and synchronize as they execute.

__device__ cudaError_t *cudaLaunchDevice*(void *func, void *parameterBuffer, dim3 gridDimension, dim3 blockDimension, unsigned int sharedMemSize, cudaStream_t stream)
Launches a specified kernel.

__host__ cudaError_t *cudaLaunchHostFunc*(cudaStream_t stream, cudaHostFn_t fn, void *userData)
Enqueues a host function call in a stream.

__host__ cudaError_t *cudaLaunchHostFunc_v2*(cudaStream_t stream, cudaHostFn_t fn, void *userData, unsigned int syncMode)
Enqueues a host function call in a stream.

__host__ cudaError_t *cudaLaunchKernel*(const void *func, dim3 gridDim, dim3 blockDim, void **args, size_t sharedMem, cudaStream_t stream)
Launches a device function.

__host__ cudaError_t *cudaLaunchKernelExC*(const cudaLaunchConfig_t *config, const void *func, void **args)
Launches a CUDA function with launch-time configuration.

__device__ void *cudaTriggerProgrammaticLaunchCompletion*(void)
Programmatic dependency trigger.

6.17.1. Functions

__host__ *cudaError_t* **cudaFuncGetAttributes**(struct *cudaFuncAttributes* *attr, const void *func)

Find out attributes for a given function.

This function obtains the attributes of a function specified via `func`. `func` is a device function symbol and must be declared as a `__global__` function. The fetched attributes are placed in `attr`. If the specified function does not exist, then it is assumed to be a *cudaKernel_t* and used as is. For templated functions, pass the function symbol as follows: func_name<template_arg_0,...,template_arg_N>

Note that some function attributes such as *maxThreadsPerBlock* may vary based on the device that is currently being used.

See also:

cudaFuncSetCacheConfig (C API), *cudaFuncGetAttributes (C++ API)*, *cudaLaunchKernel (C API)*, ::cuFuncGetAttribute

Note: Note that this function may also return error codes from previous, asynchronous launches.

Note: Use of a string naming a function as the `func` parameter was deprecated in CUDA 4.1 and removed in CUDA 5.0.

Note: Note that this function may also return *cudaErrorInitializationError*, *cudaErrorInsufficientDriver* or *cudaErrorNoDevice* if this call tries to initialize internal CUDA RT state.

Note: Note that as specified by *cudaStreamAddCallback* no CUDA function may be called from callback. *cudaErrorNotPermitted* may, but is not guaranteed to, be returned as a diagnostic in such case.

Note: The API can also be used with a kernel *cudaKernel_t* by querying the handle using *cudaLibraryGetKernel()* or *cudaGetKernel* and then passing it to the API by casting to void*. The symbol `entryFuncAddr` passed to *cudaGetKernel* should be a symbol that is registered with the same CUDA Runtime instance.

Note: Passing a symbol that belongs that belongs to a different runtime instance will result in undefined behavior. The only type that can be reliably passed to a different runtime instance is *cudaKernel_t*

Parameters

- **attr** – - Return pointer to function's attributes
- **func** – - Device function symbol

Returns

cudaSuccess, *cudaErrorInvalidDeviceFunction*

__host__ *cudaError_t* **cudaFuncGetName**(const char **name, const void *func)

Returns the function name for a device entry function pointer.

Returns in `**name` the function name associated with the symbol `func` . The function name is returned as a null-terminated string. This API may return a mangled name if the function is not declared as having C linkage. If `**name` is NULL, *cudaErrorInvalidValue* is returned. If `func` is not a device entry function, then it is assumed to be a *cudaKernel_t* and used as is.

cudaFuncGetName (C++ API)

Note: Note that this function may also return error codes from previous, asynchronous launches.

Note: Note that this function may also return *cudaErrorInitializationError*, *cudaErrorInsufficientDriver* or *cudaErrorNoDevice* if this call tries to initialize internal CUDA RT state.

Note: Note that as specified by *cudaStreamAddCallback* no CUDA function may be called from callback. *cudaErrorNotPermitted* may, but is not guaranteed to, be returned as a diagnostic in such case.

Note: The API can also be used with a kernel *cudaKernel_t* by querying the handle using *cudaLibraryGetKernel()* or *cudaGetKernel* and then passing it to the API by casting to void*. The symbol `entryFuncAddr` passed to *cudaGetKernel* should be a symbol that is registered with the same CUDA Runtime instance.

Note: Passing a symbol that belongs that belongs to a different runtime instance will result in undefined behavior. The only type that can be reliably passed to a different runtime instance is *cudaKernel_t*

Parameters

- **name** – - The returned name of the function
- **func** – - The function pointer to retrieve name for

Returns
cudaSuccess, *cudaErrorInvalidValue*, *cudaErrorInvalidDeviceFunction*

__host__ *cudaError_t* **cudaFuncGetParamCount**(const void *func, size_t *paramCount)

Returns the number of parameters used by the function.

Queries the number of kernel parameters used by `func` and returns it in `paramCount`.

Note: Note that this function may also return error codes from previous, asynchronous launches.

Note: The API can also be used with a kernel *cudaKernel_t* by querying the handle using *cudaLibraryGetKernel()* or *cudaGetKernel* and then passing it to the API by casting to void*. The symbol `entryFuncAddr` passed to *cudaGetKernel* should be a symbol that is registered with the same CUDA Runtime instance.

Note: Passing a symbol that belongs that belongs to a different runtime instance will result in undefined behavior. The only type that can be reliably passed to a different runtime instance is *cudaKernel_t*

Parameters

- **func** – - The function to query
- **paramCount** – - Returns the number of parameters used by the function

Returns
::CUDA_SUCCESS, ::CUDA_ERROR_INVALID_VALUE,

__host__ *cudaError_t* **cudaFuncGetParamInfo**(const void *func, size_t paramIndex, size_t *paramOffset, size_t *paramSize)

Returns the offset and size of a kernel parameter in the device-side parameter layout.

Queries the kernel parameter at `paramIndex` in `func`'s list of parameters and returns parameter information via `paramOffset` and `paramSize`. `paramOffset` returns the offset of the parameter in the device-side parameter layout. `paramSize` returns the size in bytes of the parameter. This information can be used to update kernel node parameters from the device via *cudaGraphKernelNodeSetParam()* and *cudaGraphKernelNodeUpdatesApply()*. `paramIndex` must be less than the number of parameters that `func` takes.

Note: Note that this function may also return error codes from previous, asynchronous launches.

Note: The API can also be used with a kernel *cudaKernel_t* by querying the handle using *cudaLibraryGetKernel()* or *cudaGetKernel* and then passing it to the API by casting to void*. The symbol `entryFuncAddr` passed to *cudaGetKernel* should be a symbol that is registered with the same CUDA Runtime instance.

Note: Passing a symbol that belongs that belongs to a different runtime instance will result in undefined behavior. The only type that can be reliably passed to a different runtime instance is *cudaKernel_t*

Parameters

- **func** – - The function to query
- **paramIndex** – - The parameter index to query
- **paramOffset** – - The offset into the device-side parameter layout at which the parameter resides
- **paramSize** – - The size of the parameter in the device-side parameter layout

Returns
::CUDA_SUCCESS, ::CUDA_ERROR_INVALID_VALUE,

__host__ *cudaError_t* **cudaFuncSetAttribute**(const void *func, enum *cudaFuncAttribute* attr, int value)

Set attributes for a given function.

This function sets the attributes of a function specified via `func`. The parameter `func` must be a pointer to a function that executes on the device. The parameter specified by `func` must be declared as a `__global__` function. The enumeration defined by `attr` is set to the value defined by `value`. If the specified function does not exist, then it is assumed to be a *cudaKernel_t*

and used as is. If the specified attribute cannot be written, or if the value is incorrect, then *cudaErrorInvalidValue* is returned.

Valid values for `attr` are:

- *cudaFuncAttributeMaxDynamicSharedMemorySize* - The requested maximum size in bytes of dynamically-allocated shared memory. The sum of this value and the function attribute ::sharedSizeBytes cannot exceed the device attribute *cudaDevAttrMaxSharedMemoryPerBlockOptin*. The maximal size of requestable dynamic shared memory may differ by GPU architecture.
- *cudaFuncAttributePreferredSharedMemoryCarveout* - On devices where the L1 cache and shared memory use the same hardware resources, this sets the shared memory carveout preference, in percent of the total shared memory. See *cudaDevAttrMaxSharedMemoryPerMultiprocessor*. This is only a hint, and the driver can choose a different ratio if required to execute the function.
- *cudaFuncAttributeRequiredClusterWidth*: The required cluster width in blocks. The width, height, and depth values must either all be 0 or all be positive. The validity of the cluster dimensions is checked at launch time. If the value is set during compile time, it cannot be set at runtime. Setting it at runtime will return cudaErrorNotPermitted.
- *cudaFuncAttributeRequiredClusterHeight*: The required cluster height in blocks. The width, height, and depth values must either all be 0 or all be positive. The validity of the cluster dimensions is checked at launch time. If the value is set during compile time, it cannot be set at runtime. Setting it at runtime will return cudaErrorNotPermitted.
- *cudaFuncAttributeRequiredClusterDepth*: The required cluster depth in blocks. The width, height, and depth values must either all be 0 or all be positive. The validity of the cluster dimensions is checked at launch time. If the value is set during compile time, it cannot be set at runtime. Setting it at runtime will return cudaErrorNotPermitted.
- *cudaFuncAttributeNonPortableClusterSizeAllowed*: Indicates whether the function can be launched with non-portable cluster size. 1 is allowed, 0 is disallowed.
- *cudaFuncAttributeClusterSchedulingPolicyPreference*: The block scheduling policy of a function. The value type is cudaClusterSchedulingPolicy.

cudaLaunchKernel (C++ API), *cudaFuncSetCacheConfig (C++ API)*, *cudaFuncGetAttributes (C API)*,

---

Note: Note that this function may also return error codes from previous, asynchronous launches.

Note: Note that this function may also return *cudaErrorInitializationError*, *cudaErrorInsufficientDriver* or *cudaErrorNoDevice* if this call tries to initialize internal CUDA RT state.

Note: Note that as specified by *cudaStreamAddCallback* no CUDA function may be called from callback. *cudaErrorNotPermitted* may, but is not guaranteed to, be returned as a diagnostic in such case.

Note: The API can also be used with a kernel *cudaKernel_t* by querying the handle using *cudaLibraryGetKernel()* or *cudaGetKernel* and then passing it to the API by casting to void*. The symbol

`entryFuncAddr` passed to *cudaGetKernel* should be a symbol that is registered with the same CUDA Runtime instance.

Note: Passing a symbol that belongs that belongs to a different runtime instance will result in undefined behavior. The only type that can be reliably passed to a different runtime instance is *cudaKernel_t*

Parameters

- **`func`** – - Function to get attributes of
- **`attr`** – - Attribute to set
- **`value`** – - Value to set

Returns

cudaSuccess, *cudaErrorInvalidDeviceFunction*, *cudaErrorInvalidValue*

__host__ *cudaError_t* **`cudaFuncSetCacheConfig`**(const void *func, enum *cudaFuncCache* cacheConfig)

Sets the preferred cache configuration for a device function.

On devices where the L1 cache and shared memory use the same hardware resources, this sets through `cacheConfig` the preferred cache configuration for the function specified via `func`. This is only a preference. The runtime will use the requested configuration if possible, but it is free to choose a different configuration if required to execute `func`.

`func` is a device function symbol and must be declared as a `__global__` function. If the specified function does not exist, then *cudaErrorInvalidDeviceFunction* is returned. For templated functions, pass the function symbol as follows: func_name<template_arg_0,...,template_arg_N>

This setting does nothing on devices where the size of the L1 cache and shared memory are fixed.

Launching a kernel with a different preference than the most recent preference setting may insert a device-side synchronization point.

The supported cache configurations are:

- *cudaFuncCachePreferNone*: no preference for shared memory or L1 (default)
- *cudaFuncCachePreferShared*: prefer larger shared memory and smaller L1 cache
- *cudaFuncCachePreferL1*: prefer larger L1 cache and smaller shared memory
- *cudaFuncCachePreferEqual*: prefer equal size L1 cache and shared memory

See also:

cudaFuncSetCacheConfig (C++ API), *cudaFuncGetAttributes (C API)*, *cudaLaunchKernel (C API)*, ::cuFuncSetCacheConfig

Note: Note that this function may also return error codes from previous, asynchronous launches.

Note: Use of a string naming a function as the `func` parameter was deprecated in CUDA 4.1 and removed in CUDA 5.0.

Note: Note that this function may also return *cudaErrorInitializationError*, *cudaErrorInsufficientDriver* or *cudaErrorNoDevice* if this call tries to initialize internal CUDA RT state.

Note: Note that as specified by *cudaStreamAddCallback* no CUDA function may be called from callback. *cudaErrorNotPermitted* may, but is not guaranteed to, be returned as a diagnostic in such case.

Note: This API does not accept a *cudaKernel_t* casted as void*. If cache config modification is required for a *cudaKernel_t* (or a **global** function), it can be replaced with a call to ::cudaFuncSetAttributes with the attribute *cudaFuncAttributePreferredSharedMemoryCarveout* to specify a more granular L1 cache and shared memory split configuration.

Parameters

- **func** – - Device function symbol
- **cacheConfig** – - Requested cache configuration

Returns

cudaSuccess, *cudaErrorInvalidDeviceFunction*

__device__ void *__cudaGetParameterBuffer__(size_t alignment, size_t size)

Obtains a parameter buffer.

Obtains a parameter buffer which can be filled with parameters for a kernel launch. Parameters passed to *cudaLaunchDevice* must be allocated via this function.

This is a low level API and can only be accessed from Parallel Thread Execution (PTX). CUDA user code should use <<< >>> to launch kernels.

See also:

cudaLaunchDevice

Note: Note that this function may also return error codes from previous, asynchronous launches.

Parameters

- **alignment** – - Specifies alignment requirement of the parameter buffer
- **size** – - Specifies size requirement in bytes

Returns

Returns pointer to the allocated parameterBuffer

__device__ inline void **cudaGridDependencySynchronize**(void)

Programmatic grid dependency synchronization.

This device function will block the thread until all direct grid dependencies have completed. This API is intended to use in conjuncture with programmatic / launch event / dependency. See ::cudaLaunchAttributeID::cudaLaunchAttributeProgrammaticStreamSerialization and ::cudaLaunchAttributeID::cudaLaunchAttributeProgrammaticEvent for more information.

__host__ *cudaError_t* **cudaLaunchCooperativeKernel**(const void *func, dim3 gridDim, dim3 blockDim, void **args, size_t sharedMem, *cudaStream_t* stream)

Launches a device function where thread blocks can cooperate and synchronize as they execute.

The function invokes kernel `func` on `gridDim` (`gridDim.xgridDim.ygridDim.z`) grid of blocks. Each block contains `blockDim` (`blockDim.xblockDim.yblockDim.z`) threads.

The device on which this kernel is invoked must have a non-zero value for the device attribute *cudaDevAttrCooperativeLaunch*.

The total number of blocks launched cannot exceed the maximum number of blocks per multiprocessor as returned by *cudaOccupancyMaxActiveBlocksPerMultiprocessor* (or *cudaOccupancyMaxActiveBlocksPerMultiprocessorWithFlags*) times the number of multiprocessors as specified by the device attribute *cudaDevAttrMultiProcessorCount*.

The kernel cannot make use of CUDA dynamic parallelism.

If the kernel has N parameters the `args` should point to array of N pointers. Each pointer, from `args[0]` to `args[N - 1]`, point to the region of memory from which the actual parameter will be copied.

For templated functions, pass the function symbol as follows: func_name<template_arg_0,...,template_arg_N>

`sharedMem` sets the amount of dynamic shared memory that will be available to each thread block.

`stream` specifies a stream the invocation is associated to.

See also:

cudaLaunchCooperativeKernel (C++ API), ::cuLaunchCooperativeKernel

---

Note: This function uses standard default stream semantics.

Note: Note that this function may also return error codes from previous, asynchronous launches.

Note: Note that this function may also return *cudaErrorInitializationError*, *cudaErrorInsufficientDriver* or *cudaErrorNoDevice* if this call tries to initialize internal CUDA RT state.

Note: Note that as specified by *cudaStreamAddCallback* no CUDA function may be called from callback. *cudaErrorNotPermitted* may, but is not guaranteed to, be returned as a diagnostic in such case.

Note: The API can also be used with a kernel *cudaKernel_t* by querying the handle using *cudaLibraryGetKernel()* or *cudaGetKernel* and then passing it to the API by casting to void*. The symbol `entryFuncAddr` passed to *cudaGetKernel* should be a symbol that is registered with the same CUDA Runtime instance.

Note: Passing a symbol that belongs that belongs to a different runtime instance will result in undefined behavior. The only type that can be reliably passed to a different runtime instance is *cudaKernel_t*

Parameters

- **func** – - Device function symbol
- **gridDim** – - Grid dimentions
- **blockDim** – - Block dimentions
- **args** – - Arguments
- **sharedMem** – - Shared memory
- **stream** – - Stream identifier

Returns

cudaSuccess, *cudaErrorInvalidDeviceFunction*, *cudaErrorInvalidConfiguration*, *cudaErrorLaunchFailure*, *cudaErrorLaunchTimeout*, *cudaErrorLaunchOutOfResources*, *cudaErrorCooperativeLaunchTooLarge*, *cudaErrorSharedObjectInitFailed*

__device__ *cudaError_t* **cudaLaunchDevice**(void *func, void *parameterBuffer, dim3 gridDimension, dim3 blockDimension, unsigned int sharedMemSize, *cudaStream_t* stream)

Launches a specified kernel.

Launches a specified kernel with the specified parameter buffer. A parameter buffer can be obtained by calling *cudaGetParameterBuffer()*.

This is a low level API and can only be accessed from Parallel Thread Execution (PTX). CUDA user code should use <<< >>> to launch the kernels.

See also:

cudaGetParameterBuffer

Note:

Note that this function may also return error codes from previous, asynchronous launches.

Please refer to Execution Configuration and Parameter Buffer Layout from the CUDA Programming Guide for the detailed descriptions of launch configuration and parameter layout respectively.

Parameters

- **func** – - Pointer to the kernel to be launched
- **parameterBuffer** – - Holds the parameters to the launched kernel. parameterBuffer can be NULL. (Optional)
- **gridDimension** – - Specifies grid dimensions
- **blockDimension** – - Specifies block dimensions
- **sharedMemSize** – - Specifies size of shared memory
- **stream** – - Specifies the stream to be used

Returns

cudaSuccess, *cudaErrorInvalidDevice*, *cudaErrorLaunchMaxDepthExceeded*, *cudaErrorInvalidConfiguration*, *cudaErrorStartupFailure*, *cudaErrorLaunchPendingCountExceeded*, *cudaErrorLaunchOutOfResources*

__host__ *cudaError_t* **cudaLaunchHostFunc**(*cudaStream_t* stream, cudaHostFn_t fn, void *userData)

Enqueues a host function call in a stream.

Enqueues a host function to run in a stream. The function will be called after currently enqueued work and will block work added after it.

The host function must not make any CUDA API calls. Attempting to use a CUDA API may result in *cudaErrorNotPermitted*, but this is not required. The host function must not perform any synchronization that may depend on outstanding CUDA work not mandated to run earlier. Host functions without a mandated order (such as in independent streams) execute in undefined order and may be serialized.

For the purposes of Unified Memory, execution makes a number of guarantees:

- The stream is considered idle for the duration of the function's execution. Thus, for example, the function may always use memory attached to the stream it was enqueued in.
- The start of execution of the function has the same effect as synchronizing an event recorded in the same stream immediately prior to the function. It thus synchronizes streams which have been "joined" prior to the function.
- Adding device work to any stream does not have the effect of making the stream active until all preceding host functions and stream callbacks have executed. Thus, for example, a function might use global attached memory even if work has been added to another stream, if the work has been ordered behind the function call with an event.
- Completion of the function does not cause a stream to become active except as described above. The stream will remain idle if no device work follows the function, and will remain idle across consecutive host functions or stream callbacks without device work in between. Thus, for example, stream synchronization can be done by signaling from a host function at the end of the stream.

Note that, in constrast to ::cuStreamAddCallback, the function will not be called in the event of an error in the CUDA context.

See also:

cudaStreamCreate, *cudaStreamQuery*, *cudaStreamSynchronize*, *cudaStreamWaitEvent*, *cudaStreamDestroy*, *cudaMallocManaged*, *cudaStreamAttachMemAsync*, *cudaStreamAddCallback*, ::cuLaunchHostFunc

Note: This function uses standard default stream semantics.

Note: Note that this function may also return error codes from previous, asynchronous launches.

Note: Note that this function may also return *cudaErrorInitializationError*, *cudaErrorInsufficientDriver* or *cudaErrorNoDevice* if this call tries to initialize internal CUDA RT state.

Note: Note that as specified by *cudaStreamAddCallback* no CUDA function may be called from callback. *cudaErrorNotPermitted* may, but is not guaranteed to, be returned as a diagnostic in such case.

Parameters

- **stream** – - Stream to enqueue function call in
- **fn** – - The function to call once preceding stream operations are complete
- **userData** – - User-specified data to be passed to the function

Returns

cudaSuccess, *cudaErrorInvalidResourceHandle*, *cudaErrorInvalidValue*, *cudaErrorNotSupported*

__host__ *cudaError_t* **cudaLaunchHostFunc_v2**(*cudaStream_t* stream, cudaHostFn_t fn, void *userData, unsigned int syncMode)

Enqueues a host function call in a stream.

Enqueues a host function to run in a stream. The function will be called after currently enqueued work and will block work added after it.

The host function must not make any CUDA API calls. Attempting to use a CUDA API may result in *cudaErrorNotPermitted*, but this is not required. The host function must not perform any synchronization that may depend on outstanding CUDA work not mandated to run earlier. Host functions without a mandated order (such as in independent streams) execute in undefined order and may be serialized.

For the purposes of Unified Memory, execution makes a number of guarantees:

- The stream is considered idle for the duration of the function's execution. Thus, for example, the function may always use memory attached to the stream it was enqueued in.
- The start of execution of the function has the same effect as synchronizing an event recorded in the same stream immediately prior to the function. It thus synchronizes streams which have been "joined" prior to the function.
- Adding device work to any stream does not have the effect of making the stream active until all preceding host functions and stream callbacks have executed. Thus, for example, a function might use global attached memory even if work has been added to another stream, if the work has been ordered behind the function call with an event.
- Completion of the function does not cause a stream to become active except as described above. The stream will remain idle if no device work follows the function, and will remain

idle across consecutive host functions or stream callbacks without device work in between. Thus, for example, stream synchronization can be done by signaling from a host function at the end of the stream.

Note that, in constrast to ::cuStreamAddCallback, the function will not be called in the event of an error in the CUDA context.

See also:

cudaStreamCreate, *cudaStreamQuery*, *cudaStreamSynchronize*, *cudaStreamWaitEvent*, *cudaStreamDestroy*, *cudaMallocManaged*, *cudaStreamAttachMemAsync*, *cudaStreamAddCallback*, ::cuLaunchHostFunc

---

Note: This function uses standard default stream semantics.

Note: Note that this function may also return error codes from previous, asynchronous launches.

Note: Note that this function may also return *cudaErrorInitializationError*, *cudaErrorInsufficientDriver* or *cudaErrorNoDevice* if this call tries to initialize internal CUDA RT state.

Note: Note that as specified by *cudaStreamAddCallback* no CUDA function may be called from callback. *cudaErrorNotPermitted* may, but is not guaranteed to, be returned as a diagnostic in such case.

Parameters

- **`stream`** – - Stream to enqueue function call in
- **`fn`** – - The function to call once preceding stream operations are complete
- **`userData`** – - User-specified data to be passed to the function
- **`syncMode`** – - Sync mode for the host function

Returns

cudaSuccess, *cudaErrorInvalidResourceHandle*, *cudaErrorInvalidValue*, *cudaErrorNotSupported*

__host__ *cudaError_t* **`cudaLaunchKernel`**(const void *func, dim3 gridDim, dim3 blockDim, void **args, size_t sharedMem, *cudaStream_t* stream)

Launches a device function.

The function invokes kernel `func` on `gridDim` (`gridDim.xgridDim.ygridDim.z`) grid of blocks. Each block contains `blockDim` (`blockDim.xblockDim.yblockDim.z`) threads.

If the kernel has N parameters the `args` should point to array of N pointers. Each pointer, from `args[0]` to `args[N - 1]`, point to the region of memory from which the actual parameter will be copied.

For templated functions, pass the function symbol as follows: func_name<template_arg_0,...,template_arg_N>

`sharedMem` sets the amount of dynamic shared memory that will be available to each thread block.

`stream` specifies a stream the invocation is associated to.

See also:

cudaLaunchKernel (C++ API), ::cuLaunchKernel

Note: This function uses standard default stream semantics.

Note: Note that this function may also return error codes from previous, asynchronous launches.

Note: Note that this function may also return *cudaErrorInitializationError*, *cudaErrorInsufficientDriver* or *cudaErrorNoDevice* if this call tries to initialize internal CUDA RT state.

Note: Note that as specified by *cudaStreamAddCallback* no CUDA function may be called from callback. *cudaErrorNotPermitted* may, but is not guaranteed to, be returned as a diagnostic in such case.

Note: The API can also be used with a kernel *cudaKernel_t* by querying the handle using *cudaLibraryGetKernel()* or *cudaGetKernel* and then passing it to the API by casting to void*. The symbol `entryFuncAddr` passed to *cudaGetKernel* should be a symbol that is registered with the same CUDA Runtime instance.

Note: Passing a symbol that belongs that belongs to a different runtime instance will result in undefined behavior. The only type that can be reliably passed to a different runtime instance is *cudaKernel_t*

Parameters

- **`func`** – - Device function symbol
- **`gridDim`** – - Grid dimentions
- **`blockDim`** – - Block dimentions
- **`args`** – - Arguments
- **`sharedMem`** – - Shared memory
- **`stream`** – - Stream identifier

Returns

cudaSuccess, *cudaErrorInvalidDeviceFunction*, *cudaErrorInvalidConfiguration*, *cudaErrorLaunchFailure*, *cudaErrorLaunchTimeout*, *cudaErrorLaunchOutOfResources*,

cudaErrorSharedObjectInitFailed, *cudaErrorInvalidPtx*, *cudaErrorUnsupportedPtxVersion*, *cudaErrorNoKernelImageForDevice*, *cudaErrorJitCompilerNotFound*, *cudaErrorJitCompilationDisabled*

__host__ *cudaError_t* **cudaLaunchKernelExC**(const *cudaLaunchConfig_t* *config, const void *func, void **args)

Launches a CUDA function with launch-time configuration.

Note that the functionally equivalent variadic template *cudaLaunchKernelEx* is available for C++11 and newer.

Invokes the kernel `func` on `config->gridDim` (`config->gridDim.x`×`config->gridDim.y`×`config->gridDim.z`) grid of blocks. Each block contains `config->blockDim` (`config->blockDim.x`×`config->blockDim.y`×`config->blockDim.z`) threads.

`config->dynamicSmemBytes` sets the amount of dynamic shared memory that will be available to each thread block.

`config->stream` specifies a stream the invocation is associated to.

Configuration beyond grid and block dimensions, dynamic shared memory size, and stream can be provided with the following two fields of `config`:

`config->attrs` is an array of `config->numAttrs` contiguous *cudaLaunchAttribute* elements. The value of this pointer is not considered if `config->numAttrs` is zero. However, in that case, it is recommended to set the pointer to NULL. `config->numAttrs` is the number of attributes populating the first `config->numAttrs` positions of the `config->attrs` array.

If the kernel has N parameters the `args` should point to array of N pointers. Each pointer, from `args[0]` to `args[N - 1]`, point to the region of memory from which the actual parameter will be copied.

N.B. This function is so named to avoid unintentionally invoking the templated version, `cudaLaunchKernelEx`, for kernels taking a single void** or void* parameter.

See also:

cudaLaunchKernelEx(const *cudaLaunchConfig_t* *config, void (*kernel)(ExpTypes...), ActTypes &&... args) “cudaLaunchKernelEx (C++ API)”, ::cuLaunchKernelEx

---

Note: This function uses standard default stream semantics.

Note: Note that this function may also return error codes from previous, asynchronous launches.

Note: Note that this function may also return *cudaErrorInitializationError*, *cudaErrorInsufficientDriver* or *cudaErrorNoDevice* if this call tries to initialize internal CUDA RT state.

Note: Note that as specified by *cudaStreamAddCallback* no CUDA function may be called from callback. *cudaErrorNotPermitted* may, but is not guaranteed to, be returned as a diagnostic in such case.

Note: The API can also be used with a kernel *cudaKernel_t* by querying the handle using *cudaLibraryGetKernel()* or *cudaGetKernel* and then passing it to the API by casting to void*. The symbol `entryFuncAddr` passed to *cudaGetKernel* should be a symbol that is registered with the same CUDA Runtime instance.

Note: Passing a symbol that belongs that belongs to a different runtime instance will result in undefined behavior. The only type that can be reliably passed to a different runtime instance is *cudaKernel_t*

Parameters

- **config** – - Launch configuration
- **func** – - Kernel to launch
- **args** – - Array of pointers to kernel parameters

Returns

cudaSuccess, *cudaErrorInvalidDeviceFunction*, *cudaErrorInvalidConfiguration*, *cudaErrorLaunchFailure*, *cudaErrorLaunchTimeout*, *cudaErrorLaunchOutOfResources*, *cudaErrorSharedObjectInitFailed*, *cudaErrorInvalidPtx*, *cudaErrorUnsupportedPtxVersion*, *cudaErrorNoKernelImageForDevice*, *cudaErrorJitCompilerNotFound*, *cudaErrorJitCompilationDisabled*

__device__ inline void **cudaTriggerProgrammaticLaunchCompletion**(void)

Programmatic dependency trigger.

This device function ensures the programmatic launch completion edges / events are fulfilled. See ::cudaLaunchAttributeID::cudaLaunchAttributeProgrammaticStreamSerialization and ::cudaLaunchAttributeID::cudaLaunchAttributeProgrammaticEvent for more information. The event / edge kick off only happens when every CTAs in the grid has either exited or called this function at least once, otherwise the kick off happens automatically after all warps finishes execution but before the grid completes. The kick off only enables scheduling of the secondary kernel. It provides no memory visibility guarantee itself. The user could enforce memory visibility by inserting a memory fence of the correct scope.

6.18. Execution Control [DEPRECATED]

This section describes the deprecated execution control functions of the CUDA runtime application programming interface.

Some functions have overloaded C++ API template versions documented separately in the *C++ API Routines* module.

Functions

__host__ cudaError_t *cudaFuncSetSharedMemConfig*(const void *func, enum cudaSharedMemConfig config)
Sets the shared memory configuration for a device function.

6.18.1. Functions

__host__ *cudaError_t* **cudaFuncSetSharedMemConfig**(const void *func, enum *cudaSharedMemConfig* config)

Sets the shared memory configuration for a device function.

Deprecated:

On devices with configurable shared memory banks, this function will force all subsequent launches of the specified device function to have the given shared memory bank size configuration. On any given launch of the function, the shared memory configuration of the device will be temporarily changed if needed to suit the function's preferred configuration. Changes in shared memory configuration between subsequent launches of functions, may introduce a device side synchronization point.

Any per-function setting of shared memory bank size set via *cudaFuncSetSharedMemConfig* will override the device wide setting set by *cudaDeviceSetSharedMemConfig*.

Changing the shared memory bank size will not increase shared memory usage or affect occupancy of kernels, but may have major effects on performance. Larger bank sizes will allow for greater potential bandwidth to shared memory, but will change what kinds of accesses to shared memory will result in bank conflicts.

This function will do nothing on devices with fixed shared memory bank size.

For templated functions, pass the function symbol as follows: func_name<template_arg_0,...,template_arg_N>

The supported bank configurations are:

- cudaSharedMemBankSizeDefault: use the device's shared memory configuration when launching this function.
- cudaSharedMemBankSizeFourByte: set shared memory bank width to be four bytes natively when launching this function.
- cudaSharedMemBankSizeEightByte: set shared memory bank width to be eight bytes natively when launching this function.

See also:

cudaDeviceSetSharedMemConfig, *cudaDeviceGetSharedMemConfig*, *cudaDeviceSetCacheConfig*, *cudaDeviceGetCacheConfig*, *cudaFuncSetCacheConfig*, ::cuFuncSetSharedMemConfig

Note: Note that this function may also return error codes from previous, asynchronous launches.

Note: Use of a string naming a function as the `func` parameter was deprecated in CUDA 4.1 and removed in CUDA 5.0.

Note: Note that this function may also return *cudaErrorInitializationError*, *cudaErrorInsufficientDriver* or *cudaErrorNoDevice* if this call tries to initialize internal CUDA RT state.

Note: Note that as specified by *cudaStreamAddCallback* no CUDA function may be called from callback. *cudaErrorNotPermitted* may, but is not guaranteed to, be returned as a diagnostic in such case.

Parameters

- **func** – - Device function symbol
- **config** – - Requested shared memory configuration

Returns

cudaSuccess, *cudaErrorInvalidDeviceFunction*, *cudaErrorInvalidValue*,

6.19. External Resource Interoperability

This section describes the external resource interoperability functions of the CUDA runtime application programming interface.

Functions

__host__ cudaError_t *cudaDestroyExternalMemory*(cudaExternalMemory_t extMem)
Destroys an external memory object.

__host__ cudaError_t *cudaDestroyExternalSemaphore*(cudaExternalSemaphore_t extSem)
Destroys an external semaphore.

__host__ cudaError_t *cudaExternalMemoryGetMappedBuffer*(void **devPtr, cudaExternalMemory_t extMem, const struct cudaExternalMemoryBufferDesc *bufferDesc)
Maps a buffer onto an imported memory object.

__host__ cudaError_t *cudaExternalMemoryGetMappedMipmappedArray*(cudaMipmappedArray_t *mipmap, cudaExternalMemory_t extMem, const struct cudaExternalMemoryMipmappedArrayDesc *mipmapDesc)
Maps a CUDA mipmapped array onto an external memory object.

__host__ cudaError_t *cudaImportExternalMemory*(cudaExternalMemory_t *extMem_out, const struct cudaExternalMemoryHandleDesc *memHandleDesc)
Imports an external memory object.

__host__ cudaError_t *cudaImportExternalSemaphore*(cudaExternalSemaphore_t *extSem_out, const struct cudaExternalSemaphoreHandleDesc *semHandleDesc)
Imports an external semaphore.

__host__ cudaError_t *cudaSignalExternalSemaphoresAsync*(const cudaExternalSemaphore_t *extSemArray, const struct cudaExternalSemaphoreSignalParams *paramsArray, unsigned int numExtSems, cudaStream_t stream=0)
Signals a set of external semaphore objects.

__host__ cudaError_t *cudaWaitExternalSemaphoresAsync*(const cudaExternalSemaphore_t *extSemArray, const struct cudaExternalSemaphoreWaitParams *paramsArray, unsigned int numExtSems, cudaStream_t stream=0)
Waits on a set of external semaphore objects.

6.19.1. Functions

__host__ *cudaError_t* **cudaDestroyExternalMemory**(*cudaExternalMemory_t* extMem)

Destroys an external memory object.

Destroys the specified external memory object. Any existing buffers and CUDA mipmapped arrays mapped onto this object must no longer be used and must be explicitly freed using *cudaFree* and *cudaFreeMipmappedArray* respectively.

See also:

cudaImportExternalMemory, *cudaExternalMemoryGetMappedBuffer*, *cudaExternalMemoryGetMappedMipmappedArray*

Note: Note that this function may also return error codes from previous, asynchronous launches.

Note: Note that this function may also return *cudaErrorInitializationError*, *cudaErrorInsufficientDriver* or *cudaErrorNoDevice* if this call tries to initialize internal CUDA RT state.

Note: Note that as specified by *cudaStreamAddCallback* no CUDA function may be called from callback. *cudaErrorNotPermitted* may, but is not guaranteed to, be returned as a diagnostic in such case.

Note: Use of the handle after this call is undefined behavior.

Parameters
extMem – - External memory object to be destroyed

Returns
cudaSuccess, *cudaErrorInvalidResourceHandle*

__host__ *cudaError_t* **cudaDestroyExternalSemaphore**(*cudaExternalSemaphore_t* extSem)

Destroys an external semaphore.

Destroys an external semaphore object and releases any references to the underlying resource. Any outstanding signals or waits must have completed before the semaphore is destroyed.

See also:

cudaImportExternalSemaphore, *cudaSignalExternalSemaphoresAsync*, *cudaWaitExternalSemaphoresAsync*

Note: Note that this function may also return error codes from previous, asynchronous launches.

Note: Note that this function may also return *cudaErrorInitializationError*, *cudaErrorInsufficientDriver* or *cudaErrorNoDevice* if this call tries to initialize internal CUDA RT state.

Note: Note that as specified by *cudaStreamAddCallback* no CUDA function may be called from callback. *cudaErrorNotPermitted* may, but is not guaranteed to, be returned as a diagnostic in such case.

Note: Use of the handle after this call is undefined behavior.

Parameters

extSem – - External semaphore to be destroyed

Returns

cudaSuccess, *cudaErrorInvalidResourceHandle*

__host__ *cudaError_t* **cudaExternalMemoryGetMappedBuffer**(void **devPtr, *cudaExternalMemory_t* extMem, const struct *cudaExternalMemoryBufferDesc* *bufferDesc)

Maps a buffer onto an imported memory object.

Maps a buffer onto an imported memory object and returns a device pointer in `devPtr`.

The properties of the buffer being mapped must be described in `bufferDesc`. The *cudaExternalMemoryBufferDesc* structure is defined as follows:

```
typedef struct cudaExternalMemoryBufferDesc_st {
    unsigned long long offset;
    unsigned long long size;
    unsigned int flags;
} cudaExternalMemoryBufferDesc;
```

where *cudaExternalMemoryBufferDesc::offset* is the offset in the memory object where the buffer's base address is. *cudaExternalMemoryBufferDesc::size* is the size of the buffer. *cudaExternalMemoryBufferDesc::flags* must be zero.

The offset and size have to be suitably aligned to match the requirements of the external API. Mapping two buffers whose ranges overlap may or may not result in the same virtual address being returned for the overlapped portion. In such cases, the application must ensure that all accesses to that region from the GPU are volatile. Otherwise writes made via one address are

not guaranteed to be visible via the other address, even if they're issued by the same thread. It is recommended that applications map the combined range instead of mapping separate buffers and then apply the appropriate offsets to the returned pointer to derive the individual buffers.

The returned pointer `devPtr` must be freed using *cudaFree*.

See also:

cudaImportExternalMemory, *cudaDestroyExternalMemory*, *cudaExternalMemoryGetMappedMipmappedArray*

Note: Note that this function may also return error codes from previous, asynchronous launches.

Note: Note that this function may also return *cudaErrorInitializationError*, *cudaErrorInsufficientDriver* or *cudaErrorNoDevice* if this call tries to initialize internal CUDA RT state.

Note: Note that as specified by *cudaStreamAddCallback* no CUDA function may be called from callback. *cudaErrorNotPermitted* may, but is not guaranteed to, be returned as a diagnostic in such case.

Parameters

- **devPtr** – - Returned device pointer to buffer
- **extMem** – - Handle to external memory object
- **bufferDesc** – - Buffer descriptor

Returns

cudaSuccess, *cudaErrorInvalidValue*, *cudaErrorInvalidResourceHandle*

__host__ *cudaError_t* **cudaExternalMemoryGetMappedMipmappedArray**(*cudaMipmappedArray_t* *mipmap, *cudaExternalMemory_t* extMem, const struct *cudaExternalMemoryMipmappedArrayDesc* *mipmapDesc)

Maps a CUDA mipmapped array onto an external memory object.

Maps a CUDA mipmapped array onto an external object and returns a handle to it in `mipmap`.

The properties of the CUDA mipmapped array being mapped must be described in `mipmapDesc`. The structure *cudaExternalMemoryMipmappedArrayDesc* is defined as follows:

```
typedef struct cudaExternalMemoryMipmappedArrayDesc_st {
    unsigned long long offset;
    cudaChannelFormatDesc formatDesc;
    cudaExtent extent;
```

(continues on next page)

(continued from previous page)

```
    unsigned int flags;
    unsigned int numLevels;
} cudaExternalMemoryMipmappedArrayDesc;
```

where *cudaExternalMemoryMipmappedArrayDesc::offset* is the offset in the memory object where the base level of the mipmap chain is. *cudaExternalMemoryMipmappedArrayDesc::formatDesc* describes the format of the data. *cudaExternalMemoryMipmappedArrayDesc::extent* specifies the dimensions of the base level of the mipmap chain. *cudaExternalMemoryMipmappedArrayDesc::flags* are flags associated with CUDA mipmapped arrays. For further details, please refer to the documentation for *cudaMalloc3DArray*. Note that if the mipmapped array is bound as a color target in the graphics API, then the flag *cudaArrayColorAttachment* must be specified in *cudaExternalMemoryMipmappedArrayDesc::flags*. *cudaExternalMemoryMipmappedArrayDesc::numLevels* specifies the total number of levels in the mipmap chain.

The returned CUDA mipmapped array must be freed using *cudaFreeMipmappedArray*.

See also:

cudaImportExternalMemory, *cudaDestroyExternalMemory*, *cudaExternalMemoryGetMappedBuffer*

Note: Note that this function may also return error codes from previous, asynchronous launches.

Note: Note that this function may also return *cudaErrorInitializationError*, *cudaErrorInsufficientDriver* or *cudaErrorNoDevice* if this call tries to initialize internal CUDA RT state.

Note: Note that as specified by *cudaStreamAddCallback* no CUDA function may be called from callback. *cudaErrorNotPermitted* may, but is not guaranteed to, be returned as a diagnostic in such case.

Note: If *cudaExternalMemoryHandleDesc::type* is *cudaExternalMemoryHandleTypeNvSciBuf*, then *cudaExternalMemoryMipmappedArrayDesc::numLevels* must not be greater than 1.

Parameters

- **mipmap** – - Returned CUDA mipmapped array
- **extMem** – - Handle to external memory object
- **mipmapDesc** – - CUDA array descriptor

Returns

cudaSuccess, *cudaErrorInvalidValue*, *cudaErrorInvalidResourceHandle*

__host__ *cudaError_t* **cudaImportExternalMemory**(*cudaExternalMemory_t* *extMem_out, const struct *cudaExternalMemoryHandleDesc* *memHandleDesc)

Imports an external memory object.

Imports an externally allocated memory object and returns a handle to that in `extMem_out`.

The properties of the handle being imported must be described in `memHandleDesc`. The *cudaExternalMemoryHandleDesc* structure is defined as follows:

```
typedef struct cudaExternalMemoryHandleDesc_st {
    cudaExternalMemoryHandleType type;
    union {
        int fd;
        struct {
            void *handle;
            const void *name;
        } win32;
        const void *nvSciBufObject;
    } handle;
    unsigned long long size;
    unsigned int flags;
} cudaExternalMemoryHandleDesc;
```

where *cudaExternalMemoryHandleDesc::type* specifies the type of handle being imported. *cudaExternalMemoryHandleType* is defined as:

```
typedef enum cudaExternalMemoryHandleType_enum {
    cudaExternalMemoryHandleTypeOpaqueFd          = 1,
    cudaExternalMemoryHandleTypeOpaqueWin32       = 2,
    cudaExternalMemoryHandleTypeOpaqueWin32Kmt    = 3,
    cudaExternalMemoryHandleTypeD3D12Heap         = 4,
    cudaExternalMemoryHandleTypeD3D12Resource     = 5,
    cudaExternalMemoryHandleTypeD3D11Resource     = 6,
   cudaExternalMemoryHandleTypeD3D11ResourceKmt = 7,
    cudaExternalMemoryHandleTypeNvSciBuf          = 8
} cudaExternalMemoryHandleType;
```

If *cudaExternalMemoryHandleDesc::type* is *cudaExternalMemoryHandleTypeOpaqueFd*, then ::cudaExternalMemoryHandleDesc::handle::fd must be a valid file descriptor referencing a memory object. Ownership of the file descriptor is transferred to the CUDA driver when the handle is imported successfully. Performing any operations on the file descriptor after it is imported results in undefined behavior.

If *cudaExternalMemoryHandleDesc::type* is *cudaExternalMemoryHandleTypeOpaqueWin32*, then exactly one of ::cudaExternalMemoryHandleDesc::handle::win32::handle and ::cudaExternalMemoryHandleDesc::handle::win32::name must not be NULL. If ::cudaExternalMemoryHandleDesc::handle::win32::handle is not NULL, then it must represent a valid shared NT handle that references a memory object. Ownership of this handle is not transferred to CUDA after the import operation, so the application must release the handle using the appropriate system call. If ::cudaExternalMemoryHandleDesc::handle::win32::name is not NULL, then it must point to a NULL-terminated array of UTF-16 characters that refers to a memory object.

If *cudaExternalMemoryHandleDesc::type* is *cudaExternalMemoryHandleTypeOpaqueWin32Kmt*, then ::cudaExternalMemoryHandleDesc::handle::win32::handle must be non-NULL and ::cudaExternalMemoryHandleDesc::handle::win32::name must be NULL. The handle specified must be a globally shared KMT handle. This handle does not hold a reference to the underlying object, and

thus will be invalid when all references to the memory object are destroyed.

If *cudaExternalMemoryHandleDesc::type* is *cudaExternalMemoryHandleTypeD3D12Heap*, then exactly one of ::cudaExternalMemoryHandleDesc::handle::win32::handle and ::cudaExternalMemoryHandleDesc::handle::win32::name must not be NULL. If ::cudaExternalMemoryHandleDesc::handle::win32::handle is not NULL, then it must represent a valid shared NT handle that is returned by ID3D12Device::CreateSharedHandle when referring to a ID3D12Heap object. This handle holds a reference to the underlying object. If ::cudaExternalMemoryHandleDesc::handle::win32::name is not NULL, then it must point to a NULL-terminated array of UTF-16 characters that refers to a ID3D12Heap object.

If *cudaExternalMemoryHandleDesc::type* is *cudaExternalMemoryHandleTypeD3D12Resource*, then exactly one of ::cudaExternalMemoryHandleDesc::handle::win32::handle and ::cudaExternalMemoryHandleDesc::handle::win32::name must not be NULL. If ::cudaExternalMemoryHandleDesc::handle::win32::handle is not NULL, then it must represent a valid shared NT handle that is returned by ID3D12Device::CreateSharedHandle when referring to a ID3D12Resource object. This handle holds a reference to the underlying object. If ::cudaExternalMemoryHandleDesc::handle::win32::name is not NULL, then it must point to a NULL-terminated array of UTF-16 characters that refers to a ID3D12Resource object.

If *cudaExternalMemoryHandleDesc::type* is *cudaExternalMemoryHandleTypeD3D11Resource* ,then exactly one of ::cudaExternalMemoryHandleDesc::handle::win32::handle and ::cudaExternalMemoryHandleDesc::handle::win32::name must not be NULL. If ::cudaExternalMemoryHandleDesc::handle::win32::handle is

not NULL, then it must represent a valid shared NT handle that is

returned by IDXGIResource1::CreateSharedHandle when referring to a ID3D11Resource object. If ::cudaExternalMemoryHandleDesc::handle::win32::name is not NULL, then it must point to a NULL-terminated array of UTF-16 characters that refers to a ID3D11Resource object.

If *cudaExternalMemoryHandleDesc::type* is *cudaExternalMemoryHandleTypeD3D11ResourceKmt*, then ::cudaExternalMemoryHandleDesc::handle::win32::handle must be non-NULL and ::cudaExternalMemoryHandleDesc::handle::win32::name must be NULL. The handle specified must be a valid shared KMT handle that is returned by IDXGIResource::GetSharedHandle when referring to a ID3D11Resource object.

If *cudaExternalMemoryHandleDesc::type* is *cudaExternalMemoryHandleTypeNvSciBuf*, then ::cudaExternalMemoryHandleDesc::handle::nvSciBufObject must be NON-NULL and reference a valid NvSciBuf object. If the NvSciBuf object imported into CUDA is also mapped by other drivers, then the application must use *cudaWaitExternalSemaphoresAsync* or *cudaSignalExternalSemaphoresAsync* as appropriate barriers to maintain coherence between CUDA and the other drivers. See *cudaExternalSemaphoreWaitSkipNvSciBufMemSync* and *cudaExternalSemaphoreSignalSkipNvSciBufMemSync* for memory synchronization.

The size of the memory object must be specified in *cudaExternalMemoryHandleDesc::size*.

Specifying the flag *cudaExternalMemoryDedicated* in *cudaExternalMemoryHandleDesc::flags* indicates that the resource is a dedicated resource. The definition of what a dedicated resource is outside the scope of this extension. This flag must be set if *cudaExternalMemoryHandleDesc::type* is one of the following: *cudaExternalMemoryHandleTypeD3D12Resource cudaExternalMemoryHandleTypeD3D11Resource cudaExternalMemoryHandleTypeD3D11ResourceKmt*

See also:

cudaDestroyExternalMemory, *cudaExternalMemoryGetMappedBuffer*, *cudaExternalMemoryGetMappedMipmappedArray*

Note: Note that this function may also return error codes from previous, asynchronous launches.

Note: Note that this function may also return *cudaErrorInitializationError*, *cudaErrorInsufficientDriver* or *cudaErrorNoDevice* if this call tries to initialize internal CUDA RT state.

Note: Note that as specified by *cudaStreamAddCallback* no CUDA function may be called from callback. *cudaErrorNotPermitted* may, but is not guaranteed to, be returned as a diagnostic in such case.

Note:

If the Vulkan memory imported into CUDA is mapped on the CPU then the application must use vkInvalidateMappedMemoryRanges/vkFlushMappedMemoryRanges as well as appropriate Vulkan pipeline barriers to maintain coherence between CPU and GPU. For more information on these APIs, please refer to "Synchronization

and Cache Control" chapter from Vulkan specification.

Parameters

- **extMem_out** – - Returned handle to an external memory object
- **memHandleDesc** – - Memory import handle descriptor

Returns

cudaSuccess, *cudaErrorInvalidValue*, *cudaErrorInvalidResourceHandle*, *cudaErrorOperatingSystem*

__host__ *cudaError_t* **cudaImportExternalSemaphore**(*cudaExternalSemaphore_t* *extSem_out, const struct *cudaExternalSemaphoreHandleDesc* *semHandleDesc)

Imports an external semaphore.

Imports an externally allocated synchronization object and returns a handle to that in `extSem_out`.

The properties of the handle being imported must be described in `semHandleDesc`. The *cudaExternalSemaphoreHandleDesc* is defined as follows:

```
typedef struct cudaExternalSemaphoreHandleDesc_st {
    cudaExternalSemaphoreHandleType type;
    union {
        int fd;
        struct {
            void *handle;
            const void *name;
        } win32;
```

(continues on next page)

(continued from previous page)

```
        const void* NvSciSyncObj;
    } handle;
    unsigned int flags;
} cudaExternalSemaphoreHandleDesc;
```

where *cudaExternalSemaphoreHandleDesc::type* specifies the type of handle being imported. *cudaExternalSemaphoreHandleType* is defined as:

```
typedef enum cudaExternalSemaphoreHandleType_enum {
    cudaExternalSemaphoreHandleTypeOpaqueFd                = 1,
    cudaExternalSemaphoreHandleTypeOpaqueWin32             = 2,
    cudaExternalSemaphoreHandleTypeOpaqueWin32Kmt          = 3,
    cudaExternalSemaphoreHandleTypeD3D12Fence              = 4,
    cudaExternalSemaphoreHandleTypeD3D11Fence              = 5,
    cudaExternalSemaphoreHandleTypeNvSciSync               = 6,
    cudaExternalSemaphoreHandleTypeKeyedMutex              = 7,
    cudaExternalSemaphoreHandleTypeKeyedMutexKmt           = 8,
    cudaExternalSemaphoreHandleTypeTimelineSemaphoreFd     = 9,
    cudaExternalSemaphoreHandleTypeTimelineSemaphoreWin32  = 10
} cudaExternalSemaphoreHandleType;
```

If *cudaExternalSemaphoreHandleDesc::type* is *cudaExternalSemaphoreHandleTypeOpaqueFd*, then ::cudaExternalSemaphoreHandleDesc::handle::fd must be a valid file descriptor referencing a synchronization object. Ownership of the file descriptor is transferred to the CUDA driver when the handle is imported successfully. Performing any operations on the file descriptor after it is imported results in undefined behavior.

If *cudaExternalSemaphoreHandleDesc::type* is *cudaExternalSemaphoreHandleTypeOpaqueWin32*, then exactly one of ::cudaExternalSemaphoreHandleDesc::handle::win32::handle and ::cudaExternalSemaphoreHandleDesc::handle::win32::name must not be NULL. If ::cudaExternalSemaphoreHandleDesc::handle::win32::handle is not NULL, then it must represent a valid shared NT handle that references a synchronization object. Ownership of this handle is not transferred to CUDA after the import operation, so the application must release the handle using the appropriate system call. If ::cudaExternalSemaphoreHandleDesc::handle::win32::name is not NULL, then it must name a valid synchronization object.

If *cudaExternalSemaphoreHandleDesc::type* is *cudaExternalSemaphoreHandleTypeOpaqueWin32Kmt*, then ::cudaExternalSemaphoreHandleDesc::handle::win32::handle must be non-NULL and ::cudaExternalSemaphoreHandleDesc::handle::win32::name must be NULL. The handle specified must be a globally shared KMT handle. This handle does not hold a reference to the underlying object, and thus will be invalid when all references to the synchronization object are destroyed.

If *cudaExternalSemaphoreHandleDesc::type* is *cudaExternalSemaphoreHandleTypeD3D12Fence*, then exactly one of ::cudaExternalSemaphoreHandleDesc::handle::win32::handle and ::cudaExternalSemaphoreHandleDesc::handle::win32::name must not be NULL. If ::cudaExternalSemaphoreHandleDesc::handle::win32::handle is not NULL, then it must represent a valid shared NT handle that is returned by ID3D12Device::CreateSharedHandle when referring to a ID3D12Fence object. This handle holds a reference to the underlying object. If ::cudaExternalSemaphoreHandleDesc::handle::win32::name is not NULL, then it must name a valid synchronization object that refers to a valid ID3D12Fence object.

If *cudaExternalSemaphoreHandleDesc::type* is *cudaExternalSemaphoreHandleTypeD3D11Fence*, then exactly one of ::cudaExternalSemaphoreHandleDesc::handle::win32::handle and ::cudaExternalSemaphoreHandleDesc::handle::win32::name must not be NULL. If ::cudaExternalSemaphore-

HandleDesc::handle::win32::handle is not NULL, then it must represent a valid shared NT handle that is returned by ID3D11Fence::CreateSharedHandle. If ::cudaExternalSemaphoreHandleDesc::handle::win32::name is not NULL, then it must name a valid synchronization object that refers to a valid ID3D11Fence object.

If *cudaExternalSemaphoreHandleDesc::type* is *cudaExternalSemaphoreHandleTypeNvSciSync*, then ::cudaExternalSemaphoreHandleDesc::handle::nvSciSyncObj represents a valid NvSciSyncObj.

cudaExternalSemaphoreHandleTypeKeyedMutex, then exactly one of ::cudaExternalSemaphoreHandleDesc::handle::win32::handle and ::cudaExternalSemaphoreHandleDesc::handle::win32::name must not be NULL. If ::cudaExternalSemaphoreHandleDesc::handle::win32::handle is not NULL, then it represent a valid shared NT handle that is returned by IDXGIResource1::CreateSharedHandle when referring to a IDXGIKeyedMutex object.

If *cudaExternalSemaphoreHandleDesc::type* is *cudaExternalSemaphoreHandleTypeKeyedMutexKmt*, then ::cudaExternalSemaphoreHandleDesc::handle::win32::handle must be non-NULL and ::cudaExternalSemaphoreHandleDesc::handle::win32::name must be NULL. The handle specified must represent a valid KMT handle that is returned by IDXGIResource::GetSharedHandle when referring to a IDXGIKeyedMutex object.

If *cudaExternalSemaphoreHandleDesc::type* is *cudaExternalSemaphoreHandleTypeTimelineSemaphoreFd*, then ::cudaExternalSemaphoreHandleDesc::handle::fd must be a valid file descriptor referencing a synchronization object. Ownership of the file descriptor is transferred to the CUDA driver when the handle is imported successfully. Performing any operations on the file descriptor after it is imported results in undefined behavior.

If *cudaExternalSemaphoreHandleDesc::type* is *cudaExternalSemaphoreHandleTypeTimelineSemaphoreWin32*, then exactly one of ::cudaExternalSemaphoreHandleDesc::handle::win32::handle and ::cudaExternalSemaphoreHandleDesc::handle::win32::name must not be NULL. If ::cudaExternalSemaphoreHandleDesc::handle::win32::handle is not NULL, then it must represent a valid shared NT handle that references a synchronization object. Ownership of this handle is not transferred to CUDA after the import operation, so the application must release the handle using the appropriate system call. If ::cudaExternalSemaphoreHandleDesc::handle::win32::name is not NULL, then it must name a valid synchronization object.

See also:

cudaDestroyExternalSemaphore, *cudaSignalExternalSemaphoresAsync*, *cudaWaitExternalSemaphoresAsync*

Note: Note that this function may also return error codes from previous, asynchronous launches.

Note: Note that this function may also return *cudaErrorInitializationError*, *cudaErrorInsufficientDriver* or *cudaErrorNoDevice* if this call tries to initialize internal CUDA RT state.

Note: Note that as specified by *cudaStreamAddCallback* no CUDA function may be called from callback. *cudaErrorNotPermitted* may, but is not guaranteed to, be returned as a diagnostic in such case.

Parameters

- **extSem_out** – - Returned handle to an external semaphore
- **semHandleDesc** – - Semaphore import handle descriptor

Returns

cudaSuccess, *cudaErrorInvalidResourceHandle*, *cudaErrorOperatingSystem*

__host__ *cudaError_t* **cudaSignalExternalSemaphoresAsync**(const *cudaExternalSemaphore_t* *extSemArray, const struct *cudaExternalSemaphoreSignalParams* *paramsArray, unsigned int numExtSems, *cudaStream_t* stream = 0)

Signals a set of external semaphore objects.

Enqueues a signal operation on a set of externally allocated semaphore object in the specified stream. The operations will be executed when all prior operations in the stream complete.

The exact semantics of signaling a semaphore depends on the type of the object.

If the semaphore object is any one of the following types: *cudaExternalSemaphoreHandleTypeOpaqueFd*, *cudaExternalSemaphoreHandleTypeOpaqueWin32*, *cudaExternalSemaphoreHandleTypeOpaqueWin32Kmt* then signaling the semaphore will set it to the signaled state.

If the semaphore object is any one of the following types: *cudaExternalSemaphoreHandleTypeD3D12Fence*, *cudaExternalSemaphoreHandleTypeD3D11Fence*, *cudaExternalSemaphoreHandleTypeTimelineSemaphoreFd*, *cudaExternalSemaphoreHandleTypeTimelineSemaphoreWin32* then the semaphore will be set to the value specified in ::cudaExternalSemaphoreSignalParams::params::fence::value.

If the semaphore object is of the type *cudaExternalSemaphoreHandleTypeNvSciSync* this API sets ::cudaExternalSemaphoreSignalParams::params::nvSciSync::fence to a value that can be used by subsequent waiters of the same NvSciSync object to order operations with those currently submitted in `stream`. Such an update will overwrite previous contents of ::cudaExternalSemaphoreSignalParams::params::nvSciSync::fence. By default, signaling such an external semaphore object causes appropriate memory synchronization operations to be performed over all the external memory objects that are imported as *cudaExternalMemoryHandleTypeNvSciBuf*. This ensures that any subsequent accesses made by other importers of the same set of NvSciBuf memory object(s) are coherent. These operations can be skipped by specifying the flag *cudaExternalSemaphoreSignalSkipNvSciBufMemSync*, which can be used as a performance optimization when data coherency is not required. But specifying this flag in scenarios where data coherency is required results in undefined behavior. Also, for semaphore object of the type *cudaExternalSemaphoreHandleTypeNvSciSync*, if the NvSciSyncAttrList used to create the NvSciSyncObj had not set the flags in *cudaDeviceGetNvSciSyncAttributes* to cudaNvSciSyncAttrSignal, this API will return cudaErrorNotSupported.

::cudaExternalSemaphoreSignalParams::params::nvSciSync::fence associated with semaphore object of the type *cudaExternalSemaphoreHandleTypeNvSciSync* can be deterministic. For this the NvSciSyncAttrList used to create the semaphore object must have value of NvSciSyncAttrKey_RequireDeterministicFences key set to true. Deterministic fences allow users to enqueue a wait over the semaphore object even before corresponding signal is enqueued. For such a semaphore object, CUDA guarantees that each signal operation will increment the fence value by '1'. Users are expected to track count of signals enqueued on the semaphore object and insert waits accordingly. When such a semaphore object is signaled from multiple streams, due to concurrent stream execution, it is possible that the order in which the semaphore gets signaled

is indeterministic. This could lead to waiters of the semaphore getting unblocked incorrectly. Users are expected to handle such situations, either by not using the same semaphore object with deterministic fence support enabled in different streams or by adding explicit dependency amongst such streams so that the semaphore is signaled in order. ::cudaExternalSemaphoreSignalParams::params::nvSciSync::fence associated with semaphore object of the type *cudaExternalSemaphoreHandleTypeNvSciSync* can be timestamp enabled. For this the NvSciSyncAttrList used to create the object must have the value of NvSciSyncAttrKey_WaiterRequireTimestamps key set to true. Timestamps are emitted asynchronously by the GPU and CUDA saves the GPU timestamp in the corresponding NvSciSyncFence at the time of signal on GPU. Users are expected to convert GPU clocks to CPU clocks using appropriate scaling functions. Users are expected to wait for the completion of the fence before extracting timestamp using appropriate NvSciSync APIs. Users are expected to ensure that there is only one outstanding timestamp enabled fence per Cuda-NvSciSync object at any point of time, failing which leads to undefined behavior. Extracting the timestamp before the corresponding fence is signalled could lead to undefined behaviour. Timestamp extracted via appropriate NvSciSync API would be in microseconds.

If the semaphore object is any one of the following types: *cudaExternalSemaphoreHandleTypeKeyedMutex*, *cudaExternalSemaphoreHandleTypeKeyedMutexKmt*, then the keyed mutex will be released with the key specified in ::cudaExternalSemaphoreSignalParams::params::keyedmutex::key.

See also:

cudaImportExternalSemaphore, *cudaDestroyExternalSemaphore*, *cudaWaitExternalSemaphoresAsync*

Note: Note that this function may also return error codes from previous, asynchronous launches.

Note: Note that this function may also return *cudaErrorInitializationError*, *cudaErrorInsufficientDriver* or *cudaErrorNoDevice* if this call tries to initialize internal CUDA RT state.

Note: Note that as specified by *cudaStreamAddCallback* no CUDA function may be called from callback. *cudaErrorNotPermitted* may, but is not guaranteed to, be returned as a diagnostic in such case.

Parameters

- **`extSemArray`** – - Set of external semaphores to be signaled
- **`paramsArray`** – - Array of semaphore parameters
- **`numExtSems`** – - Number of semaphores to signal
- **`stream`** – - Stream to enqueue the signal operations in

Returns

cudaSuccess, *cudaErrorInvalidResourceHandle*

__host__ *cudaError_t* **cudaWaitExternalSemaphoresAsync**(const *cudaExternalSemaphore_t* *extSemArray, const struct *cudaExternalSemaphoreWaitParams* *paramsArray, unsigned int numExtSems, *cudaStream_t* stream = 0)

Waits on a set of external semaphore objects.

Enqueues a wait operation on a set of externally allocated semaphore object in the specified stream. The operations will be executed when all prior operations in the stream complete.

The exact semantics of waiting on a semaphore depends on the type of the object.

If the semaphore object is any one of the following types: *cudaExternalSemaphoreHandleTypeOpaqueFd*, *cudaExternalSemaphoreHandleTypeOpaqueWin32*, *cudaExternalSemaphoreHandleTypeOpaqueWin32Kmt* then waiting on the semaphore will wait until the semaphore reaches the signaled state. The semaphore will then be reset to the unsignaled state. Therefore for every signal operation, there can only be one wait operation.

If the semaphore object is any one of the following types: *cudaExternalSemaphoreHandleTypeD3D12Fence*, *cudaExternalSemaphoreHandleTypeD3D11Fence*, *cudaExternalSemaphoreHandleTypeTimelineSemaphoreFd*, *cudaExternalSemaphoreHandleTypeTimelineSemaphoreWin32* then waiting on the semaphore will wait until the value of the semaphore is greater than or equal to ::cudaExternalSemaphoreWaitParams::params::fence::value.

If the semaphore object is of the type *cudaExternalSemaphoreHandleTypeNvSciSync* then, waiting on the semaphore will wait until the ::cudaExternalSemaphoreSignalParams::params::nvSciSync::fence is signaled by the signaler of the NvSciSyncObj that was associated with this semaphore object. By default, waiting on such an external semaphore object causes appropriate memory synchronization operations to be performed over all external memory objects that are imported as *cudaExternalMemoryHandleTypeNvSciBuf*. This ensures that any subsequent accesses made by other importers of the same set of NvSciBuf memory object(s) are coherent. These operations can be skipped by specifying the flag *cudaExternalSemaphoreWaitSkipNvSciBufMemSync*, which can be used as a performance optimization when data coherency is not required. But specifying this flag in scenarios where data coherency is required results in undefined behavior. Also, for semaphore object of the type *cudaExternalSemaphoreHandleTypeNvSciSync*, if the NvSciSyncAttrList used to create the NvSciSyncObj had not set the flags in *cudaDeviceGetNvSciSyncAttributes* to cudaNvSciSyncAttrWait, this API will return cudaErrorNotSupported.

If the semaphore object is any one of the following types: *cudaExternalSemaphoreHandleTypeKeyedMutex*, *cudaExternalSemaphoreHandleTypeKeyedMutexKmt*, then the keyed mutex will be acquired when it is released with the key specified in ::cudaExternalSemaphoreSignalParams::params::keyedmutex::key or until the timeout specified by ::cudaExternalSemaphoreSignalParams::params::keyedmutex::timeoutMs has lapsed. The timeout interval can either be a finite value specified in milliseconds or an infinite value. In case an infinite value is specified the timeout never elapses. The windows INFINITE macro must be used to specify infinite timeout

See also:

cudaImportExternalSemaphore, *cudaDestroyExternalSemaphore*, *cudaSignalExternalSemaphoresAsync*

Note: Note that this function may also return error codes from previous, asynchronous launches.

Note: Note that this function may also return *cudaErrorInitializationError*, *cudaErrorInsufficientDriver* or *cudaErrorNoDevice* if this call tries to initialize internal CUDA RT state.

Note: Note that as specified by *cudaStreamAddCallback* no CUDA function may be called from callback. *cudaErrorNotPermitted* may, but is not guaranteed to, be returned as a diagnostic in such case.

Parameters

- **extSemArray** – - External semaphores to be waited on
- **paramsArray** – - Array of semaphore parameters
- **numExtSems** – - Number of semaphores to wait on
- **stream** – - Stream to enqueue the wait operations in

Returns

cudaSuccess, *cudaErrorInvalidResourceHandle cudaErrorTimeout*

6.20. Graph Management

This section describes the graph management functions of CUDA runtime application programming interface.

Functions

__host__ cudaError_t *cudaDeviceGetGraphMemAttribute*(int device, enum cudaGraphMemAttributeType attr, void *value)
Query asynchronous allocation attributes related to graphs.

__host__ cudaError_t *cudaDeviceGraphMemTrim*(int device)
Free unused memory that was cached on the specified device for use with graphs back to the OS.

__host__ cudaError_t *cudaDeviceSetGraphMemAttribute*(int device, enum cudaGraphMemAttributeType attr, void *value)
Set asynchronous allocation attributes related to graphs.

__device__ cudaGraphExec_t *cudaGetCurrentGraphExec*(void)
Get the currently running device graph id.

__host__ cudaError_t *cudaGraphAddChildGraphNode*(cudaGraphNode_t *pGraphNode, cudaGraph_t graph, const cudaGraphNode_t *pDependencies, size_t numDependencies, cudaGraph_t childGraph)
Creates a child graph node and adds it to a graph.

__host__ cudaError_t *cudaGraphAddDependencies*(cudaGraph_t graph, const cudaGraphNode_t *from, const cudaGraphNode_t *to, const cudaGraphEdgeData *edgeData, size_t numDependencies)
Adds dependency edges to a graph.

__host__ cudaError_t *cudaGraphAddEmptyNode*(cudaGraphNode_t *pGraphNode, cudaGraph_t graph, const cudaGraphNode_t *pDependencies, size_t numDependencies)
Creates an empty node and adds it to a graph.

__host__ cudaError_t *cudaGraphAddEventRecordNode*(cudaGraphNode_t *pGraphNode, cudaGraph_t graph, const cudaGraphNode_t *pDependencies, size_t numDependencies, cudaEvent_t event)
Creates an event record node and adds it to a graph.

__host__ cudaError_t *cudaGraphAddEventWaitNode*(cudaGraphNode_t *pGraphNode, cudaGraph_t graph, const cudaGraphNode_t *pDependencies, size_t numDependencies, cudaEvent_t event)
Creates an event wait node and adds it to a graph.

__host__ cudaError_t *cudaGraphAddExternalSemaphoresSignalNode*(cudaGraphNode_t *pGraphNode, cudaGraph_t graph, const cudaGraphNode_t *pDependencies, size_t numDependencies, const struct cudaExternalSemaphoreSignalNodeParams *nodeParams)
Creates an external semaphore signal node and adds it to a graph.

__host__ cudaError_t *cudaGraphAddExternalSemaphoresWaitNode*(cudaGraphNode_t *pGraphNode, cudaGraph_t graph, const cudaGraphNode_t *pDependencies, size_t numDependencies, const struct cudaExternalSemaphoreWaitNodeParams *nodeParams)
Creates an external semaphore wait node and adds it to a graph.

__host__ cudaError_t *cudaGraphAddHostNode*(cudaGraphNode_t *pGraphNode, cudaGraph_t graph, const cudaGraphNode_t *pDependencies, size_t numDependencies, const struct cudaHostNodeParams *pNodeParams)
Creates a host execution node and adds it to a graph.

__host__ cudaError_t *cudaGraphAddKernelNode*(cudaGraphNode_t *pGraphNode, cudaGraph_t graph, const cudaGraphNode_t *pDependencies, size_t numDependencies, const struct cudaKernelNodeParams *pNodeParams)
Creates a kernel execution node and adds it to a graph.

__host__ cudaError_t *cudaGraphAddMemAllocNode*(cudaGraphNode_t *pGraphNode, cudaGraph_t graph, const cudaGraphNode_t *pDependencies, size_t numDependencies, struct cudaMemAllocNodeParams *nodeParams)
Creates an allocation node and adds it to a graph.

__host__ cudaError_t *cudaGraphAddMemFreeNode*(cudaGraphNode_t *pGraphNode, cudaGraph_t graph, const cudaGraphNode_t *pDependencies, size_t numDependencies, void *dptr)
Creates a memory free node and adds it to a graph.

__host__ cudaError_t *cudaGraphAddMemcpyNode*(cudaGraphNode_t *pGraphNode, cudaGraph_t graph, const cudaGraphNode_t *pDependencies, size_t numDependencies, const struct cudaMemcpy3DParms *pCopyParams)
Creates a memcpy node and adds it to a graph.

__host__ cudaError_t *cudaGraphAddMemcpyNode1D*(cudaGraphNode_t *pGraphNode, cudaGraph_t graph, const cudaGraphNode_t *pDependencies, size_t numDependencies, void *dst, const void *src, size_t count, enum cudaMemcpyKind kind)
Creates a 1D memcpy node and adds it to a graph.

__host__ cudaError_t *cudaGraphAddMemcpyNodeFromSymbol*(cudaGraphNode_t *pGraphNode, cudaGraph_t graph, const cudaGraphNode_t *pDependencies, size_t numDependencies, void *dst, const void *symbol, size_t count, size_t offset, enum cudaMemcpyKind kind)
Creates a memcpy node to copy from a symbol on the device and adds it to a graph.

__host__ cudaError_t *cudaGraphAddMemcpyNodeToSymbol*(cudaGraphNode_t *pGraphNode, cudaGraph_t graph, const cudaGraphNode_t *pDependencies, size_t numDependencies, const void

***symbol, const void *src, size_t count, size_t offset, enum cudaMemcpyKind kind)**
Creates a memcpy node to copy to a symbol on the device and adds it to a graph.

__host__ cudaError_t *cudaGraphAddMemsetNode*(cudaGraphNode_t *pGraphNode, cudaGraph_t graph, const cudaGraphNode_t *pDependencies, size_t numDependencies, const struct cudaMemsetParams *pMemsetParams)
Creates a memset node and adds it to a graph.

__host__ cudaError_t *cudaGraphAddNode*(cudaGraphNode_t *pGraphNode, cudaGraph_t graph, const cudaGraphNode_t *pDependencies, const cudaGraphEdgeData *dependencyData, size_t numDependencies, struct cudaGraphNodeParams *nodeParams)
Adds a node of arbitrary type to a graph.

__host__ cudaError_t *cudaGraphChildGraphNodeGetGraph*(cudaGraphNode_t node, cudaGraph_t *pGraph)
Gets a handle to the embedded graph of a child graph node.

__host__ cudaError_t *cudaGraphClone*(cudaGraph_t *pGraphClone, cudaGraph_t originalGraph)
Clones a graph.

__host__ cudaError_t *cudaGraphConditionalHandleCreate*(cudaGraphConditionalHandle *pHandle_out, cudaGraph_t graph, unsigned int defaultLaunchValue=0, unsigned int flags=0)
Create a conditional handle.

__host__ cudaError_t *cudaGraphConditionalHandleCreate_v2*(cudaGraphConditionalHandle *pHandle_out, cudaGraph_t graph, cudaExecutionContext_t ctx=NULL, unsigned int defaultLaunchValue=0, unsigned int flags=0)
Create a conditional handle.

__host__ cudaError_t *cudaGraphCreate*(cudaGraph_t *pGraph, unsigned int flags=0)
Creates a graph.

__host__ cudaError_t *cudaGraphDebugDotPrint*(cudaGraph_t graph, const char *path, unsigned int flags)
Write a DOT file describing graph structure.

__host__ cudaError_t *cudaGraphDestroy*(cudaGraph_t graph)
Destroys a graph.

__host__ cudaError_t *cudaGraphDestroyNode*(cudaGraphNode_t node)
Remove a node from the graph.

__host__ cudaError_t *cudaGraphEventRecordNodeGetEvent*(cudaGraphNode_t node, cudaEvent_t *event_out)
Returns the event associated with an event record node.

__host__ cudaError_t *cudaGraphEventRecordNodeSetEvent*(cudaGraphNode_t node, cudaEvent_t event)
Sets an event record node's event.

__host__ cudaError_t *cudaGraphEventWaitNodeGetEvent*(cudaGraphNode_t node, cudaEvent_t *event_out)
Returns the event associated with an event wait node.

__host__ cudaError_t *cudaGraphEventWaitNodeSetEvent*(cudaGraphNode_t node, cudaEvent_t event)
Sets an event wait node's event.

__host__ cudaError_t *cudaGraphExecChildGraphNodeSetParams*(cudaGraphExec_t hGraphExec, cudaGraphNode_t node, cudaGraph_t childGraph)
Updates node parameters in the child graph node in the given graphExec.

__host__ cudaError_t *cudaGraphExecDestroy*(cudaGraphExec_t graphExec)
Destroys an executable graph.

__host__ cudaError_t *cudaGraphExecEventRecordNodeSetEvent*(cudaGraphExec_t hGraphExec, cudaGraphNode_t hNode, cudaEvent_t event)
Sets the event for an event record node in the given graphExec.

__host__ cudaError_t *cudaGraphExecEventWaitNodeSetEvent*(cudaGraphExec_t hGraphExec, cudaGraphNode_t hNode, cudaEvent_t event)
Sets the event for an event wait node in the given graphExec.

__host__ cudaError_t *cudaGraphExecExternalSemaphoresSignalNodeSetParams*(cudaGraphExec_t hGraphExec, cudaGraphNode_t hNode, const struct cudaExternalSemaphoreSignalNodeParams *nodeParams)
Sets the parameters for an external semaphore signal node in the given graphExec.

__host__ cudaError_t *cudaGraphExecExternalSemaphoresWaitNodeSetParams*(cudaGraphExec_t hGraphExec, cudaGraphNode_t hNode, const struct cudaExternalSemaphoreWaitNodeParams *nodeParams)
Sets the parameters for an external semaphore wait node in the given graphExec.

__host__ cudaError_t *cudaGraphExecGetFlags*(cudaGraphExec_t graphExec, unsigned long long *flags)
Query the instantiation flags of an executable graph.

__host__ cudaError_t *cudaGraphExecGetId*(cudaGraphExec_t hGraphExec, unsigned int *graphID)
Returns the id of a given graph exec.

__host__ cudaError_t *cudaGraphExecHostNodeSetParams*(cudaGraphExec_t hGraphExec, cudaGraphNode_t node, const struct cudaHostNodeParams *pNodeParams)
Sets the parameters for a host node in the given graphExec.

__host__ cudaError_t *cudaGraphExecKernelNodeSetParams*(cudaGraphExec_t hGraphExec, cudaGraphNode_t node, const struct cudaKernelNodeParams *pNodeParams)
Sets the parameters for a kernel node in the given graphExec.

__host__ cudaError_t *cudaGraphExecMemcpyNodeSetParams*(cudaGraphExec_t hGraphExec, cudaGraphNode_t node, const struct cudaMemcpy3DParms *pNodeParams)
Sets the parameters for a memcpy node in the given graphExec.

__host__ cudaError_t *cudaGraphExecMemcpyNodeSetParams1D*(cudaGraphExec_t hGraphExec, cudaGraphNode_t node, void *dst, const void *src, size_t count, enum cudaMemcpyKind kind)
Sets the parameters for a memcpy node in the given graphExec to perform a 1-dimensional copy.

__host__ cudaError_t *cudaGraphExecMemcpyNodeSetParamsFromSymbol*(cudaGraphExec_t hGraphExec, cudaGraphNode_t node, void *dst, const void *symbol, size_t count, size_t offset, enum cudaMemcpyKind kind)
Sets the parameters for a memcpy node in the given graphExec to copy from a symbol on the device.

__host__ cudaError_t *cudaGraphExecMemcpyNodeSetParamsToSymbol*(cudaGraphExec_t hGraphExec, cudaGraphNode_t node, const void *symbol, const void *src, size_t count, size_t offset, enum cudaMemcpyKind kind)
Sets the parameters for a memcpy node in the given graphExec to copy to a symbol on the device.

__host__ cudaError_t *cudaGraphExecMemsetNodeSetParams*(cudaGraphExec_t hGraphExec, cudaGraphNode_t node, const struct cudaMemsetParams *pNodeParams)
Sets the parameters for a memset node in the given graphExec.

__host__ cudaError_t *cudaGraphExecNodeSetParams*(cudaGraphExec_t graphExec, cudaGraphNode_t node, struct cudaGraphNodeParams *nodeParams)
Update a graph node's parameters in an instantiated graph.

__host__ cudaError_t *cudaGraphExecUpdate*(cudaGraphExec_t hGraphExec, cudaGraph_t hGraph, cudaGraphExecUpdateResultInfo *resultInfo)
Check whether an executable graph can be updated with a graph and perform the update if possible.

__host__ cudaError_t *cudaGraphExternalSemaphoresSignalNodeGetParams*(cudaGraphNode_t hNode, struct cudaExternalSemaphoreSignalNodeParams *params_out)
Returns an external semaphore signal node's parameters.

__host__ cudaError_t *cudaGraphExternalSemaphoresSignalNodeSetParams*(cudaGraphNode_t hNode, const struct cudaExternalSemaphoreSignalNodeParams *nodeParams)
Sets an external semaphore signal node's parameters.

__host__ cudaError_t *cudaGraphExternalSemaphoresWaitNodeGetParams*(cudaGraphNode_t hNode, struct cudaExternalSemaphoreWaitNodeParams *params_out)
Returns an external semaphore wait node's parameters.

__host__ cudaError_t *cudaGraphExternalSemaphoresWaitNodeSetParams*(cudaGraphNode_t hNode, const struct cudaExternalSemaphoreWaitNodeParams *nodeParams)
Sets an external semaphore wait node's parameters.

__host__ cudaError_t *cudaGraphGetEdges*(cudaGraph_t graph, cudaGraphNode_t *from, cudaGraphNode_t *to, cudaGraphEdgeData *edgeData, size_t *numEdges)
Returns a graph's dependency edges.

__host__ cudaError_t *cudaGraphGetId*(cudaGraph_t hGraph, unsigned int *graphID)
Returns the id of a given graph.

__host__ cudaError_t *cudaGraphGetNodes*(cudaGraph_t graph, cudaGraphNode_t *nodes, size_t *numNodes)
Returns a graph's nodes.

__host__ cudaError_t *cudaGraphGetRootNodes*(cudaGraph_t graph, cudaGraphNode_t *pRootNodes, size_t *pNumRootNodes)
Returns a graph's root nodes.

__host__ cudaError_t *cudaGraphHostNodeGetParams*(cudaGraphNode_t node, struct cudaHostNodeParams *pNodeParams)
Returns a host node's parameters.

__host__ cudaError_t *cudaGraphHostNodeSetParams*(cudaGraphNode_t node, const struct cudaHostNodeParams *pNodeParams)
Sets a host node's parameters.

__host__ cudaError_t *cudaGraphInstantiate*(cudaGraphExec_t *pGraphExec, cudaGraph_t graph, unsigned long long flags=0)
Creates an executable graph from a graph.

__host__ cudaError_t *cudaGraphInstantiateWithFlags*(cudaGraphExec_t *pGraphExec, cudaGraph_t graph, unsigned long long flags=0)
Creates an executable graph from a graph.

__host__ cudaError_t *cudaGraphInstantiateWithParams*(cudaGraphExec_t *pGraphExec, cudaGraph_t graph, cudaGraphInstantiateParams *instantiateParams)
Creates an executable graph from a graph.

__host__ cudaError_t *cudaGraphKernelNodeCopyAttributes*(cudaGraphNode_t hDst, cudaGraphNode_t hSrc)
Copies attributes from source node to destination node.

__host__ cudaError_t *cudaGraphKernelNodeGetAttribute*(cudaGraphNode_t hNode, cudaLaunchAttributeID attr, cudaLaunchAttributeValue *value_out)
Queries node attribute.

__host__ cudaError_t *cudaGraphKernelNodeGetParams*(cudaGraphNode_t node, struct cudaKernelNodeParams *pNodeParams)
Returns a kernel node's parameters.

__host__ cudaError_t *cudaGraphKernelNodeSetAttribute*(cudaGraphNode_t hNode, cudaLaunchAttributeID attr, const cudaLaunchAttributeValue *value)
Sets node attribute.

__device__ cudaError_t *cudaGraphKernelNodeSetEnabled*(cudaGraphDeviceNode_t node, bool enable)
Enables or disables the given kernel node.

__device__ cudaError_t *cudaGraphKernelNodeSetGridDim*(cudaGraphDeviceNode_t node, dim3 gridDim)
Updates the grid dimensions of the given kernel node.

__device__ cudaError_t *cudaGraphKernelNodeSetParam*(cudaGraphDeviceNode_t node, size_t offset, const T &value)
Updates the kernel parameters of the given kernel node.

__device__ cudaError_t *cudaGraphKernelNodeSetParam*(cudaGraphDeviceNode_t node, size_t offset, const void *value, size_t size)
Updates the kernel parameters of the given kernel node.

__host__ cudaError_t *cudaGraphKernelNodeSetParams*(cudaGraphNode_t node, const struct cudaKernelNodeParams *pNodeParams)
Sets a kernel node's parameters.

__device__ cudaError_t *cudaGraphKernelNodeUpdatesApply*(const cudaGraphKernelNodeUpdate *updates, size_t updateCount)
Batch applies multiple kernel node updates.

__host__ cudaError_t *cudaGraphLaunch*(cudaGraphExec_t graphExec, cudaStream_t stream)
Launches an executable graph in a stream.

__host__ cudaError_t *cudaGraphMemAllocNodeGetParams*(cudaGraphNode_t node, struct cudaMemAllocNodeParams *params_out)
Returns a memory alloc node's parameters.

__host__ cudaError_t *cudaGraphMemFreeNodeGetParams*(cudaGraphNode_t node, void *dptr_out)
Returns a memory free node's parameters.

__host__ cudaError_t *cudaGraphMemcpyNodeGetParams*(cudaGraphNode_t node, struct cudaMemcpy3DParms *pNodeParams)
Returns a memcpy node's parameters.

__host__ cudaError_t *cudaGraphMemcpyNodeSetParams*(cudaGraphNode_t node, const struct cudaMemcpy3DParms *pNodeParams)
Sets a memcpy node's parameters.

__host__ cudaError_t *cudaGraphMemcpyNodeSetParams1D*(cudaGraphNode_t node, void *dst, const void *src, size_t count, enum cudaMemcpyKind kind)
Sets a memcpy node's parameters to perform a 1-dimensional copy.

__host__ cudaError_t *cudaGraphMemcpyNodeSetParamsFromSymbol*(cudaGraphNode_t node, void *dst, const void *symbol, size_t count, size_t offset, enum cudaMemcpyKind kind)
Sets a memcpy node's parameters to copy from a symbol on the device.

__host__ cudaError_t *cudaGraphMemcpyNodeSetParamsToSymbol*(cudaGraphNode_t node, const void *symbol, const void *src, size_t count, size_t offset, enum cudaMemcpyKind kind)
Sets a memcpy node's parameters to copy to a symbol on the device.

__host__ cudaError_t *cudaGraphMemsetNodeGetParams*(cudaGraphNode_t node, struct cudaMemsetParams *pNodeParams)
Returns a memset node's parameters.

__host__ cudaError_t *cudaGraphMemsetNodeSetParams*(cudaGraphNode_t node, const struct cudaMemsetParams *pNodeParams)
Sets a memset node's parameters.

__host__ cudaError_t *cudaGraphNodeFindInClone*(cudaGraphNode_t *pNode, cudaGraphNode_t originalNode, cudaGraph_t clonedGraph)
Finds a cloned version of a node.

__host__ cudaError_t *cudaGraphNodeGetContainingGraph*(cudaGraphNode_t hNode, cudaGraph_t *phGraph)
Returns the graph that contains a given graph node.

__host__ cudaError_t *cudaGraphNodeGetDependencies*(cudaGraphNode_t node, cudaGraphNode_t *pDependencies, cudaGraphEdgeData *edgeData, size_t *pNumDependencies)
Returns a node's dependencies.

__host__ cudaError_t *cudaGraphNodeGetDependentNodes*(cudaGraphNode_t node, cudaGraphNode_t *pDependentNodes, cudaGraphEdgeData *edgeData, size_t *pNumDependentNodes)
Returns a node's dependent nodes.

__host__ cudaError_t *cudaGraphNodeGetEnabled*(cudaGraphExec_t hGraphExec, cudaGraphNode_t hNode, unsigned int *isEnabled)
Query whether a node in the given graphExec is enabled.

__host__ cudaError_t *cudaGraphNodeGetLocalId*(cudaGraphNode_t hNode, unsigned int *nodeId)
Returns the node id of a given graph node.

__host__ cudaError_t *cudaGraphNodeGetParams*(cudaGraphNode_t node, struct cudaGraphNodeParams *nodeParams)
Returns a graph node's parameters.

__host__ cudaError_t *cudaGraphNodeGetToolsId*(cudaGraphNode_t hNode, unsigned long long *toolsNodeId)
Returns an id used by tools to identify a given node.

__host__ cudaError_t *cudaGraphNodeGetType*(cudaGraphNode_t node, enum cudaGraphNodeType *pType)
Returns a node's type.

__host__ cudaError_t *cudaGraphNodeSetEnabled*(cudaGraphExec_t hGraphExec, cudaGraphNode_t hNode, unsigned int isEnabled)
Enables or disables the specified node in the given graphExec.

__host__ cudaError_t *cudaGraphNodeSetParams*(cudaGraphNode_t node, struct cudaGraphNodeParams *nodeParams)
Update a graph node's parameters.

__host__ cudaError_t *cudaGraphReleaseUserObject*(cudaGraph_t graph, cudaUserObject_t object, unsigned int count=1)

Release a user object reference from a graph.

__host__ cudaError_t *cudaGraphRemoveDependencies*(cudaGraph_t graph, const cudaGraphNode_t *from, const cudaGraphNode_t *to, const cudaGraphEdgeData *edgeData, size_t numDependencies)
Removes dependency edges from a graph.

__host__ cudaError_t *cudaGraphRetainUserObject*(cudaGraph_t graph, cudaUserObject_t object, unsigned int count=1, unsigned int flags=0)
Retain a reference to a user object from a graph.

__device__ void *cudaGraphSetConditional*(cudaGraphConditionalHandle handle, unsigned int value)
Sets the condition value associated with a conditional node.

__host__ cudaError_t *cudaGraphUpload*(cudaGraphExec_t graphExec, cudaStream_t stream)
Uploads an executable graph in a stream.

__host__ cudaError_t *cudaUserObjectCreate*(cudaUserObject_t *object_out, void *ptr, cudaHostFn_t destroy, unsigned int initialRefcount, unsigned int flags)
Create a user object.

__host__ cudaError_t *cudaUserObjectRelease*(cudaUserObject_t object, unsigned int count=1)
Release a reference to a user object.

__host__ cudaError_t *cudaUserObjectRetain*(cudaUserObject_t object, unsigned int count=1)
Retain a reference to a user object.

6.20.1. Functions

__host__ *cudaError_t* **cudaDeviceGetGraphMemAttribute**(int device, enum *cudaGraphMemAttributeType* attr, void *value)

Query asynchronous allocation attributes related to graphs.

Valid attributes are:

- *cudaGraphMemAttrUsedMemCurrent*: Amount of memory, in bytes, currently associated with graphs
- *cudaGraphMemAttrUsedMemHigh*: High watermark of memory, in bytes, associated with graphs since the last time it was reset. High watermark can only be reset to zero.
- *cudaGraphMemAttrReservedMemCurrent*: Amount of memory, in bytes, currently allocated for use by the CUDA graphs asynchronous allocator.
- *cudaGraphMemAttrReservedMemHigh*: High watermark of memory, in bytes, currently allocated for use by the CUDA graphs asynchronous allocator.

See also:

cudaDeviceSetGraphMemAttribute, *cudaGraphAddMemAllocNode*, *cudaGraphAddMemFreeNode*, *cudaDeviceGraphMemTrim*, *cudaMallocAsync*, *cudaFreeAsync*

Note: Graph objects are not threadsafe. More here.

Note: Note that this function may also return error codes from previous, asynchronous launches.

Note: Note that this function may also return *cudaErrorInitializationError*, *cudaErrorInsufficientDriver* or *cudaErrorNoDevice* if this call tries to initialize internal CUDA RT state.

Note: Note that as specified by *cudaStreamAddCallback* no CUDA function may be called from callback. *cudaErrorNotPermitted* may, but is not guaranteed to, be returned as a diagnostic in such case.

Parameters

- **device** – - Specifies the scope of the query
- **attr** – - attribute to get
- **value** – - retrieved value

Returns

cudaSuccess, *cudaErrorInvalidDevice*

__host__ *cudaError_t* **cudaDeviceGraphMemTrim**(int device)

Free unused memory that was cached on the specified device for use with graphs back to the OS.

Blocks which are not in use by a graph that is either currently executing or scheduled to execute are freed back to the operating system.

See also:

cudaGraphAddMemAllocNode, *cudaGraphAddMemFreeNode*, *cudaDeviceGetGraphMemAttribute*, *cudaDeviceSetGraphMemAttribute*, *cudaMallocAsync*, *cudaFreeAsync*

Note: Graph objects are not threadsafe. More here.

Note: Note that this function may also return error codes from previous, asynchronous launches.

Note: Note that this function may also return *cudaErrorInitializationError*, *cudaErrorInsufficientDriver* or *cudaErrorNoDevice* if this call tries to initialize internal CUDA RT state.

Note: Note that as specified by *cudaStreamAddCallback* no CUDA function may be called from callback. *cudaErrorNotPermitted* may, but is not guaranteed to, be returned as a diagnostic in such case.

Parameters
device – - The device for which cached memory should be freed.

Returns
cudaSuccess, *cudaErrorInvalidValue*

__host__ *cudaError_t* **cudaDeviceSetGraphMemAttribute**(int device, enum *cudaGraphMemAttributeType* attr, void *value)

Set asynchronous allocation attributes related to graphs.

Valid attributes are:

- *cudaGraphMemAttrUsedMemHigh*: High watermark of memory, in bytes, associated with graphs since the last time it was reset. High watermark can only be reset to zero.
- *cudaGraphMemAttrReservedMemHigh*: High watermark of memory, in bytes, currently allocated for use by the CUDA graphs asynchronous allocator.

See also:

cudaDeviceGetGraphMemAttribute, *cudaGraphAddMemAllocNode*, *cudaGraphAddMemFreeNode*, *cudaDeviceGraphMemTrim*, *cudaMallocAsync*, *cudaFreeAsync*

Note: Graph objects are not threadsafe. More here.

Note: Note that this function may also return error codes from previous, asynchronous launches.

Note: Note that this function may also return *cudaErrorInitializationError*, *cudaErrorInsufficientDriver* or *cudaErrorNoDevice* if this call tries to initialize internal CUDA RT state.

Note: Note that as specified by *cudaStreamAddCallback* no CUDA function may be called from callback. *cudaErrorNotPermitted* may, but is not guaranteed to, be returned as a diagnostic in such case.

Parameters

- **device** – - Specifies the scope of the query
- **attr** – - attribute to get
- **value** – - pointer to value to set

Returns
cudaSuccess, *cudaErrorInvalidDevice*

__device__ inline *cudaGraphExec_t* **cudaGetCurrentGraphExec**(void)

Get the currently running device graph id.

Get the currently running device graph id.

See also:

cudaGraphLaunch

Returns

Returns the current device graph id, 0 if the call is outside of a device graph.

__host__ *cudaError_t* **cudaGraphAddChildGraphNode**(*cudaGraphNode_t* *pGraphNode, *cudaGraph_t* graph, const *cudaGraphNode_t* *pDependencies, size_t numDependencies, *cudaGraph_t* childGraph)

Creates a child graph node and adds it to a graph.

Creates a new node which executes an embedded graph, and adds it to `graph` with `numDependencies` dependencies specified via `pDependencies`. It is possible for `numDependencies` to be 0, in which case the node will be placed at the root of the graph. `pDependencies` may not have any duplicate entries. A handle to the new node will be returned in `pGraphNode`.

If `childGraph` contains allocation nodes, free nodes, or conditional nodes, this call will return an error.

The node executes an embedded child graph. The child graph is cloned in this call.

See also:

cudaGraphAddNode, *cudaGraphChildGraphNodeGetGraph*, *cudaGraphCreate*, *cudaGraphDestroyNode*, *cudaGraphAddEmptyNode*, *cudaGraphAddKernelNode*, *cudaGraphAddHostNode*, *cudaGraphAddMemcpyNode*, *cudaGraphAddMemsetNode*, *cudaGraphClone*

Warning: This function will be deprecated in a future CUDA release. Please migrate to *cudaGraphAddNode*.

Note: Graph objects are not threadsafe. More here.

Note: Note that this function may also return error codes from previous, asynchronous launches.

Note: Note that this function may also return *cudaErrorInitializationError*, *cudaErrorInsufficientDriver* or *cudaErrorNoDevice* if this call tries to initialize internal CUDA RT state.

Note: Note that as specified by *cudaStreamAddCallback* no CUDA function may be called from callback. *cudaErrorNotPermitted* may, but is not guaranteed to, be returned as a diagnostic in such case.

Parameters

- **pGraphNode** – - Returns newly created node
- **graph** – - Graph to which to add the node
- **pDependencies** – - Dependencies of the node
- **numDependencies** – - Number of dependencies
- **childGraph** – - The graph to clone into this node

Returns

cudaSuccess, *cudaErrorInvalidValue*

__host__ *cudaError_t* **cudaGraphAddDependencies**(*cudaGraph_t* graph, const *cudaGraphNode_t* *from, const *cudaGraphNode_t* *to, const *cudaGraphEdgeData* *edgeData, size_t numDependencies)

Adds dependency edges to a graph.

The number of dependencies to be added is defined by `numDependencies` Elements in `pFrom` and `pTo` at corresponding indices define a dependency. Each node in `pFrom` and `pTo` must belong to `graph`.

If `numDependencies` is 0, elements in `pFrom` and `pTo` will be ignored. Specifying an existing dependency will return an error.

See also:

cudaGraphRemoveDependencies, *cudaGraphGetEdges*, *cudaGraphNodeGetDependencies*, *cudaGraphNodeGetDependentNodes*

Note: Graph objects are not threadsafe. More here.

Note: Note that this function may also return error codes from previous, asynchronous launches.

Note: Note that this function may also return *cudaErrorInitializationError*, *cudaErrorInsufficientDriver* or *cudaErrorNoDevice* if this call tries to initialize internal CUDA RT state.

Note: Note that as specified by *cudaStreamAddCallback* no CUDA function may be called from callback. *cudaErrorNotPermitted* may, but is not guaranteed to, be returned as a diagnostic in such case.

Parameters

- **graph** – - Graph to which dependencies are added
- **from** – - Array of nodes that provide the dependencies
- **to** – - Array of dependent nodes

- **edgeData** – - Optional array of edge data. If NULL, default (zeroed) edge data is assumed.
- **numDependencies** – - Number of dependencies to be added

Returns
cudaSuccess, *cudaErrorInvalidValue*

__host__ *cudaError_t* **cudaGraphAddEmptyNode**(*cudaGraphNode_t* *pGraphNode, *cudaGraph_t* graph, const *cudaGraphNode_t* *pDependencies, size_t numDependencies)

Creates an empty node and adds it to a graph.

Creates a new node which performs no operation, and adds it to `graph` with `numDependencies` dependencies specified via `pDependencies`. It is possible for `numDependencies` to be 0, in which case the node will be placed at the root of the graph. `pDependencies` may not have any duplicate entries. A handle to the new node will be returned in `pGraphNode`.

An empty node performs no operation during execution, but can be used for transitive ordering. For example, a phased execution graph with 2 groups of n nodes with a barrier between them can be represented using an empty node and 2*n dependency edges, rather than no empty node and n^2 dependency edges.

See also:

cudaGraphAddNode, *cudaGraphCreate*, *cudaGraphDestroyNode*, *cudaGraphAddChildGraphNode*, *cudaGraphAddKernelNode*, *cudaGraphAddHostNode*, *cudaGraphAddMemcpyNode*, *cudaGraphAddMemsetNode*

Warning: This function will be deprecated in a future CUDA release. Please migrate to *cudaGraphAddNode*.

Note: Graph objects are not threadsafe. More here.

Note: Note that this function may also return *cudaErrorInitializationError*, *cudaErrorInsufficientDriver* or *cudaErrorNoDevice* if this call tries to initialize internal CUDA RT state.

Note: Note that as specified by *cudaStreamAddCallback* no CUDA function may be called from callback. *cudaErrorNotPermitted* may, but is not guaranteed to, be returned as a diagnostic in such case.

Parameters

- **pGraphNode** – - Returns newly created node
- **graph** – - Graph to which to add the node
- **pDependencies** – - Dependencies of the node

- **numDependencies** – - Number of dependencies

Returns

cudaSuccess, *cudaErrorInvalidValue*

__host__ *cudaError_t* **cudaGraphAddEventRecordNode**(*cudaGraphNode_t* *pGraphNode, *cudaGraph_t* graph, const *cudaGraphNode_t* *pDependencies, size_t numDependencies, *cudaEvent_t* event)

Creates an event record node and adds it to a graph.

Creates a new event record node and adds it to `hGraph` with `numDependencies` dependencies specified via `dependencies` and event specified in `event`. It is possible for `numDependencies` to be 0, in which case the node will be placed at the root of the graph. `dependencies` may not have any duplicate entries. A handle to the new node will be returned in `phGraphNode`.

Each launch of the graph will record `event` to capture execution of the node's dependencies.

These nodes may not be used in loops or conditionals.

See also:

cudaGraphAddNode, *cudaGraphAddEventWaitNode*, *cudaEventRecordWithFlags*, *cudaStreamWaitEvent*, *cudaGraphCreate*, *cudaGraphDestroyNode*, *cudaGraphAddChildGraphNode*, *cudaGraphAddEmptyNode*, *cudaGraphAddKernelNode*, *cudaGraphAddMemcpyNode*, *cudaGraphAddMemsetNode*

Warning: This function will be deprecated in a future CUDA release. Please migrate to *cudaGraphAddNode*.

Note: Graph objects are not threadsafe. More here.

Note: Note that this function may also return error codes from previous, asynchronous launches.

Note: Note that this function may also return *cudaErrorInitializationError*, *cudaErrorInsufficientDriver* or *cudaErrorNoDevice* if this call tries to initialize internal CUDA RT state.

Note: Note that as specified by *cudaStreamAddCallback* no CUDA function may be called from callback. *cudaErrorNotPermitted* may, but is not guaranteed to, be returned as a diagnostic in such case.

Parameters

- **phGraphNode** – - Returns newly created node
- **hGraph** – - Graph to which to add the node

- **dependencies** – - Dependencies of the node
- **numDependencies** – - Number of dependencies
- **event** – - Event for the node

Returns
cudaSuccess, *cudaErrorInvalidValue*

__host__ *cudaError_t* **cudaGraphAddEventWaitNode**(*cudaGraphNode_t* *pGraphNode, *cudaGraph_t* graph, const *cudaGraphNode_t* *pDependencies, size_t numDependencies, *cudaEvent_t* event)

Creates an event wait node and adds it to a graph.

Creates a new event wait node and adds it to `hGraph` with `numDependencies` dependencies specified via `dependencies` and event specified in `event`. It is possible for `numDependencies` to be 0, in which case the node will be placed at the root of the graph. `dependencies` may not have any duplicate entries. A handle to the new node will be returned in `phGraphNode`.

The graph node will wait for all work captured in `event`. See ::cuEventRecord() for details on what is captured by an event. The synchronization will be performed efficiently on the device when applicable. `event` may be from a different context or device than the launch stream.

These nodes may not be used in loops or conditionals.

See also:

cudaGraphAddNode, *cudaGraphAddEventRecordNode*, *cudaEventRecordWithFlags*, *cudaStreamWaitEvent*, *cudaGraphCreate*, *cudaGraphDestroyNode*, *cudaGraphAddChildGraphNode*, *cudaGraphAddEmptyNode*, *cudaGraphAddKernelNode*, *cudaGraphAddMemcpyNode*, *cudaGraphAddMemsetNode*

Warning: This function will be deprecated in a future CUDA release. Please migrate to *cudaGraphAddNode*.

Note: Graph objects are not threadsafe. More here.

Note: Note that this function may also return error codes from previous, asynchronous launches.

Note: Note that this function may also return *cudaErrorInitializationError*, *cudaErrorInsufficientDriver* or *cudaErrorNoDevice* if this call tries to initialize internal CUDA RT state.

Note: Note that as specified by *cudaStreamAddCallback* no CUDA function may be called from callback. *cudaErrorNotPermitted* may, but is not guaranteed to, be returned as a diagnostic in such case.

Parameters

- **phGraphNode** – - Returns newly created node
- **hGraph** – - Graph to which to add the node
- **dependencies** – - Dependencies of the node
- **numDependencies** – - Number of dependencies
- **event** – - Event for the node

Returns

cudaSuccess, *cudaErrorInvalidValue*

__host__ *cudaError_t* **cudaGraphAddExternalSemaphoresSignalNode**(*cudaGraphNode_t* *pGraphNode, *cudaGraph_t* graph, const *cudaGraphNode_t* *pDependencies, size_t numDependencies, const struct *cudaExternalSemaphoreSignalNodeParams* *nodeParams)

Creates an external semaphore signal node and adds it to a graph.

Creates a new external semaphore signal node and adds it to `graph` with `numDependencies` dependencies specified via `dependencies` and arguments specified in `nodeParams`. It is possible for `numDependencies` to be 0, in which case the node will be placed at the root of the graph. `dependencies` may not have any duplicate entries. A handle to the new node will be returned in `pGraphNode`.

Performs a signal operation on a set of externally allocated semaphore objects when the node is launched. The operation(s) will occur after all of the node's dependencies have completed.

See also:

cudaGraphAddNode, *cudaGraphExternalSemaphoresSignalNodeGetParams*, *cudaGraphExternalSemaphoresSignalNodeSetParams*, *cudaGraphExecExternalSemaphoresSignalNodeSetParams*, *cudaGraphAddExternalSemaphoresWaitNode*, *cudaImportExternalSemaphore*, *cudaSignalExternalSemaphoresAsync*, *cudaWaitExternalSemaphoresAsync*, *cudaGraphCreate*, *cudaGraphDestroyNode*, *cudaGraphAddEventRecordNode*, *cudaGraphAddEventWaitNode*, *cudaGraphAddChildGraphNode*, *cudaGraphAddEmptyNode*, *cudaGraphAddKernelNode*, *cudaGraphAddMemcpyNode*, *cudaGraphAddMemsetNode*

Warning: This function will be deprecated in a future CUDA release. Please migrate to *cudaGraphAddNode*.

Note: Graph objects are not threadsafe. More here.

Note: Note that this function may also return error codes from previous, asynchronous launches.

Note: Note that this function may also return *cudaErrorInitializationError*, *cudaErrorInsufficientDriver* or *cudaErrorNoDevice* if this call tries to initialize internal CUDA RT state.

Note: Note that as specified by *cudaStreamAddCallback* no CUDA function may be called from callback. *cudaErrorNotPermitted* may, but is not guaranteed to, be returned as a diagnostic in such case.

Parameters

- **pGraphNode** – - Returns newly created node
- **graph** – - Graph to which to add the node
- **pDependencies** – - Dependencies of the node
- **numDependencies** – - Number of dependencies
- **nodeParams** – - Parameters for the node

Returns
cudaSuccess, *cudaErrorInvalidValue*

__host__ *cudaError_t* **cudaGraphAddExternalSemaphoresWaitNode**(*cudaGraphNode_t* *pGraphNode, *cudaGraph_t* graph, const *cudaGraphNode_t* *pDependencies, size_t numDependencies, const struct *cudaExternalSemaphoreWaitNodeParams* *nodeParams)

Creates an external semaphore wait node and adds it to a graph.

Creates a new external semaphore wait node and adds it to `graph` with `numDependencies` dependencies specified via `dependencies` and arguments specified in `nodeParams`. It is possible for `numDependencies` to be 0, in which case the node will be placed at the root of the graph. `dependencies` may not have any duplicate entries. A handle to the new node will be returned in `pGraphNode`.

Performs a wait operation on a set of externally allocated semaphore objects when the node is launched. The node's dependencies will not be launched until the wait operation has completed.

See also:

cudaGraphAddNode, *cudaGraphExternalSemaphoresWaitNodeGetParams*, *cudaGraphExternalSemaphoresWaitNodeSetParams*, *cudaGraphExecExternalSemaphoresWaitNodeSetParams*,

cudaGraphAddExternalSemaphoresSignalNode, *cudaImportExternalSemaphore*, *cudaSignalExternalSemaphoresAsync*, *cudaWaitExternalSemaphoresAsync*, *cudaGraphCreate*, *cudaGraphDestroyNode*, *cudaGraphAddEventRecordNode*, *cudaGraphAddEventWaitNode*, *cudaGraphAddChildGraphNode*, *cudaGraphAddEmptyNode*, *cudaGraphAddKernelNode*, *cudaGraphAddMemcpyNode*, *cudaGraphAddMemsetNode*

Warning: This function will be deprecated in a future CUDA release. Please migrate to *cudaGraphAddNode*.

Note: Graph objects are not threadsafe. More here.

Note: Note that this function may also return error codes from previous, asynchronous launches.

Note: Note that this function may also return *cudaErrorInitializationError*, *cudaErrorInsufficientDriver* or *cudaErrorNoDevice* if this call tries to initialize internal CUDA RT state.

Note: Note that as specified by *cudaStreamAddCallback* no CUDA function may be called from callback. *cudaErrorNotPermitted* may, but is not guaranteed to, be returned as a diagnostic in such case.

Parameters

- **pGraphNode** – - Returns newly created node
- **graph** – - Graph to which to add the node
- **pDependencies** – - Dependencies of the node
- **numDependencies** – - Number of dependencies
- **nodeParams** – - Parameters for the node

Returns

cudaSuccess, *cudaErrorInvalidValue*

__host__ *cudaError_t* **cudaGraphAddHostNode**(*cudaGraphNode_t* *pGraphNode, *cudaGraph_t* graph, const *cudaGraphNode_t* *pDependencies, size_t numDependencies, const struct *cudaHostNodeParams* *pNodeParams)

Creates a host execution node and adds it to a graph.

Creates a new CPU execution node and adds it to `graph` with `numDependencies` dependencies specified via `pDependencies` and arguments specified in `pNodeParams`. It is possible for `numDependencies` to be 0, in which case the node will be placed at the root of the graph. `pDependencies` may not have any duplicate entries. A handle to the new node will be returned in `pGraphNode`.

When the graph is launched, the node will invoke the specified CPU function. Host nodes are not supported under MPS with pre-Volta GPUs.

See also:

cudaGraphAddNode, *cudaLaunchHostFunc*, *cudaGraphHostNodeGetParams*, *cudaGraphHostNodeSetParams*, *cudaGraphCreate*, *cudaGraphDestroyNode*, *cudaGraphAddChildGraphNode*, *cudaGraphAddEmptyNode*, *cudaGraphAddKernelNode*, *cudaGraphAddMemcpyNode*, *cudaGraphAddMemsetNode*

Warning: This function will be deprecated in a future CUDA release. Please migrate to *cudaGraphAddNode*.

Note: Graph objects are not threadsafe. More here.

Note: Note that this function may also return error codes from previous, asynchronous launches.

Note: Note that this function may also return *cudaErrorInitializationError*, *cudaErrorInsufficientDriver* or *cudaErrorNoDevice* if this call tries to initialize internal CUDA RT state.

Note: Note that as specified by *cudaStreamAddCallback* no CUDA function may be called from callback. *cudaErrorNotPermitted* may, but is not guaranteed to, be returned as a diagnostic in such case.

Parameters

- **pGraphNode** – - Returns newly created node
- **graph** – - Graph to which to add the node
- **pDependencies** – - Dependencies of the node
- **numDependencies** – - Number of dependencies
- **pNodeParams** – - Parameters for the host node

Returns

cudaSuccess, *cudaErrorNotSupported*, *cudaErrorInvalidValue*

__host__ *cudaError_t* **cudaGraphAddKernelNode**(*cudaGraphNode_t* *pGraphNode, *cudaGraph_t* graph, const *cudaGraphNode_t* *pDependencies, size_t numDependencies, const struct *cudaKernelNodeParams* *pNodeParams)

Creates a kernel execution node and adds it to a graph.

Creates a new kernel execution node and adds it to `graph` with `numDependencies` dependencies specified via `pDependencies` and arguments specified in `pNodeParams`. It is possible for `numDependencies` to be 0, in which case the node will be placed at the root of the graph. `pDependencies` may not have any duplicate entries. A handle to the new node will be returned in `pGraphNode`.

The *cudaKernelNodeParams* structure is defined as:

```
struct cudaKernelNodeParams
{
    void* func;
    dim3 gridDim;
    dim3 blockDim;
    unsigned int sharedMemBytes;
    void **kernelParams;
    void **extra;
};
```

When the graph is launched, the node will invoke kernel `func` on a (`gridDim.x` x `gridDim.y` x `gridDim.z`) grid of blocks. Each block contains (`blockDim.x` x `blockDim.y` x `blockDim.z`) threads.

`sharedMem` sets the amount of dynamic shared memory that will be available to each thread block.

Kernel parameters to `func` can be specified in one of two ways:

1) Kernel parameters can be specified via `kernelParams`. If the kernel has N parameters, then `kernelParams` needs to be an array of N pointers. Each pointer, from `kernelParams`[0] to `kernelParams`[N-1], points to the region of memory from which the actual parameter will be copied. The number of kernel parameters and their offsets and sizes do not need to be specified as that information is retrieved directly from the kernel's image.

2) Kernel parameters can also be packaged by the application into a single buffer that is passed in via `extra`. This places the burden on the application of knowing each kernel parameter's size and alignment/padding within the buffer. The `extra` parameter exists to allow this function to take additional less commonly used arguments. `extra` specifies a list of names of extra settings and their corresponding values. Each extra setting name is immediately followed by the corresponding value. The list must be terminated with either NULL or CU_LAUNCH_PARAM_END.

- ::CU_LAUNCH_PARAM_END, which indicates the end of the `extra` array;
- ::CU_LAUNCH_PARAM_BUFFER_POINTER, which specifies that the next value in `extra` will be a pointer to a buffer containing all the kernel parameters for launching kernel `func`;
- ::CU_LAUNCH_PARAM_BUFFER_SIZE, which specifies that the next value in `extra` will be a pointer to a size_t containing the size of the buffer specified with ::CU_LAUNCH_PARAM_BUFFER_POINTER;

The error *cudaErrorInvalidValue* will be returned if kernel parameters are specified with both `kernelParams` and `extra` (i.e. both `kernelParams` and `extra` are non-NULL).

The `kernelParams` or `extra` array, as well as the argument values it points to, are copied during this call.

See also:

cudaGraphAddNode, *cudaLaunchKernel*, *cudaGraphKernelNodeGetParams*, *cudaGraphKernelNodeSetParams*, *cudaGraphCreate*, *cudaGraphDestroyNode*, *cudaGraphAddChildGraphNode*, *cudaGraphAddEmptyNode*, *cudaGraphAddHostNode*, *cudaGraphAddMemcpyNode*, *cudaGraphAddMemsetNode*

Warning: This function will be deprecated in a future CUDA release. Please migrate to *cudaGraphAddNode*.

Note: Kernels launched using graphs must not use texture and surface references. Reading or writing through any texture or surface reference is undefined behavior. This restriction does not apply to texture and surface objects.

Note: Graph objects are not threadsafe. More here.

Note: Note that this function may also return error codes from previous, asynchronous launches.

Note: Note that this function may also return *cudaErrorInitializationError*, *cudaErrorInsufficientDriver* or *cudaErrorNoDevice* if this call tries to initialize internal CUDA RT state.

Note: Note that as specified by *cudaStreamAddCallback* no CUDA function may be called from callback. *cudaErrorNotPermitted* may, but is not guaranteed to, be returned as a diagnostic in such case.

Note: The API can also be used with a kernel *cudaKernel_t* by querying the handle using *cudaLibraryGetKernel()* or *cudaGetKernel* and then passing it to the API by casting to void*. The symbol `entryFuncAddr` passed to *cudaGetKernel* should be a symbol that is registered with the same CUDA Runtime instance.

Note: Passing a symbol that belongs that belongs to a different runtime instance will result in undefined behavior. The only type that can be reliably passed to a different runtime instance is *cudaKernel_t*

Parameters

- **pGraphNode** – - Returns newly created node
- **graph** – - Graph to which to add the node
- **pDependencies** – - Dependencies of the node
- **numDependencies** – - Number of dependencies

- **pNodeParams** – - Parameters for the GPU execution node

Returns

cudaSuccess, *cudaErrorInvalidValue*, *cudaErrorInvalidDeviceFunction*

__host__ *cudaError_t* **cudaGraphAddMemAllocNode**(*cudaGraphNode_t* *pGraphNode, *cudaGraph_t* graph, const *cudaGraphNode_t* *pDependencies, size_t numDependencies, struct *cudaMemAllocNodeParams* *nodeParams)

Creates an allocation node and adds it to a graph.

Creates a new allocation node and adds it to `graph` with `numDependencies` dependencies specified via `pDependencies` and arguments specified in `nodeParams`. It is possible for `numDependencies` to be 0, in which case the node will be placed at the root of the graph. `pDependencies` may not have any duplicate entries. A handle to the new node will be returned in `pGraphNode`.

When *cudaGraphAddMemAllocNode* creates an allocation node, it returns the address of the allocation in `nodeParams.dptr`. The allocation's address remains fixed across instantiations and launches.

If the allocation is freed in the same graph, by creating a free node using *cudaGraphAddMemFreeNode*, the allocation can be accessed by nodes ordered after the allocation node but before the free node. These allocations cannot be freed outside the owning graph, and they can only be freed once in the owning graph.

If the allocation is not freed in the same graph, then it can be accessed not only by nodes in the graph which are ordered after the allocation node, but also by stream operations ordered after the graph's execution but before the allocation is freed.

Allocations which are not freed in the same graph can be freed by:

- passing the allocation to ::cudaMemFreeAsync or ::cudaMemFree;
- launching a graph with a free node for that allocation; or
- specifying *cudaGraphInstantiateFlagAutoFreeOnLaunch* during instantiation, which makes each launch behave as though it called ::cudaMemFreeAsync for every unfreed allocation.

It is not possible to free an allocation in both the owning graph and another graph. If the allocation is freed in the same graph, a free node cannot be added to another graph. If the allocation is freed in another graph, a free node can no longer be added to the owning graph.

The following restrictions apply to graphs which contain allocation and/or memory free nodes:

- Nodes and edges of the graph cannot be deleted.
- The graph can only be used in a child node if the ownership is moved to the parent.
- Only one instantiation of the graph may exist at any point in time.
- The graph cannot be cloned.

See also:

cudaGraphAddNode, *cudaGraphAddMemFreeNode*, *cudaGraphMemAllocNodeGetParams*, *cudaDeviceGraphMemTrim*, *cudaDeviceGetGraphMemAttribute*, *cudaDeviceSetGraphMemAttribute*, *cudaMallocAsync*, *cudaFreeAsync*, *cudaGraphCreate*, *cudaGraphDestroyNode*, *cuda-*

GraphAddChildGraphNode, *cudaGraphAddEmptyNode*, *cudaGraphAddEventRecordNode*, *cudaGraphAddEventWaitNode*, *cudaGraphAddExternalSemaphoresSignalNode*, *cudaGraphAddExternalSemaphoresWaitNode*, *cudaGraphAddKernelNode*, *cudaGraphAddMemcpyNode*, *cudaGraphAddMemsetNode*

Warning: This function will be deprecated in a future CUDA release. Please migrate to *cudaGraphAddNode*.

Note: Graph objects are not threadsafe. More here.

Note: Note that this function may also return error codes from previous, asynchronous launches.

Parameters

- **pGraphNode** – - Returns newly created node
- **graph** – - Graph to which to add the node
- **pDependencies** – - Dependencies of the node
- **numDependencies** – - Number of dependencies
- **nodeParams** – - Parameters for the node

Returns

cudaSuccess, *cudaErrorCudartUnloading*, *cudaErrorInitializationError*, *cudaErrorNotSupported*, *cudaErrorInvalidValue*, ::cudaErrorOutOfMemory

__host__ *cudaError_t* **cudaGraphAddMemFreeNode**(*cudaGraphNode_t* *pGraphNode, *cudaGraph_t* graph, const *cudaGraphNode_t* *pDependencies, size_t numDependencies, void *dptr)

Creates a memory free node and adds it to a graph.

Creates a new memory free node and adds it to `graph` with `numDependencies` dependencies specified via `pDependencies` and address specified in `dptr`. It is possible for `numDependencies` to be 0, in which case the node will be placed at the root of the graph. `pDependencies` may not have any duplicate entries. A handle to the new node will be returned in `pGraphNode`.

cudaGraphAddMemFreeNode will return *cudaErrorInvalidValue* if the user attempts to free:

- an allocation twice in the same graph.
- an address that was not returned by an allocation node.
- an invalid address.

The following restrictions apply to graphs which contain allocation and/or memory free nodes:

- Nodes and edges of the graph cannot be deleted.
- The graph can only be used in a child node if the ownership is moved to the parent.
- Only one instantiation of the graph may exist at any point in time.

- The graph cannot be cloned.

See also:

cudaGraphAddNode, *cudaGraphAddMemAllocNode*, *cudaGraphMemFreeNodeGetParams*, *cudaDeviceGraphMemTrim*, *cudaDeviceGetGraphMemAttribute*, *cudaDeviceSetGraphMemAttribute*, *cudaMallocAsync*, *cudaFreeAsync*, *cudaGraphCreate*, *cudaGraphDestroyNode*, *cudaGraphAddChildGraphNode*, *cudaGraphAddEmptyNode*, *cudaGraphAddEventRecordNode*, *cudaGraphAddEventWaitNode*, *cudaGraphAddExternalSemaphoresSignalNode*, *cudaGraphAddExternalSemaphoresWaitNode*, *cudaGraphAddKernelNode*, *cudaGraphAddMemcpyNode*, *cudaGraphAddMemsetNode*

Warning: This function will be deprecated in a future CUDA release. Please migrate to *cudaGraphAddNode*.

Note: Graph objects are not threadsafe. More here.

Note: Note that this function may also return error codes from previous, asynchronous launches.

Parameters

- **pGraphNode** – - Returns newly created node
- **graph** – - Graph to which to add the node
- **pDependencies** – - Dependencies of the node
- **numDependencies** – - Number of dependencies
- **dptr** – - Address of memory to free

Returns

cudaSuccess, *cudaErrorCudartUnloading*, *cudaErrorInitializationError*, *cudaErrorNotSupported*, *cudaErrorInvalidValue*, ::cudaErrorOutOfMemory

__host__ *cudaError_t* **cudaGraphAddMemcpyNode**(*cudaGraphNode_t* *pGraphNode, *cudaGraph_t* graph, const *cudaGraphNode_t* *pDependencies, size_t numDependencies, const struct *cudaMemcpy3DParms* *pCopyParams)

Creates a memcpy node and adds it to a graph.

Creates a new memcpy node and adds it to `graph` with `numDependencies` dependencies specified via `pDependencies`. It is possible for `numDependencies` to be 0, in which case the node will be placed at the root of the graph. `pDependencies` may not have any duplicate entries. A handle to the new node will be returned in `pGraphNode`.

When the graph is launched, the node will perform the memcpy described by `pCopyParams`. See *cudaMemcpy3D()* for a description of the structure and its restrictions.

Memcpy nodes have some additional restrictions with regards to managed memory, if the system contains at least one device which has a zero value for the device attribute *cudaDevAttrConcurrentManagedAccess*.

See also:

cudaGraphAddNode, *cudaMemcpy3D*, *cudaGraphAddMemcpyNodeToSymbol*, *cudaGraphAddMemcpyNodeFromSymbol*, *cudaGraphAddMemcpyNode1D*, *cudaGraphMemcpyNodeGetParams*, *cudaGraphMemcpyNodeSetParams*, *cudaGraphCreate*, *cudaGraphDestroyNode*, *cudaGraphAddChildGraphNode*, *cudaGraphAddEmptyNode*, *cudaGraphAddKernelNode*, *cudaGraphAddHostNode*, *cudaGraphAddMemsetNode*

Warning: This function will be deprecated in a future CUDA release. Please migrate to *cudaGraphAddNode*.

Note: Graph objects are not threadsafe. More here.

Note: Note that this function may also return error codes from previous, asynchronous launches.

Note: Note that this function may also return *cudaErrorInitializationError*, *cudaErrorInsufficientDriver* or *cudaErrorNoDevice* if this call tries to initialize internal CUDA RT state.

Note: Note that as specified by *cudaStreamAddCallback* no CUDA function may be called from callback. *cudaErrorNotPermitted* may, but is not guaranteed to, be returned as a diagnostic in such case.

Parameters

- **pGraphNode** – - Returns newly created node
- **graph** – - Graph to which to add the node
- **pDependencies** – - Dependencies of the node
- **numDependencies** – - Number of dependencies
- **pCopyParams** – - Parameters for the memory copy

Returns

cudaSuccess, *cudaErrorInvalidValue*

__host__ *cudaError_t* **cudaGraphAddMemcpyNode1D**(*cudaGraphNode_t* *pGraphNode, *cudaGraph_t* graph, const *cudaGraphNode_t* *pDependencies, size_t numDependencies, void *dst, const void *src, size_t count, enum *cudaMemcpyKind* kind)

Creates a 1D memcpy node and adds it to a graph.

Creates a new 1D memcpy node and adds it to `graph` with `numDependencies` dependencies specified via `pDependencies`. It is possible for `numDependencies` to be 0, in which case the node will be placed at the root of the graph. `pDependencies` may not have any duplicate entries. A handle to the new node will be returned in `pGraphNode`.

When the graph is launched, the node will copy `count` bytes from the memory area pointed to by `src` to the memory area pointed to by `dst`, where `kind` specifies the direction of the copy, and must be one of *cudaMemcpyHostToHost*, *cudaMemcpyHostToDevice*, *cudaMemcpyDeviceToHost*, *cudaMemcpyDeviceToDevice*, or *cudaMemcpyDefault*. Passing *cudaMemcpyDefault* is recommended, in which case the type of transfer is inferred from the pointer values. However, *cudaMemcpyDefault* is only allowed on systems that support unified virtual addressing. Launching a memcpy node with dst and src pointers that do not match the direction of the copy results in an undefined behavior.

Memcpy nodes have some additional restrictions with regards to managed memory, if the system contains at least one device which has a zero value for the device attribute *cudaDevAttrConcurrentManagedAccess*.

See also:

cudaMemcpy, *cudaGraphAddMemcpyNode*, *cudaGraphMemcpyNodeGetParams*, *cudaGraphMemcpyNodeSetParams*, *cudaGraphMemcpyNodeSetParams1D*, *cudaGraphCreate*, *cudaGraphDestroyNode*, *cudaGraphAddChildGraphNode*, *cudaGraphAddEmptyNode*, *cudaGraphAddKernelNode*, *cudaGraphAddHostNode*, *cudaGraphAddMemsetNode*

Note: Graph objects are not threadsafe. More here.

Note: Note that this function may also return error codes from previous, asynchronous launches.

Note: Note that this function may also return *cudaErrorInitializationError*, *cudaErrorInsufficientDriver* or *cudaErrorNoDevice* if this call tries to initialize internal CUDA RT state.

Note: Note that as specified by *cudaStreamAddCallback* no CUDA function may be called from callback. *cudaErrorNotPermitted* may, but is not guaranteed to, be returned as a diagnostic in such case.

Parameters

- **`pGraphNode`** – - Returns newly created node
- **`graph`** – - Graph to which to add the node
- **`pDependencies`** – - Dependencies of the node
- **`numDependencies`** – - Number of dependencies
- **`dst`** – - Destination memory address
- **`src`** – - Source memory address
- **`count`** – - Size in bytes to copy

- **kind** – - Type of transfer

Returns
cudaSuccess, *cudaErrorInvalidValue*

__host__ *cudaError_t* **cudaGraphAddMemcpyNodeFromSymbol**(*cudaGraphNode_t* *pGraphNode, *cudaGraph_t* graph, const *cudaGraphNode_t* *pDependencies, size_t numDependencies, void *dst, const void *symbol, size_t count, size_t offset, enum *cudaMemcpyKind* kind)

Creates a memcpy node to copy from a symbol on the device and adds it to a graph.

Creates a new memcpy node to copy from `symbol` and adds it to `graph` with `numDependencies` dependencies specified via `pDependencies`. It is possible for `numDependencies` to be 0, in which case the node will be placed at the root of the graph. `pDependencies` may not have any duplicate entries. A handle to the new node will be returned in `pGraphNode`.

When the graph is launched, the node will copy `count` bytes from the memory area pointed to by `offset` bytes from the start of symbol `symbol` to the memory area pointed to by `dst`. The memory areas may not overlap. `symbol` is a variable that resides in global or constant memory space. `kind` can be either *cudaMemcpyDeviceToHost*, *cudaMemcpyDeviceToDevice*, or *cudaMemcpyDefault*. Passing *cudaMemcpyDefault* is recommended, in which case the type of transfer is inferred from the pointer values. However, *cudaMemcpyDefault* is only allowed on systems that support unified virtual addressing.

Memcpy nodes have some additional restrictions with regards to managed memory, if the system contains at least one device which has a zero value for the device attribute *cudaDevAttrConcurrentManagedAccess*.

See also:

cudaMemcpyFromSymbol, *cudaGraphAddMemcpyNode*, *cudaGraphAddMemcpyNodeToSymbol*, *cudaGraphMemcpyNodeGetParams*, *cudaGraphMemcpyNodeSetParams*, *cudaGraphMemcpyNodeSetParamsFromSymbol*, *cudaGraphMemcpyNodeSetParamsToSymbol*, *cudaGraphCreate*, *cudaGraphDestroyNode*, *cudaGraphAddChildGraphNode*, *cudaGraphAddEmptyNode*, *cudaGraphAddKernelNode*, *cudaGraphAddHostNode*, *cudaGraphAddMemsetNode*

Note: Graph objects are not threadsafe. More here.

Note: Note that this function may also return error codes from previous, asynchronous launches.

Note: Note that this function may also return *cudaErrorInitializationError*, *cudaErrorInsufficientDriver* or *cudaErrorNoDevice* if this call tries to initialize internal CUDA RT state.

Note: Note that as specified by *cudaStreamAddCallback* no CUDA function may be called from callback. *cudaErrorNotPermitted* may, but is not guaranteed to, be returned as a diagnostic in such case.

Parameters

- **pGraphNode** – - Returns newly created node
- **graph** – - Graph to which to add the node
- **pDependencies** – - Dependencies of the node
- **numDependencies** – - Number of dependencies
- **dst** – - Destination memory address
- **symbol** – - Device symbol address
- **count** – - Size in bytes to copy
- **offset** – - Offset from start of symbol in bytes
- **kind** – - Type of transfer

Returns

cudaSuccess, *cudaErrorInvalidValue*

__host__ *cudaError_t* **cudaGraphAddMemcpyNodeToSymbol**(*cudaGraphNode_t* *pGraphNode, *cudaGraph_t* graph, const *cudaGraphNode_t* *pDependencies, size_t numDependencies, const void *symbol, const void *src, size_t count, size_t offset, enum *cudaMemcpyKind* kind)

Creates a memcpy node to copy to a symbol on the device and adds it to a graph.

Creates a new memcpy node to copy to `symbol` and adds it to `graph` with `numDependencies` dependencies specified via `pDependencies`. It is possible for `numDependencies` to be 0, in which case the node will be placed at the root of the graph. `pDependencies` may not have any duplicate entries. A handle to the new node will be returned in `pGraphNode`.

When the graph is launched, the node will copy `count` bytes from the memory area pointed to by `src` to the memory area pointed to by `offset` bytes from the start of symbol `symbol`. The memory areas may not overlap. `symbol` is a variable that resides in global or constant memory space. `kind` can be either *cudaMemcpyHostToDevice*, *cudaMemcpyDeviceToDevice*, or *cudaMemcpyDefault*. Passing *cudaMemcpyDefault* is recommended, in which case the type of transfer is inferred from the pointer values. However, *cudaMemcpyDefault* is only allowed on systems that support unified virtual addressing.

Memcpy nodes have some additional restrictions with regards to managed memory, if the system contains at least one device which has a zero value for the device attribute *cudaDevAttrConcurrentManagedAccess*.

See also:

cudaMemcpyToSymbol, *cudaGraphAddMemcpyNode*, *cudaGraphAddMemcpyNodeFromSymbol*, *cudaGraphMemcpyNodeGetParams*, *cudaGraphMemcpyNodeSetParams*, *cudaGraphMemcpyNodeSetParamsToSymbol*, *cudaGraphMemcpyNodeSetParamsFromSymbol*, *cudaGraphCreate*, *cudaGraphDestroyNode*, *cudaGraphAddChildGraphNode*, *cudaGraphAddEmptyNode*, *cudaGraphAddKernelNode*, *cudaGraphAddHostNode*, *cudaGraphAddMemsetNode*

Note: Graph objects are not threadsafe. More here.

Note: Note that this function may also return error codes from previous, asynchronous launches.

Note: Note that this function may also return *cudaErrorInitializationError*, *cudaErrorInsufficientDriver* or *cudaErrorNoDevice* if this call tries to initialize internal CUDA RT state.

Note: Note that as specified by *cudaStreamAddCallback* no CUDA function may be called from callback. *cudaErrorNotPermitted* may, but is not guaranteed to, be returned as a diagnostic in such case.

Parameters

- **pGraphNode** – - Returns newly created node
- **graph** – - Graph to which to add the node
- **pDependencies** – - Dependencies of the node
- **numDependencies** – - Number of dependencies
- **symbol** – - Device symbol address
- **src** – - Source memory address
- **count** – - Size in bytes to copy
- **offset** – - Offset from start of symbol in bytes
- **kind** – - Type of transfer

Returns
cudaSuccess, *cudaErrorInvalidValue*

__host__ *cudaError_t* **cudaGraphAddMemsetNode**(*cudaGraphNode_t* *pGraphNode, *cudaGraph_t* graph, const *cudaGraphNode_t* *pDependencies, size_t numDependencies, const struct *cudaMemsetParams* *pMemsetParams)

Creates a memset node and adds it to a graph.

Creates a new memset node and adds it to `graph` with `numDependencies` dependencies specified via `pDependencies`. It is possible for `numDependencies` to be 0, in which case the node will be placed at the root of the graph. `pDependencies` may not have any duplicate entries. A handle to the new node will be returned in `pGraphNode`.

The element size must be 1, 2, or 4 bytes. When the graph is launched, the node will perform the memset described by `pMemsetParams`.

See also:

cudaGraphAddNode, *cudaMemset2D*, *cudaGraphMemsetNodeGetParams*, *cudaGraphMemsetNodeSetParams*, *cudaGraphCreate*, *cudaGraphDestroyNode*, *cudaGraphAddChildGraphNode*, *cudaGraphAddEmptyNode*, *cudaGraphAddKernelNode*, *cudaGraphAddHostNode*, *cudaGraphAddMemcpyNode*

Warning: This function will be deprecated in a future CUDA release. Please migrate to *cudaGraphAddNode*.

Note: Graph objects are not threadsafe. More here.

Note: Note that this function may also return error codes from previous, asynchronous launches.

Note: Note that this function may also return *cudaErrorInitializationError*, *cudaErrorInsufficientDriver* or *cudaErrorNoDevice* if this call tries to initialize internal CUDA RT state.

Note: Note that as specified by *cudaStreamAddCallback* no CUDA function may be called from callback. *cudaErrorNotPermitted* may, but is not guaranteed to, be returned as a diagnostic in such case.

Parameters

- **pGraphNode** – - Returns newly created node
- **graph** – - Graph to which to add the node
- **pDependencies** – - Dependencies of the node
- **numDependencies** – - Number of dependencies
- **pMemsetParams** – - Parameters for the memory set

Returns

cudaSuccess, *cudaErrorInvalidValue*, *cudaErrorInvalidDevice*

__host__ *cudaError_t* **cudaGraphAddNode**(*cudaGraphNode_t* *pGraphNode, *cudaGraph_t* graph, const *cudaGraphNode_t* *pDependencies, const *cudaGraphEdgeData* *dependencyData, size_t numDependencies, struct *cudaGraphNodeParams* *nodeParams)

Adds a node of arbitrary type to a graph.

Creates a new node in `graph` described by `nodeParams` with `numDependencies` dependencies specified via `pDependencies`. `numDependencies` may be 0. `pDependencies` may be null if `numDependencies` is 0. `pDependencies` may not have any duplicate entries.

`nodeParams` is a tagged union. The node type should be specified in the `type` field, and type-specific parameters in the corresponding union member. All unused bytes - that is, `reserved0`

and all bytes past the utilized union member - must be set to zero. It is recommended to use brace initialization or memset to ensure all bytes are initialized.

Note that for some node types, `nodeParams` may contain "out parameters" which are modified during the call, such as `nodeParams->alloc.dptr`.

A handle to the new node will be returned in `phGraphNode`.

See also:

cudaGraphCreate, *cudaGraphNodeSetParams*, *cudaGraphExecNodeSetParams*

Note: Graph objects are not threadsafe. More here.

Note: Note that this function may also return error codes from previous, asynchronous launches.

Note: Note that this function may also return *cudaErrorInitializationError*, *cudaErrorInsufficientDriver* or *cudaErrorNoDevice* if this call tries to initialize internal CUDA RT state.

Note: Note that as specified by *cudaStreamAddCallback* no CUDA function may be called from callback. *cudaErrorNotPermitted* may, but is not guaranteed to, be returned as a diagnostic in such case.

Parameters

- **pGraphNode** – - Returns newly created node
- **graph** – - Graph to which to add the node
- **pDependencies** – - Dependencies of the node
- **dependencyData** – - Optional edge data for the dependencies. If NULL, the data is assumed to be default (zeroed) for all dependencies.
- **numDependencies** – - Number of dependencies
- **nodeParams** – - Specification of the node

Returns

cudaSuccess, *cudaErrorInvalidValue*, *cudaErrorInvalidDeviceFunction*, *cudaErrorNotSupported*

__host__ *cudaError_t* **cudaGraphChildGraphNodeGetGraph**(*cudaGraphNode_t* node, *cudaGraph_t* *pGraph)

Gets a handle to the embedded graph of a child graph node.

Gets a handle to the embedded graph in a child graph node. This call does not clone the graph. Changes to the graph will be reflected in the node, and the node retains ownership of the graph.

Allocation and free nodes cannot be added to the returned graph. Attempting to do so will return an error.

See also:

cudaGraphAddChildGraphNode, *cudaGraphNodeFindInClone*

Warning: This function will be deprecated in a future CUDA release. Please migrate to *cudaGraphNodeGetParams*.

Note: Graph objects are not threadsafe. More here.

Note: Note that this function may also return error codes from previous, asynchronous launches.

Note: Note that this function may also return *cudaErrorInitializationError*, *cudaErrorInsufficientDriver* or *cudaErrorNoDevice* if this call tries to initialize internal CUDA RT state.

Note: Note that as specified by *cudaStreamAddCallback* no CUDA function may be called from callback. *cudaErrorNotPermitted* may, but is not guaranteed to, be returned as a diagnostic in such case.

Parameters

- **node** – - Node to get the embedded graph for
- **pGraph** – - Location to store a handle to the graph

Returns

cudaSuccess, *cudaErrorInvalidValue*

__host__ *cudaError_t* **cudaGraphClone**(*cudaGraph_t* *pGraphClone, *cudaGraph_t* originalGraph)

Clones a graph.

This function creates a copy of `originalGraph` and returns it in `pGraphClone`. All parameters are copied into the cloned graph. The original graph may be modified after this call without affecting the clone.

Child graph nodes in the original graph are recursively copied into the clone.

See also:

cudaGraphCreate, *cudaGraphNodeFindInClone*

Note: : Cloning is not supported for graphs which contain memory allocation nodes, memory free nodes, or conditional nodes.

Note: Graph objects are not threadsafe. More here.

Note: Note that this function may also return error codes from previous, asynchronous launches.

Note: Note that this function may also return *cudaErrorInitializationError*, *cudaErrorInsufficientDriver* or *cudaErrorNoDevice* if this call tries to initialize internal CUDA RT state.

Note: Note that as specified by *cudaStreamAddCallback* no CUDA function may be called from callback. *cudaErrorNotPermitted* may, but is not guaranteed to, be returned as a diagnostic in such case.

Parameters

- **pGraphClone** – - Returns newly created cloned graph
- **originalGraph** – - Graph to clone

Returns
cudaSuccess, *cudaErrorInvalidValue*, *cudaErrorMemoryAllocation*

__host__ *cudaError_t* **cudaGraphConditionalHandleCreate**(*cudaGraphConditionalHandle* *pHandle_out, *cudaGraph_t* graph, unsigned int defaultLaunchValue = 0, unsigned int flags = 0)

Create a conditional handle.

Creates a conditional handle associated with `hGraph`.

The conditional handle must be associated with a conditional node in this graph or one of its children.

Handles not associated with a conditional node may cause graph instantiation to fail.

See also:

::cuGraphAddNode,

Note: Graph objects are not threadsafe. More here.

Note: Note that this function may also return error codes from previous, asynchronous launches.

Parameters

- **pHandle_out** – - Pointer used to return the handle to the caller.
- **graph** – - Graph which will contain the conditional node using this handle.

- **defaultLaunchValue** – - Optional initial value for the conditional variable. Applied at the beginning of each graph execution if cudaGraphCondAssignDefault is set in `flags`.
- **flags** – - Currently must be cudaGraphCondAssignDefault or 0.

Returns
::CUDA_SUCCESS, ::CUDA_ERROR_INVALID_VALUE, ::CUDA_ERROR_NOT_SUPPORTED

__host__ *cudaError_t* **cudaGraphConditionalHandleCreate_v2**(*cudaGraphConditionalHandle* *pHandle_out, *cudaGraph_t* graph, *cudaExecutionContext_t* ctx = NULL, unsigned int defaultLaunchValue = 0, unsigned int flags = 0)

Create a conditional handle.

Creates a conditional handle associated with `hGraph`.

The conditional handle must be associated with a conditional node in this graph or one of its children.

Handles not associated with a conditional node may cause graph instantiation to fail.

See also:

::cuGraphAddNode,

Note: Graph objects are not threadsafe. More here.

Note: Note that this function may also return error codes from previous, asynchronous launches.

Parameters

- **pHandle_out** – - Pointer used to return the handle to the caller.
- **graph** – - Graph which will contain the conditional node using this handle.
- **ctx** – - Execution context for the handle and associated conditional node. If NULL, current context will be used.
- **defaultLaunchValue** – - Optional initial value for the conditional variable. Applied at the beginning of each graph execution if cudaGraphCondAssignDefault is set in `flags`.
- **flags** – - Currently must be cudaGraphCondAssignDefault or 0.

Returns
::CUDA_SUCCESS, ::CUDA_ERROR_INVALID_VALUE, ::CUDA_ERROR_NOT_SUPPORTED

__host__ *cudaError_t* **cudaGraphCreate**(*cudaGraph_t* *pGraph, unsigned int flags = 0)

Creates a graph.

Creates an empty graph, which is returned via `pGraph`.

See also:

cudaGraphAddChildGraphNode, *cudaGraphAddEmptyNode*, *cudaGraphAddKernelNode*, *cudaGraphAddHostNode*, *cudaGraphAddMemcpyNode*, *cudaGraphAddMemsetNode*, *cudaGraphInstantiate*, *cudaGraphDestroy*, *cudaGraphGetNodes*, *cudaGraphGetRootNodes*, *cudaGraphGetEdges*, *cudaGraphClone*

Note: Graph objects are not threadsafe. More here.

Note: Note that this function may also return error codes from previous, asynchronous launches.

Note: Note that this function may also return *cudaErrorInitializationError*, *cudaErrorInsufficientDriver* or *cudaErrorNoDevice* if this call tries to initialize internal CUDA RT state.

Note: Note that as specified by *cudaStreamAddCallback* no CUDA function may be called from callback. *cudaErrorNotPermitted* may, but is not guaranteed to, be returned as a diagnostic in such case.

Parameters

- **pGraph** – - Returns newly created graph
- **flags** – - Graph creation flags, must be 0

Returns

cudaSuccess, *cudaErrorInvalidValue*, *cudaErrorMemoryAllocation*

__host__ *cudaError_t* **cudaGraphDebugDotPrint**(*cudaGraph_t* graph, const char *path, unsigned int flags)

Write a DOT file describing graph structure.

Using the provided `graph`, write to `path` a DOT formatted description of the graph. By default this includes the graph topology, node types, node id, kernel names and memcpy direction. `flags` can be specified to write more detailed information about each node type such as parameter values, kernel attributes, node and function handles.

Parameters

- **graph** – - The graph to create a DOT file from
- **path** – - The path to write the DOT file to
- **flags** – - Flags from cudaGraphDebugDotFlags for specifying which additional node information to write

Returns

cudaSuccess, *cudaErrorInvalidValue*, *cudaErrorOperatingSystem*

__host__ *cudaError_t* **cudaGraphDestroy**(*cudaGraph_t* graph)

Destroys a graph.

Destroys the graph specified by `graph`, as well as all of its nodes.

See also:

cudaGraphCreate

Note: Graph objects are not threadsafe. More here.

Note: Note that this function may also return error codes from previous, asynchronous launches.

Note: Note that this function may also return *cudaErrorInitializationError*, *cudaErrorInsufficientDriver* or *cudaErrorNoDevice* if this call tries to initialize internal CUDA RT state.

Note: Note that as specified by *cudaStreamAddCallback* no CUDA function may be called from callback. *cudaErrorNotPermitted* may, but is not guaranteed to, be returned as a diagnostic in such case.

Note: Use of the handle after this call is undefined behavior.

Parameters

graph – - Graph to destroy

Returns

cudaSuccess, *cudaErrorInvalidValue*

__host__ *cudaError_t* **cudaGraphDestroyNode**(*cudaGraphNode_t* node)

Remove a node from the graph.

Removes `node` from its graph. This operation also severs any dependencies of other nodes on `node` and vice versa.

Dependencies cannot be removed from graphs which contain allocation or free nodes. Any attempt to do so will return an error.

See also:

cudaGraphAddChildGraphNode, *cudaGraphAddEmptyNode*, *cudaGraphAddKernelNode*, *cudaGraphAddHostNode*, *cudaGraphAddMemcpyNode*, *cudaGraphAddMemsetNode*

Note: Graph objects are not threadsafe. More here.

Note: Note that this function may also return error codes from previous, asynchronous launches.

Note: Note that this function may also return *cudaErrorInitializationError*, *cudaErrorInsufficientDriver* or *cudaErrorNoDevice* if this call tries to initialize internal CUDA RT state.

Note: Note that as specified by *cudaStreamAddCallback* no CUDA function may be called from callback. *cudaErrorNotPermitted* may, but is not guaranteed to, be returned as a diagnostic in such case.

Note: Use of the handle after this call is undefined behavior.

Parameters
node – - Node to remove

Returns
cudaSuccess, *cudaErrorInvalidValue*

__host__ *cudaError_t* **cudaGraphEventRecordNodeGetEvent**(*cudaGraphNode_t* node, *cudaEvent_t* *event_out)

Returns the event associated with an event record node.

Returns the event of event record node `hNode` in `event_out`.

See also:

cudaGraphAddEventRecordNode, *cudaGraphEventRecordNodeSetEvent*, *cudaGraphEventWaitNodeGetEvent*, *cudaEventRecordWithFlags*, *cudaStreamWaitEvent*

Warning: This function will be deprecated in a future CUDA release. Please migrate to *cudaGraphNodeGetParams*.

Note: Graph objects are not threadsafe. More here.

Note: Note that this function may also return error codes from previous, asynchronous launches.

Note: Note that this function may also return *cudaErrorInitializationError*, *cudaErrorInsufficientDriver* or *cudaErrorNoDevice* if this call tries to initialize internal CUDA RT state.

Note: Note that as specified by *cudaStreamAddCallback* no CUDA function may be called from callback. *cudaErrorNotPermitted* may, but is not guaranteed to, be returned as a diagnostic in such case.

Parameters

- **hNode** – - Node to get the event for
- **event_out** – - Pointer to return the event

Returns

cudaSuccess, *cudaErrorInvalidValue*

__host__ *cudaError_t* **cudaGraphEventRecordNodeSetEvent**(*cudaGraphNode_t* node, *cudaEvent_t* event)

Sets an event record node's event.

Sets the event of event record node `hNode` to `event`.

See also:

cudaGraphNodeSetParams, *cudaGraphAddEventRecordNode*, *cudaGraphEventRecordNodeGetEvent*, *cudaGraphEventWaitNodeSetEvent*, *cudaEventRecordWithFlags*, *cudaStreamWaitEvent*

Warning: This function will be deprecated in a future CUDA release. Please migrate to *cudaGraphNodeSetParams*.

Note: Graph objects are not threadsafe. More here.

Note: Note that this function may also return error codes from previous, asynchronous launches.

Note: Note that this function may also return *cudaErrorInitializationError*, *cudaErrorInsufficientDriver* or *cudaErrorNoDevice* if this call tries to initialize internal CUDA RT state.

Note: Note that as specified by *cudaStreamAddCallback* no CUDA function may be called from callback. *cudaErrorNotPermitted* may, but is not guaranteed to, be returned as a diagnostic in such case.

Parameters

- **hNode** – - Node to set the event for

- **event** – - Event to use

Returns
cudaSuccess, *cudaErrorInvalidValue*

__host__ *cudaError_t* **cudaGraphEventWaitNodeGetEvent**(*cudaGraphNode_t* node, *cudaEvent_t* *event_out)

Returns the event associated with an event wait node.

Returns the event of event wait node `hNode` in `event_out`.

See also:

cudaGraphAddEventWaitNode, *cudaGraphEventWaitNodeSetEvent*, *cudaGraphEventRecordNodeGetEvent*, *cudaEventRecordWithFlags*, *cudaStreamWaitEvent*

Warning: This function will be deprecated in a future CUDA release. Please migrate to *cudaGraphNodeGetParams*.

Note: Graph objects are not threadsafe. More here.

Note: Note that this function may also return error codes from previous, asynchronous launches.

Note: Note that this function may also return *cudaErrorInitializationError*, *cudaErrorInsufficientDriver* or *cudaErrorNoDevice* if this call tries to initialize internal CUDA RT state.

Note: Note that as specified by *cudaStreamAddCallback* no CUDA function may be called from callback. *cudaErrorNotPermitted* may, but is not guaranteed to, be returned as a diagnostic in such case.

Parameters

- **hNode** – - Node to get the event for
- **event_out** – - Pointer to return the event

Returns
cudaSuccess, *cudaErrorInvalidValue*

__host__ *cudaError_t* **cudaGraphEventWaitNodeSetEvent**(*cudaGraphNode_t* node, *cudaEvent_t* event)

Sets an event wait node's event.

Sets the event of event wait node `hNode` to `event`.

See also:

cudaGraphNodeSetParams, *cudaGraphAddEventWaitNode*, *cudaGraphEventWaitNodeGetEvent*, *cudaGraphEventRecordNodeSetEvent*, *cudaEventRecordWithFlags*, *cudaStreamWaitEvent*

Warning: This function will be deprecated in a future CUDA release. Please migrate to *cudaGraphNodeSetParams*.

Note: Graph objects are not threadsafe. More here.

Note: Note that this function may also return error codes from previous, asynchronous launches.

Note: Note that this function may also return *cudaErrorInitializationError*, *cudaErrorInsufficientDriver* or *cudaErrorNoDevice* if this call tries to initialize internal CUDA RT state.

Note: Note that as specified by *cudaStreamAddCallback* no CUDA function may be called from callback. *cudaErrorNotPermitted* may, but is not guaranteed to, be returned as a diagnostic in such case.

Parameters

- **hNode** – - Node to set the event for
- **event** – - Event to use

Returns

cudaSuccess, *cudaErrorInvalidValue*

__host__ *cudaError_t* **cudaGraphExecChildGraphNodeSetParams**(*cudaGraphExec_t* hGraphExec, *cudaGraphNode_t* node, *cudaGraph_t* childGraph)

Updates node parameters in the child graph node in the given graphExec.

Updates the work represented by `node` in `hGraphExec` as though the nodes contained in `node`'s graph had the parameters contained in `childGraph`'s nodes at instantiation. `node` must remain in the graph which was used to instantiate `hGraphExec`. Changed edges to and from `node` are ignored.

The modifications only affect future launches of `hGraphExec`. Already enqueued or running launches of `hGraphExec` are not affected by this call. `node` is also not modified by this call.

The topology of `childGraph`, as well as the node insertion order, must match that of the graph contained in `node`. See *cudaGraphExecUpdate()* for a list of restrictions on what can be updated in an instantiated graph. The update is recursive, so child graph nodes contained within the top level child graph will also be updated.

See also:

cudaGraphExecNodeSetParams, *cudaGraphAddChildGraphNode*, *cudaGraphChildGraphNodeGetGraph*, *cudaGraphExecKernelNodeSetParams*, *cudaGraphExecMemcpyNodeSetParams*, *cudaGraphExecMemsetNodeSetParams*, *cudaGraphExecHostNodeSetParams*, *cudaGraphExecEventRecordNodeSetEvent*, *cudaGraphExecEventWaitNodeSetEvent*, *cudaGraphExecExternalSemaphoresSignalNodeSetParams*, *cudaGraphExecExternalSemaphoresWaitNodeSetParams*, *cudaGraphExecUpdate*, *cudaGraphInstantiate*

Warning: This function will be deprecated in a future CUDA release. Please migrate to *cudaGraphExecNodeSetParams*.

Note: Graph objects are not threadsafe. More here.

Note: Note that this function may also return error codes from previous, asynchronous launches.

Note: Note that this function may also return *cudaErrorInitializationError*, *cudaErrorInsufficientDriver* or *cudaErrorNoDevice* if this call tries to initialize internal CUDA RT state.

Note: Note that as specified by *cudaStreamAddCallback* no CUDA function may be called from callback. *cudaErrorNotPermitted* may, but is not guaranteed to, be returned as a diagnostic in such case.

Parameters

- **hGraphExec** – - The executable graph in which to set the specified node
- **node** – - Host node from the graph which was used to instantiate graphExec
- **childGraph** – - The graph supplying the updated parameters

Returns

cudaSuccess, *cudaErrorInvalidValue*,

__host__ *cudaError_t* **cudaGraphExecDestroy**(*cudaGraphExec_t* graphExec)

Destroys an executable graph.

Destroys the executable graph specified by `graphExec`.

See also:

cudaGraphInstantiate, *cudaGraphUpload*, *cudaGraphLaunch*

Note: Graph objects are not threadsafe. More here.

Note: Note that this function may also return error codes from previous, asynchronous launches.

Note: Note that this function may also return *cudaErrorInitializationError*, *cudaErrorInsufficientDriver* or *cudaErrorNoDevice* if this call tries to initialize internal CUDA RT state.

Note: Note that as specified by *cudaStreamAddCallback* no CUDA function may be called from callback. *cudaErrorNotPermitted* may, but is not guaranteed to, be returned as a diagnostic in such case.

Note: Use of the handle after this call is undefined behavior.

Parameters

graphExec – - Executable graph to destroy

Returns

cudaSuccess, *cudaErrorInvalidValue*

__host__ *cudaError_t* **cudaGraphExecEventRecordNodeSetEvent**(*cudaGraphExec_t* hGraphExec, *cudaGraphNode_t* hNode, *cudaEvent_t* event)

Sets the event for an event record node in the given graphExec.

Sets the event of an event record node in an executable graph `hGraphExec`. The node is identified by the corresponding node `hNode` in the non-executable graph, from which the executable graph was instantiated.

The modifications only affect future launches of `hGraphExec`. Already enqueued or running launches of `hGraphExec` are not affected by this call. `hNode` is also not modified by this call.

See also:

cudaGraphExecNodeSetParams, *cudaGraphAddEventRecordNode*, *cudaGraphEventRecordNodeGetEvent*, *cudaGraphEventWaitNodeSetEvent*, *cudaEventRecordWithFlags*, *cudaStreamWaitEvent*, *cudaGraphExecKernelNodeSetParams*, *cudaGraphExecMemcpyNodeSetParams*, *cudaGraphExecMemsetNodeSetParams*, *cudaGraphExecHostNodeSetParams*, *cudaGraphExecChildGraphNodeSetParams*, *cudaGraphExecEventWaitNodeSetEvent*, *cudaGraphExecExternalSemaphoresSignalNodeSetParams*, *cudaGraphExecExternalSemaphoresWaitNodeSetParams*, *cudaGraphExecUpdate*, *cudaGraphInstantiate*

Warning: This function will be deprecated in a future CUDA release. Please migrate to *cudaGraphExecNodeSetParams*.

Note: Graph objects are not threadsafe. More here.

Note: Note that this function may also return error codes from previous, asynchronous launches.

Note: Note that this function may also return *cudaErrorInitializationError*, *cudaErrorInsufficientDriver* or *cudaErrorNoDevice* if this call tries to initialize internal CUDA RT state.

Note: Note that as specified by *cudaStreamAddCallback* no CUDA function may be called from callback. *cudaErrorNotPermitted* may, but is not guaranteed to, be returned as a diagnostic in such case.

Parameters

- **hGraphExec** – - The executable graph in which to set the specified node
- **hNode** – - Event record node from the graph from which graphExec was instantiated
- **event** – - Updated event to use

Returns
cudaSuccess, *cudaErrorInvalidValue*,

__host__ *cudaError_t* **cudaGraphExecEventWaitNodeSetEvent**(*cudaGraphExec_t* hGraphExec, *cudaGraphNode_t* hNode, *cudaEvent_t* event)

Sets the event for an event wait node in the given graphExec.

Sets the event of an event wait node in an executable graph `hGraphExec`. The node is identified by the corresponding node `hNode` in the non-executable graph, from which the executable graph was instantiated.

The modifications only affect future launches of `hGraphExec`. Already enqueued or running launches of `hGraphExec` are not affected by this call. `hNode` is also not modified by this call.

See also:

cudaGraphExecNodeSetParams, *cudaGraphAddEventWaitNode*, *cudaGraphEventWaitNodeGetEvent*, *cudaGraphEventRecordNodeSetEvent*, *cudaEventRecordWithFlags*, *cudaStreamWaitEvent*, *cudaGraphExecKernelNodeSetParams*, *cudaGraphExecMemcpyNodeSetParams*, *cudaGraphExecMemsetNodeSetParams*, *cudaGraphExecHostNodeSetParams*, *cudaGraphExecChildGraphNodeSetParams*, *cudaGraphExecEventRecordNodeSetEvent*, *cudaGraphExecExternalSemaphoresSignalNodeSetParams*, *cudaGraphExecExternalSemaphoresWaitNodeSetParams*, *cudaGraphExecUpdate*, *cudaGraphInstantiate*

Warning: This function will be deprecated in a future CUDA release. Please migrate to *cudaGraphExecNodeSetParams*.

Note: Graph objects are not threadsafe. More here.

Note: Note that this function may also return error codes from previous, asynchronous launches.

Note: Note that this function may also return *cudaErrorInitializationError*, *cudaErrorInsufficientDriver* or *cudaErrorNoDevice* if this call tries to initialize internal CUDA RT state.

Note: Note that as specified by *cudaStreamAddCallback* no CUDA function may be called from callback. *cudaErrorNotPermitted* may, but is not guaranteed to, be returned as a diagnostic in such case.

Parameters

- **hGraphExec** – - The executable graph in which to set the specified node
- **hNode** – - Event wait node from the graph from which graphExec was instantiated
- **event** – - Updated event to use

Returns
cudaSuccess, *cudaErrorInvalidValue*,

__host__ *cudaError_t* **cudaGraphExecExternalSemaphoresSignalNodeSetParams**(*cudaGraphExec_t* hGraphExec, *cudaGraphNode_t* hNode, const struct *cudaExternalSemaphoreSignalNodeParams* *nodeParams)

Sets the parameters for an external semaphore signal node in the given graphExec.

Sets the parameters of an external semaphore signal node in an executable graph `hGraphExec`. The node is identified by the corresponding node `hNode` in the non-executable graph, from which the executable graph was instantiated.

`hNode` must not have been removed from the original graph.

The modifications only affect future launches of `hGraphExec`. Already enqueued or running launches of `hGraphExec` are not affected by this call. `hNode` is also not modified by this call.

Changing `nodeParams->numExtSems` is not supported.

See also:

cudaGraphExecNodeSetParams, *cudaGraphAddExternalSemaphoresSignalNode*, *cudaImportExternalSemaphore*, *cudaSignalExternalSemaphoresAsync*, *cudaWaitExternalSemaphoresAsync*, *cudaGraphExecKernelNodeSetParams*, *cudaGraphExecMemcpyNodeSetParams*, *cudaGraphExecMemsetNodeSetParams*, *cudaGraphExecHostNodeSetParams*, *cudaGraphExecChildGraphNodeSetParams*, *cudaGraphExecEventRecordNodeSetEvent*, *cudaGraphExecEventWaitNodeSetEvent*, *cudaGraphExecExternalSemaphoresWaitNodeSetParams*, *cudaGraphExecUpdate*, *cudaGraphInstantiate*

Warning: This function will be deprecated in a future CUDA release. Please migrate to *cudaGraphExecNodeSetParams*.

Note: Graph objects are not threadsafe. More here.

Note: Note that this function may also return error codes from previous, asynchronous launches.

Note: Note that this function may also return *cudaErrorInitializationError*, *cudaErrorInsufficientDriver* or *cudaErrorNoDevice* if this call tries to initialize internal CUDA RT state.

Note: Note that as specified by *cudaStreamAddCallback* no CUDA function may be called from callback. *cudaErrorNotPermitted* may, but is not guaranteed to, be returned as a diagnostic in such case.

Parameters

- **hGraphExec** – - The executable graph in which to set the specified node
- **hNode** – - semaphore signal node from the graph from which graphExec was instantiated
- **nodeParams** – - Updated Parameters to set

Returns

cudaSuccess, *cudaErrorInvalidValue*,

__host__ *cudaError_t* **cudaGraphExecExternalSemaphoresWaitNodeSetParams**(*cudaGraphExec_t* hGraphExec, *cudaGraphNode_t* hNode, const struct *cudaExternalSemaphoreWaitNodeParams* *nodeParams)

Sets the parameters for an external semaphore wait node in the given graphExec.

Sets the parameters of an external semaphore wait node in an executable graph `hGraphExec`. The node is identified by the corresponding node `hNode` in the non-executable graph, from which the executable graph was instantiated.

`hNode` must not have been removed from the original graph.

The modifications only affect future launches of `hGraphExec`. Already enqueued or running launches of `hGraphExec` are not affected by this call. `hNode` is also not modified by this call.

Changing `nodeParams->numExtSems` is not supported.

See also:

cudaGraphExecNodeSetParams, *cudaGraphAddExternalSemaphoresWaitNode*, *cudaImportExternalSemaphore*, *cudaSignalExternalSemaphoresAsync*, *cudaWaitExternalSemaphoresAsync*, *cudaGraphExecKernelNodeSetParams*, *cudaGraphExecMemcpyNodeSetParams*, *cudaGraphExecMemsetNodeSetParams*, *cudaGraphExecHostNodeSetParams*, *cudaGraphExecChildGraphNodeSetParams*, *cudaGraphExecEventRecordNodeSetEvent*, *cudaGraphExecEventWaitNodeSetEvent*, *cudaGraphExecExternalSemaphoresSignalNodeSetParams*, *cudaGraphExecUpdate*, *cudaGraphInstantiate*

Warning: This function will be deprecated in a future CUDA release. Please migrate to *cudaGraphExecNodeSetParams*.

Note: Graph objects are not threadsafe. More here.

Note: Note that this function may also return error codes from previous, asynchronous launches.

Note: Note that this function may also return *cudaErrorInitializationError*, *cudaErrorInsufficientDriver* or *cudaErrorNoDevice* if this call tries to initialize internal CUDA RT state.

Note: Note that as specified by *cudaStreamAddCallback* no CUDA function may be called from callback. *cudaErrorNotPermitted* may, but is not guaranteed to, be returned as a diagnostic in such case.

Parameters

- **`hGraphExec`** – - The executable graph in which to set the specified node
- **`hNode`** – - semaphore wait node from the graph from which graphExec was instantiated
- **`nodeParams`** – - Updated Parameters to set

Returns

cudaSuccess, *cudaErrorInvalidValue*,

__host__ *cudaError_t* **cudaGraphExecGetFlags**(*cudaGraphExec_t* graphExec, unsigned long long *flags)

Query the instantiation flags of an executable graph.

Returns the flags that were passed to instantiation for the given executable graph. *cudaGraphInstantiateFlagUpload* will not be returned by this API as it does not affect the resulting executable graph.

See also:

cudaGraphInstantiate, *cudaGraphInstantiateWithFlags*, *cudaGraphInstantiateWithParams*

Note: Graph objects are not threadsafe. More here.

Note: Note that this function may also return error codes from previous, asynchronous launches.

Note: Note that this function may also return *cudaErrorInitializationError*, *cudaErrorInsufficientDriver* or *cudaErrorNoDevice* if this call tries to initialize internal CUDA RT state.

Note: Note that as specified by *cudaStreamAddCallback* no CUDA function may be called from callback. *cudaErrorNotPermitted* may, but is not guaranteed to, be returned as a diagnostic in such case.

Parameters

- **graphExec** – - The executable graph to query
- **flags** – - Returns the instantiation flags

Returns

cudaSuccess, *cudaErrorInvalidValue*

__host__ *cudaError_t* **cudaGraphExecGetId**(*cudaGraphExec_t* hGraphExec, unsigned int *graphID)

Returns the id of a given graph exec.

Returns the id of `hGraphExec` in `*graphId`. The value in `*graphId` matches that referenced by *cudaGraphDebugDotPrint*.

See also:

cudaGraphGetNodes, *cudaGraphDebugDotPrint* *cudaGraphNodeGetContainingGraph* *cudaGraphNodeGetLocalId* *cudaGraphNodeGetToolsId* *cudaGraphGetId*

Parameters

- **hGraphExec** – - Graph to query
- **graphId** – - Pointer to return the graphId

Returns

cudaSuccess, *cudaErrorInvalidValue*

__host__ *cudaError_t* **cudaGraphExecHostNodeSetParams**(*cudaGraphExec_t* hGraphExec, *cudaGraphNode_t* node, const struct *cudaHostNodeParams* *pNodeParams)

Sets the parameters for a host node in the given graphExec.

Updates the work represented by `node` in `hGraphExec` as though `node` had contained `pNodeParams` at instantiation. `node` must remain in the graph which was used to instantiate `hGraphExec`. Changed edges to and from `node` are ignored.

The modifications only affect future launches of `hGraphExec`. Already enqueued or running launches of `hGraphExec` are not affected by this call. `node` is also not modified by this call.

See also:

cudaGraphExecNodeSetParams, *cudaGraphAddHostNode*, *cudaGraphHostNodeSetParams*, *cudaGraphExecKernelNodeSetParams*, *cudaGraphExecMemcpyNodeSetParams*, *cudaGraphExecMemsetNodeSetParams*, *cudaGraphExecChildGraphNodeSetParams*, *cudaGraphExecEventRecordNodeSetEvent*, *cudaGraphExecEventWaitNodeSetEvent*, *cudaGraphExecExternalSemaphoresSignalNodeSetParams*, *cudaGraphExecExternalSemaphoresWaitNodeSetParams*, *cudaGraphExecUpdate*, *cudaGraphInstantiate*

Warning: This function will be deprecated in a future CUDA release. Please migrate to *cudaGraphExecNodeSetParams*.

Note: Graph objects are not threadsafe. More here.

Note: Note that this function may also return error codes from previous, asynchronous launches.

Note: Note that this function may also return *cudaErrorInitializationError*, *cudaErrorInsufficientDriver* or *cudaErrorNoDevice* if this call tries to initialize internal CUDA RT state.

Note: Note that as specified by *cudaStreamAddCallback* no CUDA function may be called from callback. *cudaErrorNotPermitted* may, but is not guaranteed to, be returned as a diagnostic in such case.

Parameters

- **hGraphExec** – - The executable graph in which to set the specified node
- **node** – - Host node from the graph which was used to instantiate graphExec
- **pNodeParams** – - Updated Parameters to set

Returns
cudaSuccess, *cudaErrorInvalidValue*,

__host__ *cudaError_t* **cudaGraphExecKernelNodeSetParams**(*cudaGraphExec_t* hGraphExec, *cudaGraphNode_t* node, const struct *cudaKernelNodeParams* *pNodeParams)

Sets the parameters for a kernel node in the given graphExec.

Sets the parameters of a kernel node in an executable graph `hGraphExec`. The node is identified by the corresponding node `node` in the non-executable graph, from which the executable graph was instantiated.

`node` must not have been removed from the original graph. All `nodeParams` fields may change, but the following restrictions apply to `func` updates:

- The owning device of the function cannot change.
- A node whose function originally did not use CUDA dynamic parallelism cannot be updated to a function which uses CDP
- A node whose function originally did not make device-side update calls cannot be updated to a function which makes device-side update calls.
- If `hGraphExec` was not instantiated for device launch, a node whose function originally did not use device-side *cudaGraphLaunch()* cannot be updated to a function which uses device-side *cudaGraphLaunch()* unless the node resides on the same device as nodes which contained such calls at instantiate-time. If no such calls were present at instantiation, these updates cannot be performed at all.

The modifications only affect future launches of `hGraphExec`. Already enqueued or running launches of `hGraphExec` are not affected by this call. `node` is also not modified by this call.

If `node` is a device-updatable kernel node, the next upload/launch of `hGraphExec` will overwrite any previous device-side updates. Additionally, applying host updates to a device-updatable kernel node while it is being updated from the device will result in undefined behavior.

See also:

cudaGraphExecNodeSetParams, *cudaGraphAddKernelNode*, *cudaGraphKernelNodeSetParams*, *cudaGraphExecMemcpyNodeSetParams*, *cudaGraphExecMemsetNodeSetParams*, *cudaGraphExecHostNodeSetParams*, *cudaGraphExecChildGraphNodeSetParams*, *cudaGraphExecEventRecordNodeSetEvent*, *cudaGraphExecEventWaitNodeSetEvent*, *cudaGraphExecExternalSemaphoresSignalNodeSetParams*, *cudaGraphExecExternalSemaphoresWaitNodeSetParams*, *cudaGraphExecUpdate*, *cudaGraphInstantiate*

Warning: This function will be deprecated in a future CUDA release. Please migrate to *cudaGraphExecNodeSetParams*.

Note: Graph objects are not threadsafe. More here.

Note: Note that this function may also return error codes from previous, asynchronous launches.

Note: Note that this function may also return *cudaErrorInitializationError*, *cudaErrorInsufficientDriver* or *cudaErrorNoDevice* if this call tries to initialize internal CUDA RT state.

Note: Note that as specified by *cudaStreamAddCallback* no CUDA function may be called from callback. *cudaErrorNotPermitted* may, but is not guaranteed to, be returned as a diagnostic in such case.

Note: The API can also be used with a kernel *cudaKernel_t* by querying the handle using *cudaLibraryGetKernel()* or *cudaGetKernel* and then passing it to the API by casting to void*. The symbol `entryFuncAddr` passed to *cudaGetKernel* should be a symbol that is registered with the same CUDA Runtime instance.

Note: Passing a symbol that belongs that belongs to a different runtime instance will result in undefined behavior. The only type that can be reliably passed to a different runtime instance is *cudaKernel_t*

Parameters

- **hGraphExec** – - The executable graph in which to set the specified node
- **node** – - kernel node from the graph from which graphExec was instantiated
- **pNodeParams** – - Updated Parameters to set

Returns
cudaSuccess, *cudaErrorInvalidValue*,

__host__ *cudaError_t* **cudaGraphExecMemcpyNodeSetParams**(*cudaGraphExec_t* hGraphExec, *cudaGraphNode_t* node, const struct *cudaMemcpy3DParms* *pNodeParams)

Sets the parameters for a memcpy node in the given graphExec.

Updates the work represented by `node` in `hGraphExec` as though `node` had contained `pNodeParams` at instantiation. `node` must remain in the graph which was used to instantiate `hGraphExec`. Changed edges to and from `node` are ignored.

The source and destination memory in `pNodeParams` must be allocated from the same contexts as the original source and destination memory. Both the instantiation-time memory operands and the memory operands in `pNodeParams` must be 1-dimensional. Zero-length operations are not supported.

The modifications only affect future launches of `hGraphExec`. Already enqueued or running launches of `hGraphExec` are not affected by this call. `node` is also not modified by this call.

Returns *cudaErrorInvalidValue* if the memory operands' mappings changed or either the original or new memory operands are multidimensional.

See also:

cudaGraphExecNodeSetParams, *cudaGraphAddMemcpyNode*, *cudaGraphMemcpyNodeSetParams*, *cudaGraphExecMemcpyNodeSetParamsToSymbol*, *cudaGraphExecMemcpyNodeSetParamsFromSymbol*, *cudaGraphExecMemcpyNodeSetParams1D*, *cudaGraphExecKernelNodeSetParams*, *cudaGraphExecMemsetNodeSetParams*, *cudaGraphExecHostNodeSetParams*, *cudaGraphExecChildGraphNodeSetParams*, *cudaGraphExecEventRecordNodeSetEvent*, *cudaGraphExecEventWaitNodeSetEvent*, *cudaGraphExecExternalSemaphoresSignalNodeSetParams*, *cudaGraphExecExternalSemaphoresWaitNodeSetParams*, *cudaGraphExecUpdate*, *cudaGraphInstantiate*

Warning: This function will be deprecated in a future CUDA release. Please migrate to *cudaGraphExecNodeSetParams*.

Note: Graph objects are not threadsafe. More here.

Note: Note that this function may also return error codes from previous, asynchronous launches.

Note: Note that this function may also return *cudaErrorInitializationError*, *cudaErrorInsufficientDriver* or *cudaErrorNoDevice* if this call tries to initialize internal CUDA RT state.

Note: Note that as specified by *cudaStreamAddCallback* no CUDA function may be called from callback. *cudaErrorNotPermitted* may, but is not guaranteed to, be returned as a diagnostic in such case.

Parameters

- **hGraphExec** – - The executable graph in which to set the specified node
- **node** – - Memcpy node from the graph which was used to instantiate graphExec
- **pNodeParams** – - Updated Parameters to set

Returns

cudaSuccess, *cudaErrorInvalidValue*,

__host__ *cudaError_t* **cudaGraphExecMemcpyNodeSetParams1D**(*cudaGraphExec_t* hGraphExec, *cudaGraphNode_t* node, void *dst, const void *src, size_t count, enum *cudaMemcpyKind* kind)

Sets the parameters for a memcpy node in the given graphExec to perform a 1-dimensional copy.

Updates the work represented by `node` in `hGraphExec` as though `node` had contained the given params at instantiation. `node` must remain in the graph which was used to instantiate `hGraphExec`. Changed edges to and from `node` are ignored.

`src` and `dst` must be allocated from the same contexts as the original source and destination memory. The instantiation-time memory operands must be 1-dimensional. Zero-length operations are not supported.

The modifications only affect future launches of `hGraphExec`. Already enqueued or running launches of `hGraphExec` are not affected by this call. `node` is also not modified by this call.

Returns *cudaErrorInvalidValue* if the memory operands' mappings changed or the original memory operands are multidimensional.

See also:

cudaGraphAddMemcpyNode, *cudaGraphAddMemcpyNode1D*, *cudaGraphMemcpyNodeSetParams*, *cudaGraphMemcpyNodeSetParams1D*, *cudaGraphExecMemcpyNodeSetParams*, *cudaGraphExecKernelNodeSetParams*, *cudaGraphExecMemsetNodeSetParams*, *cudaGraphExecHostNodeSetParams*, *cudaGraphExecChildGraphNodeSetParams*, *cudaGraphExecEventRecordNodeSetEvent*, *cudaGraphExecEventWaitNodeSetEvent*, *cudaGraphExecExternalSemaphoresSignalNodeSetParams*, *cudaGraphExecExternalSemaphoresWaitNodeSetParams*, *cudaGraphExecUpdate*, *cudaGraphInstantiate*

Note: Graph objects are not threadsafe. More here.

Note: Note that this function may also return error codes from previous, asynchronous launches.

Note: Note that this function may also return *cudaErrorInitializationError*, *cudaErrorInsufficientDriver* or *cudaErrorNoDevice* if this call tries to initialize internal CUDA RT state.

Note: Note that as specified by *cudaStreamAddCallback* no CUDA function may be called from callback. *cudaErrorNotPermitted* may, but is not guaranteed to, be returned as a diagnostic in such case.

Parameters

- **`hGraphExec`** – - The executable graph in which to set the specified node
- **`node`** – - Memcpy node from the graph which was used to instantiate graphExec
- **`dst`** – - Destination memory address
- **`src`** – - Source memory address
- **`count`** – - Size in bytes to copy
- **`kind`** – - Type of transfer

Returns

cudaSuccess, *cudaErrorInvalidValue*

__host__ *cudaError_t* **cudaGraphExecMemcpyNodeSetParamsFromSymbol**(*cudaGraphExec_t* hGraphExec, *cudaGraphNode_t* node, void *dst, const void *symbol, size_t count, size_t offset, enum *cudaMemcpyKind* kind)

Sets the parameters for a memcpy node in the given graphExec to copy from a symbol on the device.

Updates the work represented by `node` in `hGraphExec` as though `node` had contained the given params at instantiation. `node` must remain in the graph which was used to instantiate `hGraphExec`. Changed edges to and from `node` are ignored.

`symbol` and `dst` must be allocated from the same contexts as the original source and destination memory. The instantiation-time memory operands must be 1-dimensional. Zero-length operations are not supported.

The modifications only affect future launches of `hGraphExec`. Already enqueued or running launches of `hGraphExec` are not affected by this call. `node` is also not modified by this call.

Returns *cudaErrorInvalidValue* if the memory operands' mappings changed or the original memory operands are multidimensional.

See also:

cudaGraphAddMemcpyNode, *cudaGraphAddMemcpyNodeFromSymbol*, *cudaGraphMemcpyNodeSetParams*, *cudaGraphMemcpyNodeSetParamsFromSymbol*, *cudaGraphExecMemcpyNodeSetParams*, *cudaGraphExecMemcpyNodeSetParamsToSymbol*, *cudaGraphExecKernelNodeSetParams*, *cudaGraphExecMemsetNodeSetParams*, *cudaGraphExecHostNodeSetParams*, *cudaGraphExecChildGraphNodeSetParams*, *cudaGraphExecEventRecordNodeSetEvent*, *cudaGraphExecEventWaitNodeSetEvent*, *cudaGraphExecExternalSemaphoresSignalNodeSetParams*, *cudaGraphExecExternalSemaphoresWaitNodeSetParams*, *cudaGraphExecUpdate*, *cudaGraphInstantiate*

Note: Graph objects are not threadsafe. More here.

Note: Note that this function may also return error codes from previous, asynchronous launches.

Note: Note that this function may also return *cudaErrorInitializationError*, *cudaErrorInsufficientDriver* or *cudaErrorNoDevice* if this call tries to initialize internal CUDA RT state.

Note: Note that as specified by *cudaStreamAddCallback* no CUDA function may be called from callback. *cudaErrorNotPermitted* may, but is not guaranteed to, be returned as a diagnostic in such case.

Parameters

- **hGraphExec** – - The executable graph in which to set the specified node

- **node** – - Memcpy node from the graph which was used to instantiate graphExec
- **dst** – - Destination memory address
- **symbol** – - Device symbol address
- **count** – - Size in bytes to copy
- **offset** – - Offset from start of symbol in bytes
- **kind** – - Type of transfer

Returns
cudaSuccess, *cudaErrorInvalidValue*

__host__ *cudaError_t* **cudaGraphExecMemcpyNodeSetParamsToSymbol**(*cudaGraphExec_t* hGraphExec, *cudaGraphNode_t* node, const void *symbol, const void *src, size_t count, size_t offset, enum *cudaMemcpyKind* kind)

Sets the parameters for a memcpy node in the given graphExec to copy to a symbol on the device.

Updates the work represented by `node` in `hGraphExec` as though `node` had contained the given params at instantiation. `node` must remain in the graph which was used to instantiate `hGraphExec`. Changed edges to and from `node` are ignored.

`src` and `symbol` must be allocated from the same contexts as the original source and destination memory. The instantiation-time memory operands must be 1-dimensional. Zero-length operations are not supported.

The modifications only affect future launches of `hGraphExec`. Already enqueued or running launches of `hGraphExec` are not affected by this call. `node` is also not modified by this call.

Returns *cudaErrorInvalidValue* if the memory operands' mappings changed or the original memory operands are multidimensional.

See also:

cudaGraphAddMemcpyNode, *cudaGraphAddMemcpyNodeToSymbol*, *cudaGraphMemcpyNodeSetParams*, *cudaGraphMemcpyNodeSetParamsToSymbol*, *cudaGraphExecMemcpyNodeSetParams*, *cudaGraphExecMemcpyNodeSetParamsFromSymbol*, *cudaGraphExecKernelNodeSetParams*, *cudaGraphExecMemsetNodeSetParams*, *cudaGraphExecHostNodeSetParams*, *cudaGraphExecChildGraphNodeSetParams*, *cudaGraphExecEventRecordNodeSetEvent*, *cudaGraphExecEventWaitNodeSetEvent*, *cudaGraphExecExternalSemaphoresSignalNodeSetParams*, *cudaGraphExecExternalSemaphoresWaitNodeSetParams*, *cudaGraphExecUpdate*, *cudaGraphInstantiate*

Note: Graph objects are not threadsafe. More here.

Note: Note that this function may also return error codes from previous, asynchronous launches.

Note: Note that this function may also return *cudaErrorInitializationError*, *cudaErrorInsufficientDriver* or *cudaErrorNoDevice* if this call tries to initialize internal CUDA RT state.

Note: Note that as specified by *cudaStreamAddCallback* no CUDA function may be called from callback. *cudaErrorNotPermitted* may, but is not guaranteed to, be returned as a diagnostic in such case.

Parameters

- **hGraphExec** – - The executable graph in which to set the specified node
- **node** – - Memcpy node from the graph which was used to instantiate graphExec
- **symbol** – - Device symbol address
- **src** – - Source memory address
- **count** – - Size in bytes to copy
- **offset** – - Offset from start of symbol in bytes
- **kind** – - Type of transfer

Returns

cudaSuccess, *cudaErrorInvalidValue*

__host__ *cudaError_t* **cudaGraphExecMemsetNodeSetParams**(*cudaGraphExec_t* hGraphExec, *cudaGraphNode_t* node, const struct *cudaMemsetParams* *pNodeParams)

Sets the parameters for a memset node in the given graphExec.

Updates the work represented by `node` in `hGraphExec` as though `node` had contained `pNodeParams` at instantiation. `node` must remain in the graph which was used to instantiate `hGraphExec`. Changed edges to and from `node` are ignored.

Zero sized operations are not supported.

The new destination pointer in `pNodeParams` must be to the same kind of allocation as the original destination pointer and have the same context association and device mapping as the original destination pointer.

Both the value and pointer address may be updated.

Changing other aspects of the memset (width, height, element size or pitch) may cause the update to be rejected. Specifically, for 2d memsets, all dimension changes are rejected. For 1d memsets, changes in height are explicitly rejected and other changes are opportunistically allowed if the resulting work maps onto the work resources already allocated for the node.

The modifications only affect future launches of `hGraphExec`. Already enqueued or running launches of `hGraphExec` are not affected by this call. `node` is also not modified by this call.

See also:

cudaGraphExecNodeSetParams, *cudaGraphAddMemsetNode*, *cudaGraphMemsetNodeSetParams*, *cudaGraphExecKernelNodeSetParams*, *cudaGraphExecMemcpyNodeSetParams*, *cudaGraphExecHostNodeSetParams*, *cudaGraphExecChildGraphNodeSetParams*, *cudaGraphExecEventRecordNodeSetEvent*, *cudaGraphExecEventWaitNodeSetEvent*, *cudaGraphExecExter-*

nalSemaphoresSignalNodeSetParams, *cudaGraphExecExternalSemaphoresWaitNodeSetParams*, *cudaGraphExecUpdate*, *cudaGraphInstantiate*

Warning: This function will be deprecated in a future CUDA release. Please migrate to *cudaGraphExecNodeSetParams*.

Note: Graph objects are not threadsafe. More here.

Note: Note that this function may also return error codes from previous, asynchronous launches.

Note: Note that this function may also return *cudaErrorInitializationError*, *cudaErrorInsufficientDriver* or *cudaErrorNoDevice* if this call tries to initialize internal CUDA RT state.

Note: Note that as specified by *cudaStreamAddCallback* no CUDA function may be called from callback. *cudaErrorNotPermitted* may, but is not guaranteed to, be returned as a diagnostic in such case.

Parameters

- **hGraphExec** – - The executable graph in which to set the specified node
- **node** – - Memset node from the graph which was used to instantiate graphExec
- **pNodeParams** – - Updated Parameters to set

Returns
cudaSuccess, *cudaErrorInvalidValue*,

__host__ *cudaError_t* **cudaGraphExecNodeSetParams**(*cudaGraphExec_t* graphExec, *cudaGraphNode_t* node, struct *cudaGraphNodeParams* *nodeParams)

Update a graph node's parameters in an instantiated graph.

Sets the parameters of a node in an executable graph `graphExec`. The node is identified by the corresponding node `node` in the non-executable graph from which the executable graph was instantiated. `node` must not have been removed from the original graph.

The modifications only affect future launches of `graphExec`. Already enqueued or running launches of `graphExec` are not affected by this call. `node` is also not modified by this call.

Allowed changes to parameters on executable graphs are as follows:

Node type	Allowed changes
kernel	See *cudaGraphExecKernelNodeSetParams*
memcpy	Addresses for 1-dimensional copies if allocated in same context; see *cudaGraphExecMemcpyNodeSetParams*
memset	Addresses for 1-dimensional memsets if allocated in same context; see *cudaGraphExecMemsetNodeSetParams*
host	Unrestricted
child graph	Topology must match and restrictions apply recursively; see *cudaGraphExecUpdate*
event wait	Unrestricted
event record	Unrestricted
external semaphore signal	Number of semaphore operations cannot change
external semaphore wait	Number of semaphore operations cannot change
memory allocation	API unsupported
memory free	API unsupported

See also:

cudaGraphAddNode, *cudaGraphNodeSetParams cudaGraphExecUpdate*, *cudaGraphInstantiate*

Note: Graph objects are not threadsafe. More here.

Note: Note that this function may also return error codes from previous, asynchronous launches.

Note: Note that this function may also return *cudaErrorInitializationError*, *cudaErrorInsufficientDriver* or *cudaErrorNoDevice* if this call tries to initialize internal CUDA RT state.

Note: Note that as specified by *cudaStreamAddCallback* no CUDA function may be called from callback. *cudaErrorNotPermitted* may, but is not guaranteed to, be returned as a diagnostic in such case.

Parameters

- **graphExec** – - The executable graph in which to update the specified node
- **node** – - Corresponding node from the graph from which graphExec was instantiated

- **nodeParams** – - Updated Parameters to set

Returns

cudaSuccess, *cudaErrorInvalidValue*, *cudaErrorInvalidDeviceFunction*, *cudaErrorNotSupported*

__host__ *cudaError_t* **cudaGraphExecUpdate**(*cudaGraphExec_t* hGraphExec, *cudaGraph_t* hGraph, *cudaGraphExecUpdateResultInfo* *resultInfo)

Check whether an executable graph can be updated with a graph and perform the update if possible.

Updates the node parameters in the instantiated graph specified by `hGraphExec` with the node parameters in a topologically identical graph specified by `hGraph`.

Limitations:

- Kernel nodes:
  - The owning context of the function cannot change.
  - A node whose function originally did not use CUDA dynamic parallelism cannot be updated to a function which uses CDP.
  - A node whose function originally did not make device-side update calls cannot be updated to a function which makes device-side update calls.
  - A cooperative node cannot be updated to a non-cooperative node, and vice-versa.
  - If the graph was instantiated with cudaGraphInstantiateFlagUseNodePriority, the priority attribute cannot change. Equality is checked on the originally requested priority values, before they are clamped to the device's supported range.
  - If `hGraphExec` was not instantiated for device launch, a node whose function originally did not use device-side *cudaGraphLaunch()* cannot be updated to a function which uses device-side *cudaGraphLaunch()* unless the node resides on the same device as nodes which contained such calls at instantiate-time. If no such calls were present at instantiation, these updates cannot be performed at all.
  - Neither `hGraph` nor `hGraphExec` may contain device-updatable kernel nodes.
- Memset and memcpy nodes:
  - The CUDA device(s) to which the operand(s) was allocated/mapped cannot change.
  - The source/destination memory must be allocated from the same contexts as the original source/destination memory.
  - For 2d memsets, only address and assigned value may be updated.
  - For 1d memsets, updating dimensions is also allowed, but may fail if the resulting operation doesn't map onto the work resources already allocated for the node.
- Additional memcpy node restrictions:
  - Changing either the source or destination memory type(i.e. CU_MEMORYTYPE_DEVICE, CU_MEMORYTYPE_ARRAY, etc.) is not supported.
- Conditional nodes:
  - Changing node parameters is not supported.
  - Changing parameters of nodes within the conditional body graph is subject to the rules above.

- Conditional handle flags and default values are updated as part of the graph update.

Note: The API may add further restrictions in future releases. The return code should always be checked.

cudaGraphExecUpdate sets the result member of `resultInfo` to cudaGraphExecUpdateErrorTopologyChanged under the following conditions:

- The count of nodes directly in `hGraphExec` and `hGraph` differ, in which case resultInfo->errorNode is set to NULL.
- `hGraph` has more exit nodes than `hGraph`, in which case resultInfo->errorNode is set to one of the exit nodes in hGraph.
- A node in `hGraph` has a different number of dependencies than the node from `hGraphExec` it is paired with, in which case resultInfo->errorNode is set to the node from `hGraph`.
- A node in `hGraph` has a dependency that does not match with the corresponding dependency of the paired node from `hGraphExec`. resultInfo->errorNode will be set to the node from `hGraph`. resultInfo->errorFromNode will be set to the mismatched dependency. The dependencies are paired based on edge order and a dependency does not match when the nodes are already paired based on other edges examined in the graph.

cudaGraphExecUpdate sets `the` result member of `resultInfo` to:

- cudaGraphExecUpdateError if passed an invalid value.
- cudaGraphExecUpdateErrorTopologyChanged if the graph topology changed
- cudaGraphExecUpdateErrorNodeTypeChanged if the type of a node changed, in which case `hErrorNode_out` is set to the node from `hGraph`.
- cudaGraphExecUpdateErrorFunctionChanged if the function of a kernel node changed (CUDA driver < 11.2)
- cudaGraphExecUpdateErrorUnsupportedFunctionChange if the func field of a kernel changed in an unsupported way(see note above), in which case `hErrorNode_out` is set to the node from `hGraph`
- cudaGraphExecUpdateErrorParametersChanged if any parameters to a node changed in a way that is not supported, in which case `hErrorNode_out` is set to the node from `hGraph`
- cudaGraphExecUpdateErrorAttributesChanged if any attributes of a node changed in a way that is not supported, in which case `hErrorNode_out` is set to the node from `hGraph`
- cudaGraphExecUpdateErrorNotSupported if something about a node is unsupported, like the node's type or configuration, in which case `hErrorNode_out` is set to the node from `hGraph`

If the update fails for a reason not listed above, the result member of `resultInfo` will be set to cudaGraphExecUpdateError. If the update succeeds, the result member will be set to cudaGraphExecUpdateSuccess.

cudaGraphExecUpdate returns cudaSuccess when the updated was performed successfully. It returns cudaErrorGraphExecUpdateFailure if the graph update was not performed because it included changes which violated constraints specific to instantiated graph update.

See also:

cudaGraphInstantiate

Note: Graph objects are not threadsafe. More here.

Note: Note that this function may also return error codes from previous, asynchronous launches.

Note: Note that this function may also return *cudaErrorInitializationError*, *cudaErrorInsufficientDriver* or *cudaErrorNoDevice* if this call tries to initialize internal CUDA RT state.

Note: Note that as specified by *cudaStreamAddCallback* no CUDA function may be called from callback. *cudaErrorNotPermitted* may, but is not guaranteed to, be returned as a diagnostic in such case.

Parameters

- **hGraphExec** – The instantiated graph to be updated
- **hGraph** – The graph containing the updated parameters
- **resultInfo** – the error info structure

Returns

cudaSuccess, *cudaErrorGraphExecUpdateFailure*,

__host__ *cudaError_t* **cudaGraphExternalSemaphoresSignalNodeGetParams**(*cudaGraphNode_t* hNode, struct *cudaExternalSemaphoreSignalNodeParams* *params_out)

Returns an external semaphore signal node's parameters.

Returns the parameters of an external semaphore signal node `hNode` in `params_out`. The `extSemArray` and `paramsArray` returned in `params_out`, are owned by the node. This memory remains valid until the node is destroyed or its parameters are modified, and should not be modified directly. Use *cudaGraphExternalSemaphoresSignalNodeSetParams* to update the parameters of this node.

See also:

cudaGraphNodeGetParams, *cudaLaunchKernel*, *cudaGraphAddExternalSemaphoresSignalNode*, *cudaGraphExternalSemaphoresSignalNodeSetParams*, *cudaGraphAddExternalSemaphoresWaitNode*, *cudaSignalExternalSemaphoresAsync*, *cudaWaitExternalSemaphoresAsync*

Warning: This function will be deprecated in a future CUDA release. Please migrate to *cudaGraphNodeGetParams*.

Note: Graph objects are not threadsafe. More here.

Note: Note that this function may also return error codes from previous, asynchronous launches.

Note: Note that this function may also return *cudaErrorInitializationError*, *cudaErrorInsufficientDriver* or *cudaErrorNoDevice* if this call tries to initialize internal CUDA RT state.

Note: Note that as specified by *cudaStreamAddCallback* no CUDA function may be called from callback. *cudaErrorNotPermitted* may, but is not guaranteed to, be returned as a diagnostic in such case.

Parameters

- **hNode** – - Node to get the parameters for
- **params_out** – - Pointer to return the parameters

Returns

cudaSuccess, *cudaErrorInvalidValue*

__host__ *cudaError_t* **cudaGraphExternalSemaphoresSignalNodeSetParams**(*cudaGraphNode_t* hNode, const struct *cudaExternalSemaphoreSignalNodeParams* *nodeParams)

Sets an external semaphore signal node's parameters.

Sets the parameters of an external semaphore signal node `hNode` to `nodeParams`.

See also:

cudaGraphNodeSetParams, *cudaGraphAddExternalSemaphoresSignalNode*, *cudaGraphExternalSemaphoresSignalNodeSetParams*, *cudaGraphAddExternalSemaphoresWaitNode*, *cudaSignalExternalSemaphoresAsync*, *cudaWaitExternalSemaphoresAsync*

Warning: This function will be deprecated in a future CUDA release. Please migrate to *cudaGraphNodeSetParams*.

Note: Graph objects are not threadsafe. More here.

Note: Note that this function may also return error codes from previous, asynchronous launches.

Note: Note that this function may also return *cudaErrorInitializationError*, *cudaErrorInsufficientDriver* or *cudaErrorNoDevice* if this call tries to initialize internal CUDA RT state.

Note: Note that as specified by *cudaStreamAddCallback* no CUDA function may be called from callback. *cudaErrorNotPermitted* may, but is not guaranteed to, be returned as a diagnostic in such case.

Parameters

- **hNode** – - Node to set the parameters for
- **nodeParams** – - Parameters to copy

Returns
cudaSuccess, *cudaErrorInvalidValue*

__host__ *cudaError_t* **cudaGraphExternalSemaphoresWaitNodeGetParams**(*cudaGraphNode_t* hNode, struct *cudaExternalSemaphoreWaitNodeParams* *params_out)

Returns an external semaphore wait node's parameters.

Returns the parameters of an external semaphore wait node `hNode` in `params_out`. The `extSemArray` and `paramsArray` returned in `params_out`, are owned by the node. This memory remains valid until the node is destroyed or its parameters are modified, and should not be modified directly. Use *cudaGraphExternalSemaphoresSignalNodeSetParams* to update the parameters of this node.

See also:

cudaGraphNodeGetParams, *cudaLaunchKernel*, *cudaGraphAddExternalSemaphoresWaitNode*, *cudaGraphExternalSemaphoresWaitNodeSetParams*, *cudaGraphAddExternalSemaphoresWaitNode*, *cudaSignalExternalSemaphoresAsync*, *cudaWaitExternalSemaphoresAsync*

Warning: This function will be deprecated in a future CUDA release. Please migrate to *cudaGraphNodeGetParams*.

Note: Graph objects are not threadsafe. More here.

Note: Note that this function may also return error codes from previous, asynchronous launches.

Note: Note that this function may also return *cudaErrorInitializationError*, *cudaErrorInsufficientDriver* or *cudaErrorNoDevice* if this call tries to initialize internal CUDA RT state.

Note: Note that as specified by *cudaStreamAddCallback* no CUDA function may be called from callback. *cudaErrorNotPermitted* may, but is not guaranteed to, be returned as a diagnostic in such case.

Parameters

- **hNode** – - Node to get the parameters for
- **params_out** – - Pointer to return the parameters

Returns
cudaSuccess, *cudaErrorInvalidValue*

__host__ *cudaError_t* **cudaGraphExternalSemaphoresWaitNodeSetParams**(*cudaGraphNode_t* hNode, const struct *cudaExternalSemaphoreWaitNodeParams* *nodeParams)

Sets an external semaphore wait node's parameters.

Sets the parameters of an external semaphore wait node `hNode` to `nodeParams`.

See also:

cudaGraphNodeSetParams, *cudaGraphAddExternalSemaphoresWaitNode*, *cudaGraphExternalSemaphoresWaitNodeSetParams*, *cudaGraphAddExternalSemaphoresWaitNode*, *cudaSignalExternalSemaphoresAsync*, *cudaWaitExternalSemaphoresAsync*

Warning: This function will be deprecated in a future CUDA release. Please migrate to *cudaGraphNodeSetParams*.

Note: Graph objects are not threadsafe. More here.

Note: Note that this function may also return error codes from previous, asynchronous launches.

Note: Note that this function may also return *cudaErrorInitializationError*, *cudaErrorInsufficientDriver* or *cudaErrorNoDevice* if this call tries to initialize internal CUDA RT state.

Note: Note that as specified by *cudaStreamAddCallback* no CUDA function may be called from callback. *cudaErrorNotPermitted* may, but is not guaranteed to, be returned as a diagnostic in such case.

Parameters

- **hNode** – - Node to set the parameters for
- **nodeParams** – - Parameters to copy

Returns

cudaSuccess, *cudaErrorInvalidValue*

__host__ *cudaError_t* **cudaGraphGetEdges**(*cudaGraph_t* graph, *cudaGraphNode_t* *from, *cudaGraphNode_t* *to, *cudaGraphEdgeData* *edgeData, size_t *numEdges)

Returns a graph's dependency edges.

Returns a list of `graph`'s dependency edges. Edges are returned via corresponding indices in `from`, `to` and `edgeData`; that is, the node in `to`[i] has a dependency on the node in `from`[i] with data `edgeData`[i]. `from` and `to` may both be NULL, in which case this function only returns the number of edges in `numEdges`. Otherwise, `numEdges` entries will be filled in. If `numEdges` is higher than the actual number of edges, the remaining entries in `from` and `to` will be set to NULL, and the number of edges actually returned will be written to `numEdges`. `edgeData` may alone be NULL, in which case the edges must all have default (zeroed) edge data. Attempting a losst query via NULL `edgeData` will result in *cudaErrorLossyQuery*. If `edgeData` is non-NULL then `from` and `to` must be as well.

See also:

cudaGraphGetNodes, *cudaGraphGetRootNodes*, *cudaGraphAddDependencies*, *cudaGraphRemoveDependencies*, *cudaGraphNodeGetDependencies*, *cudaGraphNodeGetDependentNodes*

Note: Graph objects are not threadsafe. More here.

Note: Note that this function may also return error codes from previous, asynchronous launches.

Note: Note that this function may also return *cudaErrorInitializationError*, *cudaErrorInsufficientDriver* or *cudaErrorNoDevice* if this call tries to initialize internal CUDA RT state.

Note: Note that as specified by *cudaStreamAddCallback* no CUDA function may be called from callback. *cudaErrorNotPermitted* may, but is not guaranteed to, be returned as a diagnostic in such case.

Parameters

- **graph** – - Graph to get the edges from
- **from** – - Location to return edge endpoints
- **to** – - Location to return edge endpoints
- **edgeData** – - Optional location to return edge data
- **numEdges** – - See description

Returns
cudaSuccess, *cudaErrorLossyQuery*, *cudaErrorInvalidValue*

__host__ *cudaError_t* **cudaGraphGetId**(*cudaGraph_t* hGraph, unsigned int *graphID)

Returns the id of a given graph.

Returns the id of `hGraph` in `*graphId`. The value in `*graphId` matches that referenced by *cudaGraphDebugDotPrint*.

See also:

cudaGraphGetNodes, *cudaGraphDebugDotPrint* *cudaGraphNodeGetContainingGraph* *cudaGraphNodeGetLocalId* *cudaGraphNodeGetToolsId* *cudaGraphExecGetId*

Parameters

- **hGraph** – - Graph to query
- **graphId** – - Pointer to return the graphId

Returns
cudaSuccess, *cudaErrorInvalidValue*

__host__ *cudaError_t* **cudaGraphGetNodes**(*cudaGraph_t* graph, *cudaGraphNode_t* *nodes, size_t *numNodes)

Returns a graph's nodes.

Returns a list of `graph's` nodes. `nodes` may be NULL, in which case this function will return the number of nodes in `numNodes`. Otherwise, `numNodes` entries will be filled in. If `numNodes` is higher than the actual number of nodes, the remaining entries in `nodes` will be set to NULL, and the number of nodes actually obtained will be returned in `numNodes`.

See also:

cudaGraphCreate, *cudaGraphGetRootNodes*, *cudaGraphGetEdges*, *cudaGraphNodeGetType*, *cudaGraphNodeGetDependencies*, *cudaGraphNodeGetDependentNodes*

Note: Graph objects are not threadsafe. More here.

Note: Note that this function may also return error codes from previous, asynchronous launches.

Note: Note that this function may also return *cudaErrorInitializationError*, *cudaErrorInsufficientDriver* or *cudaErrorNoDevice* if this call tries to initialize internal CUDA RT state.

Note: Note that as specified by *cudaStreamAddCallback* no CUDA function may be called from callback. *cudaErrorNotPermitted* may, but is not guaranteed to, be returned as a diagnostic in such case.

Parameters

- **graph** – - Graph to query
- **nodes** – - Pointer to return the nodes
- **numNodes** – - See description

Returns

cudaSuccess, *cudaErrorInvalidValue*

__host__ *cudaError_t* **cudaGraphGetRootNodes**(*cudaGraph_t* graph, *cudaGraphNode_t* *pRootNodes, size_t *pNumRootNodes)

Returns a graph's root nodes.

Returns a list of `graph's` root nodes. `pRootNodes` may be NULL, in which case this function will return the number of root nodes in `pNumRootNodes`. Otherwise, `pNumRootNodes` entries will be filled in. If `pNumRootNodes` is higher than the actual number of root nodes, the remaining entries in `pRootNodes` will be set to NULL, and the number of nodes actually obtained will be returned in `pNumRootNodes`.

See also:

cudaGraphCreate, *cudaGraphGetNodes*, *cudaGraphGetEdges*, *cudaGraphNodeGetType*, *cudaGraphNodeGetDependencies*, *cudaGraphNodeGetDependentNodes*

Note: Graph objects are not threadsafe. More here.

Note: Note that this function may also return error codes from previous, asynchronous launches.

Note: Note that this function may also return *cudaErrorInitializationError*, *cudaErrorInsufficientDriver* or *cudaErrorNoDevice* if this call tries to initialize internal CUDA RT state.

Note: Note that as specified by *cudaStreamAddCallback* no CUDA function may be called from callback. *cudaErrorNotPermitted* may, but is not guaranteed to, be returned as a diagnostic in such case.

Parameters

- **graph** – - Graph to query
- **pRootNodes** – - Pointer to return the root nodes
- **pNumRootNodes** – - See description

Returns
cudaSuccess, *cudaErrorInvalidValue*

__host__ *cudaError_t* **cudaGraphHostNodeGetParams**(*cudaGraphNode_t* node, struct *cudaHostNodeParams* *pNodeParams)

Returns a host node's parameters.

Returns the parameters of host node `node` in `pNodeParams`.

See also:

cudaGraphNodeGetParams, *cudaLaunchHostFunc*, *cudaGraphAddHostNode*, *cudaGraphHostNodeSetParams*

Warning: This function will be deprecated in a future CUDA release. Please migrate to *cudaGraphNodeGetParams*.

Note: Graph objects are not threadsafe. More here.

Note: Note that this function may also return error codes from previous, asynchronous launches.

Note: Note that this function may also return *cudaErrorInitializationError*, *cudaErrorInsufficientDriver* or *cudaErrorNoDevice* if this call tries to initialize internal CUDA RT state.

Note: Note that as specified by *cudaStreamAddCallback* no CUDA function may be called from callback. *cudaErrorNotPermitted* may, but is not guaranteed to, be returned as a diagnostic in such case.

Parameters

- **node** – - Node to get the parameters for
- **pNodeParams** – - Pointer to return the parameters

Returns
cudaSuccess, *cudaErrorInvalidValue*

__host__ *cudaError_t* **cudaGraphHostNodeSetParams**(*cudaGraphNode_t* node, const struct *cudaHostNodeParams* *pNodeParams)

Sets a host node's parameters.

Sets the parameters of host node `node` to `nodeParams`.

See also:

cudaGraphNodeSetParams, *cudaLaunchHostFunc*, *cudaGraphAddHostNode*, *cudaGraphHostNodeGetParams*

Warning: This function will be deprecated in a future CUDA release. Please migrate to *cudaGraphNodeSetParams*.

Note: Graph objects are not threadsafe. More here.

Note: Note that this function may also return error codes from previous, asynchronous launches.

Note: Note that this function may also return *cudaErrorInitializationError*, *cudaErrorInsufficientDriver* or *cudaErrorNoDevice* if this call tries to initialize internal CUDA RT state.

Note: Note that as specified by *cudaStreamAddCallback* no CUDA function may be called from callback. *cudaErrorNotPermitted* may, but is not guaranteed to, be returned as a diagnostic in such case.

Parameters

- **node** – - Node to set the parameters for
- **pNodeParams** – - Parameters to copy

Returns

cudaSuccess, *cudaErrorInvalidValue*

__host__ *cudaError_t* **cudaGraphInstantiate**(*cudaGraphExec_t* *pGraphExec, *cudaGraph_t* graph, unsigned long long flags = 0)

Creates an executable graph from a graph.

Instantiates `graph` as an executable graph. The graph is validated for any structural constraints or intra-node constraints which were not previously validated. If instantiation is successful, a handle to the instantiated graph is returned in `pGraphExec`.

The `flags` parameter controls the behavior of instantiation and subsequent graph launches. Valid flags are:

- *cudaGraphInstantiateFlagAutoFreeOnLaunch*, which configures a graph containing memory allocation nodes to automatically free any unfreed memory allocations before the graph is relaunched.
- *cudaGraphInstantiateFlagDeviceLaunch*, which configures the graph for launch from the device. If this flag is passed, the executable graph handle returned can be used to launch the graph from both the host and device. This flag cannot be used in conjunction with *cudaGraphInstantiateFlagAutoFreeOnLaunch*.
- *cudaGraphInstantiateFlagUseNodePriority*, which causes the graph to use the priorities from the per-node attributes rather than the priority of the launch stream during execution. Note that priorities are only available on kernel nodes, and are copied from stream priority during stream capture.

If `graph` contains any allocation or free nodes, there can be at most one executable graph in existence for that graph at a time. An attempt to instantiate a second executable graph before destroying the first with *cudaGraphExecDestroy* will result in an error. The same also applies if `graph` contains any device-updatable kernel nodes.

Graphs instantiated for launch on the device have additional restrictions which do not apply to host graphs:

- The graph's nodes must reside on a single device.
- The graph can only contain kernel nodes, memcpy nodes, memset nodes, and child graph nodes.
- The graph cannot be empty and must contain at least one kernel, memcpy, or memset node. Operation-specific restrictions are outlined below.
- Kernel nodes:
  - Use of CUDA Dynamic Parallelism is not permitted.
  - Cooperative launches are permitted as long as MPS is not in use.
- Memcpy nodes:
  - Only copies involving device memory and/or pinned device-mapped host memory are permitted.
  - Copies involving CUDA arrays are not permitted.
  - Both operands must be accessible from the current device, and the current device must match the device of other nodes in the graph.

If `graph` is not instantiated for launch on the device but contains kernels which call device-side *cudaGraphLaunch()* from multiple devices, this will result in an error.

See also:

cudaGraphInstantiateWithFlags, *cudaGraphCreate*, *cudaGraphUpload*, *cudaGraphLaunch*, *cudaGraphExecDestroy*

Note: Graph objects are not threadsafe. More here.

Note: Note that this function may also return error codes from previous, asynchronous launches.

Note: Note that this function may also return *cudaErrorInitializationError*, *cudaErrorInsufficientDriver* or *cudaErrorNoDevice* if this call tries to initialize internal CUDA RT state.

Note: Note that as specified by *cudaStreamAddCallback* no CUDA function may be called from callback. *cudaErrorNotPermitted* may, but is not guaranteed to, be returned as a diagnostic in such case.

Parameters

- **pGraphExec** – - Returns instantiated graph
- **graph** – - Graph to instantiate
- **flags** – - Flags to control instantiation. See ::CUgraphInstantiate_flags.

Returns

cudaSuccess, *cudaErrorInvalidValue*

__host__ *cudaError_t* **cudaGraphInstantiateWithFlags**(*cudaGraphExec_t* *pGraphExec, *cudaGraph_t* graph, unsigned long long flags = 0)

Creates an executable graph from a graph.

Instantiates `graph` as an executable graph. The graph is validated for any structural constraints or intra-node constraints which were not previously validated. If instantiation is successful, a handle to the instantiated graph is returned in `pGraphExec`.

The `flags` parameter controls the behavior of instantiation and subsequent graph launches. Valid flags are:

- *cudaGraphInstantiateFlagAutoFreeOnLaunch*, which configures a graph containing memory allocation nodes to automatically free any unfreed memory allocations before the graph is relaunched.
- *cudaGraphInstantiateFlagDeviceLaunch*, which configures the graph for launch from the device. If this flag is passed, the executable graph handle returned can be used to launch the graph from both the host and device. This flag can only be used on platforms which support unified addressing. This flag cannot be used in conjunction with *cudaGraphInstantiateFlagAutoFreeOnLaunch*.
- *cudaGraphInstantiateFlagUseNodePriority*, which causes the graph to use the priorities from the per-node attributes rather than the priority of the launch stream during execution. Note that priorities are only available on kernel nodes, and are copied from stream priority during stream capture.

If `graph` contains any allocation or free nodes, there can be at most one executable graph in existence for that graph at a time. An attempt to instantiate a second executable graph before destroying the first with *cudaGraphExecDestroy* will result in an error. The same also applies if `graph` contains any device-updatable kernel nodes.

If `graph` contains kernels which call device-side *cudaGraphLaunch()* from multiple devices, this will result in an error.

Graphs instantiated for launch on the device have additional restrictions which do not apply to host graphs:

- The graph's nodes must reside on a single device.
- The graph can only contain kernel nodes, memcpy nodes, memset nodes, and child graph nodes.
- The graph cannot be empty and must contain at least one kernel, memcpy, or memset node. Operation-specific restrictions are outlined below.
- Kernel nodes:
  - Use of CUDA Dynamic Parallelism is not permitted.
  - Cooperative launches are permitted as long as MPS is not in use.
- Memcpy nodes:
  - Only copies involving device memory and/or pinned device-mapped host memory are permitted.
  - Copies involving CUDA arrays are not permitted.
  - Both operands must be accessible from the current device, and the current device must match the device of other nodes in the graph.

See also:

cudaGraphInstantiate, *cudaGraphCreate*, *cudaGraphUpload*, *cudaGraphLaunch*, *cudaGraphExecDestroy*

Note: Graph objects are not threadsafe. More here.

Note: Note that this function may also return error codes from previous, asynchronous launches.

Note: Note that this function may also return *cudaErrorInitializationError*, *cudaErrorInsufficientDriver* or *cudaErrorNoDevice* if this call tries to initialize internal CUDA RT state.

Note: Note that as specified by *cudaStreamAddCallback* no CUDA function may be called from callback. *cudaErrorNotPermitted* may, but is not guaranteed to, be returned as a diagnostic in such case.

Parameters

- **pGraphExec** – - Returns instantiated graph
- **graph** – - Graph to instantiate

- **flags** – - Flags to control instantiation. See ::CUgraphInstantiate_flags.

Returns
cudaSuccess, *cudaErrorInvalidValue*

__host__ *cudaError_t* **cudaGraphInstantiateWithParams**(*cudaGraphExec_t* *pGraphExec, *cudaGraph_t* graph, *cudaGraphInstantiateParams* *instantiateParams)

Creates an executable graph from a graph.

Instantiates `graph` as an executable graph according to the `instantiateParams` structure. The graph is validated for any structural constraints or intra-node constraints which were not previously validated. If instantiation is successful, a handle to the instantiated graph is returned in `pGraphExec`.

`instantiateParams` controls the behavior of instantiation and subsequent graph launches, as well as returning more detailed information in the event of an error. *cudaGraphInstantiateParams* is defined as:

```
typedef struct {
    unsigned long long flags;
    cudaStream_t uploadStream;
    cudaGraphNode_t errNode_out;
    cudaGraphInstantiateResult result_out;
} cudaGraphInstantiateParams;
```

The `flags` field controls the behavior of instantiation and subsequent graph launches. Valid flags are:

- *cudaGraphInstantiateFlagAutoFreeOnLaunch*, which configures a graph containing memory allocation nodes to automatically free any unfreed memory allocations before the graph is relaunched.
- *cudaGraphInstantiateFlagUpload*, which will perform an upload of the graph into `uploadStream` once the graph has been instantiated.
- *cudaGraphInstantiateFlagDeviceLaunch*, which configures the graph for launch from the device. If this flag is passed, the executable graph handle returned can be used to launch the graph from both the host and device. This flag can only be used on platforms which support unified addressing. This flag cannot be used in conjunction with *cudaGraphInstantiateFlagAutoFreeOnLaunch*.
- *cudaGraphInstantiateFlagUseNodePriority*, which causes the graph to use the priorities from the per-node attributes rather than the priority of the launch stream during execution. Note that priorities are only available on kernel nodes, and are copied from stream priority during stream capture.

If `graph` contains any allocation or free nodes, there can be at most one executable graph in existence for that graph at a time. An attempt to instantiate a second executable graph before destroying the first with *cudaGraphExecDestroy* will result in an error. The same also applies if `graph` contains any device-updatable kernel nodes.

If `graph` contains kernels which call device-side *cudaGraphLaunch()* from multiple devices, this will result in an error.

Graphs instantiated for launch on the device have additional restrictions which do not apply to host graphs:

- The graph's nodes must reside on a single device.
- The graph can only contain kernel nodes, memcpy nodes, memset nodes, and child graph nodes.
- The graph cannot be empty and must contain at least one kernel, memcpy, or memset node. Operation-specific restrictions are outlined below.
- Kernel nodes:
  - Use of CUDA Dynamic Parallelism is not permitted.
  - Cooperative launches are permitted as long as MPS is not in use.
- Memcpy nodes:
  - Only copies involving device memory and/or pinned device-mapped host memory are permitted.
  - Copies involving CUDA arrays are not permitted.
  - Both operands must be accessible from the current device, and the current device must match the device of other nodes in the graph.

In the event of an error, the `result_out` and `errNode_out` fields will contain more information about the nature of the error. Possible error reporting includes:

- *cudaGraphInstantiateError*, if passed an invalid value or if an unexpected error occurred which is described by the return value of the function. `errNode_out` will be set to NULL.
- *cudaGraphInstantiateInvalidStructure*, if the graph structure is invalid. `errNode_out` will be set to one of the offending nodes.
- *cudaGraphInstantiateNodeOperationNotSupported*, if the graph is instantiated for device launch but contains a node of an unsupported node type, or a node which performs unsupported operations, such as use of CUDA dynamic parallelism within a kernel node. `errNode_out` will be set to this node.
- *cudaGraphInstantiateMultipleDevicesNotSupported*, if the graph is instantiated for device launch but a node's device differs from that of another node. This error can also be returned if a graph is not instantiated for device launch and it contains kernels which call device-side *cudaGraphLaunch()* from multiple devices. `errNode_out` will be set to this node.

If instantiation is successful, `result_out` will be set to *cudaGraphInstantiateSuccess*, and `hErrNode_out` will be set to NULL.

See also:

cudaGraphCreate, *cudaGraphInstantiate*, *cudaGraphInstantiateWithFlags*, *cudaGraphExecDestroy*

Note: Graph objects are not threadsafe. More here.

Note: Note that this function may also return error codes from previous, asynchronous launches.

Note: Note that this function may also return *cudaErrorInitializationError*, *cudaErrorInsufficientDriver* or *cudaErrorNoDevice* if this call tries to initialize internal CUDA RT state.

Note: Note that as specified by *cudaStreamAddCallback* no CUDA function may be called from callback. *cudaErrorNotPermitted* may, but is not guaranteed to, be returned as a diagnostic in such case.

Parameters

- **pGraphExec** – - Returns instantiated graph
- **graph** – - Graph to instantiate
- **instantiateParams** – - Instantiation parameters

Returns
cudaSuccess, *cudaErrorInvalidValue*

__host__ *cudaError_t* **cudaGraphKernelNodeCopyAttributes**(*cudaGraphNode_t* hDst, *cudaGraphNode_t* hSrc)

Copies attributes from source node to destination node.

Copies attributes from source node `hSrc` to destination node `hDst`. Both node must have the same context.

See also:

cudaAccessPolicyWindow

Note: Note that this function may also return error codes from previous, asynchronous launches.

Parameters

- **hDst** – **[out]** Destination node
- **hSrc** – **[in]** Source node For list of attributes see cudaKernelNodeAttrID

Returns
cudaSuccess, ::cudaErrorInvalidContext

__host__ *cudaError_t* **cudaGraphKernelNodeGetAttribute**(*cudaGraphNode_t* hNode, *cudaLaunchAttributeID* attr, cudaLaunchAttributeValue *value_out)

Queries node attribute.

Queries attribute `attr` from node `hNode` and stores it in corresponding member of `value_out`.

See also:

cudaAccessPolicyWindow

Note: Note that this function may also return error codes from previous, asynchronous launches.

Parameters

- **hNode** – **[in]**
- **attr** – **[in]**
- **value_out** –

Returns

cudaSuccess, *cudaErrorInvalidValue*, *cudaErrorInvalidResourceHandle*

__host__ *cudaError_t* **cudaGraphKernelNodeGetParams**(*cudaGraphNode_t* node, struct *cudaKernelNodeParams* *pNodeParams)

Returns a kernel node's parameters.

Returns the parameters of kernel node `node` in `pNodeParams`. The `kernelParams` or `extra` array returned in `pNodeParams`, as well as the argument values it points to, are owned by the node. This memory remains valid until the node is destroyed or its parameters are modified, and should not be modified directly. Use *cudaGraphKernelNodeSetParams* to update the parameters of this node.

The params will contain either `kernelParams` or `extra`, according to which of these was most recently set on the node.

See also:

cudaGraphNodeGetParams, *cudaLaunchKernel*, *cudaGraphAddKernelNode*, *cudaGraphKernelNodeSetParams*

Warning: This function will be deprecated in a future CUDA release. Please migrate to *cudaGraphNodeGetParams*.

Note: Graph objects are not threadsafe. More here.

Note: Note that this function may also return error codes from previous, asynchronous launches.

Note: Note that this function may also return *cudaErrorInitializationError*, *cudaErrorInsufficientDriver* or *cudaErrorNoDevice* if this call tries to initialize internal CUDA RT state.

Note: Note that as specified by *cudaStreamAddCallback* no CUDA function may be called from callback. *cudaErrorNotPermitted* may, but is not guaranteed to, be returned as a diagnostic in such case.

Parameters

- **node** – - Node to get the parameters for
- **pNodeParams** – - Pointer to return the parameters

Returns
cudaSuccess, *cudaErrorInvalidValue*, *cudaErrorInvalidDeviceFunction*

__host__ *cudaError_t* **cudaGraphKernelNodeSetAttribute**(*cudaGraphNode_t* hNode, *cudaLaunchAttributeID* attr, const cudaLaunchAttributeValue *value)

Sets node attribute.

Sets attribute `attr` on node `hNode` from corresponding attribute of `value`.

See also:

cudaAccessPolicyWindow

Note: Note that this function may also return error codes from previous, asynchronous launches.

Parameters

- **hNode** – **[out]**
- **attr** – **[in]**
- **value** –

Returns
cudaSuccess, *cudaErrorInvalidValue*, *cudaErrorInvalidResourceHandle*

__device__ *cudaError_t* **cudaGraphKernelNodeSetEnabled**(*cudaGraphDeviceNode_t* node, bool enable)

Enables or disables the given kernel node.

Enables or disables `node` based upon `enable`. If `enable` is true, the node will be enabled; if it is false, the node will be disabled. Disabled nodes will act as a NOP during execution. `node` must be device-updatable, and must reside upon the same device as the calling kernel.

If this function is called for the node's immediate dependent and that dependent is configured for programmatic dependent launch, then a memory fence must be invoked via __threadfence() before kickoff of the dependent is triggered via *cudaTriggerProgrammaticLaunchCompletion()* to ensure that the update is visible to that dependent node before it is launched.

See also:

cudaGraphKernelNodeSetParam, *cudaGraphKernelNodeSetGridDim*, *cudaGraphKernelNodeUpdatesApply*

Note: Note that this function may also return error codes from previous, asynchronous launches.

Parameters

- **node** – - The node to update
- **enable** – - Whether to enable or disable the node

Returns
cudaSuccess, cudaErrorInvalidValue

__device__ *cudaError_t* **cudaGraphKernelNodeSetGridDim**(*cudaGraphDeviceNode_t* node, dim3 gridDim)

Updates the grid dimensions of the given kernel node.

Sets the grid dimensions of `node` to `gridDim`. `node` must be device-updatable, and must reside upon the same device as thecalling kernel.

If this function is called for the node's immediate dependent and that dependent is configured for programmatic dependent launch, then a memory fence must be invoked via __threadfence() before kickoff of the dependent is triggered via *cudaTriggerProgrammaticLaunchCompletion()* to ensure that the update is visible to that dependent node before it is launched.

See also:

cudaGraphKernelNodeSetParam, *cudaGraphKernelNodeSetEnabled*, *cudaGraphKernelNodeUpdatesApply*

Note: Note that this function may also return error codes from previous, asynchronous launches.

Parameters

- **node** – - The node to update
- **gridDim** – - The grid dimensions to set

Returns
cudaSuccess, cudaErrorInvalidValue

template<typename **T**>
__device__ *cudaError_t* **cudaGraphKernelNodeSetParam**(*cudaGraphDeviceNode_t* node, size_t offset, const *T* &value)

Updates the kernel parameters of the given kernel node.

Updates the kernel parameters of `node` at `offset` to `value`. `node` must be device-updatable, and must reside upon the same device as the calling kernel.

If this function is called for the node's immediate dependent and that dependent is configured for programmatic dependent launch, then a memory fence must be invoked via __threadfence() before kickoff of the dependent is triggered via *cudaTriggerProgrammaticLaunchCompletion()* to ensure that the update is visible to that dependent node before it is launched.

See also:

cudaGraphKernelNodeSetEnabled, *cudaGraphKernelNodeSetGridDim*, *cudaGraphKernelNodeUpdatesApply*

Note: Note that this function may also return error codes from previous, asynchronous launches.

Parameters

- **node** – - The node to update
- **offset** – - The offset into the params at which to make the update
- **value** – - Parameter value to write

Returns

cudaSuccess, cudaErrorInvalidValue

__device__ *cudaError_t* **cudaGraphKernelNodeSetParam**(*cudaGraphDeviceNode_t* node, size_t offset, const void *value, size_t size)

Updates the kernel parameters of the given kernel node.

Updates `size` bytes in the kernel parameters of `node` at `offset` to the contents of `value`. `node` must be device-updatable, and must reside upon the same device as the calling kernel.

If this function is called for the node's immediate dependent and that dependent is configured for programmatic dependent launch, then a memory fence must be invoked via __threadfence() before kickoff of the dependent is triggered via *cudaTriggerProgrammaticLaunchCompletion()* to ensure that the update is visible to that dependent node before it is launched.

See also:

cudaGraphKernelNodeSetEnabled, *cudaGraphKernelNodeSetGridDim*, *cudaGraphKernelNodeUpdatesApply*

Note: Note that this function may also return error codes from previous, asynchronous launches.

Parameters

- **node** – - The node to update
- **offset** – - The offset into the params at which to make the update
- **value** – - Buffer containing the params to write
- **size** – - Size in bytes to update

Returns
cudaSuccess, cudaErrorInvalidValue

__host__ *cudaError_t* **cudaGraphKernelNodeSetParams**(*cudaGraphNode_t* node, const struct *cudaKernelNodeParams* *pNodeParams)

Sets a kernel node's parameters.

Sets the parameters of kernel node `node` to `pNodeParams`.

See also:

cudaGraphNodeSetParams, *cudaLaunchKernel*, *cudaGraphAddKernelNode*, *cudaGraphKernelNodeGetParams*

Warning: This function will be deprecated in a future CUDA release. Please migrate to *cudaGraphNodeSetParams*.

Note: Graph objects are not threadsafe. More here.

Note: Note that this function may also return error codes from previous, asynchronous launches.

Note: Note that this function may also return *cudaErrorInitializationError*, *cudaErrorInsufficientDriver* or *cudaErrorNoDevice* if this call tries to initialize internal CUDA RT state.

Note: Note that as specified by *cudaStreamAddCallback* no CUDA function may be called from callback. *cudaErrorNotPermitted* may, but is not guaranteed to, be returned as a diagnostic in such case.

Note: The API can also be used with a kernel *cudaKernel_t* by querying the handle using *cudaLibraryGetKernel()* or *cudaGetKernel* and then passing it to the API by casting to void*. The symbol `entryFuncAddr` passed to *cudaGetKernel* should be a symbol that is registered with the same CUDA Runtime instance.

Note: Passing a symbol that belongs that belongs to a different runtime instance will result in undefined behavior. The only type that can be reliably passed to a different runtime instance is *cudaKernel_t*

Parameters

- **node** – - Node to set the parameters for

- **pNodeParams** – - Parameters to copy

Returns

cudaSuccess, *cudaErrorInvalidValue*, *cudaErrorInvalidResourceHandle*, *cudaErrorMemoryAllocation*

__device__ *cudaError_t* **cudaGraphKernelNodeUpdatesApply**(const *cudaGraphKernelNodeUpdate* *updates, size_t updateCount)

Batch applies multiple kernel node updates.

Batch applies one or more kernel node updates based on the information provided in `updates`. `updateCount` specifies the number of updates to apply. Each entry in `updates` must specify a node to update, the type of update to apply, and the parameters for that type of update. See the documentation for *cudaGraphKernelNodeUpdate* for more detail.

If this function is called for the node's immediate dependent and that dependent is configured for programmatic dependent launch, then a memory fence must be invoked via __threadfence() before kickoff of the dependent is triggered via *cudaTriggerProgrammaticLaunchCompletion()* to ensure that the update is visible to that dependent node before it is launched.

See also:

cudaGraphKernelNodeSetParam, *cudaGraphKernelNodeSetEnabled*, *cudaGraphKernelNodeSetGridDim*

Note: Note that this function may also return error codes from previous, asynchronous launches.

Parameters

- **updates** – - The updates to apply
- **updateCount** – - The number of updates to apply

Returns

cudaSuccess, cudaErrorInvalidValue

__host__ *cudaError_t* **cudaGraphLaunch**(*cudaGraphExec_t* graphExec, *cudaStream_t* stream)

Launches an executable graph in a stream.

Executes `graphExec` in `stream`. Only one instance of `graphExec` may be executing at a time. Each launch is ordered behind both any previous work in `stream` and any previous launches of `graphExec`. To execute a graph concurrently, it must be instantiated multiple times into multiple executable graphs.

If any allocations created by `graphExec` remain unfreed (from a previous launch) and `graphExec` was not instantiated with *cudaGraphInstantiateFlagAutoFreeOnLaunch*, the launch will fail with *cudaErrorInvalidValue*.

See also:

cudaGraphInstantiate, *cudaGraphUpload*, *cudaGraphExecDestroy*

Note: Graph objects are not threadsafe. More here.

Note: Note that this function may also return error codes from previous, asynchronous launches.

Note: Note that this function may also return *cudaErrorInitializationError*, *cudaErrorInsufficientDriver* or *cudaErrorNoDevice* if this call tries to initialize internal CUDA RT state.

Note: Note that as specified by *cudaStreamAddCallback* no CUDA function may be called from callback. *cudaErrorNotPermitted* may, but is not guaranteed to, be returned as a diagnostic in such case.

Parameters

- **graphExec** – - Executable graph to launch
- **stream** – - Stream in which to launch the graph

Returns
cudaSuccess, *cudaErrorInvalidValue*

__host__ *cudaError_t* **cudaGraphMemAllocNodeGetParams**(*cudaGraphNode_t* node, struct *cudaMemAllocNodeParams* *params_out)

Returns a memory alloc node's parameters.

Returns the parameters of a memory alloc node `hNode` in `params_out`. The `poolProps` and `accessDescs` returned in `params_out`, are owned by the node. This memory remains valid until the node is destroyed. The returned parameters must not be modified.

See also:

cudaGraphNodeGetParams, *cudaGraphAddMemAllocNode*, *cudaGraphMemFreeNodeGetParams*

Warning: This function will be deprecated in a future CUDA release. Please migrate to *cudaGraphNodeGetParams*.

Note: Graph objects are not threadsafe. More here.

Note: Note that this function may also return error codes from previous, asynchronous launches.

Note: Note that this function may also return *cudaErrorInitializationError*, *cudaErrorInsufficientDriver* or *cudaErrorNoDevice* if this call tries to initialize internal CUDA RT state.

Note: Note that as specified by *cudaStreamAddCallback* no CUDA function may be called from callback. *cudaErrorNotPermitted* may, but is not guaranteed to, be returned as a diagnostic in such case.

Parameters

- **node** – - Node to get the parameters for
- **params_out** – - Pointer to return the parameters

Returns
cudaSuccess, *cudaErrorInvalidValue*

__host__ *cudaError_t* **cudaGraphMemFreeNodeGetParams**(*cudaGraphNode_t* node, void *dptr_out)

Returns a memory free node's parameters.

Returns the address of a memory free node `hNode` in `dptr_out`.

See also:

cudaGraphNodeGetParams, *cudaGraphAddMemFreeNode*, *cudaGraphMemFreeNodeGetParams*

Warning: This function will be deprecated in a future CUDA release. Please migrate to *cudaGraphNodeGetParams*.

Note: Graph objects are not threadsafe. More here.

Note: Note that this function may also return error codes from previous, asynchronous launches.

Note: Note that this function may also return *cudaErrorInitializationError*, *cudaErrorInsufficientDriver* or *cudaErrorNoDevice* if this call tries to initialize internal CUDA RT state.

Note: Note that as specified by *cudaStreamAddCallback* no CUDA function may be called from callback. *cudaErrorNotPermitted* may, but is not guaranteed to, be returned as a diagnostic in such case.

Parameters

- **node** – - Node to get the parameters for
- **dptr_out** – - Pointer to return the device address

Returns
cudaSuccess, *cudaErrorInvalidValue*

__host__ *cudaError_t* **cudaGraphMemcpyNodeGetParams**(*cudaGraphNode_t* node, struct *cudaMemcpy3DParms* *pNodeParams)

Returns a memcpy node's parameters.

Returns the parameters of memcpy node `node` in `pNodeParams`.

See also:

cudaGraphNodeGetParams, *cudaMemcpy3D*, *cudaGraphAddMemcpyNode*, *cudaGraphMemcpyNodeSetParams*

Warning: This function will be deprecated in a future CUDA release. Please migrate to *cudaGraphNodeGetParams*.

Note: Graph objects are not threadsafe. More here.

Note: Note that this function may also return error codes from previous, asynchronous launches.

Note: Note that this function may also return *cudaErrorInitializationError*, *cudaErrorInsufficientDriver* or *cudaErrorNoDevice* if this call tries to initialize internal CUDA RT state.

Note: Note that as specified by *cudaStreamAddCallback* no CUDA function may be called from callback. *cudaErrorNotPermitted* may, but is not guaranteed to, be returned as a diagnostic in such case.

Parameters

- **node** – - Node to get the parameters for
- **pNodeParams** – - Pointer to return the parameters

Returns
cudaSuccess, *cudaErrorInvalidValue*

__host__ *cudaError_t* **cudaGraphMemcpyNodeSetParams**(*cudaGraphNode_t* node, const struct *cudaMemcpy3DParms* *pNodeParams)

Sets a memcpy node's parameters.

Sets the parameters of memcpy node node to pNodeParams.

See also:

cudaGraphNodeSetParams, *cudaMemcpy3D*, *cudaGraphMemcpyNodeSetParamsToSymbol*, *cudaGraphMemcpyNodeSetParamsFromSymbol*, *cudaGraphMemcpyNodeSetParams1D*, *cudaGraphAddMemcpyNode*, *cudaGraphMemcpyNodeGetParams*

Warning: This function will be deprecated in a future CUDA release. Please migrate to *cudaGraphNodeSetParams*.

Note: Graph objects are not threadsafe. More here.

Note: Note that this function may also return error codes from previous, asynchronous launches.

Note: Note that this function may also return *cudaErrorInitializationError*, *cudaErrorInsufficientDriver* or *cudaErrorNoDevice* if this call tries to initialize internal CUDA RT state.

Note: Note that as specified by *cudaStreamAddCallback* no CUDA function may be called from callback. *cudaErrorNotPermitted* may, but is not guaranteed to, be returned as a diagnostic in such case.

Parameters

- **node** – - Node to set the parameters for
- **pNodeParams** – - Parameters to copy

Returns

cudaSuccess, *cudaErrorInvalidValue*,

__host__ *cudaError_t* **cudaGraphMemcpyNodeSetParams1D**(*cudaGraphNode_t* node, void *dst, const void *src, size_t count, enum *cudaMemcpyKind* kind)

Sets a memcpy node's parameters to perform a 1-dimensional copy.

Sets the parameters of memcpy node node to the copy described by the provided parameters.

When the graph is launched, the node will copy count bytes from the memory area pointed to by src to the memory area pointed to by dst, where kind specifies the direction of the copy, and must be one of *cudaMemcpyHostToHost*, *cudaMemcpyHostToDevice*, *cudaMemcpyDeviceToHost*, *cudaMemcpyDeviceToDevice*, or *cudaMemcpyDefault*. Passing *cudaMemcpyDefault* is recommended, in which case the type of transfer is inferred from the pointer values. However, *cudaMemcpyDefault* is only allowed on systems that support unified virtual addressing. Launching a memcpy node with dst and src pointers that do not match the direction of the copy results in an undefined behavior.

See also:

cudaMemcpy, *cudaGraphMemcpyNodeSetParams*, *cudaGraphAddMemcpyNode*, *cudaGraphMemcpyNodeGetParams*

Note: Graph objects are not threadsafe. More here.

Note: Note that this function may also return error codes from previous, asynchronous launches.

Note: Note that this function may also return *cudaErrorInitializationError*, *cudaErrorInsufficientDriver* or *cudaErrorNoDevice* if this call tries to initialize internal CUDA RT state.

Note: Note that as specified by *cudaStreamAddCallback* no CUDA function may be called from callback. *cudaErrorNotPermitted* may, but is not guaranteed to, be returned as a diagnostic in such case.

Parameters

- **node** – - Node to set the parameters for
- **dst** – - Destination memory address
- **src** – - Source memory address
- **count** – - Size in bytes to copy
- **kind** – - Type of transfer

Returns

cudaSuccess, *cudaErrorInvalidValue*

__host__ *cudaError_t* **cudaGraphMemcpyNodeSetParamsFromSymbol**(*cudaGraphNode_t* node, void *dst, const void *symbol, size_t count, size_t offset, enum *cudaMemcpyKind* kind)

Sets a memcpy node's parameters to copy from a symbol on the device.

Sets the parameters of memcpy node `node` to the copy described by the provided parameters.

When the graph is launched, the node will copy `count` bytes from the memory area pointed to by `offset` bytes from the start of symbol `symbol` to the memory area pointed to by `dst`. The memory areas may not overlap. `symbol` is a variable that resides in global or constant memory space. `kind` can be either *cudaMemcpyDeviceToHost*, *cudaMemcpyDeviceToDevice*, or *cudaMemcpyDefault*. Passing *cudaMemcpyDefault* is recommended, in which case the type of transfer is inferred from the pointer values. However, *cudaMemcpyDefault* is only allowed on systems that support unified virtual addressing.

See also:

cudaMemcpyFromSymbol, *cudaGraphMemcpyNodeSetParams*, *cudaGraphMemcpyNodeSetParamsToSymbol*, *cudaGraphAddMemcpyNode*, *cudaGraphMemcpyNodeGetParams*

Note: Graph objects are not threadsafe. More here.

Note: Note that this function may also return error codes from previous, asynchronous launches.

Note: Note that this function may also return *cudaErrorInitializationError*, *cudaErrorInsufficientDriver* or *cudaErrorNoDevice* if this call tries to initialize internal CUDA RT state.

Note: Note that as specified by *cudaStreamAddCallback* no CUDA function may be called from callback. *cudaErrorNotPermitted* may, but is not guaranteed to, be returned as a diagnostic in such case.

Parameters

- **node** – - Node to set the parameters for
- **dst** – - Destination memory address
- **symbol** – - Device symbol address
- **count** – - Size in bytes to copy
- **offset** – - Offset from start of symbol in bytes
- **kind** – - Type of transfer

Returns

cudaSuccess, *cudaErrorInvalidValue*

__host__ *cudaError_t* **cudaGraphMemcpyNodeSetParamsToSymbol**(*cudaGraphNode_t* node, const void *symbol, const void *src, size_t count, size_t offset, enum *cudaMemcpyKind* kind)

Sets a memcpy node's parameters to copy to a symbol on the device.

Sets the parameters of memcpy node `node` to the copy described by the provided parameters.

When the graph is launched, the node will copy `count` bytes from the memory area pointed to by `src` to the memory area pointed to by `offset` bytes from the start of symbol `symbol`. The memory areas may not overlap. `symbol` is a variable that resides in global or constant memory space. `kind` can be either *cudaMemcpyHostToDevice*, *cudaMemcpyDeviceToDevice*, or *cudaMemcpyDefault*. Passing *cudaMemcpyDefault* is recommended, in which case the type of transfer is inferred from the pointer values. However, *cudaMemcpyDefault* is only allowed on systems that support unified virtual addressing.

See also:

cudaMemcpyToSymbol, *cudaGraphMemcpyNodeSetParams*, *cudaGraphMemcpyNodeSetParams-FromSymbol*, *cudaGraphAddMemcpyNode*, *cudaGraphMemcpyNodeGetParams*

Note: Graph objects are not threadsafe. More here.

Note: Note that this function may also return error codes from previous, asynchronous launches.

Note: Note that this function may also return *cudaErrorInitializationError*, *cudaErrorInsufficient-Driver* or *cudaErrorNoDevice* if this call tries to initialize internal CUDA RT state.

Note: Note that as specified by *cudaStreamAddCallback* no CUDA function may be called from callback. *cudaErrorNotPermitted* may, but is not guaranteed to, be returned as a diagnostic in such case.

Parameters

- **node** – - Node to set the parameters for
- **symbol** – - Device symbol address
- **src** – - Source memory address
- **count** – - Size in bytes to copy
- **offset** – - Offset from start of symbol in bytes
- **kind** – - Type of transfer

Returns

cudaSuccess, *cudaErrorInvalidValue*

__host__ *cudaError_t* **cudaGraphMemsetNodeGetParams**(*cudaGraphNode_t* node, struct *cudaMemsetParams* *pNodeParams)

Returns a memset node's parameters.

Returns the parameters of memset node `node` in `pNodeParams`.

See also:

cudaGraphNodeGetParams, *cudaMemset2D*, *cudaGraphAddMemsetNode*, *cudaGraphMemsetNodeSetParams*

Warning: This function will be deprecated in a future CUDA release. Please migrate to *cudaGraphNodeGetParams*.

Note: Graph objects are not threadsafe. More here.

Note: Note that this function may also return error codes from previous, asynchronous launches.

Note: Note that this function may also return *cudaErrorInitializationError*, *cudaErrorInsufficientDriver* or *cudaErrorNoDevice* if this call tries to initialize internal CUDA RT state.

Note: Note that as specified by *cudaStreamAddCallback* no CUDA function may be called from callback. *cudaErrorNotPermitted* may, but is not guaranteed to, be returned as a diagnostic in such case.

Parameters

- **node** – - Node to get the parameters for
- **pNodeParams** – - Pointer to return the parameters

Returns
cudaSuccess, *cudaErrorInvalidValue*

__host__ *cudaError_t* **cudaGraphMemsetNodeSetParams**(*cudaGraphNode_t* node, const struct *cudaMemsetParams* *pNodeParams)

Sets a memset node's parameters.

Sets the parameters of memset node `node` to `pNodeParams`.

See also:

cudaGraphNodeSetParams, *cudaMemset2D*, *cudaGraphAddMemsetNode*, *cudaGraphMemsetNodeGetParams*

Warning: This function will be deprecated in a future CUDA release. Please migrate to *cudaGraphNodeSetParams*.

Note: Graph objects are not threadsafe. More here.

Note: Note that this function may also return error codes from previous, asynchronous launches.

Note: Note that this function may also return *cudaErrorInitializationError*, *cudaErrorInsufficientDriver* or *cudaErrorNoDevice* if this call tries to initialize internal CUDA RT state.

Note: Note that as specified by *cudaStreamAddCallback* no CUDA function may be called from callback. *cudaErrorNotPermitted* may, but is not guaranteed to, be returned as a diagnostic in such case.

Parameters

- **node** – - Node to set the parameters for
- **pNodeParams** – - Parameters to copy

Returns
cudaSuccess, *cudaErrorInvalidValue*

__host__ *cudaError_t* **cudaGraphNodeFindInClone**(*cudaGraphNode_t* *pNode, *cudaGraphNode_t* originalNode, *cudaGraph_t* clonedGraph)

Finds a cloned version of a node.

This function returns the node in `clonedGraph` corresponding to `originalNode` in the original graph.

`clonedGraph` must have been cloned from `originalGraph` via *cudaGraphClone*. `originalNode` must have been in `originalGraph` at the time of the call to *cudaGraphClone*, and the corresponding cloned node in `clonedGraph` must not have been removed. The cloned node is then returned via `pClonedNode`.

See also:

cudaGraphClone

Note: Graph objects are not threadsafe. More here.

Note: Note that this function may also return error codes from previous, asynchronous launches.

Note: Note that this function may also return *cudaErrorInitializationError*, *cudaErrorInsufficientDriver* or *cudaErrorNoDevice* if this call tries to initialize internal CUDA RT state.

Note: Note that as specified by *cudaStreamAddCallback* no CUDA function may be called from callback. *cudaErrorNotPermitted* may, but is not guaranteed to, be returned as a diagnostic in such case.

Parameters

- **pNode** – - Returns handle to the cloned node
- **originalNode** – - Handle to the original node
- **clonedGraph** – - Cloned graph to query

Returns
cudaSuccess, *cudaErrorInvalidValue*

__host__ *cudaError_t* **cudaGraphNodeGetContainingGraph**(*cudaGraphNode_t* hNode, *cudaGraph_t* *phGraph)

Returns the graph that contains a given graph node.

Returns the graph that contains `hNode` in `*phGraph`. If hNode is in a child graph, the child graph it is in is returned.

See also:

cudaGraphGetNodes, *cudaGraphDebugDotPrint* *cudaGraphNodeGetLocalId* *cudaGraphNodeGetToolsId* *cudaGraphGetId* *cudaGraphExecGetId*

Parameters

- **hNode** – - Node to query
- **phGraph** – - Pointer to return the containing graph

Returns
cudaSuccess *cudaErrorInvalidValue*

__host__ *cudaError_t* **cudaGraphNodeGetDependencies**(*cudaGraphNode_t* node, *cudaGraphNode_t* *pDependencies, *cudaGraphEdgeData* *edgeData, size_t *pNumDependencies)

Returns a node's dependencies.

Returns a list of `node`'s dependencies. `pDependencies` may be NULL, in which case this function will return the number of dependencies in `pNumDependencies`. Otherwise, `pNumDependencies` entries will be filled in. If `pNumDependencies` is higher than the actual number of dependencies, the remaining entries in `pDependencies` will be set to NULL, and the number of nodes actually obtained will be returned in `pNumDependencies`.

Note that if an edge has non-zero (non-default) edge data and `edgeData` is NULL, this API will return *cudaErrorLossyQuery*. If `edgeData` is non-NULL, then `pDependencies` must be as well.

See also:

cudaGraphNodeGetDependentNodes, *cudaGraphGetNodes*, *cudaGraphGetRootNodes*, *cudaGraphGetEdges*, *cudaGraphAddDependencies*, *cudaGraphRemoveDependencies*

Note: Graph objects are not threadsafe. More here.

Note: Note that this function may also return error codes from previous, asynchronous launches.

Note: Note that this function may also return *cudaErrorInitializationError*, *cudaErrorInsufficientDriver* or *cudaErrorNoDevice* if this call tries to initialize internal CUDA RT state.

Note: Note that as specified by *cudaStreamAddCallback* no CUDA function may be called from callback. *cudaErrorNotPermitted* may, but is not guaranteed to, be returned as a diagnostic in such case.

Parameters

- **node** – - Node to query
- **pDependencies** – - Pointer to return the dependencies
- **edgeData** – - Optional array to return edge data for each dependency
- **pNumDependencies** – - See description

Returns

cudaSuccess, *cudaErrorLossyQuery*, *cudaErrorInvalidValue*

__host__ *cudaError_t* **cudaGraphNodeGetDependentNodes**(*cudaGraphNode_t* node, *cudaGraphNode_t* *pDependentNodes, *cudaGraphEdgeData* *edgeData, size_t *pNumDependentNodes)

Returns a node's dependent nodes.

Returns a list of `node`'s dependent nodes. `pDependentNodes` may be NULL, in which case this function will return the number of dependent nodes in `pNumDependentNodes`. Otherwise, `pNumDependentNodes` entries will be filled in. If `pNumDependentNodes` is higher than the actual number of dependent nodes, the remaining entries in `pDependentNodes` will be set to NULL, and the number of nodes actually obtained will be returned in `pNumDependentNodes`.

Note that if an edge has non-zero (non-default) edge data and `edgeData` is NULL, this API will return *cudaErrorLossyQuery*. If `edgeData` is non-NULL, then `pDependentNodes` must be as well.

See also:

cudaGraphNodeGetDependencies, *cudaGraphGetNodes*, *cudaGraphGetRootNodes*, *cudaGraphGetEdges*, *cudaGraphAddDependencies*, *cudaGraphRemoveDependencies*

Note: Graph objects are not threadsafe. More here.

Note: Note that this function may also return error codes from previous, asynchronous launches.

Note: Note that this function may also return *cudaErrorInitializationError*, *cudaErrorInsufficientDriver* or *cudaErrorNoDevice* if this call tries to initialize internal CUDA RT state.

Note: Note that as specified by *cudaStreamAddCallback* no CUDA function may be called from callback. *cudaErrorNotPermitted* may, but is not guaranteed to, be returned as a diagnostic in such case.

Parameters

- **node** – - Node to query
- **pDependentNodes** – - Pointer to return the dependent nodes
- **edgeData** – - Optional pointer to return edge data for dependent nodes
- **pNumDependentNodes** – - See description

Returns

cudaSuccess, *cudaErrorLossyQuery*, *cudaErrorInvalidValue*

__host__ *cudaError_t* **cudaGraphNodeGetEnabled**(*cudaGraphExec_t* hGraphExec, *cudaGraphNode_t* hNode, unsigned int *isEnabled)

Query whether a node in the given graphExec is enabled.

Sets isEnabled to 1 if `hNode` is enabled, or 0 if `hNode` is disabled.

The node is identified by the corresponding node `hNode` in the non-executable graph, from which the executable graph was instantiated.

`hNode` must not have been removed from the original graph.

See also:

cudaGraphNodeSetEnabled, *cudaGraphExecUpdate*, *cudaGraphInstantiate cudaGraphLaunch*

Note: Currently only kernel, memset and memcpy nodes are supported.

Note: Graph objects are not threadsafe. More here.

Note: Note that this function may also return error codes from previous, asynchronous launches.

Note: Note that this function may also return *cudaErrorInitializationError*, *cudaErrorInsufficient-Driver* or *cudaErrorNoDevice* if this call tries to initialize internal CUDA RT state.

Note: Note that as specified by *cudaStreamAddCallback* no CUDA function may be called from callback. *cudaErrorNotPermitted* may, but is not guaranteed to, be returned as a diagnostic in such case.

Parameters

- **hGraphExec** – - The executable graph in which to set the specified node
- **hNode** – - Node from the graph from which graphExec was instantiated
- **isEnabled** – - Location to return the enabled status of the node

Returns
cudaSuccess, *cudaErrorInvalidValue*,

__host__ *cudaError_t* **cudaGraphNodeGetLocalId**(*cudaGraphNode_t* hNode, unsigned int *nodeId)

Returns the node id of a given graph node.

Returns the node id of `hNode` in `*nodeId`. The nodeId matches that referenced by *cudaGraphDebugDotPrint*. The local nodeId and graphId together can uniquely identify the node.

See also:

cudaGraphGetNodes, *cudaGraphDebugDotPrint* *cudaGraphNodeGetContainingGraph* *cudaGraphNodeGetToolsId* *cudaGraphGetId* *cudaGraphExecGetId*

Parameters

- **hNode** – - Node to query
- **nodeId** – - Pointer to return the nodeId

Returns
cudaSuccess *cudaErrorInvalidValue*

__host__ *cudaError_t* **cudaGraphNodeGetParams**(*cudaGraphNode_t* node, struct *cudaGraphNodeParams* *nodeParams)

Returns a graph node's parameters.

Returns the parameters of graph node `node` in `*nodeParams`.

Any pointers returned in `*nodeParams` point to driver-owned memory associated with the node. This memory remains valid until the node is destroyed. Any memory pointed to from `*nodeParams` must not be modified.

The returned parameters are a description of the node, but may not be identical to the struct provided at creation and may not be suitable for direct creation of identical nodes. This is because parameters may be partially unspecified and filled in by the driver at creation, may reference non-copyable handles, or may describe ownership semantics or other parameters that govern behavior of node creation but are not part of the final functional descriptor.

See also:

cudaGraphNodeGetParams, *cudaGraphNodeSetParams*, *cudaGraphAddNode*, *cudaGraphExecNodeSetParams*

Note: Graph objects are not threadsafe. More here.

Note: Note that this function may also return error codes from previous, asynchronous launches.

Note: Note that this function may also return *cudaErrorInitializationError*, *cudaErrorInsufficientDriver* or *cudaErrorNoDevice* if this call tries to initialize internal CUDA RT state.

Note: Note that as specified by *cudaStreamAddCallback* no CUDA function may be called from callback. *cudaErrorNotPermitted* may, but is not guaranteed to, be returned as a diagnostic in such case.

Parameters

- **node** – - Node to get the parameters for
- **nodeParams** – - Pointer to return the parameters

Returns
cudaSuccess, *cudaErrorInvalidValue*, *cudaErrorNotSupported*

__host__ *cudaError_t* **cudaGraphNodeGetToolsId**(*cudaGraphNode_t* hNode, unsigned long long *toolsNodeId)

Returns an id used by tools to identify a given node.

See also:

cudaGraphGetNodes, *cudaGraphDebugDotPrint* *cudaGraphNodeGetContainingGraph* *cudaGraphNodeGetLocalId* *cudaGraphGetId* *cudaGraphExecGetId*

Parameters

- **hNode** – - Node to query
- ***toolsNodeId** – - Pointer to return the id used by tools

Returns
::CUDA_SUCCESS *cudaErrorInvalidValue*

__host__ *cudaError_t* **cudaGraphNodeGetType**(*cudaGraphNode_t* node, enum *cudaGraphNodeType* *pType)

Returns a node's type.

Returns the node type of node in pType.

See also:

cudaGraphGetNodes, *cudaGraphGetRootNodes*, *cudaGraphChildGraphNodeGetGraph*, *cudaGraphKernelNodeGetParams*, *cudaGraphKernelNodeSetParams*, *cudaGraphHostNodeGetParams*, *cudaGraphHostNodeSetParams*, *cudaGraphMemcpyNodeGetParams*, *cudaGraphMemcpyNodeSetParams*, *cudaGraphMemsetNodeGetParams*, *cudaGraphMemsetNodeSetParams*

Note: Graph objects are not threadsafe. More here.

Note: Note that this function may also return error codes from previous, asynchronous launches.

Note: Note that this function may also return *cudaErrorInitializationError*, *cudaErrorInsufficientDriver* or *cudaErrorNoDevice* if this call tries to initialize internal CUDA RT state.

Note: Note that as specified by *cudaStreamAddCallback* no CUDA function may be called from callback. *cudaErrorNotPermitted* may, but is not guaranteed to, be returned as a diagnostic in such case.

Parameters

- **node** – - Node to query
- **pType** – - Pointer to return the node type

Returns
cudaSuccess, *cudaErrorInvalidValue*

__host__ *cudaError_t* **cudaGraphNodeSetEnabled**(*cudaGraphExec_t* hGraphExec, *cudaGraphNode_t* hNode, unsigned int isEnabled)

Enables or disables the specified node in the given graphExec.

Sets `hNode` to be either enabled or disabled. Disabled nodes are functionally equivalent to empty nodes until they are reenabled. Existing node parameters are not affected by disabling/enabling the node.

The node is identified by the corresponding node `hNode` in the non-executable graph, from which the executable graph was instantiated.

`hNode` must not have been removed from the original graph.

The modifications only affect future launches of `hGraphExec`. Already enqueued or running launches of `hGraphExec` are not affected by this call. `hNode` is also not modified by this call.

See also:

cudaGraphNodeGetEnabled, *cudaGraphExecUpdate*, *cudaGraphInstantiate cudaGraphLaunch*

Note: Currently only kernel, memset and memcpy nodes are supported.

Note: Graph objects are not threadsafe. More here.

Note: Note that this function may also return error codes from previous, asynchronous launches.

Note: Note that this function may also return *cudaErrorInitializationError*, *cudaErrorInsufficientDriver* or *cudaErrorNoDevice* if this call tries to initialize internal CUDA RT state.

Note: Note that as specified by *cudaStreamAddCallback* no CUDA function may be called from callback. *cudaErrorNotPermitted* may, but is not guaranteed to, be returned as a diagnostic in such case.

Parameters

- **hGraphExec** – - The executable graph in which to set the specified node
- **hNode** – - Node from the graph from which graphExec was instantiated
- **isEnabled** – - Node is enabled if != 0, otherwise the node is disabled

Returns

cudaSuccess, *cudaErrorInvalidValue*,

__host__ *cudaError_t* **cudaGraphNodeSetParams**(*cudaGraphNode_t* node, struct *cudaGraphNodeParams* *nodeParams)

Update a graph node's parameters.

Sets the parameters of graph node `node` to `nodeParams`. The node type specified by `nodeParams->type` must match the type of `node`. `nodeParams` must be fully initialized and all unused bytes (reserved, padding) zeroed.

Modifying parameters is not supported for node types cudaGraphNodeTypeMemAlloc and cudaGraphNodeTypeMemFree.

See also:

cudaGraphNodeGetParams, *cudaGraphAddNode*, *cudaGraphExecNodeSetParams*

Note: Graph objects are not threadsafe. More here.

Note: Note that this function may also return error codes from previous, asynchronous launches.

Note: Note that this function may also return *cudaErrorInitializationError*, *cudaErrorInsufficientDriver* or *cudaErrorNoDevice* if this call tries to initialize internal CUDA RT state.

Note: Note that as specified by *cudaStreamAddCallback* no CUDA function may be called from callback. *cudaErrorNotPermitted* may, but is not guaranteed to, be returned as a diagnostic in such case.

Parameters

- **node** – - Node to set the parameters for
- **nodeParams** – - Parameters to copy

Returns

cudaSuccess, *cudaErrorInvalidValue*, *cudaErrorInvalidDeviceFunction*, *cudaErrorNotSupported*

__host__ *cudaError_t* **cudaGraphReleaseUserObject**(*cudaGraph_t* graph, *cudaUserObject_t* object, unsigned int count = 1)

Release a user object reference from a graph.

Releases user object references owned by a graph.

See CUDA User Objects in the CUDA C++ Programming Guide for more information on user objects.

See also:

cudaUserObjectCreate cudaUserObjectRetain, *cudaUserObjectRelease*, *cudaGraphRetainUserObject*, *cudaGraphCreate*

Parameters

- **graph** – - The graph that will release the reference
- **object** – - The user object to release a reference for
- **count** – - The number of references to release, typically 1. Must be nonzero and not larger than INT_MAX.

Returns

cudaSuccess, *cudaErrorInvalidValue*

__host__ *cudaError_t* **cudaGraphRemoveDependencies**(*cudaGraph_t* graph, const *cudaGraphNode_t* *from, const *cudaGraphNode_t* *to, const *cudaGraphEdgeData* *edgeData, size_t numDependencies)

Removes dependency edges from a graph.

The number of `pDependencies` to be removed is defined by `numDependencies`. Elements in `pFrom` and `pTo` at corresponding indices define a dependency. Each node in `pFrom` and `pTo` must belong to `graph`.

If `numDependencies` is 0, elements in `pFrom` and `pTo` will be ignored. Specifying an edge that does not exist in the graph, with data matching `edgeData`, results in an error. `edgeData` is nullable, which is equivalent to passing default (zeroed) data for each edge.

See also:

cudaGraphAddDependencies, *cudaGraphGetEdges*, *cudaGraphNodeGetDependencies*, *cudaGraphNodeGetDependentNodes*

Note: Graph objects are not threadsafe. More here.

Note: Note that this function may also return error codes from previous, asynchronous launches.

Note: Note that this function may also return *cudaErrorInitializationError*, *cudaErrorInsufficientDriver* or *cudaErrorNoDevice* if this call tries to initialize internal CUDA RT state.

Note: Note that as specified by *cudaStreamAddCallback* no CUDA function may be called from callback. *cudaErrorNotPermitted* may, but is not guaranteed to, be returned as a diagnostic in such case.

Parameters

- **graph** – - Graph from which to remove dependencies
- **from** – - Array of nodes that provide the dependencies
- **to** – - Array of dependent nodes
- **edgeData** – - Optional array of edge data. If NULL, edge data is assumed to be default (zeroed).
- **numDependencies** – - Number of dependencies to be removed

Returns

cudaSuccess, *cudaErrorInvalidValue*

__host__ *cudaError_t* **cudaGraphRetainUserObject**(*cudaGraph_t* graph, *cudaUserObject_t* object, unsigned int count = 1, unsigned int flags = 0)

Retain a reference to a user object from a graph.

Creates or moves user object references that will be owned by a CUDA graph.

See CUDA User Objects in the CUDA C++ Programming Guide for more information on user objects.

See also:

cudaUserObjectCreate *cudaUserObjectRetain*, *cudaUserObjectRelease*, *cudaGraphReleaseUserObject*, *cudaGraphCreate*

Parameters

- **graph** – - The graph to associate the reference with
- **object** – - The user object to retain a reference for
- **count** – - The number of references to add to the graph, typically 1. Must be nonzero and not larger than INT_MAX.
- **flags** – - The optional flag *cudaGraphUserObjectMove* transfers references from the calling thread, rather than create new references. Pass 0 to create new references.

Returns
cudaSuccess, *cudaErrorInvalidValue*

__device__ void **cudaGraphSetConditional**(*cudaGraphConditionalHandle* handle, unsigned int value)

Sets the condition value associated with a conditional node.

Sets the condition value associated with a conditional node.

Note: `handle` must be associated with the same context as the kernel calling this function. Note: It is undefined behavior to have racing / possibly concurrent calls to *cudaGraphSetConditional*.

See also:

cudaGraphConditionalHandleCreate

__host__ *cudaError_t* **cudaGraphUpload**(*cudaGraphExec_t* graphExec, *cudaStream_t* stream)

Uploads an executable graph in a stream.

Uploads `graphExec` to the device in `stream` without executing it. Uploads of the same `graphExec` will be serialized. Each upload is ordered behind both any previous work in `stream` and any previous launches of `graphExec`. Uses memory cached by `stream` to back the allocations owned by `graphExec`.

See also:

cudaGraphInstantiate, *cudaGraphLaunch*, *cudaGraphExecDestroy*

Note: Note that this function may also return error codes from previous, asynchronous launches.

Note: Note that this function may also return *cudaErrorInitializationError*, *cudaErrorInsufficientDriver* or *cudaErrorNoDevice* if this call tries to initialize internal CUDA RT state.

Parameters

- **graphExec** – - Executable graph to upload
- **stream** – - Stream in which to upload the graph

Returns
cudaSuccess, *cudaErrorInvalidValue*,

__host__ *cudaError_t* **cudaUserObjectCreate**(*cudaUserObject_t* *object_out, void *ptr, cudaHostFn_t destroy, unsigned int initialRefcount, unsigned int flags)

Create a user object.

Create a user object with the specified destructor callback and initial reference count. The initial references are owned by the caller.

Destructor callbacks cannot make CUDA API calls and should avoid blocking behavior, as they are executed by a shared internal thread. Another thread may be signaled to perform such actions, if it does not block forward progress of tasks scheduled through CUDA.

See CUDA User Objects in the CUDA C++ Programming Guide for more information on user objects.

See also:

cudaUserObjectRetain, *cudaUserObjectRelease*, *cudaGraphRetainUserObject*, *cudaGraphReleaseUserObject*, *cudaGraphCreate*

Parameters

- **object_out** – - Location to return the user object handle
- **ptr** – - The pointer to pass to the destroy function
- **destroy** – - Callback to free the user object when it is no longer in use
- **initialRefcount** – - The initial refcount to create the object with, typically 1. The initial references are owned by the calling thread.
- **flags** – - Currently it is required to pass *cudaUserObjectNoDestructorSync*, which is the only defined flag. This indicates that the destroy callback cannot be waited on by any CUDA API. Users requiring synchronization of the callback should signal its completion manually.

Returns

cudaSuccess, *cudaErrorInvalidValue*

__host__ *cudaError_t* **cudaUserObjectRelease**(*cudaUserObject_t* object, unsigned int count = 1)

Release a reference to a user object.

Releases user object references owned by the caller. The object's destructor is invoked if the reference count reaches zero.

It is undefined behavior to release references not owned by the caller, or to use a user object handle after all references are released.

See CUDA User Objects in the CUDA C++ Programming Guide for more information on user objects.

See also:

cudaUserObjectCreate, *cudaUserObjectRetain*, *cudaGraphRetainUserObject*, *cudaGraphReleaseUserObject*, *cudaGraphCreate*

Parameters

- **object** – - The object to release
- **count** – - The number of references to release, typically 1. Must be nonzero and not larger than INT_MAX.

Returns

cudaSuccess, *cudaErrorInvalidValue*

__host__ *cudaError_t* **cudaUserObjectRetain**(*cudaUserObject_t* object, unsigned int count = 1)

Retain a reference to a user object.

Retains new references to a user object. The new references are owned by the caller.

See CUDA User Objects in the CUDA C++ Programming Guide for more information on user objects.

See also:

cudaUserObjectCreate, *cudaUserObjectRelease*, *cudaGraphRetainUserObject*, *cudaGraphReleaseUserObject*, *cudaGraphCreate*

Parameters

- **object** – - The object to retain
- **count** – - The number of references to retain, typically 1. Must be nonzero and not larger than INT_MAX.

Returns

cudaSuccess, *cudaErrorInvalidValue*

6.21. Graphics Interoperability

This section describes the graphics interoperability functions of the CUDA runtime application programming interface.

Functions

__host__ cudaError_t *cudaGraphicsMapResources*(int count, cudaGraphicsResource_t *resources, cudaStream_t stream=0)
Map graphics resources for access by CUDA.

__host__ cudaError_t *cudaGraphicsResourceGetMappedMipmappedArray*(cudaMipmappedArray_t *mipmappedArray, cudaGraphicsResource_t resource)
Get a mipmapped array through which to access a mapped graphics resource.

__host__ cudaError_t *cudaGraphicsResourceGetMappedPointer*(void **devPtr, size_t *size, cudaGraphicsResource_t resource)
Get an device pointer through which to access a mapped graphics resource.

__host__ cudaError_t *cudaGraphicsResourceSetMapFlags*(cudaGraphicsResource_t resource, unsigned int flags)
Set usage flags for mapping a graphics resource.

__host__ cudaError_t *cudaGraphicsSubResourceGetMappedArray*(cudaArray_t *array, cudaGraphicsResource_t resource, unsigned int arrayIndex, unsigned int mipLevel)
Get an array through which to access a subresource of a mapped graphics resource.

__host__ cudaError_t *cudaGraphicsUnmapResources*(int count, cudaGraphicsResource_t *resources, cudaStream_t stream=0)
Unmap graphics resources.

__host__ cudaError_t *cudaGraphicsUnregisterResource*(cudaGraphicsResource_t resource)
Unregisters a graphics resource for access by CUDA.

6.21.1. Functions

__host__ *cudaError_t* **cudaGraphicsMapResources**(int count, *cudaGraphicsResource_t* *resources, *cudaStream_t* stream = 0)

Map graphics resources for access by CUDA.

Maps the `count` graphics resources in `resources` for access by CUDA.

The resources in `resources` may be accessed by CUDA until they are unmapped. The graphics API from which `resources` were registered should not access any resources while they are mapped by CUDA. If an application does so, the results are undefined.

This function provides the synchronization guarantee that any graphics calls issued before *cudaGraphicsMapResources()* will complete before any subsequent CUDA work issued in `stream` begins.

If `resources` contains any duplicate entries then *cudaErrorInvalidResourceHandle* is returned. If any of `resources` are presently mapped for access by CUDA then *cudaErrorUnknown* is returned.

See also:

cudaGraphicsResourceGetMappedPointer, *cudaGraphicsSubResourceGetMappedArray*, *cudaGraphicsUnmapResources*, ::cuGraphicsMapResources

---

Note: This function uses standard default stream semantics.

Note: Note that this function may also return error codes from previous, asynchronous launches.

Note: Note that this function may also return *cudaErrorInitializationError*, *cudaErrorInsufficientDriver* or *cudaErrorNoDevice* if this call tries to initialize internal CUDA RT state.

Note: Note that as specified by *cudaStreamAddCallback* no CUDA function may be called from callback. *cudaErrorNotPermitted* may, but is not guaranteed to, be returned as a diagnostic in such case.

Parameters

- **count** – - Number of resources to map
- **resources** – - Resources to map for CUDA
- **stream** – - Stream for synchronization

Returns
cudaSuccess, *cudaErrorInvalidResourceHandle*, *cudaErrorUnknown*

__host__ *cudaError_t* **cudaGraphicsResourceGetMappedMipmappedArray**(*cudaMipmappedArray_t* *mipmappedArray, *cudaGraphicsResource_t* resource)

Get a mipmapped array through which to access a mapped graphics resource.

Returns in `*mipmappedArray` a mipmapped array through which the mapped graphics resource `resource` may be accessed. The value set in `mipmappedArray` may change every time that `resource` is mapped.

If `resource` is not a texture then it cannot be accessed via an array and *cudaErrorUnknown* is returned. If `resource` is not mapped then *cudaErrorUnknown* is returned.

See also:

cudaGraphicsResourceGetMappedPointer, ::cuGraphicsResourceGetMappedMipmappedArray

---

Note: Note that this function may also return error codes from previous, asynchronous launches.

Note: Note that this function may also return *cudaErrorInitializationError*, *cudaErrorInsufficientDriver* or *cudaErrorNoDevice* if this call tries to initialize internal CUDA RT state.

Note: Note that as specified by *cudaStreamAddCallback* no CUDA function may be called from callback. *cudaErrorNotPermitted* may, but is not guaranteed to, be returned as a diagnostic in such case.

Parameters

- **mipmappedArray** – - Returned mipmapped array through which `resource` may be accessed
- **resource** – - Mapped resource to access

Returns

cudaSuccess, *cudaErrorInvalidValue*, *cudaErrorInvalidResourceHandle*, *cudaErrorUnknown*

__host__ *cudaError_t* **cudaGraphicsResourceGetMappedPointer**(void **devPtr, size_t *size, *cudaGraphicsResource_t* resource)

Get an device pointer through which to access a mapped graphics resource.

Returns in `*devPtr` a pointer through which the mapped graphics resource `resource` may be accessed. Returns in `*size` the size of the memory in bytes which may be accessed from that pointer. The value set in `devPtr` may change every time that `resource` is mapped.

If `resource` is not a buffer then it cannot be accessed via a pointer and *cudaErrorUnknown* is returned. If `resource` is not mapped then *cudaErrorUnknown* is returned.

See also:

cudaGraphicsMapResources, *cudaGraphicsSubResourceGetMappedArray*, ::cuGraphicsResourceGetMappedPointer

---

Note: Note that this function may also return error codes from previous, asynchronous launches.

Note: Note that this function may also return *cudaErrorInitializationError*, *cudaErrorInsufficientDriver* or *cudaErrorNoDevice* if this call tries to initialize internal CUDA RT state.

Note: Note that as specified by *cudaStreamAddCallback* no CUDA function may be called from callback. *cudaErrorNotPermitted* may, but is not guaranteed to, be returned as a diagnostic in such case.

Parameters

- **devPtr** – - Returned pointer through which `resource` may be accessed
- **size** – - Returned size of the buffer accessible starting at `*devPtr`
- **resource** – - Mapped resource to access

Returns

cudaSuccess, *cudaErrorInvalidValue*, *cudaErrorInvalidResourceHandle*, *cudaErrorUnknown*

__host__ *cudaError_t* **cudaGraphicsResourceSetMapFlags**(*cudaGraphicsResource_t* resource, unsigned int flags)

Set usage flags for mapping a graphics resource.

Set `flags` for mapping the graphics resource `resource`.

Changes to `flags` will take effect the next time `resource` is mapped. The `flags` argument may be any of the following:

- *cudaGraphicsMapFlagsNone*: Specifies no hints about how `resource` will be used. It is therefore assumed that CUDA may read from or write to `resource`.
- *cudaGraphicsMapFlagsReadOnly*: Specifies that CUDA will not write to `resource`.
- *cudaGraphicsMapFlagsWriteDiscard*: Specifies CUDA will not read from `resource` and will write over the entire contents of `resource`, so none of the data previously stored in `resource` will be preserved.

If `resource` is presently mapped for access by CUDA then *cudaErrorUnknown* is returned. If `flags` is not one of the above values then *cudaErrorInvalidValue* is returned.

See also:

cudaGraphicsMapResources, ::cuGraphicsResourceSetMapFlags

Note: Note that this function may also return error codes from previous, asynchronous launches.

Note: Note that this function may also return *cudaErrorInitializationError*, *cudaErrorInsufficientDriver* or *cudaErrorNoDevice* if this call tries to initialize internal CUDA RT state.

Note: Note that as specified by *cudaStreamAddCallback* no CUDA function may be called from callback. *cudaErrorNotPermitted* may, but is not guaranteed to, be returned as a diagnostic in such case.

Parameters

- **resource** – - Registered resource to set flags for
- **flags** – - Parameters for resource mapping

Returns

cudaSuccess, *cudaErrorInvalidValue*, *cudaErrorInvalidResourceHandle*, *cudaErrorUnknown*,

__host__ *cudaError_t* **cudaGraphicsSubResourceGetMappedArray**(*cudaArray_t* *array, *cudaGraphicsResource_t* resource, unsigned int arrayIndex, unsigned int mipLevel)

Get an array through which to access a subresource of a mapped graphics resource.

Returns in `*array` an array through which the subresource of the mapped graphics resource `resource` which corresponds to array index `arrayIndex` and mipmap level `mipLevel` may be accessed. The value set in `array` may change every time that `resource` is mapped.

If `resource` is not a texture then it cannot be accessed via an array and *cudaErrorUnknown* is returned. If `arrayIndex` is not a valid array index for `resource` then *cudaErrorInvalidValue* is returned. If `mipLevel` is not a valid mipmap level for `resource` then *cudaErrorInvalidValue* is returned. If `resource` is not mapped then *cudaErrorUnknown* is returned.

See also:

cudaGraphicsResourceGetMappedPointer, ::cuGraphicsSubResourceGetMappedArray

Note: Note that this function may also return error codes from previous, asynchronous launches.

Note: Note that this function may also return *cudaErrorInitializationError*, *cudaErrorInsufficientDriver* or *cudaErrorNoDevice* if this call tries to initialize internal CUDA RT state.

Note: Note that as specified by *cudaStreamAddCallback* no CUDA function may be called from callback. *cudaErrorNotPermitted* may, but is not guaranteed to, be returned as a diagnostic in such case.

Parameters

- **array** – - Returned array through which a subresource of resource may be accessed
- **resource** – - Mapped resource to access
- **arrayIndex** – - Array index for array textures or cubemap face index as defined by *cudaGraphicsCubeFace* for cubemap textures for the subresource to access
- **mipLevel** – - Mipmap level for the subresource to access

Returns

cudaSuccess, *cudaErrorInvalidValue*, *cudaErrorInvalidResourceHandle*, *cudaErrorUnknown*

__host__ *cudaError_t* **cudaGraphicsUnmapResources**(int count, *cudaGraphicsResource_t* *resources, *cudaStream_t* stream = 0)

Unmap graphics resources.

Unmaps the count graphics resources in resources.

Once unmapped, the resources in resources may not be accessed by CUDA until they are mapped again.

This function provides the synchronization guarantee that any CUDA work issued in stream before *cudaGraphicsUnmapResources()* will complete before any subsequently issued graphics work begins.

If resources contains any duplicate entries then *cudaErrorInvalidResourceHandle* is returned. If any of resources are not presently mapped for access by CUDA then *cudaErrorUnknown* is returned.

See also:

cudaGraphicsMapResources, ::cuGraphicsUnmapResources

Note: This function uses standard default stream semantics.

Note: Note that this function may also return error codes from previous, asynchronous launches.

Note: Note that this function may also return *cudaErrorInitializationError*, *cudaErrorInsufficientDriver* or *cudaErrorNoDevice* if this call tries to initialize internal CUDA RT state.

Note: Note that as specified by *cudaStreamAddCallback* no CUDA function may be called from callback. *cudaErrorNotPermitted* may, but is not guaranteed to, be returned as a diagnostic in such case.

Parameters

- **count** – - Number of resources to unmap
- **resources** – - Resources to unmap
- **stream** – - Stream for synchronization

Returns

cudaSuccess, *cudaErrorInvalidResourceHandle*, *cudaErrorUnknown*

__host__ *cudaError_t* **cudaGraphicsUnregisterResource**(*cudaGraphicsResource_t* resource)

Unregisters a graphics resource for access by CUDA.

Unregisters the graphics resource `resource` so it is not accessible by CUDA unless registered again.

If `resource` is invalid then *cudaErrorInvalidResourceHandle* is returned.

See also:

cudaGraphicsD3D9RegisterResource, *cudaGraphicsD3D10RegisterResource*, *cudaGraphicsD3D11RegisterResource*, *cudaGraphicsGLRegisterBuffer*, *cudaGraphicsGLRegisterImage*, ::cuGraphicsUnregisterResource

---

Note: Note that this function may also return error codes from previous, asynchronous launches.

Note: Note that this function may also return *cudaErrorInitializationError*, *cudaErrorInsufficientDriver* or *cudaErrorNoDevice* if this call tries to initialize internal CUDA RT state.

Note: Note that as specified by *cudaStreamAddCallback* no CUDA function may be called from callback. *cudaErrorNotPermitted* may, but is not guaranteed to, be returned as a diagnostic in such case.

Note: Use of the handle after this call is undefined behavior.

Parameters

resource – - Resource to unregister

Returns

cudaSuccess, *cudaErrorInvalidResourceHandle*, *cudaErrorUnknown*

6.22. Interactions with the CUDA Driver API

This section describes the interactions between the CUDA Driver API and the CUDA Runtime API.

6.22.1. Execution Contexts

The CUDA Runtime provides *cudaExecutionContext_t* as an abstraction over driver-level contexts—specifically, green contexts and the primary context.

There are two primary ways to obtain an execution context:

- *cudaDeviceGetExecutionCtx*: Returns the execution context that corresponds to the primary context of the specified device.
- *cudaGreenCtxCreate*: Creates a green context with the specified resources and returns an execution context.

Note: Developers should treat *cudaExecutionContext_t* as an opaque handle and avoid assumptions about its underlying representation. The CUDA Runtime does not provide a way to convert this handle into a ::CUcontext or ::CUgreenCtx.

6.22.2. Primary Context (aka Device Execution Context)

The primary context is the default execution context associated with a device in the Runtime. It can be obtained via a call to *cudaDeviceGetExecutionCtx()*. There is a one-to-one mapping between CUDA devices in the runtime and their primary contexts within a process.

From the CUDA Runtime's perspective, a device and its primary context are functionally synonymous.

Unless explicitly overridden, either by making a different context current via the Driver API (e.g., ::cuCtxSetCurrent()) or by using an explicit execution context handle, the Runtime will implicitly initialize and use the primary context for API calls as needed.

6.22.3. Initialization and Tear-Down

Unless an explicit execution context is specified (see "Execution Context Management" for APIs), CUDA Runtime API calls operate on the CUDA Driver ::CUcontext which is current to the calling host thread. If no ::CUcontext is current to the calling thread when a CUDA Runtime API call which requires an active context is made, then the primary context (device execution context) for a device will be selected, made current to the calling thread, and initialized. The context will be initialized using the parameters specified by the CUDA Runtime API functions *cudaSetDeviceFlags()*, *cudaD3D9SetDirect3DDevice()*, *cudaD3D10SetDirect3DDevice()*, *cudaD3D11SetDirect3DDevice()*, *cudaGLSetGLDevice()*, and *cudaVDPAUSetVDPAUDevice()*. Note that these functions will fail with *cudaErrorSetOnActiveProcess* if they are called when the primary context for the specified device has already been initialized, except for *cudaSetDeviceFlags()* which will simply overwrite the previous settings.

The function *cudaInitDevice()* ensures that the primary context is initialized for the requested device but does not make it current to the calling thread.

The function *cudaSetDevice()* initializes the primary context for the specified device and makes it current to the calling thread by calling ::cuCtxSetCurrent().

Primary contexts will remain active until they are explicitly deinitialized using *cudaDeviceReset()*. The function *cudaDeviceReset()* will deinitialize the primary context for the calling thread's current device immediately. The context will remain current to all of the threads that it was current to. The next CUDA Runtime API call on any thread which requires an active context will trigger the reinitialization of that device's primary context.

Note that primary contexts are shared resources. It is recommended that the primary context not be reset except just before exit or to recover from an unspecified launch failure.

6.22.4. CUcontext Interoperability

Note that the use of multiple ::CUcontext s per device within a single process will substantially degrade performance and is strongly discouraged. Instead, it is highly recommended to either use execution contexts *cudaExecutionContext_t* or the implicit one-to-one device-to-primary context mapping for the process provided by the CUDA Runtime API.

If a non-primary ::CUcontext created by the CUDA Driver API is current to a thread then the CUDA Runtime API calls to that thread will operate on that ::CUcontext, with some exceptions listed below. Interoperability between data types is discussed in the following sections.

The function *cudaDeviceEnablePeerAccess()* and the rest of the peer access API may not be called when a non-primary CUcontext is current. To use the peer access APIs with a context created using the CUDA Driver API, it is necessary that the CUDA Driver API be used to access these features.

All CUDA Runtime API state (e.g, global variables' addresses and values) travels with its underlying ::CUcontext. In particular, if a ::CUcontext is moved from one thread to another then all CUDA Runtime API state will move to that thread as well.

Please note that attaching to legacy CUcontext (those with a version of 3010 as returned by ::cuCtxGetApiVersion()) is not possible. The CUDA Runtime will return *cudaErrorIncompatibleDriverContext* in such cases.

6.22.5. Interactions between CUstream and cudaStream_t

The types ::CUstream and *cudaStream_t* are identical and may be used interchangeably.

6.22.6. Interactions between CUevent and cudaEvent_t

The types ::CUevent and *cudaEvent_t* are identical and may be used interchangeably.

6.22.7. Interactions between CUarray and cudaArray_t

The types ::CUarray and struct ::cudaArray * represent the same data type and may be used interchangeably by casting the two types between each other.

In order to use a ::CUarray in a CUDA Runtime API function which takes a struct ::cudaArray *, it is necessary to explicitly cast the ::CUarray to a struct ::cudaArray *.

In order to use a struct ::cudaArray * in a CUDA Driver API function which takes a ::CUarray, it is necessary to explicitly cast the struct ::cudaArray * to a ::CUarray .

6.22.8. Interactions between CUgraphicsResource and cudaGraphicsResource_t

The types ::CUgraphicsResource and *cudaGraphicsResource_t* represent the same data type and may be used interchangeably by casting the two types between each other.

In order to use a ::CUgraphicsResource in a CUDA Runtime API function which takes a *cudaGraphicsResource_t*, it is necessary to explicitly cast the ::CUgraphicsResource to a *cudaGraphicsResource_t*.

In order to use a *cudaGraphicsResource_t* in a CUDA Driver API function which takes a ::CUgraphicsResource, it is necessary to explicitly cast the *cudaGraphicsResource_t* to a ::CUgraphicsResource.

6.22.9. Interactions between CUtexObject and cudaTextureObject_t

The types ::CUtexObject and *cudaTextureObject_t* represent the same data type and may be used interchangeably by casting the two types between each other.

In order to use a ::CUtexObject in a CUDA Runtime API function which takes a *cudaTextureObject_t*, it is necessary to explicitly cast the ::CUtexObject to a *cudaTextureObject_t*.

In order to use a *cudaTextureObject_t* in a CUDA Driver API function which takes a ::CUtexObject, it is necessary to explicitly cast the *cudaTextureObject_t* to a ::CUtexObject.

6.22.10. Interactions between CUsurfObject and cudaSurfaceObject_t

The types ::CUsurfObject and *cudaSurfaceObject_t* represent the same data type and may be used interchangeably by casting the two types between each other.

In order to use a ::CUsurfObject in a CUDA Runtime API function which takes a *cudaSurfaceObject_t*, it is necessary to explicitly cast the ::CUsurfObject to a *cudaSurfaceObject_t*.

In order to use a *cudaSurfaceObject_t* in a CUDA Driver API function which takes a ::CUsurfObject, it is necessary to explicitly cast the *cudaSurfaceObject_t* to a ::CUsurfObject.

6.22.11. Interactions between CUfunction and cudaFunction_t

The types ::CUfunction and *cudaFunction_t* represent the same data type and may be used interchangeably by casting the two types between each other.

In order to use a *cudaFunction_t* in a CUDA Driver API function which takes a ::CUfunction, it is necessary to explicitly cast the *cudaFunction_t* to a ::CUfunction.

6.22.12. Interactions between CUkernel and cudaKernel_t

The types ::CUkernel and *cudaKernel_t* represent the same data type and may be used interchangeably by casting the two types between each other.

In order to use a *cudaKernel_t* in a CUDA Driver API function which takes a ::CUkernel, it is necessary to explicitly cast the *cudaKernel_t* to a ::CUkernel.

Functions

__host__ cudaError_t *cudaGetFuncBySymbol*(cudaFunction_t *functionPtr, const void *symbolPtr)
Get pointer to device entry function that matches entry function `symbolPtr` .

__host__ cudaError_t *cudaGetKernel*(cudaKernel_t *kernelPtr, const void *entryFuncAddr)
Get pointer to device kernel that matches entry function `entryFuncAddr` .

6.22.13. Functions

__host__ *cudaError_t* **cudaGetFuncBySymbol**(*cudaFunction_t* *functionPtr, const void *symbolPtr)

Get pointer to device entry function that matches entry function `symbolPtr`.

Returns in `functionPtr` the device entry function corresponding to the symbol `symbolPtr`.

Parameters

- **functionPtr** – - Returns the device entry function
- **symbolPtr** – - Pointer to device entry function to search for

Returns
cudaSuccess

__host__ *cudaError_t* **cudaGetKernel**(*cudaKernel_t* *kernelPtr, const void *entryFuncAddr)

Get pointer to device kernel that matches entry function `entryFuncAddr`.

Returns in `kernelPtr` the device kernel corresponding to the entry function `entryFuncAddr`.

Note that it is possible that there are multiple symbols belonging to different translation units with the same `entryFuncAddr` registered with this CUDA Runtime and so the order which the translation units are loaded and registered with the CUDA Runtime can lead to differing return pointers in `kernelPtr` . Suggested methods of ensuring uniqueness are to limit visibility of

global device functions by using static or hidden visibility attribute in the respective translation units.

See also:

cudaGetKernel (C++ API)

Parameters

- **kernelPtr** – - Returns the device kernel
- **entryFuncAddr** – - Address of device entry function to search kernel for

Returns

cudaSuccess

6.23. Library Management

This section describes the library management functions of the CUDA runtime application programming interface.

Functions

__host__ cudaError_t *cudaKernelSetAttributeForDevice*(cudaKernel_t kernel, enum cudaFuncAttribute attr, int value, int device)
Sets information about a kernel.

__host__ cudaError_t *cudaLibraryEnumerateKernels*(cudaKernel_t *kernels, unsigned int numKernels, cudaLibrary_t lib)
Retrieve the kernel handles within a library.

__host__ cudaError_t *cudaLibraryGetGlobal*(void **dptr, size_t *bytes, cudaLibrary_t library, const char *name)
Returns a global device pointer.

__host__ cudaError_t *cudaLibraryGetKernel*(cudaKernel_t *pKernel, cudaLibrary_t library, const char *name)
Returns a kernel handle.

__host__ cudaError_t *cudaLibraryGetKernelCount*(unsigned int *count, cudaLibrary_t lib)
Returns the number of kernels within a library.

__host__ cudaError_t *cudaLibraryGetManaged*(void **dptr, size_t *bytes, cudaLibrary_t library, const char *name)
Returns a pointer to managed memory.

__host__ cudaError_t *cudaLibraryGetUnifiedFunction*(void **fptr, cudaLibrary_t library, const char *symbol)
Returns a pointer to a unified function.

__host__ cudaError_t *cudaLibraryLoadData*(cudaLibrary_t *library, const void *code, enum cudaJitOption *jitOptions, void **jitOptionsValues, unsigned int numJitOptions, enum cudaLibraryOption *libraryOptions, void **libraryOptionValues, unsigned int numLibraryOptions)
Load a library with specified code and options.

__host__ cudaError_t *cudaLibraryLoadFromFile*(cudaLibrary_t *library, const char *fileName, enum cudaJitOption *jitOptions, void **jitOptionsValues, unsigned int numJitOptions, enum cudaLibraryOption *libraryOptions, void **libraryOptionValues, unsigned int numLibraryOptions)
Load a library with specified file and options.

__host__ cudaError_t *cudaLibraryUnload*(cudaLibrary_t library)
Unloads a library.

6.23.1. Functions

__host__ *cudaError_t* **cudaKernelSetAttributeForDevice**(*cudaKernel_t* kernel, enum *cudaFuncAttribute* attr, int value, int device)

Sets information about a kernel.

This call sets the value of a specified attribute `attr` on the kernel `kernel` for the requested device `device` to an integer value specified by `value`. This function returns *cudaSuccess* if the new value of the attribute could be successfully set. If the set fails, this call will return an error. Not all attributes can have values set. Attempting to set a value on a read-only attribute will result in an error (*cudaErrorInvalidValue*)

Note that attributes set using *cudaFuncSetAttribute()* will override the attribute set by this API irrespective of whether the call to *cudaFuncSetAttribute()* is made before or after this API call. Because of this and the stricter locking requirements mentioned below it is suggested that this call be used during the initialization path and not on each thread accessing `kernel` such as on kernel launches or on the critical path.

Valid values for `attr` are:

- *cudaFuncAttributeMaxDynamicSharedMemorySize* - The requested maximum size in bytes of dynamically-allocated shared memory. The sum of this value and the function attribute ::sharedSizeBytes cannot exceed the device attribute *cudaDevAttrMaxSharedMemoryPerBlockOptin*. The maximal size of requestable dynamic shared memory may differ by GPU architecture.
- *cudaFuncAttributePreferredSharedMemoryCarveout* - On devices where the L1 cache and shared memory use the same hardware resources, this sets the shared memory carveout preference, in percent of the total shared memory. See *cudaDevAttrMaxSharedMemoryPerMultiprocessor*. This is only a hint, and the driver can choose a different ratio if required to execute the function.
- *cudaFuncAttributeRequiredClusterWidth*: The required cluster width in blocks. The width, height, and depth values must either all be 0 or all be positive. The validity of the cluster dimensions is checked at launch time. If the value is set during compile time, it cannot be set at runtime. Setting it at runtime will return cudaErrorNotPermitted.
- *cudaFuncAttributeRequiredClusterHeight*: The required cluster height in blocks. The width, height, and depth values must either all be 0 or all be positive. The validity of the cluster dimensions is checked at launch time. If the value is set during compile time, it cannot be set at runtime. Setting it at runtime will return cudaErrorNotPermitted.
- *cudaFuncAttributeRequiredClusterDepth*: The required cluster depth in blocks. The width, height, and depth values must either all be 0 or all be positive. The validity of the cluster dimensions is checked at launch time. If the value is set during compile time, it cannot be set at runtime. Setting it at runtime will return cudaErrorNotPermitted.

- *cudaFuncAttributeNonPortableClusterSizeAllowed*: Indicates whether the function can be launched with non-portable cluster size. 1 is allowed, 0 is disallowed.
- *cudaFuncAttributeClusterSchedulingPolicyPreference*: The block scheduling policy of a function. The value type is cudaClusterSchedulingPolicy.

See also:

cudaLibraryLoadData, *cudaLibraryLoadFromFile*, *cudaLibraryUnload*, *cudaLibraryGetKernel*, *cudaLaunchKernel*, *cudaFuncSetAttribute*, ::cuKernelSetAttribute

---

Note: The API has stricter locking requirements in comparison to its legacy counterpart *cudaFuncSetAttribute()* due to device-wide semantics. If multiple threads are trying to set the same attribute on the same device simultaneously, the attribute setting will depend on the interleavings chosen by the OS scheduler and memory consistency.

Parameters

- **kernel** – - Kernel to set attribute of
- **attr** – - Attribute requested
- **value** – - Value to set
- **device** – - Device to set attribute of

Returns

cudaSuccess, *cudaErrorInvalidDeviceFunction*, *cudaErrorInvalidValue*

__host__ *cudaError_t* **cudaLibraryEnumerateKernels**(*cudaKernel_t* *kernels, unsigned int numKernels, *cudaLibrary_t* lib)

Retrieve the kernel handles within a library.

Returns in `kernels` a maximum number of `numKernels` kernel handles within `lib`. The returned kernel handle becomes invalid when the library is unloaded.

See also:

cudaLibraryGetKernelCount, ::cuLibraryEnumerateKernels

Parameters

- **kernels** – - Buffer where the kernel handles are returned to
- **numKernels** – - Maximum number of kernel handles may be returned to the buffer
- **lib** – - Library to query from

Returns

cudaSuccess, *cudaErrorCudartUnloading*, *cudaErrorInitializationError*, *cudaErrorInvalidValue*, *cudaErrorInvalidResourceHandle*

__host__ *cudaError_t* **cudaLibraryGetGlobal**(void **dptr, size_t *bytes, *cudaLibrary_t* library, const char *name)

Returns a global device pointer.

Returns in `*dptr` and `*bytes` the base pointer and size of the global with name `name` for the requested library `library` and the current device. If no global for the requested name `name` exists, the call returns *cudaErrorSymbolNotFound*. One of the parameters `dptr` or `bytes` (not both) can be NULL in which case it is ignored. The returned `dptr` cannot be passed to the Symbol APIs such as *cudaMemcpyToSymbol*, *cudaMemcpyFromSymbol*, *cudaGetSymbolAddress*, or *cudaGetSymbolSize*.

See also:

cudaLibraryLoadData, *cudaLibraryLoadFromFile*, *cudaLibraryUnload*, *cudaLibraryGetManaged*, ::cuLibraryGetGlobal

Parameters

- **dptr** – - Returned global device pointer for the requested library
- **bytes** – - Returned global size in bytes
- **library** – - Library to retrieve global from
- **name** – - Name of global to retrieve

Returns

cudaSuccess, *cudaErrorCudartUnloading*, *cudaErrorInitializationError*, *cudaErrorInvalidValue*, *cudaErrorInvalidResourceHandle*, *cudaErrorSymbolNotFound* *cudaErrorDeviceUninitialized*, *cudaErrorContextIsDestroyed*

__host__ *cudaError_t* **cudaLibraryGetKernel**(*cudaKernel_t* *pKernel, *cudaLibrary_t* library, const char *name)

Returns a kernel handle.

Returns in `pKernel` the handle of the kernel with name `name` located in library `library`. If kernel handle is not found, the call returns *cudaErrorSymbolNotFound*.

See also:

cudaLibraryLoadData, *cudaLibraryLoadFromFile*, *cudaLibraryUnload*, ::cuLibraryGetKernel

Parameters

- **pKernel** – - Returned kernel handle
- **library** – - Library to retrieve kernel from
- **name** – - Name of kernel to retrieve

Returns

cudaSuccess, *cudaErrorCudartUnloading*, *cudaErrorInitializationError*, *cudaErrorInvalidValue*, *cudaErrorInvalidResourceHandle*, *cudaErrorSymbolNotFound*

__host__ *cudaError_t* **cudaLibraryGetKernelCount**(unsigned int *count, *cudaLibrary_t* lib)

Returns the number of kernels within a library.

Returns in `count` the number of kernels in `lib`.

See also:

cudaLibraryEnumerateKernels, *cudaLibraryLoadFromFile*, *cudaLibraryLoadData*, ::cuLibraryGetKernelCount

Parameters

- **count** – - Number of kernels found within the library
- **lib** – - Library to query

Returns

cudaSuccess, *cudaErrorCudartUnloading*, *cudaErrorInitializationError*, *cudaErrorInvalidValue*, *cudaErrorInvalidResourceHandle*

__host__ *cudaError_t* **cudaLibraryGetManaged**(void **dptr, size_t *bytes, *cudaLibrary_t* library, const char *name)

Returns a pointer to managed memory.

Returns in `*dptr` and `*bytes` the base pointer and size of the managed memory with name `name` for the requested library `library`. If no managed memory with the requested name `name` exists, the call returns *cudaErrorSymbolNotFound*. One of the parameters `dptr` or `bytes` (not both) can be NULL in which case it is ignored. Note that managed memory for library `library` is shared across devices and is registered when the library is loaded. The returned `dptr` cannot be passed to the Symbol APIs such as *cudaMemcpyToSymbol*, *cudaMemcpyFromSymbol*, *cudaGetSymbolAddress*, or *cudaGetSymbolSize*.

See also:

cudaLibraryLoadData, *cudaLibraryLoadFromFile*, *cudaLibraryUnload*, *cudaLibraryGetGlobal*, ::cuLibraryGetManaged

Parameters

- **dptr** – - Returned pointer to the managed memory
- **bytes** – - Returned memory size in bytes
- **library** – - Library to retrieve managed memory from
- **name** – - Name of managed memory to retrieve

Returns

cudaSuccess, *cudaErrorCudartUnloading*, *cudaErrorInitializationError*, *cudaErrorInvalidValue*, *cudaErrorInvalidResourceHandle*, *cudaErrorSymbolNotFound*

__host__ *cudaError_t* **cudaLibraryGetUnifiedFunction**(void **fptr, *cudaLibrary_t* library, const char *symbol)

Returns a pointer to a unified function.

Returns in `*fptr` the function pointer to a unified function denoted by `symbol`. If no unified function with name `symbol` exists, the call returns *cudaErrorSymbolNotFound*. If there is no device with attribute *cudaDeviceProp::unifiedFunctionPointers* present in the system, the call may return *cudaErrorSymbolNotFound*.

See also:

cudaLibraryLoadData, *cudaLibraryLoadFromFile*, *cudaLibraryUnload*, ::cuLibraryGetUnifiedFunction

Parameters

- **fptr** – - Returned pointer to a unified function
- **library** – - Library to retrieve function pointer memory from
- **symbol** – - Name of function pointer to retrieve

Returns

cudaSuccess, *cudaErrorCudartUnloading*, *cudaErrorInitializationError*, *cudaErrorInvalidValue*, *cudaErrorInvalidResourceHandle*, *cudaErrorSymbolNotFound*

__host__ *cudaError_t* **cudaLibraryLoadData**(*cudaLibrary_t* *library, const void *code, enum *cudaJitOption* *jitOptions, void **jitOptionsValues, unsigned int numJitOptions, enum *cudaLibraryOption* *libraryOptions, void **libraryOptionValues, unsigned int numLibraryOptions)

Load a library with specified code and options.

Takes a pointer `code` and loads the corresponding library `library` based on the application defined library loading mode:

- If module loading is set to EAGER, via the environment variables described in "Module loading", `library` is loaded eagerly into all contexts at the time of the call and future contexts at the time of creation until the library is unloaded with *cudaLibraryUnload()*.
- If the environment variables are set to LAZY, `library` is not immediately loaded onto all existent contexts and will only be loaded when a function is needed for that context, such as a kernel launch.

These environment variables are described in the CUDA programming guide under the "CUDA environment variables" section.

The `code` may be a *cubin* or *fatbin* as output by **nvcc**, or a NULL-terminated *PTX*, either as output by **nvcc** or hand-written, or *Tile* IR data. A fatbin should also contain relocatable code when doing separate compilation. Please also see the documentation for nvrtc (https://docs.nvidia.com/cuda/nvrtc/index.html), nvjitlink (https://docs.nvidia.com/cuda/nvjitlink/index.html), and nvfatbin (https://docs.nvidia.com/cuda/nvfatbin/index.html) for more information on generating loadable code at runtime.

Options are passed as an array via `jitOptions` and any corresponding parameters are passed in `jitOptionsValues`. The number of total JIT options is supplied via `numJitOptions`. Any outputs will be returned via `jitOptionsValues`.

Library load options are passed as an array via `libraryOptions` and any corresponding parameters are passed in `libraryOptionValues`. The number of total library load options is supplied via `numLibraryOptions`.

See also:

cudaLibraryLoadFromFile, *cudaLibraryUnload*, ::cuLibraryLoadData

Parameters

- **library** – - Returned library
- **code** – - Code to load
- **jitOptions** – - Options for JIT
- **jitOptionsValues** – - Option values for JIT
- **numJitOptions** – - Number of options
- **libraryOptions** – - Options for loading
- **libraryOptionValues** – - Option values for loading
- **numLibraryOptions** – - Number of options for loading

Returns

cudaSuccess, *cudaErrorInvalidValue*, *cudaErrorMemoryAllocation*, *cudaErrorInitializationError*, *cudaErrorCudartUnloading*, *cudaErrorInvalidPtx*, *cudaErrorUnsupportedPtxVersion*, *cudaErrorNoKernelImageForDevice*, *cudaErrorSharedObjectSymbolNotFound*, *cudaErrorSharedObjectInitFailed*, *cudaErrorJitCompilerNotFound*

__host__ *cudaError_t* **cudaLibraryLoadFromFile**(*cudaLibrary_t* *library, const char *fileName, enum *cudaJitOption* *jitOptions, void **jitOptionsValues, unsigned int numJitOptions, enum *cudaLibraryOption* *libraryOptions, void **libraryOptionValues, unsigned int numLibraryOptions)

Load a library with specified file and options.

Takes a pointer `code` and loads the corresponding library `library` based on the application defined library loading mode:

- If module loading is set to EAGER, via the environment variables described in "Module loading", `library` is loaded eagerly into all contexts at the time of the call and future contexts at the time of creation until the library is unloaded with *cudaLibraryUnload()*.
- If the environment variables are set to LAZY, `library` is not immediately loaded onto all existent contexts and will only be loaded when a function is needed for that context, such as a kernel launch.

These environment variables are described in the CUDA programming guide under the "CUDA environment variables" section.

The file should be a *cubin* file as output by **nvcc**, or a *PTX* file either as output by **nvcc** or hand-written, or a *fatbin* file as output by **nvcc** or hand-written, or *Tile* IR file. A fatbin should also contain relocatable code when doing separate compilation. Please also see the documentation for nvrtc (https://docs.nvidia.com/cuda/nvrtc/index.html), nvjitlink (https://docs.nvidia.com/cuda/nvjitlink/index.html), and nvfatbin (https://docs.nvidia.com/cuda/nvfatbin/index.html) for more information on generating loadable code at runtime.

Options are passed as an array via `jitOptions` and any corresponding parameters are passed in `jitOptionsValues`. The number of total options is supplied via `numJitOptions`. Any outputs will be returned via `jitOptionsValues`.

Library load options are passed as an array via `libraryOptions` and any corresponding parameters are passed in `libraryOptionValues`. The number of total library load options is supplied via `numLibraryOptions`.

See also:

cudaLibraryLoadData, *cudaLibraryUnload*, ::cuLibraryLoadFromFile

Parameters

- **`library`** – - Returned library
- **`fileName`** – - File to load from
- **`jitOptions`** – - Options for JIT
- **`jitOptionsValues`** – - Option values for JIT
- **`numJitOptions`** – - Number of options
- **`libraryOptions`** – - Options for loading
- **`libraryOptionValues`** – - Option values for loading
- **`numLibraryOptions`** – - Number of options for loading

Returns

cudaSuccess, *cudaErrorInvalidValue*, *cudaErrorMemoryAllocation*, *cudaErrorInitializationError*, *cudaErrorCudartUnloading*, *cudaErrorInvalidPtx*, *cudaErrorUnsupportedPtxVersion*, *cudaErrorNoKernelImageForDevice*, *cudaErrorSharedObjectSymbolNotFound*, *cudaErrorSharedObjectInitFailed*, *cudaErrorJitCompilerNotFound*

__host__ *cudaError_t* **`cudaLibraryUnload`**(*cudaLibrary_t* library)

Unloads a library.

Unloads the library specified with `library`

See also:

cudaLibraryLoadData, *cudaLibraryLoadFromFile*, ::cuLibraryUnload

Parameters

`library` – - Library to unload

Returns

cudaSuccess, *cudaErrorCudartUnloading*, *cudaErrorInitializationError*, *cudaErrorInvalidValue*

6.24. Logical Endpoint

This section describes the functions for Logical Endpoint operations management.

Functions

__device__ cudaError_t *cudaFabricOpErrorStatusCount*(void *status, cudaFabricOpStatusSource statusSource, unsigned int *count)
Get the number of errors for a fabric operation.

__device__ cudaError_t *cudaFabricOpErrorStatusGet*(void *status, cudaFabricOpStatusSource statusSource, unsigned int statusIndex, cudaFabricOpStatusInfo *statusInfo)
Get the error status of a fabric operation.

6.24.1. Functions

__device__ *cudaError_t* **cudaFabricOpErrorStatusCount**(void *status, *cudaFabricOpStatusSource* statusSource, unsigned int *count)

Get the number of errors for a fabric operation.

Given the pointer to the `status` field obtained from fabric operation completion object, return the total count of errors.

Parameters

- **`status`** – - The pointer to the status field obtained from fabric operation completion object
- **`statusSource`** – - The type of the fabric operation completion object
- **`count`** – - The number of errors in the decoded status field

Returns
cudaSuccess, cudaErrorInvalidValue, cudaErrorNotSupported

__device__ *cudaError_t* **cudaFabricOpErrorStatusGet**(void *status, *cudaFabricOpStatusSource* statusSource, unsigned int statusIndex, *cudaFabricOpStatusInfo* *statusInfo)

Get the error status of a fabric operation.

Given a pointer to the `status` field obtained from a fabric operation completion object, retrieves the error at index `statusIndex` in the decoded `status` field and writes it to `statusInfo`.

`statusIndex` must be in the range [0, statusCount), where statusCount is the value returned via the `count` parameter of *cudaFabricOpErrorStatusCount*.

Parameters

- **`status`** – - Pointer to the status field from the fabric operation completion object
- **`statusSource`** – - The type of the fabric operation completion object

- **statusIndex** – - Index of the error to retrieve (0-based; selects which of the decoded errors to return)
- **statusInfo** – - Pointer to `cudaFabricOpStatusInfo` where the decoded error information is returned

Returns
cudaSuccess, cudaErrorInvalidValue, cudaErrorNotSupported

6.25. Memory Management

This section describes the memory management functions of the CUDA runtime application programming interface.

Some functions have overloaded C++ API template versions documented separately in the *C++ API Routines* module.

Functions

__host__ cudaError_t *cudaArrayGetInfo*(struct cudaChannelFormatDesc *desc, struct cudaExtent *extent, unsigned int *flags, cudaArray_t array)
Gets info about the specified cudaArray.

__host__ cudaError_t *cudaArrayGetMemoryRequirements*(struct cudaArrayMemoryRequirements *memoryRequirements, cudaArray_t array, int device)
Returns the memory requirements of a CUDA array.

__host__ cudaError_t *cudaArrayGetPlane*(cudaArray_t *pPlaneArray, cudaArray_t hArray, unsigned int planeIdx)
Gets a CUDA array plane from a CUDA array.

__host__ cudaError_t *cudaArrayGetSparseProperties*(struct cudaArraySparseProperties *sparseProperties, cudaArray_t array)
Returns the layout properties of a sparse CUDA array.

__host__ cudaError_t *cudaFree*(void *devPtr)
Frees memory on the device.

__host__ cudaError_t *cudaFreeArray*(cudaArray_t array)
Frees an array on the device.

__host__ cudaError_t *cudaFreeHost*(void *ptr)
Frees page-locked memory.

__host__ cudaError_t *cudaFreeMipmappedArray*(cudaMipmappedArray_t mipmappedArray)
Frees a mipmapped array on the device.

__host__ cudaError_t *cudaGetMipmappedArrayLevel*(cudaArray_t *levelArray, cudaMipmappedArray_const_t mipmappedArray, unsigned int level)
Gets a mipmap level of a CUDA mipmapped array.

__host__ cudaError_t *cudaGetSymbolAddress*(void **devPtr, const void *symbol)
Finds the address associated with a CUDA symbol.

__host__ cudaError_t *cudaGetSymbolSize*(size_t *size, const void *symbol)
Finds the size of the object associated with a CUDA symbol.

__host__ cudaError_t *cudaHostAlloc*(void **pHost, size_t size, unsigned int flags)
Allocates page-locked memory on the host.

__host__ cudaError_t *cudaHostGetDevicePointer*(void **pDevice, void *pHost, unsigned int flags)
Passes back device pointer of mapped host memory allocated by cudaHostAlloc or registered by cudaHostRegister.

__host__ cudaError_t *cudaHostGetFlags*(unsigned int *pFlags, void *pHost)
Passes back flags used to allocate pinned host memory allocated by cudaHostAlloc.

__host__ cudaError_t *cudaHostRegister*(void *ptr, size_t size, unsigned int flags)
Registers an existing host memory range for use by CUDA.

__host__ cudaError_t *cudaHostUnregister*(void *ptr)
Unregisters a memory range that was registered with cudaHostRegister.

__host__ cudaError_t *cudaMalloc*(void **devPtr, size_t size)
Allocate memory on the device.

__host__ cudaError_t *cudaMalloc3D*(struct cudaPitchedPtr *pitchedDevPtr, struct cudaExtent extent)
Allocates logical 1D, 2D, or 3D memory objects on the device.

__host__ cudaError_t *cudaMalloc3DArray*(cudaArray_t *array, const struct cudaChannelFormatDesc *desc, struct cudaExtent extent, unsigned int flags=0)
Allocate an array on the device.

__host__ cudaError_t *cudaMallocArray*(cudaArray_t *array, const struct cudaChannelFormatDesc *desc, size_t width, size_t height=0, unsigned int flags=0)
Allocate an array on the device.

__host__ cudaError_t *cudaMallocHost*(void **ptr, size_t size)
Allocates page-locked memory on the host.

__host__ cudaError_t *cudaMallocManaged*(void **devPtr, size_t size, unsigned int flags=0x01)
Allocates memory that will be automatically managed by the Unified Memory system.

__host__ cudaError_t *cudaMallocMipmappedArray*(cudaMipmappedArray_t *mipmappedArray, const struct cudaChannelFormatDesc *desc, struct cudaExtent extent, unsigned int numLevels, unsigned int flags=0)
Allocate a mipmapped array on the device.

__host__ cudaError_t *cudaMallocPitch*(void **devPtr, size_t *pitch, size_t width, size_t height)
Allocates pitched memory on the device.

__host__ cudaError_t *cudaMemAdvise*(const void *devPtr, size_t count, enum cudaMemoryAdvise advice, struct cudaMemLocation location)
Advise about the usage of a given memory range.

__host__ cudaError_t *cudaMemDiscardAndPrefetchBatchAsync*(void **dptrs, size_t *sizes, size_t count, struct cudaMemLocation *prefetchLocs, size_t *prefetchLocIdxs, size_t numPrefetchLocs, unsigned long long flags, cudaStream_t stream)
Performs a batch of memory discards and prefetches asynchronously.

__host__ cudaError_t *cudaMemDiscardBatchAsync*(void **dptrs, size_t *sizes, size_t count, unsigned long long flags, cudaStream_t stream)
Performs a batch of memory discards asynchronously.

__host__ cudaError_t *cudaMemGetInfo*(size_t *free, size_t *total)
Gets free and total device memory.

__host__ cudaError_t *cudaMemGetLocationInfo*(void *devPtr, size_t size, size_t summaryGranularity, size_t samplingGranularity, struct cudaMemLocation *location_out)
Gets location information for a memory address range.

__host__ cudaError_t *cudaMemPrefetchAsync*(const void *devPtr, size_t count, struct cudaMemLocation location, unsigned int flags, cudaStream_t stream=0)
Prefetches memory to the specified destination location.

__host__ cudaError_t *cudaMemPrefetchBatchAsync*(void **dptrs, size_t *sizes, size_t count, struct cudaMemLocation *prefetchLocs, size_t *prefetchLocIdxs, size_t numPrefetchLocs, unsigned long long flags, cudaStream_t stream)
Performs a batch of memory prefetches asynchronously.

__host__ cudaError_t *cudaMemRangeGetAttribute*(void *data, size_t dataSize, enum cudaMemRangeAttribute attribute, const void *devPtr, size_t count)
Query an attribute of a given memory range.

__host__ cudaError_t *cudaMemRangeGetAttributes*(void **data, size_t *dataSizes, enum cudaMemRangeAttribute *attributes, size_t numAttributes, const void *devPtr, size_t count)
Query attributes of a given memory range.

__host__ cudaError_t *cudaMemcpy*(void *dst, const void *src, size_t count, enum cudaMemcpyKind kind)
Copies data between host and device.

__host__ cudaError_t *cudaMemcpy2D*(void *dst, size_t dpitch, const void *src, size_t spitch, size_t width, size_t height, enum cudaMemcpyKind kind)
Copies data between host and device.

__host__ cudaError_t *cudaMemcpy2DArrayToArray*(cudaArray_t dst, size_t wOffsetDst, size_t hOffsetDst, cudaArray_const_t src, size_t wOffsetSrc, size_t hOffsetSrc, size_t width, size_t height, enum cudaMemcpyKind kind=cudaMemcpyDeviceToDevice)
Copies data between host and device.

__host__ cudaError_t *cudaMemcpy2DAsync*(void *dst, size_t dpitch, const void *src, size_t spitch, size_t width, size_t height, enum cudaMemcpyKind kind, cudaStream_t stream=0)
Copies data between host and device.

__host__ cudaError_t *cudaMemcpy2DFromArray*(void *dst, size_t dpitch, cudaArray_const_t src, size_t wOffset, size_t hOffset, size_t width, size_t height, enum cudaMemcpyKind kind)
Copies data between host and device.

__host__ cudaError_t *cudaMemcpy2DFromArrayAsync*(void *dst, size_t dpitch, cudaArray_const_t src, size_t wOffset, size_t hOffset, size_t width, size_t height, enum cudaMemcpyKind kind, cudaStream_t stream=0)
Copies data between host and device.

__host__ cudaError_t *cudaMemcpy2DToArray*(cudaArray_t dst, size_t wOffset, size_t hOffset, const void *src, size_t spitch, size_t width, size_t height, enum cudaMemcpyKind kind)
Copies data between host and device.

__host__ cudaError_t *cudaMemcpy2DToArrayAsync*(cudaArray_t dst, size_t wOffset, size_t hOffset, const void *src, size_t spitch, size_t width, size_t height, enum cudaMemcpyKind kind, cudaStream_t stream=0)
Copies data between host and device.

__host__ cudaError_t *cudaMemcpy3D*(const struct cudaMemcpy3DParms *p)
Copies data between 3D objects.

__host__ cudaError_t *cudaMemcpy3DAsync*(const struct cudaMemcpy3DParms *p, cudaStream_t stream=0)

Copies data between 3D objects.

__host__ cudaError_t *cudaMemcpy3DBatchAsync*(size_t numOps, struct cudaMemcpy3DBatchOp *opList, unsigned long long flags, cudaStream_t stream)
Performs a batch of 3D memory copies asynchronously.

__host__ cudaError_t *cudaMemcpy3DPeer*(const struct cudaMemcpy3DPeerParms *p)
Copies memory between devices.

__host__ cudaError_t *cudaMemcpy3DPeerAsync*(const struct cudaMemcpy3DPeerParms *p, cudaStream_t stream=0)
Copies memory between devices asynchronously.

__host__ cudaError_t *cudaMemcpy3DWithAttributesAsync*(struct cudaMemcpy3DBatchOp *op, unsigned long long flags, cudaStream_t stream)
Performs 3D asynchronous memory copy with the specified attributes.

__host__ cudaError_t *cudaMemcpyAsync*(void *dst, const void *src, size_t count, enum cudaMemcpyKind kind, cudaStream_t stream=0)
Copies data between host and device.

__host__ cudaError_t *cudaMemcpyBatchAsync*(void *const *dsts, const void *const *srcs, const size_t *sizes, size_t count, struct cudaMemcpyAttributes *attrs, size_t *attrsIdxs, size_t numAttrs, cudaStream_t stream)
Performs a batch of memory copies asynchronously.

__host__ cudaError_t *cudaMemcpyFromSymbol*(void *dst, const void *symbol, size_t count, size_t offset=0, enum cudaMemcpyKind kind=cudaMemcpyDeviceToHost)
Copies data from the given symbol on the device.

__host__ cudaError_t *cudaMemcpyFromSymbolAsync*(void *dst, const void *symbol, size_t count, size_t offset, enum cudaMemcpyKind kind, cudaStream_t stream=0)
Copies data from the given symbol on the device.

__host__ cudaError_t *cudaMemcpyPeer*(void *dst, int dstDevice, const void *src, int srcDevice, size_t count)
Copies memory between two devices.

__host__ cudaError_t *cudaMemcpyPeerAsync*(void *dst, int dstDevice, const void *src, int srcDevice, size_t count, cudaStream_t stream=0)
Copies memory between two devices asynchronously.

__host__ cudaError_t *cudaMemcpyToSymbol*(const void *symbol, const void *src, size_t count, size_t offset=0, enum cudaMemcpyKind kind=cudaMemcpyHostToDevice)
Copies data to the given symbol on the device.

__host__ cudaError_t *cudaMemcpyToSymbolAsync*(const void *symbol, const void *src, size_t count, size_t offset, enum cudaMemcpyKind kind, cudaStream_t stream=0)
Copies data to the given symbol on the device.

__host__ cudaError_t *cudaMemcpyWithAttributesAsync*(void *dst, const void *src, size_t size, struct cudaMemcpyAttributes *attr, cudaStream_t stream)
Performs asynchronous memory copy operation with the specified attributes.

__host__ cudaError_t *cudaMemset*(void *devPtr, int value, size_t count)
Initializes or sets device memory to a value.

__host__ cudaError_t *cudaMemset2D*(void *devPtr, size_t pitch, int value, size_t width, size_t height)
Initializes or sets device memory to a value.

__host__ cudaError_t *cudaMemset2DAsync*(void *devPtr, size_t pitch, int value, size_t width, size_t height, cudaStream_t stream=0)
Initializes or sets device memory to a value.

__host__ cudaError_t *cudaMemset3D*(struct cudaPitchedPtr pitchedDevPtr, int value, struct cudaExtent extent)
Initializes or sets device memory to a value.

__host__ cudaError_t *cudaMemset3DAsync*(struct cudaPitchedPtr pitchedDevPtr, int value, struct cudaExtent extent, cudaStream_t stream=0)
Initializes or sets device memory to a value.

__host__ cudaError_t *cudaMemsetAsync*(void *devPtr, int value, size_t count, cudaStream_t stream=0)
Initializes or sets device memory to a value.

__host__ cudaError_t *cudaMipmappedArrayGetMemoryRequirements*(struct cudaArrayMemoryRequirements *memoryRequirements, cudaMipmappedArray_t mipmap, int device)
Returns the memory requirements of a CUDA mipmapped array.

__host__ cudaError_t *cudaMipmappedArrayGetSparseProperties*(struct cudaArraySparseProperties *sparseProperties, cudaMipmappedArray_t mipmap)
Returns the layout properties of a sparse CUDA mipmapped array.

__host__ struct cudaExtent *make_cudaExtent*(size_t w, size_t h, size_t d)
Returns a *cudaExtent* based on input parameters.

__host__ struct cudaPitchedPtr *make_cudaPitchedPtr*(void *d, size_t p, size_t xsz, size_t ysz)
Returns a *cudaPitchedPtr* based on input parameters.

__host__ struct cudaPos *make_cudaPos*(size_t x, size_t y, size_t z)
Returns a *cudaPos* based on input parameters.

6.25.1. Functions

__host__ *cudaError_t* **cudaArrayGetInfo**(struct *cudaChannelFormatDesc* *desc, struct *cudaExtent* *extent, unsigned int *flags, *cudaArray_t* array)

Gets info about the specified cudaArray.

Returns in `*desc`, `*extent` and `*flags` respectively, the type, shape and flags of `array`.

Any of `*desc`, `*extent` and `*flags` may be specified as NULL.

See also:

::cuArrayGetDescriptor, ::cuArray3DGetDescriptor

Note: Note that this function may also return error codes from previous, asynchronous launches.

Note: Note that this function may also return *cudaErrorInitializationError*, *cudaErrorInsufficientDriver* or *cudaErrorNoDevice* if this call tries to initialize internal CUDA RT state.

Note: Note that as specified by *cudaStreamAddCallback* no CUDA function may be called from callback. *cudaErrorNotPermitted* may, but is not guaranteed to, be returned as a diagnostic in such case.

Parameters

- **desc** – - Returned array type
- **extent** – - Returned array shape. 2D arrays will have depth of zero
- **flags** – - Returned array flags
- **array** – - The ::cudaArray to get info for

Returns
cudaSuccess, *cudaErrorInvalidValue*

__host__ *cudaError_t* **cudaArrayGetMemoryRequirements**(struct *cudaArrayMemoryRequirements* *memoryRequirements, *cudaArray_t* array, int device)

Returns the memory requirements of a CUDA array.

Returns the memory requirements of a CUDA array in `memoryRequirements` If the CUDA array is not allocated with flag *cudaArrayDeferredMapping* *cudaErrorInvalidValue* will be returned.

The returned value in *cudaArrayMemoryRequirements::size* represents the total size of the CUDA array. The returned value in *cudaArrayMemoryRequirements::alignment* represents the alignment necessary for mapping the CUDA array.

See also:

cudaMipmappedArrayGetMemoryRequirements

Parameters

- **memoryRequirements** – **[out]** - Pointer to *cudaArrayMemoryRequirements*
- **array** – **[in]** - CUDA array to get the memory requirements of
- **device** – **[in]** - Device to get the memory requirements for

Returns
cudaSuccess *cudaErrorInvalidValue*

__host__ *cudaError_t* **cudaArrayGetPlane**(*cudaArray_t* *pPlaneArray, *cudaArray_t* hArray, unsigned int planeIdx)

Gets a CUDA array plane from a CUDA array.

Returns in `pPlaneArray` a CUDA array that represents a single format plane of the CUDA array `hArray`.

If `planeIdx` is greater than the maximum number of planes in this array or if the array does not have a multi-planar format e.g: *cudaChannelFormatKindNV12*, then *cudaErrorInvalidValue* is returned.

Note that if the `hArray` has format *cudaChannelFormatKindNV12*, then passing in 0 for `planeIdx` returns a CUDA array of the same size as `hArray` but with one 8-bit channel and *cudaChannelFormatKindUnsigned* as its format kind. If 1 is passed for `planeIdx`, then the returned CUDA array has half the height and width of `hArray` with two 8-bit channels and *cudaChannelFormatKindUnsigned* as its format kind.

See also:

::cuArrayGetPlane

Note: Note that this function may also return error codes from previous, asynchronous launches.

Parameters

- **pPlaneArray** – - Returned CUDA array referenced by the `planeIdx`
- **hArray** – - CUDA array
- **planeIdx** – - Plane index

Returns

cudaSuccess, *cudaErrorInvalidValue* *cudaErrorInvalidResourceHandle*

__host__ *cudaError_t* **cudaArrayGetSparseProperties**(struct *cudaArraySparseProperties* *sparseProperties, *cudaArray_t* array)

Returns the layout properties of a sparse CUDA array.

Returns the layout properties of a sparse CUDA array in `sparseProperties`. If the CUDA array is not allocated with flag *cudaArraySparse* *cudaErrorInvalidValue* will be returned.

If the returned value in *cudaArraySparseProperties::flags* contains *cudaArraySparsePropertiesSingleMipTail*, then *cudaArraySparseProperties::miptailSize* represents the total size of the array. Otherwise, it will be zero. Also, the returned value in *cudaArraySparseProperties::miptailFirstLevel* is always zero. Note that the `array` must have been allocated using *cudaMallocArray* or *cudaMalloc3DArray*. For CUDA arrays obtained using ::cudaMipmappedArrayGetLevel, *cudaErrorInvalidValue* will be returned. Instead, *cudaMipmappedArrayGetSparseProperties* must be used to obtain the sparse properties of the entire CUDA mipmapped array to which `array` belongs to.

See also:

cudaMipmappedArrayGetSparseProperties, ::cuMemMapArrayAsync

Parameters

- **sparseProperties** – **[out]** - Pointer to return the *cudaArraySparseProperties*
- **array** – **[in]** - The CUDA array to get the sparse properties of

Returns

cudaSuccess *cudaErrorInvalidValue*

__host__ *cudaError_t* **cudaFree**(void *devPtr)

Frees memory on the device.

Frees the memory space pointed to by `devPtr`, which must have been returned by a previous call to one of the following memory allocation APIs - *cudaMalloc()*, *cudaMallocPitch()*, *cudaMallocManaged()*, *cudaMallocAsync()*, *cudaMallocFromPoolAsync()*.

Note - This API will not perform any implicit synchronization when the pointer was allocated with *cudaMallocAsync* or *cudaMallocFromPoolAsync*. Callers must ensure that all accesses to these pointer have completed before invoking *cudaFree*. For best performance and memory reuse, users should use *cudaFreeAsync* to free memory allocated via the stream ordered memory allocator. For all other pointers, this API may perform implicit synchronization.

If *cudaFree*(`devPtr`) has already been called before, an error is returned. If `devPtr` is 0, no operation is performed. *cudaFree()* returns ::cudaErrorValue in case of failure.

The device version of *cudaFree* cannot be used with a `*devPtr` allocated using the host API, and vice versa.

See also:

cudaMalloc, *cudaMallocPitch*, *cudaMallocManaged*, *cudaMallocArray*, *cudaFreeArray*, *cudaMallocAsync*, *cudaMallocFromPoolAsync* *cudaMallocHost (C API)*, *cudaFreeHost*, *cudaMalloc3D*, *cudaMalloc3DArray*, *cudaFreeAsync* *cudaHostAlloc*, ::cuMemFree

---

Note: Note that this function may also return error codes from previous, asynchronous launches.

Note: Note that this function may also return *cudaErrorInitializationError*, *cudaErrorInsufficientDriver* or *cudaErrorNoDevice* if this call tries to initialize internal CUDA RT state.

Note: Note that as specified by *cudaStreamAddCallback* no CUDA function may be called from callback. *cudaErrorNotPermitted* may, but is not guaranteed to, be returned as a diagnostic in such case.

Parameters

devPtr – - Device pointer to memory to free

Returns

cudaSuccess, *cudaErrorInvalidValue*

__host__ *cudaError_t* **cudaFreeArray**(*cudaArray_t* array)

Frees an array on the device.

Frees the CUDA array `array`, which must have been returned by a previous call to *cudaMallocArray()*. If `devPtr` is 0, no operation is performed.

See also:

cudaMalloc, *cudaMallocPitch*, *cudaFree*, *cudaMallocArray*, *cudaMallocHost (C API)*, *cudaFreeHost*, *cudaHostAlloc*, ::cuArrayDestroy

Note: Note that this function may also return error codes from previous, asynchronous launches.

Note: Note that this function may also return *cudaErrorInitializationError*, *cudaErrorInsufficientDriver* or *cudaErrorNoDevice* if this call tries to initialize internal CUDA RT state.

Note: Note that as specified by *cudaStreamAddCallback* no CUDA function may be called from callback. *cudaErrorNotPermitted* may, but is not guaranteed to, be returned as a diagnostic in such case.

Parameters

array – - Pointer to array to free

Returns

cudaSuccess, *cudaErrorInvalidValue*

__host__ *cudaError_t* **cudaFreeHost**(void *ptr)

Frees page-locked memory.

Frees the memory space pointed to by `hostPtr`, which must have been returned by a previous call to *cudaMallocHost()* or *cudaHostAlloc()*.

See also:

cudaMalloc, *cudaMallocPitch*, *cudaFree*, *cudaMallocArray*, *cudaFreeArray*, *cudaMallocHost (C API)*, *cudaMalloc3D*, *cudaMalloc3DArray*, *cudaHostAlloc*, ::cuMemFreeHost

Note: Note that this function may also return error codes from previous, asynchronous launches.

Note: Note that this function may also return *cudaErrorInitializationError*, *cudaErrorInsufficientDriver* or *cudaErrorNoDevice* if this call tries to initialize internal CUDA RT state.

Note: Note that as specified by *cudaStreamAddCallback* no CUDA function may be called from callback. *cudaErrorNotPermitted* may, but is not guaranteed to, be returned as a diagnostic in such case.

Parameters

ptr – - Pointer to memory to free

Returns

cudaSuccess, *cudaErrorInvalidValue*

__host__ *cudaError_t* **cudaFreeMipmappedArray**(*cudaMipmappedArray_t* mipmappedArray)

Frees a mipmapped array on the device.

Frees the CUDA mipmapped array `mipmappedArray`, which must have been returned by a previous call to *cudaMallocMipmappedArray()*. If `devPtr` is 0, no operation is performed.

See also:

cudaMalloc, *cudaMallocPitch*, *cudaFree*, *cudaMallocArray*, *cudaMallocHost (C API)*, *cudaFreeHost*, *cudaHostAlloc*, ::cuMipmappedArrayDestroy

---

Note: Note that this function may also return error codes from previous, asynchronous launches.

Note: Note that this function may also return *cudaErrorInitializationError*, *cudaErrorInsufficientDriver* or *cudaErrorNoDevice* if this call tries to initialize internal CUDA RT state.

Note: Note that as specified by *cudaStreamAddCallback* no CUDA function may be called from callback. *cudaErrorNotPermitted* may, but is not guaranteed to, be returned as a diagnostic in such case.

Parameters

`mipmappedArray` – - Pointer to mipmapped array to free

Returns

cudaSuccess, *cudaErrorInvalidValue*

__host__ *cudaError_t* **`cudaGetMipmappedArrayLevel`**(*cudaArray_t* *levelArray, *cudaMipmappedArray_const_t* mipmappedArray, unsigned int level)

Gets a mipmap level of a CUDA mipmapped array.

Returns in `*levelArray` a CUDA array that represents a single mipmap level of the CUDA mipmapped array `mipmappedArray`.

If `level` is greater than the maximum number of levels in this mipmapped array, *cudaErrorInvalidValue* is returned.

If `mipmappedArray` is NULL, *cudaErrorInvalidResourceHandle* is returned.

See also:

cudaMalloc3D, *cudaMalloc*, *cudaMallocPitch*, *cudaFree*, *cudaFreeArray*, *cudaMallocHost (C API)*, *cudaFreeHost*, *cudaHostAlloc*, *make_cudaExtent*, ::cuMipmappedArrayGetLevel

---

Note: Note that this function may also return error codes from previous, asynchronous launches.

Note: Note that this function may also return *cudaErrorInitializationError*, *cudaErrorInsufficientDriver* or *cudaErrorNoDevice* if this call tries to initialize internal CUDA RT state.

Note: Note that as specified by *cudaStreamAddCallback* no CUDA function may be called from callback. *cudaErrorNotPermitted* may, but is not guaranteed to, be returned as a diagnostic in such case.

Parameters

- **levelArray** – - Returned mipmap level CUDA array
- **mipmappedArray** – - CUDA mipmapped array
- **level** – - Mipmap level

Returns

cudaSuccess, *cudaErrorInvalidValue* *cudaErrorInvalidResourceHandle*

__host__ *cudaError_t* **cudaGetSymbolAddress**(void **devPtr, const void *symbol)

Finds the address associated with a CUDA symbol.

Returns in `*devPtr` the address of symbol `symbol` on the device. `symbol` is a variable that resides in global or constant memory space. If `symbol` cannot be found, or if `symbol` is not declared in the global or constant memory space, `*devPtr` is unchanged and the error *cudaErrorInvalidSymbol* is returned.

See also:

cudaGetSymbolAddress (C++ API), *cudaGetSymbolSize (C API)*, ::cuModuleGetGlobal

Note: Note that this function may also return error codes from previous, asynchronous launches.

Note: Use of a string naming a variable as the `symbol` parameter was deprecated in CUDA 4.1 and removed in CUDA 5.0.

Note: Note that this function may also return *cudaErrorInitializationError*, *cudaErrorInsufficientDriver* or *cudaErrorNoDevice* if this call tries to initialize internal CUDA RT state.

Note: Note that as specified by *cudaStreamAddCallback* no CUDA function may be called from callback. *cudaErrorNotPermitted* may, but is not guaranteed to, be returned as a diagnostic in such case.

Parameters

- **devPtr** – - Return device pointer associated with symbol
- **symbol** – - Device symbol address

Returns

cudaSuccess, *cudaErrorInvalidSymbol*, *cudaErrorNoKernelImageForDevice*

__host__ *cudaError_t* **cudaGetSymbolSize**(size_t *size, const void *symbol)

Finds the size of the object associated with a CUDA symbol.

Returns in `*size` the size of symbol `symbol`. `symbol` is a variable that resides in global or constant memory space. If `symbol` cannot be found, or if `symbol` is not declared in global or constant memory space, `*size` is unchanged and the error *cudaErrorInvalidSymbol* is returned.

See also:

cudaGetSymbolAddress (C API), *cudaGetSymbolSize (C++ API)*, ::cuModuleGetGlobal

---

Note: Note that this function may also return error codes from previous, asynchronous launches.

Note: Use of a string naming a variable as the `symbol` parameter was deprecated in CUDA 4.1 and removed in CUDA 5.0.

Note: Note that this function may also return *cudaErrorInitializationError*, *cudaErrorInsufficientDriver* or *cudaErrorNoDevice* if this call tries to initialize internal CUDA RT state.

Note: Note that as specified by *cudaStreamAddCallback* no CUDA function may be called from callback. *cudaErrorNotPermitted* may, but is not guaranteed to, be returned as a diagnostic in such case.

Parameters

- **size** – - Size of object associated with symbol
- **symbol** – - Device symbol address

Returns

cudaSuccess, *cudaErrorInvalidSymbol*, *cudaErrorNoKernelImageForDevice*

__host__ *cudaError_t* **cudaHostAlloc**(void **pHost, size_t size, unsigned int flags)

Allocates page-locked memory on the host.

Allocates `size` bytes of host memory that is page-locked and accessible to the device. The driver tracks the virtual memory ranges allocated with this function and automatically accelerates calls to functions such as *cudaMemcpy()*. Since the memory can be accessed directly by the device, it can be read or written with much higher bandwidth than pageable memory obtained with functions such as ::malloc(). Allocating excessive amounts of pinned memory may degrade system performance, since it reduces the amount of memory available to the system for paging. As a result, this function is best used sparingly to allocate staging areas for data exchange between host and device.

The `flags` parameter enables different options to be specified that affect the allocation, as follows.

- *cudaHostAllocDefault*: This flag's value is defined to be 0 and causes *cudaHostAlloc()* to emulate *cudaMallocHost()*.

- *cudaHostAllocPortable*: The memory returned by this call will be considered as pinned memory by all CUDA contexts, not just the one that performed the allocation.
- *cudaHostAllocMapped*: Maps the allocation into the CUDA address space. The device pointer to the memory may be obtained by calling *cudaHostGetDevicePointer()*.
- *cudaHostAllocWriteCombined*: Allocates the memory as write-combined (WC). WC memory can be transferred across the PCI Express bus more quickly on some system configurations, but cannot be read efficiently by most CPUs. WC memory is a good option for buffers that will be written by the CPU and read by the device via mapped pinned memory or host->device transfers.

All of these flags are orthogonal to one another: a developer may allocate memory that is portable, mapped and/or write-combined with no restrictions.

In order for the *cudaHostAllocMapped* flag to have any effect, the CUDA context must support the *cudaDeviceMapHost* flag, which can be checked via *cudaGetDeviceFlags()*. The *cudaDeviceMapHost* flag is implicitly set for contexts created via the runtime API.

The *cudaHostAllocMapped* flag may be specified on CUDA contexts for devices that do not support mapped pinned memory. The failure is deferred to *cudaHostGetDevicePointer()* because the memory may be mapped into other CUDA contexts via the *cudaHostAllocPortable* flag.

Memory allocated by this function must be freed with *cudaFreeHost()*.

See also:

cudaSetDeviceFlags, *cudaMallocHost (C API)*, *cudaFreeHost*, *cudaGetDeviceFlags*, ::cuMemHostAlloc

---

Note: Note that this function may also return error codes from previous, asynchronous launches.

Note: Note that this function may also return *cudaErrorInitializationError*, *cudaErrorInsufficientDriver* or *cudaErrorNoDevice* if this call tries to initialize internal CUDA RT state.

Note: Note that as specified by *cudaStreamAddCallback* no CUDA function may be called from callback. *cudaErrorNotPermitted* may, but is not guaranteed to, be returned as a diagnostic in such case.

Parameters

- **`pHost`** – - Device pointer to allocated memory
- **`size`** – - Requested allocation size in bytes
- **`flags`** – - Requested properties of allocated memory

Returns

cudaSuccess, *cudaErrorInvalidValue*, *cudaErrorMemoryAllocation*, *cudaErrorExternalDevice*

__host__ *cudaError_t* **cudaHostGetDevicePointer**(void **pDevice, void *pHost, unsigned int flags)

Passes back device pointer of mapped host memory allocated by cudaHostAlloc or registered by cudaHostRegister.

Passes back the device pointer corresponding to the mapped, pinned host buffer allocated by *cudaHostAlloc()* or registered by *cudaHostRegister()*.

cudaHostGetDevicePointer() will fail if the *cudaDeviceMapHost* flag was not specified before deferred context creation occurred, or if called on a device that does not support mapped, pinned memory.

For devices that have a non-zero value for the device attribute *cudaDevAttrCanUseHostPointerForRegisteredMem*, the memory can also be accessed from the device using the host pointer `pHost`. The device pointer returned by *cudaHostGetDevicePointer()* may or may not match the original host pointer `pHost` and depends on the devices visible to the application. If all devices visible to the application have a non-zero value for the device attribute, the device pointer returned by *cudaHostGetDevicePointer()* will match the original pointer `pHost`. If any device visible to the application has a zero value for the device attribute, the device pointer returned by *cudaHostGetDevicePointer()* will not match the original host pointer `pHost`, but it will be suitable for use on all devices provided Unified Virtual Addressing is enabled. In such systems, it is valid to access the memory using either pointer on devices that have a non-zero value for the device attribute. Note however that such devices should access the memory using only of the two pointers and not both.

`flags` provides for future releases. For now, it must be set to 0.

See also:

cudaSetDeviceFlags, *cudaHostAlloc*, ::cuMemHostGetDevicePointer

---

Note: Note that this function may also return error codes from previous, asynchronous launches.

Note: Note that this function may also return *cudaErrorInitializationError*, *cudaErrorInsufficientDriver* or *cudaErrorNoDevice* if this call tries to initialize internal CUDA RT state.

Note: Note that as specified by *cudaStreamAddCallback* no CUDA function may be called from callback. *cudaErrorNotPermitted* may, but is not guaranteed to, be returned as a diagnostic in such case.

Parameters

- **pDevice** – - Returned device pointer for mapped memory
- **pHost** – - Requested host pointer mapping
- **flags** – - Flags for extensions (must be 0 for now)

Returns

cudaSuccess, *cudaErrorInvalidValue*, *cudaErrorMemoryAllocation*

__host__ *cudaError_t* **cudaHostGetFlags**(unsigned int *pFlags, void *pHost)

Passes back flags used to allocate pinned host memory allocated by cudaHostAlloc.

cudaHostGetFlags() will fail if the input pointer does not reside in an address range allocated by *cudaHostAlloc()*.

See also:

cudaHostAlloc, ::cuMemHostGetFlags

Note: Note that this function may also return error codes from previous, asynchronous launches.

Note: Note that this function may also return *cudaErrorInitializationError*, *cudaErrorInsufficientDriver* or *cudaErrorNoDevice* if this call tries to initialize internal CUDA RT state.

Note: Note that as specified by *cudaStreamAddCallback* no CUDA function may be called from callback. *cudaErrorNotPermitted* may, but is not guaranteed to, be returned as a diagnostic in such case.

Parameters

- **pFlags** – - Returned flags word
- **pHost** – - Host pointer

Returns

cudaSuccess, *cudaErrorInvalidValue*

__host__ *cudaError_t* **cudaHostRegister**(void *ptr, size_t size, unsigned int flags)

Registers an existing host memory range for use by CUDA.

Page-locks the memory range specified by `ptr` and `size` and maps it for the device(s) as specified by `flags`. This memory range also is added to the same tracking mechanism as *cudaHostAlloc()* to automatically accelerate calls to functions such as *cudaMemcpy()*. Since the memory can be accessed directly by the device, it can be read or written with much higher bandwidth than pageable memory that has not been registered. Page-locking excessive amounts of memory may degrade system performance, since it reduces the amount of memory available to the system for paging. As a result, this function is best used sparingly to register staging areas for data exchange between host and device.

On systems where ::pageableMemoryAccessUsesHostPageTables is true, *cudaHostRegister* might not page-lock the memory range specified by `ptr`. On such systems with discrete GPUs, the memory range is not page-locked and only unpopulated pages are populated. On systems with an SoC-integrated GPU (e.g. NVIDIA Tegra), the memory range is page-locked. Since the GPU on these systems can access pageable memory directly, applications that want to avoid the page-locking cost can skip *cudaHostRegister* and access the memory directly from the GPU. See Tegra app note for registered memory behavior on Tegra platforms.

cudaHostRegister is supported only on I/O coherent devices that have a non-zero value for the device attribute *cudaDevAttrHostRegisterSupported*.

The `flags` parameter enables different options to be specified that affect the allocation, as follows.

- *cudaHostRegisterDefault*: On a system with unified virtual addressing, the memory will be both mapped and portable. On a system with no unified virtual addressing, the memory will be neither mapped nor portable.
- *cudaHostRegisterPortable*: The memory returned by this call will be considered as pinned memory by all CUDA contexts, not just the one that performed the allocation.
- *cudaHostRegisterMapped*: Maps the allocation into the CUDA address space. The device pointer to the memory may be obtained by calling *cudaHostGetDevicePointer()*.
- *cudaHostRegisterIoMemory*: The passed memory pointer is treated as pointing to some memory-mapped I/O space, e.g. belonging to a third-party PCIe device, and it will marked as non cache-coherent and contiguous.
- *cudaHostRegisterReadOnly*: The passed memory pointer is treated as pointing to memory that is considered read-only by the device. On platforms without *cudaDevAttrPageableMemoryAccessUsesHostPageTables*, this flag is required in order to register memory mapped to the CPU as read-only. Support for the use of this flag can be queried from the device attribute *cudaDevAttrHostRegisterReadOnlySupported*. Using this flag with a current context associated with a device that does not have this attribute set will cause *cudaHostRegister* to error with cudaErrorNotSupported.

All of these flags are orthogonal to one another: a developer may page-lock memory that is portable or mapped with no restrictions.

The CUDA context must have been created with the ::cudaMapHost flag in order for the *cudaHostRegisterMapped* flag to have any effect.

The *cudaHostRegisterMapped* flag may be specified on CUDA contexts for devices that do not support mapped pinned memory. The failure is deferred to *cudaHostGetDevicePointer()* because the memory may be mapped into other CUDA contexts via the *cudaHostRegisterPortable* flag.

For devices that have a non-zero value for the device attribute *cudaDevAttrCanUseHostPointerForRegisteredMem*, the memory can also be accessed from the device using the host pointer `ptr`. The device pointer returned by *cudaHostGetDevicePointer()* may or may not match the original host pointer `ptr` and depends on the devices visible to the application. If all devices visible to the application have a non-zero value for the device attribute, the device pointer returned by *cudaHostGetDevicePointer()* will match the original pointer `ptr`. If any device visible to the application has a zero value for the device attribute, the device pointer returned by *cudaHostGetDevicePointer()* will not match the original host pointer `ptr`, but it will be suitable for use on all devices provided Unified Virtual Addressing is enabled. In such systems, it is valid to access the memory using either pointer on devices that have a non-zero value for the device attribute. Note however that such devices should access the memory using only of the two pointers and not both.

The memory page-locked by this function must be unregistered with *cudaHostUnregister()*.

See also:

cudaHostUnregister, *cudaHostGetFlags*, *cudaHostGetDevicePointer*, ::cuMemHostRegister

---

Note: Note that this function may also return error codes from previous, asynchronous launches.

Note: Note that this function may also return *cudaErrorInitializationError*, *cudaErrorInsufficientDriver* or *cudaErrorNoDevice* if this call tries to initialize internal CUDA RT state.

Note: Note that as specified by *cudaStreamAddCallback* no CUDA function may be called from callback. *cudaErrorNotPermitted* may, but is not guaranteed to, be returned as a diagnostic in such case.

Parameters

- **ptr** – - Host pointer to memory to page-lock
- **size** – - Size in bytes of the address range to page-lock in bytes
- **flags** – - Flags for allocation request

Returns

cudaSuccess, *cudaErrorInvalidValue*, *cudaErrorMemoryAllocation*, *cudaErrorHostMemoryAlreadyRegistered*, *cudaErrorNotSupported*, *cudaErrorExternalDevice*

__host__ *cudaError_t* **cudaHostUnregister**(void *ptr)

Unregisters a memory range that was registered with cudaHostRegister.

Unmaps the memory range whose base address is specified by `ptr`, and makes it pageable again.

The base address must be the same one specified to *cudaHostRegister()*.

See also:

cudaHostUnregister, ::cuMemHostUnregister

Note: Note that this function may also return error codes from previous, asynchronous launches.

Note: Note that this function may also return *cudaErrorInitializationError*, *cudaErrorInsufficientDriver* or *cudaErrorNoDevice* if this call tries to initialize internal CUDA RT state.

Note: Note that as specified by *cudaStreamAddCallback* no CUDA function may be called from callback. *cudaErrorNotPermitted* may, but is not guaranteed to, be returned as a diagnostic in such case.

Parameters

ptr – - Host pointer to memory to unregister

Returns

cudaSuccess, *cudaErrorInvalidValue*, *cudaErrorHostMemoryNotRegistered*

__host__ *cudaError_t* **cudaMalloc**(void **devPtr, size_t size)

Allocate memory on the device.

Allocates `size` bytes of linear memory on the device and returns in `*devPtr` a pointer to the allocated memory. The allocated memory is suitably aligned for any kind of variable. The memory is not cleared. *cudaMalloc()* returns *cudaErrorMemoryAllocation* in case of failure.

The device version of *cudaFree* cannot be used with a `*devPtr` allocated using the host API, and vice versa.

See also:

cudaMallocPitch, *cudaFree*, *cudaMallocArray*, *cudaFreeArray*, *cudaMalloc3D*, *cudaMalloc3DArray*, *cudaMallocHost (C API)*, *cudaFreeHost*, *cudaHostAlloc*, ::cuMemAlloc

---

Note: Note that this function may also return error codes from previous, asynchronous launches.

Note: Note that this function may also return *cudaErrorInitializationError*, *cudaErrorInsufficientDriver* or *cudaErrorNoDevice* if this call tries to initialize internal CUDA RT state.

Note: Note that as specified by *cudaStreamAddCallback* no CUDA function may be called from callback. *cudaErrorNotPermitted* may, but is not guaranteed to, be returned as a diagnostic in such case.

Parameters

- **devPtr** – - Pointer to allocated device memory
- **size** – - Requested allocation size in bytes

Returns

cudaSuccess, *cudaErrorInvalidValue*, *cudaErrorMemoryAllocation*, *cudaErrorExternalDevice*

__host__ *cudaError_t* **cudaMalloc3D**(struct *cudaPitchedPtr* *pitchedDevPtr, struct *cudaExtent* extent)

Allocates logical 1D, 2D, or 3D memory objects on the device.

Allocates at least `width` * `height` * `depth` bytes of linear memory on the device and returns a *cudaPitchedPtr* in which `ptr` is a pointer to the allocated memory. The function may pad the allocation to ensure hardware alignment requirements are met. The pitch returned in the `pitch` field of `pitchedDevPtr` is the width in bytes of the allocation.

The returned *cudaPitchedPtr* contains additional fields `xsize` and `ysize`, the logical width and height of the allocation, which are equivalent to the `width` and `height` `extent` parameters provided by the programmer during allocation.

For allocations of 2D and 3D objects, it is highly recommended that programmers perform allocations using *cudaMalloc3D()* or *cudaMallocPitch()*. Due to alignment restrictions in the hardware, this is especially true if the application will be performing memory copies involving 2D or 3D objects (whether linear memory or CUDA arrays).

See also:

cudaMallocPitch, *cudaFree*, *cudaMemcpy3D*, *cudaMemset3D*, *cudaMalloc3DArray*, *cudaMallocArray*, *cudaFreeArray*, *cudaMallocHost (C API)*, *cudaFreeHost*, *cudaHostAlloc*, *make_cudaPitchedPtr*, *make_cudaExtent*, ::cuMemAllocPitch

---

Note: Note that this function may also return error codes from previous, asynchronous launches.

Note: Note that this function may also return *cudaErrorInitializationError*, *cudaErrorInsufficientDriver* or *cudaErrorNoDevice* if this call tries to initialize internal CUDA RT state.

Note: Note that as specified by *cudaStreamAddCallback* no CUDA function may be called from callback. *cudaErrorNotPermitted* may, but is not guaranteed to, be returned as a diagnostic in such case.

Parameters

- **pitchedDevPtr** – - Pointer to allocated pitched device memory
- **extent** – - Requested allocation size (`width` field in bytes)

Returns

cudaSuccess, *cudaErrorInvalidValue*, *cudaErrorMemoryAllocation*

__host__ *cudaError_t* **cudaMalloc3DArray**(*cudaArray_t* *array, const struct *cudaChannelFormatDesc* *desc, struct *cudaExtent* extent, unsigned int flags = 0)

Allocate an array on the device.

Allocates a CUDA array according to the *cudaChannelFormatDesc* structure `desc` and returns a handle to the new CUDA array in `*array`.

The *cudaChannelFormatDesc* is defined as:

```
struct cudaChannelFormatDesc {
    int x, y, z, w;
    enum cudaChannelFormatKind f;
};
```

where *cudaChannelFormatKind* is one of *cudaChannelFormatKindSigned*, *cudaChannelFormatKindUnsigned*, or *cudaChannelFormatKindFloat*.

cudaMalloc3DArray() can allocate the following:

- A 1D array is allocated if the height and depth extents are both zero.
- A 2D array is allocated if only the depth extent is zero.
- A 3D array is allocated if all three extents are non-zero.

- A 1D layered CUDA array is allocated if only the height extent is zero and the cudaArrayLayered flag is set. Each layer is a 1D array. The number of layers is determined by the depth extent.
- A 2D layered CUDA array is allocated if all three extents are non-zero and the cudaArrayLayered flag is set. Each layer is a 2D array. The number of layers is determined by the depth extent.
- A cubemap CUDA array is allocated if all three extents are non-zero and the cudaArrayCubemap flag is set. Width must be equal to height, and depth must be six. A cubemap is a special type of 2D layered CUDA array, where the six layers represent the six faces of a cube. The order of the six layers in memory is the same as that listed in *cudaGraphicsCubeFace*.
- A cubemap layered CUDA array is allocated if all three extents are non-zero, and both, cudaArrayCubemap and cudaArrayLayered flags are set. Width must be equal to height, and depth must be a multiple of six. A cubemap layered CUDA array is a special type of 2D layered CUDA array that consists of a collection of cubemaps. The first six layers represent the first cubemap, the next six layers form the second cubemap, and so on.

The `flags` parameter enables different options to be specified that affect the allocation, as follows.

- *cudaArrayDefault*: This flag's value is defined to be 0 and provides default array allocation
- *cudaArrayLayered*: Allocates a layered CUDA array, with the depth extent indicating the number of layers
- *cudaArrayCubemap*: Allocates a cubemap CUDA array. Width must be equal to height, and depth must be six. If the cudaArrayLayered flag is also set, depth must be a multiple of six.
- *cudaArraySurfaceLoadStore*: Allocates a CUDA array that could be read from or written to using a surface reference.
- *cudaArrayTextureGather*: This flag indicates that texture gather operations will be performed on the CUDA array. Texture gather can only be performed on 2D CUDA arrays.
- *cudaArraySparse*: Allocates a CUDA array without physical backing memory. The subregions within this sparse array can later be mapped onto a physical memory allocation by calling ::cuMemMapArrayAsync. This flag can only be used for creating 2D, 3D or 2D layered sparse CUDA arrays. The physical backing memory must be allocated via ::cuMemCreate.
- *cudaArrayDeferredMapping*: Allocates a CUDA array without physical backing memory. The entire array can later be mapped onto a physical memory allocation by calling ::cuMemMapArrayAsync. The physical backing memory must be allocated via ::cuMemCreate.

The width, height and depth extents must meet certain size requirements as listed in the following table. All values are specified in elements.

Note that 2D CUDA arrays have different size requirements if the *cudaArrayTextureGather* flag is set. In that case, the valid range for (width, height, depth) is ((1,maxTexture2DGather[0]), (1,maxTexture2DGather[1]), 0).

CUDA array type	Valid extents that must always be met {(width range in elements), (height range), (depth range)}	Valid extents with cudaArraySurfaceLoadStore set {(width range in elements), (height range), (depth range)}
1D	{ (1,maxTexture1D), 0, 0 }	{ (1,maxSurface1D), 0, 0 }
2D	{ (1,maxTexture2D[0]), (1,maxTexture2D[1]), 0 }	{ (1,maxSurface2D[0]), (1,maxSurface2D[1]), 0 }
3D	{ (1,maxTexture3D[0]), (1,maxTexture3D[1]), (1,maxTexture3D[2]) } OR { (1,maxTexture3DAlt[0]), (1,maxTexture3DAlt[1]), (1,maxTexture3DAlt[2]) }	{ (1,maxSurface3D[0]), (1,maxSurface3D[1]), (1,maxSurface3D[2]) }
1D Layered	{ (1,maxTexture1DLayered[0]), 0, (1,maxTexture1DLayered[1]) }	{ (1,maxSurface1DLayered[0]), 0, (1,maxSurface1DLayered[1]) }
2D Layered	{ (1,maxTexture2DLayered[0]), (1,maxTexture2DLayered[1]), (1,maxTexture2DLayered[2]) }	{ (1,maxSurface2DLayered[0]), (1,maxSurface2DLayered[1]), (1,maxSurface2DLayered[2]) }
Cubemap	{ (1,maxTextureCubemap), (1,maxTextureCubemap), 6 }	{ (1,maxSurfaceCubemap), (1,maxSurfaceCubemap), 6 }
Cubemap Layered	{ (1,maxTextureCubemapLayered[0]), (1,maxTextureCubemapLayered[0]), (1,maxTextureCubemapLayered[1]) }	{ (1,maxSurfaceCubemapLayered[0]), (1,maxSurfaceCubemapLayered[0]), (1,maxSurfaceCubemapLayered[1]) }

See also:

cudaMalloc3D, *cudaMalloc*, *cudaMallocPitch*, *cudaFree*, *cudaFreeArray*, *cudaMallocHost (C API)*, *cudaFreeHost*, *cudaHostAlloc*, *make_cudaExtent*, ::cuArray3DCreate

---

Note: Note that this function may also return error codes from previous, asynchronous launches.

Note: Note that this function may also return *cudaErrorInitializationError*, *cudaErrorInsufficientDriver* or *cudaErrorNoDevice* if this call tries to initialize internal CUDA RT state.

Note: Note that as specified by *cudaStreamAddCallback* no CUDA function may be called from callback. *cudaErrorNotPermitted* may, but is not guaranteed to, be returned as a diagnostic in such case.

Parameters

- **`array`** – - Pointer to allocated array in device memory
- **`desc`** – - Requested channel format
- **`extent`** – - Requested allocation size (`width` field in elements)

- **flags** – - Flags for extensions

Returns
cudaSuccess, *cudaErrorInvalidValue*, *cudaErrorMemoryAllocation*

__host__ *cudaError_t* **cudaMallocArray**(*cudaArray_t* *array, const struct *cudaChannelFormatDesc* *desc, size_t width, size_t height = 0, unsigned int flags = 0)

Allocate an array on the device.

Allocates a CUDA array according to the *cudaChannelFormatDesc* structure `desc` and returns a handle to the new CUDA array in `*array`.

The *cudaChannelFormatDesc* is defined as:

```
struct cudaChannelFormatDesc {
    int x, y, z, w;
enum cudaChannelFormatKind f;
};
```

where *cudaChannelFormatKind* is one of *cudaChannelFormatKindSigned*, *cudaChannelFormatKindUnsigned*, or *cudaChannelFormatKindFloat*.

The `flags` parameter enables different options to be specified that affect the allocation, as follows.

- *cudaArrayDefault*: This flag's value is defined to be 0 and provides default array allocation
- *cudaArraySurfaceLoadStore*: Allocates an array that can be read from or written to using a surface reference
- *cudaArrayTextureGather*: This flag indicates that texture gather operations will be performed on the array.
- *cudaArraySparse*: Allocates a CUDA array without physical backing memory. The subregions within this sparse array can later be mapped onto a physical memory allocation by calling ::cuMemMapArrayAsync. The physical backing memory must be allocated via ::cuMemCreate.
- *cudaArrayDeferredMapping*: Allocates a CUDA array without physical backing memory. The entire array can later be mapped onto a physical memory allocation by calling ::cuMemMapArrayAsync. The physical backing memory must be allocated via ::cuMemCreate.

`width` and `height` must meet certain size requirements. See *cudaMalloc3DArray()* for more details.

See also:

cudaMalloc, *cudaMallocPitch*, *cudaFree*, *cudaFreeArray*, *cudaMallocHost (C API)*, *cudaFreeHost*, *cudaMalloc3D*, *cudaMalloc3DArray*, *cudaHostAlloc*, ::cuArrayCreate

Note: Note that this function may also return error codes from previous, asynchronous launches.

Note: Note that this function may also return *cudaErrorInitializationError*, *cudaErrorInsufficientDriver* or *cudaErrorNoDevice* if this call tries to initialize internal CUDA RT state.

Note: Note that as specified by *cudaStreamAddCallback* no CUDA function may be called from callback. *cudaErrorNotPermitted* may, but is not guaranteed to, be returned as a diagnostic in such case.

Parameters

- **array** – - Pointer to allocated array in device memory
- **desc** – - Requested channel format
- **width** – - Requested array allocation width
- **height** – - Requested array allocation height
- **flags** – - Requested properties of allocated array

Returns

cudaSuccess, *cudaErrorInvalidValue*, *cudaErrorMemoryAllocation*

__host__ *cudaError_t* **cudaMallocHost**(void **ptr, size_t size)

Allocates page-locked memory on the host.

Allocates `size` bytes of host memory that is page-locked and accessible to the device. The driver tracks the virtual memory ranges allocated with this function and automatically accelerates calls to functions such as *cudaMemcpy**(). Since the memory can be accessed directly by the device, it can be read or written with much higher bandwidth than pageable memory obtained with functions such as ::malloc().

On systems where ::pageableMemoryAccessUsesHostPageTables is true, *cudaMallocHost* may not page-lock the allocated memory.

Page-locking excessive amounts of memory with *cudaMallocHost()* may degrade system performance, since it reduces the amount of memory available to the system for paging. As a result, this function is best used sparingly to allocate staging areas for data exchange between host and device.

See also:

cudaMalloc, *cudaMallocPitch*, *cudaMallocArray*, *cudaMalloc3D*, *cudaMalloc3DArray*, *cudaHostAlloc*, *cudaFree*, *cudaFreeArray*, *cudaMallocHost (C++ API)*, *cudaFreeHost*, *cudaHostAlloc*, ::cuMemAllocHost

Note: Note that this function may also return error codes from previous, asynchronous launches.

Note: Note that this function may also return *cudaErrorInitializationError*, *cudaErrorInsufficientDriver* or *cudaErrorNoDevice* if this call tries to initialize internal CUDA RT state.

Note: Note that as specified by *cudaStreamAddCallback* no CUDA function may be called from callback. *cudaErrorNotPermitted* may, but is not guaranteed to, be returned as a diagnostic in such case.

Parameters

- **ptr** – - Pointer to allocated host memory
- **size** – - Requested allocation size in bytes

Returns

cudaSuccess, *cudaErrorInvalidValue*, *cudaErrorMemoryAllocation*, *cudaErrorExternalDevice*

__host__ *cudaError_t* **cudaMallocManaged**(void **devPtr, size_t size, unsigned int flags = 0x01)

Allocates memory that will be automatically managed by the Unified Memory system.

Allocates `size` bytes of managed memory on the device and returns in `*devPtr` a pointer to the allocated memory. If the device doesn't support allocating managed memory, *cudaErrorNotSupported* is returned. Support for managed memory can be queried using the device attribute *cudaDevAttrManagedMemory*. The allocated memory is suitably aligned for any kind of variable. The memory is not cleared. If `size` is 0, *cudaMallocManaged* returns *cudaErrorInvalidValue*. The pointer is valid on the CPU and on all GPUs in the system that support managed memory. All accesses to this pointer must obey the Unified Memory programming model.

`flags` specifies the default stream association for this allocation. `flags` must be one of *cudaMemAttachGlobal* or *cudaMemAttachHost*. The default value for `flags` is *cudaMemAttachGlobal*. If *cudaMemAttachGlobal* is specified, then this memory is accessible from any stream on any device. If *cudaMemAttachHost* is specified, then the allocation should not be accessed from devices that have a zero value for the device attribute *cudaDevAttrConcurrentManagedAccess*; an explicit call to *cudaStreamAttachMemAsync* will be required to enable access on such devices.

If the association is later changed via *cudaStreamAttachMemAsync* to a single stream, the default association, as specifed during *cudaMallocManaged*, is restored when that stream is destroyed. For **managed** variables, the default association is always *cudaMemAttachGlobal*. Note that destroying a stream is an asynchronous operation, and as a result, the change to default association won't happen until all work in the stream has completed.

Memory allocated with *cudaMallocManaged* should be released with *cudaFree*.

Device memory oversubscription is possible for GPUs that have a non-zero value for the device attribute *cudaDevAttrConcurrentManagedAccess*. Managed memory on such GPUs may be evicted from device memory to host memory at any time by the Unified Memory driver in order to make room for other allocations.

In a system where all GPUs have a non-zero value for the device attribute *cudaDevAttrConcurrentManagedAccess*, managed memory may not be populated when this API returns and instead may be populated on access. In such systems, managed memory can migrate to any processor's memory at any time. The Unified Memory driver will employ heuristics to maintain data locality and prevent excessive page faults to the extent possible. The application can also guide the driver about memory usage patterns via *cudaMemAdvise*. The application can also explicitly migrate memory to a desired processor's memory via *cudaMemPrefetchAsync*.

In a multi-GPU system where all of the GPUs have a zero value for the device attribute *cudaDevAttrConcurrentManagedAccess* and all the GPUs have peer-to-peer support with each other, the physical storage for managed memory is created on the GPU which is active at the time *cudaMallocManaged* is called. All other GPUs will reference the data at reduced bandwidth via peer mappings over the PCIe bus. The Unified Memory driver does not migrate memory among such GPUs.

In a multi-GPU system where not all GPUs have peer-to-peer support with each other and where the value of the device attribute *cudaDevAttrConcurrentManagedAccess* is zero for at least one of

those GPUs, the location chosen for physical storage of managed memory is system-dependent.

- On Linux, the location chosen will be device memory as long as the current set of active contexts are on devices that either have peer-to-peer support with each other or have a non-zero value for the device attribute *cudaDevAttrConcurrentManagedAccess*. If there is an active context on a GPU that does not have a non-zero value for that device attribute and it does not have peer-to-peer support with the other devices that have active contexts on them, then the location for physical storage will be 'zero-copy' or host memory. Note that this means that managed memory that is located in device memory is migrated to host memory if a new context is created on a GPU that doesn't have a non-zero value for the device attribute and does not support peer-to-peer with at least one of the other devices that has an active context. This in turn implies that context creation may fail if there is insufficient host memory to migrate all managed allocations.
- On Windows, the physical storage is always created in 'zero-copy' or host memory. All GPUs will reference the data at reduced bandwidth over the PCIe bus. In these circumstances, use of the environment variable CUDA_VISIBLE_DEVICES is recommended to restrict CUDA to only use those GPUs that have peer-to-peer support. Alternatively, users can also set CUDA_MANAGED_FORCE_DEVICE_ALLOC to a non-zero value to force the driver to always use device memory for physical storage. When this environment variable is set to a non-zero value, all devices used in that process that support managed memory have to be peer-to-peer compatible with each other. The error *cudaErrorInvalidDevice* will be returned if a device that supports managed memory is used and it is not peer-to-peer compatible with any of the other managed memory supporting devices that were previously used in that process, even if *cudaDeviceReset* has been called on those devices. These environment variables are described in the CUDA programming guide under the "CUDA environment variables" section.

See also:

cudaMallocPitch, *cudaFree*, *cudaMallocArray*, *cudaFreeArray*, *cudaMalloc3D*, *cudaMalloc3DArray*, *cudaMallocHost (C API)*, *cudaFreeHost*, *cudaHostAlloc*, *cudaDeviceGetAttribute*, *cudaStreamAttachMemAsync*, ::cuMemAllocManaged

---

Note: Note that this function may also return error codes from previous, asynchronous launches.

Note: Note that this function may also return *cudaErrorInitializationError*, *cudaErrorInsufficientDriver* or *cudaErrorNoDevice* if this call tries to initialize internal CUDA RT state.

Note: Note that as specified by *cudaStreamAddCallback* no CUDA function may be called from callback. *cudaErrorNotPermitted* may, but is not guaranteed to, be returned as a diagnostic in such case.

Parameters

- **`flags`** – Memory can be accessed by any stream on any device
- **`devPtr`** – - Pointer to allocated device memory
- **`size`** – - Requested allocation size in bytes

- **flags** – - Must be either *cudaMemAttachGlobal* or *cudaMemAttachHost* (defaults to *cudaMemAttachGlobal*)

Returns

cudaSuccess, *cudaErrorMemoryAllocation*, *cudaErrorNotSupported*, *cudaErrorInvalidValue*

__host__ *cudaError_t* **cudaMallocMipmappedArray**(*cudaMipmappedArray_t* *mipmappedArray, const struct *cudaChannelFormatDesc* *desc, struct *cudaExtent* extent, unsigned int numLevels, unsigned int flags = 0)

Allocate a mipmapped array on the device.

Allocates a CUDA mipmapped array according to the *cudaChannelFormatDesc* structure `desc` and returns a handle to the new CUDA mipmapped array in `*mipmappedArray`. `numLevels` specifies the number of mipmap levels to be allocated. This value is clamped to the range [1, 1 + floor(log2(max(width, height, depth)))].

The *cudaChannelFormatDesc* is defined as:

```
struct cudaChannelFormatDesc {
    int x, y, z, w;
    enum cudaChannelFormatKind f;
};
```

where *cudaChannelFormatKind* is one of *cudaChannelFormatKindSigned*, *cudaChannelFormatKindUnsigned*, or *cudaChannelFormatKindFloat*.

cudaMallocMipmappedArray() can allocate the following:

- A 1D mipmapped array is allocated if the height and depth extents are both zero.
- A 2D mipmapped array is allocated if only the depth extent is zero.
- A 3D mipmapped array is allocated if all three extents are non-zero.
- A 1D layered CUDA mipmapped array is allocated if only the height extent is zero and the cudaArrayLayered flag is set. Each layer is a 1D mipmapped array. The number of layers is determined by the depth extent.
- A 2D layered CUDA mipmapped array is allocated if all three extents are non-zero and the cudaArrayLayered flag is set. Each layer is a 2D mipmapped array. The number of layers is determined by the depth extent.
- A cubemap CUDA mipmapped array is allocated if all three extents are non-zero and the cudaArrayCubemap flag is set. Width must be equal to height, and depth must be six. The order of the six layers in memory is the same as that listed in *cudaGraphicsCubeFace*.
- A cubemap layered CUDA mipmapped array is allocated if all three extents are non-zero, and both, cudaArrayCubemap and cudaArrayLayered flags are set. Width must be equal to height, and depth must be a multiple of six. A cubemap layered CUDA mipmapped array is a special type of 2D layered CUDA mipmapped array that consists of a collection of cubemap mipmapped arrays. The first six layers represent the first cubemap mipmapped array, the next six layers form the second cubemap mipmapped array, and so on.

The `flags` parameter enables different options to be specified that affect the allocation, as follows.

- *cudaArrayDefault*: This flag's value is defined to be 0 and provides default mipmapped array allocation
- *cudaArrayLayered*: Allocates a layered CUDA mipmapped array, with the depth extent indicating the number of layers
- *cudaArrayCubemap*: Allocates a cubemap CUDA mipmapped array. Width must be equal to height, and depth must be six. If the cudaArrayLayered flag is also set, depth must be a multiple of six.
- *cudaArraySurfaceLoadStore*: This flag indicates that individual mipmap levels of the CUDA mipmapped array will be read from or written to using a surface reference.
- *cudaArrayTextureGather*: This flag indicates that texture gather operations will be performed on the CUDA array. Texture gather can only be performed on 2D CUDA mipmapped arrays, and the gather operations are performed only on the most detailed mipmap level.
- *cudaArraySparse*: Allocates a CUDA mipmapped array without physical backing memory. The subregions within this sparse array can later be mapped onto a physical memory allocation by calling ::cuMemMapArrayAsync. This flag can only be used for creating 2D, 3D or 2D layered sparse CUDA mipmapped arrays. The physical backing memory must be allocated via ::cuMemCreate.
- *cudaArrayDeferredMapping*: Allocates a CUDA mipmapped array without physical backing memory. The entire array can later be mapped onto a physical memory allocation by calling ::cuMemMapArrayAsync. The physical backing memory must be allocated via ::cuMemCreate.

The width, height and depth extents must meet certain size requirements as listed in the following table. All values are specified in elements.

CUDA array type	Valid extents that must always be met {(width range in elements), (height range), (depth range)}	Valid extents with cudaArraySurfaceLoadStore set {(width range in elements), (height range), (depth range)}
1D	{ (1,maxTexture1DMipmap), 0, 0 }	{ (1,maxSurface1D), 0, 0 }
2D	{ (1,maxTexture2DMipmap[0]), (1,maxTexture2DMipmap[1]), 0 }	{ (1,maxSurface2D[0]), (1,maxSurface2D[1]), 0 }
3D	{ (1,maxTexture3D[0]), (1,maxTexture3D[1]), (1,maxTexture3D[2]) } OR { (1,maxTexture3DAlt[0]), (1,maxTexture3DAlt[1]), (1,maxTexture3DAlt[2]) }	{ (1,maxSurface3D[0]), (1,maxSurface3D[1]), (1,maxSurface3D[2]) }
1D Layered	{ (1,maxTexture1DLayered[0]), 0, (1,maxTexture1DLayered[1]) }	{ (1,maxSurface1DLayered[0]), 0, (1,maxSurface1DLayered[1]) }
2D Layered	{ (1,maxTexture2DLayered[0]), (1,maxTexture2DLayered[1]), (1,maxTexture2DLayered[2]) }	{ (1,maxSurface2DLayered[0]), (1,maxSurface2DLayered[1]), (1,maxSurface2DLayered[2]) }
Cubemap	{ (1,maxTextureCubemap), (1,maxTextureCubemap), 6 }	{ (1,maxSurfaceCubemap), (1,maxSurfaceCubemap), 6 }
Cubemap Layered	{ (1,maxTextureCubemapLayered[0]), (1,maxTextureCubemapLayered[0]), (1,maxTextureCubemapLayered[1]) }	{ (1,maxSurfaceCubemapLayered[0]), (1,maxSurfaceCubemapLayered[0]), (1,maxSurfaceCubemapLayered[1]) }

See also:

cudaMalloc3D, *cudaMalloc*, *cudaMallocPitch*, *cudaFree*, *cudaFreeArray*, *cudaMallocHost (C API)*, *cudaFreeHost*, *cudaHostAlloc*, *make_cudaExtent*, ::cuMipmappedArrayCreate

Note: Note that this function may also return error codes from previous, asynchronous launches.

Note: Note that this function may also return *cudaErrorInitializationError*, *cudaErrorInsufficientDriver* or *cudaErrorNoDevice* if this call tries to initialize internal CUDA RT state.

Note: Note that as specified by *cudaStreamAddCallback* no CUDA function may be called from callback. *cudaErrorNotPermitted* may, but is not guaranteed to, be returned as a diagnostic in such case.

Parameters

- **`mipmappedArray`** – - Pointer to allocated mipmapped array in device memory
- **`desc`** – - Requested channel format
- **`extent`** – - Requested allocation size (`width` field in elements)

- **numLevels** – - Number of mipmap levels to allocate
- **flags** – - Flags for extensions

Returns

cudaSuccess, *cudaErrorInvalidValue*, *cudaErrorMemoryAllocation*

__host__ *cudaError_t* **cudaMallocPitch**(void **devPtr, size_t *pitch, size_t width, size_t height)

Allocates pitched memory on the device.

Allocates at least `width` (in bytes) * `height` bytes of linear memory on the device and returns in `*devPtr` a pointer to the allocated memory. The function may pad the allocation to ensure that corresponding pointers in any given row will continue to meet the alignment requirements for coalescing as the address is updated from row to row. The pitch returned in `*pitch` by *cudaMallocPitch()* is the width in bytes of the allocation. The intended usage of `pitch` is as a separate parameter of the allocation, used to compute addresses within the 2D array. Given the row and column of an array element of type T, the address is computed as:

```
T* pElement = (T*)((char*)BaseAddress + Row * pitch) + Column;
```

For allocations of 2D arrays, it is recommended that programmers consider performing pitch allocations using *cudaMallocPitch()*. Due to pitch alignment restrictions in the hardware, this is especially true if the application will be performing 2D memory copies between different regions of device memory (whether linear memory or CUDA arrays).

See also:

cudaMalloc, *cudaFree*, *cudaMallocArray*, *cudaFreeArray*, *cudaMallocHost (C API)*, *cudaFreeHost*, *cudaMalloc3D*, *cudaMalloc3DArray*, *cudaHostAlloc*, ::cuMemAllocPitch

---

Note: Note that this function may also return error codes from previous, asynchronous launches.

Note: Note that this function may also return *cudaErrorInitializationError*, *cudaErrorInsufficientDriver* or *cudaErrorNoDevice* if this call tries to initialize internal CUDA RT state.

Note: Note that as specified by *cudaStreamAddCallback* no CUDA function may be called from callback. *cudaErrorNotPermitted* may, but is not guaranteed to, be returned as a diagnostic in such case.

Parameters

- **devPtr** – - Pointer to allocated pitched device memory
- **pitch** – - Pitch for allocation
- **width** – - Requested pitched allocation width (in bytes)
- **height** – - Requested pitched allocation height

Returns

cudaSuccess, *cudaErrorInvalidValue*, *cudaErrorMemoryAllocation*

__host__ *cudaError_t* **cudaMemAdvise**(const void *devPtr, size_t count, enum *cudaMemoryAdvise* advice, struct *cudaMemLocation* location)

Advise about the usage of a given memory range.

Advise the Unified Memory subsystem about the usage pattern for the memory range starting at `devPtr` with a size of `count` bytes. The start address and end address of the memory range will be rounded down and rounded up respectively to be aligned to CPU page size before the advice is applied. The memory range must refer to managed memory allocated via *cudaMallocManaged* or declared via **managed** variables. The memory range could also refer to system-allocated pageable memory provided it represents a valid, host-accessible region of memory and all additional constraints imposed by `advice` as outlined below are also satisfied. Specifying an invalid system-allocated pageable memory range results in an error being returned.

The `advice` parameter can take the following values:

- *cudaMemAdviseSetReadMostly*: This implies that the data is mostly going to be read from and only occasionally written to. Any read accesses from any processor to this region will create a read-only copy of at least the accessed pages in that processor's memory. Additionally, if *cudaMemPrefetchAsync* or *cudaMemPrefetchAsync* is called on this region, it will create a read-only copy of the data on the destination processor. If the target location for *cudaMemPrefetchAsync* is a host NUMA node and a read-only copy already exists on another host NUMA node, that copy will be migrated to the targeted host NUMA node. If any processor writes to this region, all copies of the corresponding page will be invalidated except for the one where the write occurred. If the writing processor is the CPU and the preferred location of the page is a host NUMA node, then the page will also be migrated to that host NUMA node. The `location` argument is ignored for this advice. Note that for a page to be read-duplicated, the accessing processor must either be the CPU or a GPU that has a non-zero value for the device attribute *cudaDevAttrConcurrentManagedAccess*. Also, if a context is created on a device that does not have the device attribute *cudaDevAttrConcurrentManagedAccess* set, then read-duplication will not occur until all such contexts are destroyed. If the memory region refers to valid system-allocated pageable memory, then the accessing device must have a non-zero value for the device attribute *cudaDevAttrPageableMemoryAccess* for a read-only copy to be created on that device. Note however that if the accessing device also has a non-zero value for the device attribute *cudaDevAttrPageableMemoryAccessUsesHostPageTables*, then setting this advice will not create a read-only copy when that device accesses this memory region.
- ::cudaMemAdviceUnsetReadMostly: Undoes the effect of *cudaMemAdviseSetReadMostly* and also prevents the Unified Memory driver from attempting heuristic read-duplication on the memory range. Any read-duplicated copies of the data will be collapsed into a single copy. The location for the collapsed copy will be the preferred location if the page has a preferred location and one of the read-duplicated copies was resident at that location. Otherwise, the location chosen is arbitrary. Note: The `location` argument is ignored for this advice.
- *cudaMemAdviseSetPreferredLocation*: This advice sets the preferred location for the data to be the memory belonging to `location`. When *cudaMemLocation::type* is *cudaMemLocationTypeHost*, *cudaMemLocation::id* is ignored and the preferred location is set to be host memory. To set the preferred location to a specific host NUMA node, applications must set *cudaMemLocation::type* to *cudaMemLocationTypeHostNuma* and *cudaMemLocation::id* must specify the NUMA ID of the host NUMA node. If *cudaMemLocation::type* is set to *cudaMemLocationTypeHostNumaCurrent*, *cudaMemLocation::id* will be ignored and the host NUMA node closest to the calling thread's CPU will be used as the preferred location. If *cudaMemLocation::type* is a *cudaMemLocationTypeDevice*, then *cudaMemLocation::id* must be a valid device ordinal and the device must have a non-zero value for the device attribute *cudaDevAttrConcurrentManagedAccess*. Setting the preferred location does not cause data

to migrate to that location immediately. Instead, it guides the migration policy when a fault occurs on that memory region. If the data is already in its preferred location and the faulting processor can establish a mapping without requiring the data to be migrated, then data migration will be avoided. On the other hand, if the data is not in its preferred location or if a direct mapping cannot be established, then it will be migrated to the processor accessing it. It is important to note that setting the preferred location does not prevent data prefetching done using *cudaMemPrefetchAsync*. Having a preferred location can override the page thrash detection and resolution logic in the Unified Memory driver. Normally, if a page is detected to be constantly thrashing between for example host and device memory, the page may eventually be pinned to host memory by the Unified Memory driver. But if the preferred location is set as device memory, then the page will continue to thrash indefinitely. If *cudaMemAdviseSetReadMostly* is also set on this memory region or any subset of it, then the policies associated with that advice will override the policies of this advice, unless read accesses from `location` will not result in a read-only copy being created on that procesor as outlined in description for the advice *cudaMemAdviseSetReadMostly*. If the memory region refers to valid system-allocated pageable memory, and *cudaMemLocation::type* is *cudaMemLocationTypeDevice* then *cudaMemLocation::id* must be a valid device that has a non-zero alue for the device attribute *cudaDevAttrPageableMemoryAccess*.

- *cudaMemAdviseUnsetPreferredLocation*: Undoes the effect of *cudaMemAdviseSetPreferredLocation* and changes the preferred location to none. The `location` argument is ignored for this advice.
- *cudaMemAdviseSetAccessedBy*: This advice implies that the data will be accessed by processor `location`. The *cudaMemLocation::type* must be either *cudaMemLocationTypeDevice* with *cudaMemLocation::id* representing a valid device ordinal or *cudaMemLocationTypeHost* and *cudaMemLocation::id* will be ignored. All other location types are invalid. If *cudaMemLocation::id* is a GPU, then the device attribute *cudaDevAttrConcurrentManagedAccess* must be non-zero. This advice does not cause data migration and has no impact on the location of the data per se. Instead, it causes the data to always be mapped in the specified processor's page tables, as long as the location of the data permits a mapping to be established. If the data gets migrated for any reason, the mappings are updated accordingly. This advice is recommended in scenarios where data locality is not important, but avoiding faults is. Consider for example a system containing multiple GPUs with peer-to-peer access enabled, where the data located on one GPU is occasionally accessed by peer GPUs. In such scenarios, migrating data over to the other GPUs is not as important because the accesses are infrequent and the overhead of migration may be too high. But preventing faults can still help improve performance, and so having a mapping set up in advance is useful. Note that on CPU access of this data, the data may be migrated to host memory because the CPU typically cannot access device memory directly. Any GPU that had the *cudaMemAdviseSetAccessedBy* flag set for this data will now have its mapping updated to point to the page in host memory. If *cudaMemAdviseSetReadMostly* is also set on this memory region or any subset of it, then the policies associated with that advice will override the policies of this advice. Additionally, if the preferred location of this memory region or any subset of it is also `location`, then the policies associated with ::CU_MEM_ADVISE_SET_PREFERRED_LOCATION will override the policies of this advice. If the memory region refers to valid system-allocated pageable memory, and *cudaMemLocation::type* is *cudaMemLocationTypeDevice* then device in *cudaMemLocation::id* must have a non-zero value for the device attribute *cudaDevAttrPageableMemoryAccess*. Additionally, if *cudaMemLocation::id* has a non-zero value for the device attribute *cudaDevAttrPageableMemoryAccessUsesHostPageTables*, then this call has no effect.
- ::CU_MEM_ADVISE_UNSET_ACCESSED_BY: Undoes the effect of *cudaMemAdviseSetAccessedBy*. Any mappings to the data from `location` may be removed at any time causing accesses to result in non-fatal page faults. If the memory region refers to valid system-

allocated pageable memory, and *cudaMemLocation::type* is *cudaMemLocationTypeDevice* then device in *cudaMemLocation::id* must have a non-zero value for the device attribute *cudaDevAttrPageableMemoryAccess*. Additionally, if *cudaMemLocation::id* has a non-zero value for the device attribute *cudaDevAttrPageableMemoryAccessUsesHostPageTables*, then this call has no effect.

See also:

cudaMemcpy, *cudaMemcpyPeer*, *cudaMemcpyAsync*, *cudaMemcpy3DPeerAsync*, *cudaMemPrefetchAsync*, ::cuMemAdvise

---

Note: Note that this function may also return error codes from previous, asynchronous launches.

Note: This function exhibits asynchronous behavior for most use cases.

Note: This function uses standard default stream semantics.

Note: Note that this function may also return *cudaErrorInitializationError*, *cudaErrorInsufficientDriver* or *cudaErrorNoDevice* if this call tries to initialize internal CUDA RT state.

Note: Note that as specified by *cudaStreamAddCallback* no CUDA function may be called from callback. *cudaErrorNotPermitted* may, but is not guaranteed to, be returned as a diagnostic in such case.

Parameters

- **devPtr** – - Pointer to memory to set the advice for
- **count** – - Size in bytes of the memory range
- **advice** – - Advice to be applied for the specified memory range
- **location** – - location to apply the advice for

Returns

cudaSuccess, *cudaErrorInvalidValue*, *cudaErrorInvalidDevice*, *cudaErrorNotSupported*

__host__ *cudaError_t* **cudaMemDiscardAndPrefetchBatchAsync**(void **dptrs, size_t *sizes, size_t count, struct *cudaMemLocation* *prefetchLocs, size_t *prefetchLocIdxs, size_t numPrefetchLocs, unsigned long long flags, *cudaStream_t* stream)

Performs a batch of memory discards and prefetches asynchronously.

Performs a batch of memory discards followed by prefetches. The batch as a whole executes in stream order but operations within a batch are not guaranteed to execute in any specific order. All devices in the system must have a non-zero value for the device attribute *cudaDevAttrConcurrentManagedAccess* otherwise the API will return an error.

Calling *cudaMemDiscardAndPrefetchBatchAsync* is semantically equivalent to calling *cudaMemDiscardBatchAsync* followed by *cudaMemPrefetchBatchAsync*, but is more optimal. For more details on what discarding and prefetching imply, please refer to *cudaMemDiscardBatchAsync* and *cudaMemPrefetchBatchAsync* respectively. Note that any reads, writes or prefetches to any part of the memory range that occur simultaneously with this combined discard+prefetch operation result in undefined behavior.

Performs memory discard and prefetch on address ranges specified in `dptrs` and `sizes`. Both arrays must be of the same length as specified by `count`. Each memory range specified must refer to managed memory allocated via *cudaMallocManaged* or declared via **managed** variables or it may also refer to system-allocated memory when all devices have a non-zero value for *cudaDevAttrPageableMemoryAccess*. Every operation in the batch has to be associated with a valid location to prefetch the address range to and specified in the `prefetchLocs` array. Each entry in this array can apply to more than one operation. This can be done by specifying in the `prefetchLocIdxs` array, the index of the first operation that the corresponding entry in the `prefetchLocs` array applies to. Both `prefetchLocs` and `prefetchLocIdxs` must be of the same length as specified by `numPrefetchLocs`. For example, if a batch has 10 operations listed in dptrs/sizes, the first 6 of which are to be prefetched to one location and the remaining 4 are to be prefetched to another, then `numPrefetchLocs` will be 2, `prefetchLocIdxs` will be {0, 6} and `prefetchLocs` will contain the two set of locations. Note the first entry in `prefetchLocIdxs` must always be 0. Also, each entry must be greater than the previous entry and the last entry should be less than `count`. Furthermore, `numPrefetchLocs` must be lesser than or equal to `count`.

Parameters

- **`dptrs`** – - Array of pointers to be discarded
- **`sizes`** – - Array of sizes for memory discard operations.
- **`count`** – - Size of `dptrs` and `sizes` arrays.
- **`prefetchLocs`** – - Array of locations to prefetch to.
- **`prefetchLocIdxs`** – - Array of indices to specify which operands each entry in the `prefetchLocs` array applies to. The locations specified in prefetchLocs[k] will be applied to operations starting from prefetchLocIdxs[k] through prefetchLocIdxs[k+1] - 1. Also prefetchLocs[numPrefetchLocs - 1] will apply to copies starting from prefetchLocIdxs[numPrefetchLocs - 1] through count - 1.
- **`numPrefetchLocs`** – - Size of `prefetchLocs` and `prefetchLocIdxs` arrays.
- **`flags`** – - Flags reserved for future use. Must be zero.
- **`stream`** – - The stream to enqueue the operations in. Must not be legacy NULL stream.

__host__ *cudaError_t* **cudaMemDiscardBatchAsync**(void **dptrs, size_t *sizes, size_t count, unsigned long long flags, *cudaStream_t* stream)

Performs a batch of memory discards asynchronously.

Performs a batch of memory discards. The batch as a whole executes in stream order but operations within a batch are not guaranteed to execute in any specific order. All devices in the system must have a non-zero value for the device attribute *cudaDevAttrConcurrentManagedAccess* otherwise the API will return an error.

Discarding a memory range informs the driver that the contents of that range are no longer useful. Discarding memory ranges allows the driver to optimize certain data migrations and can also help reduce memory pressure. This operation can be undone on any part of the range by either writing to it or prefetching it via *cudaMemPrefetchAsync* or *cudaMemPrefetchBatchAsync*. Reading from a discarded range, without a subsequent write or prefetch to that part of the range, will return an indeterminate value. Note that any reads, writes or prefetches to any part of the memory range that occur simultaneously with the discard operation result in undefined behavior.

Performs memory discard on address ranges specified in `dptrs` and `sizes`. Both arrays must be of the same length as specified by `count`. Each memory range specified must refer to managed memory allocated via *cudaMallocManaged* or declared via **managed** variables or it may also refer to system-allocated memory when all devices have a non-zero value for *cudaDevAttrPageableMemoryAccess*.

Parameters

- **`dptrs`** – - Array of pointers to be discarded
- **`sizes`** – - Array of sizes for memory discard operations.
- **`count`** – - Size of `dptrs` and `sizes` arrays.
- **`flags`** – - Flags reserved for future use. Must be zero.
- **`stream`** – - The stream to enqueue the operations in. Must not be legacy NULL stream.

__host__ *cudaError_t* **`cudaMemGetInfo`**(size_t *free, size_t *total)

Gets free and total device memory.

Returns in `*total` the total amount of memory available to the the current context. Returns in `*free` the amount of memory on the device that is free according to the OS. CUDA is not guaranteed to be able to allocate all of the memory that the OS reports as free. In a multi-tenet situation, free estimate returned is prone to race condition where a new allocation/free done by a different process or a different thread in the same process between the time when free memory was estimated and reported, will result in deviation in free value reported and actual free memory.

The integrated GPU on Tegra shares memory with CPU and other component of the SoC. The free and total values returned by the API excludes the SWAP memory space maintained by the OS on some platforms. The OS may move some of the memory pages into swap area as the GPU or CPU allocate or access memory. See Tegra app note on how to calculate total and free memory on Tegra.

See also:

::cuMemGetInfo

Note: Note that this function may also return error codes from previous, asynchronous launches.

Note: Note that this function may also return *cudaErrorInitializationError*, *cudaErrorInsufficientDriver* or *cudaErrorNoDevice* if this call tries to initialize internal CUDA RT state.

Note: Note that as specified by *cudaStreamAddCallback* no CUDA function may be called from callback. *cudaErrorNotPermitted* may, but is not guaranteed to, be returned as a diagnostic in such case.

Parameters

- **free** – - Returned free memory in bytes
- **total** – - Returned total memory in bytes

Returns
cudaSuccess, *cudaErrorInvalidValue*, *cudaErrorLaunchFailure*

__host__ *cudaError_t* **cudaMemGetLocationInfo**(void *devPtr, size_t size, size_t summaryGranularity, size_t samplingGranularity, struct *cudaMemLocation* *location_out)

Gets location information for a memory address range.

Retrieves memory location information for the specified address range starting at `ptr` with size `size`. The API determines the most common location by sampling memory at intervals defined by `samplingGranularity` within the whole interval.

The location information is returned in the `location_out` array, with one entry per summary region. The total number of locations returned will be ceil(size/summaryGranularity). The user is expected to allocate the `location_out` array with sufficient memory.

For example, with an address range of 1GB, a `summaryGranularity` of 128MB, and a `samplingGranularity` of 2MB, the function will:

- Divide the 1GB range into 8 summary regions of 128MB each
- Within each 128MB region, sample every 2MB to determine the most common location. If there is a tie a random winner is chosen.
- Populate the `location_out` array with 8 entries, one for each 128MB region `summaryGranularity` should be less than or equal to `size` and greater than 0. `samplingGranularity` should be less than or equal to `summaryGranularity`. If the `samplingGranularity` is set to 0 it is set to a system dependent default value. In all other cases, the call returns *cudaErrorInvalidValue*.

When the memory is not resident on any processor, the call returns *cudaSuccess* and the returned location type for that interval is *cudaMemLocationTypeNone*.

The memory range must refer to one of the following:

- Managed memory allocated via *cudaMallocManaged*, via *cudaMallocFromPoolAsync* from a managed memory pool or declared via **managed** variables.
- System-allocated pageable memory that is not registered via *cudaHostRegister*. If the memory range does not refer to one of the above, the call returns *cudaErrorInvalidValue*.

All devices on the system must have non-zero value for the device attribute *cudaDevAttrConcurrentManagedAccess*. If not, this call returns *cudaErrorNotSupported*.

Note: Note that this function may also return error codes from previous, asynchronous launches.

Parameters

- **ptr** – - Starting address of the memory range to query
- **size** – - Size in bytes of the memory range to query
- **summaryGranularity** – - Granularity in bytes at which to summarize location information
- **samplingGranularity** – - Granularity in bytes at which to sample memory within each summary region
- **location_out** – - Array to store location information, one entry per summary region

Returns

cudaSuccess, *cudaErrorInvalidValue*, *cudaErrorNotSupported*

__host__ *cudaError_t* **cudaMemPrefetchAsync**(const void *devPtr, size_t count, struct *cudaMemLocation* location, unsigned int flags, *cudaStream_t* stream = 0)

Prefetches memory to the specified destination location.

Prefetches memory to the specified destination location. `devPtr` is the base device pointer of the memory to be prefetched and `location` specifies the destination location. `count` specifies the number of bytes to copy. `stream` is the stream in which the operation is enqueued. The memory range must refer to managed memory allocated via *cudaMallocManaged* or declared via **managed** variables, or it may also refer to memory allocated from a managed memory pool, or it may also refer to system-allocated memory on systems with non-zero cudaDevAttrPageableMemoryAccess.

Specifying *cudaMemLocationTypeDevice* for *cudaMemLocation::type* will prefetch memory to GPU specified by device ordinal *cudaMemLocation::id* which must have non-zero value for the device attribute ::concurrentManagedAccess. Additionally, `stream` must be associated with a device that has a non-zero value for the device attribute ::concurrentManagedAccess. Specifying *cudaMemLocationTypeHost* as *cudaMemLocation::type* will prefetch data to host memory. Applications can request prefetching memory to a specific host NUMA node by specifying *cudaMemLocationTypeHostNuma* for *cudaMemLocation::type* and a valid host NUMA node id in *cudaMemLocation::id* Users can also request prefetching memory to the host NUMA node closest to the current thread's CPU by specifying *cudaMemLocationTypeHostNumaCurrent* for *cudaMemLocation::type*. Note when *cudaMemLocation::type* is etiher *cudaMemLocationTypeHost* OR *cudaMemLocationTypeHostNumaCurrent*, *cudaMemLocation::id* will be ignored. Prefetching to *cudaMemLocationTypeDeviceLocalityDomain* locations is not supported.

The start address and end address of the memory range will be rounded down and rounded up respectively to be aligned to CPU page size before the prefetch operation is enqueued in the stream.

If no physical memory has been allocated for this region, then this memory region will be populated and mapped on the destination device. If there's insufficient memory to prefetch the desired region, the Unified Memory driver may evict pages from other *cudaMallocManaged* allocations to host memory in order to make room. Device memory allocated using *cudaMalloc* or *cudaMallocArray* will not be evicted.

By default, any mappings to the previous location of the migrated pages are removed and mappings for the new location are only setup on the destination location. The exact behavior however also depends on the settings applied to this memory range via ::cuMemAdvise as described below:

If *cudaMemAdviseSetReadMostly* was set on any subset of this memory range, then that subset will create a read-only copy of the pages on destination location. If however the destination location is a host NUMA node, then any pages of that subset that are already in another host NUMA node will be transferred to the destination.

If *cudaMemAdviseSetPreferredLocation* was called on any subset of this memory range, then the pages will be migrated to `location` even if `location` is not the preferred location of any pages in the memory range.

If *cudaMemAdviseSetAccessedBy* was called on any subset of this memory range, then mappings to those pages from all the appropriate processors are updated to refer to the new location if establishing such a mapping is possible. Otherwise, those mappings are cleared.

Note that this API is not required for functionality and only serves to improve performance by allowing the application to migrate data to a suitable location before it is accessed. Memory accesses to this range are always coherent and are allowed even when the data is actively being migrated.

Note that this function is asynchronous with respect to the host and all work on other devices.

See also:

cudaMemcpy, *cudaMemcpyPeer*, *cudaMemcpyAsync*, *cudaMemcpy3DPeerAsync*, *cudaMemAdvise*, ::cuMemPrefetchAsync

---

Note: Note that this function may also return error codes from previous, asynchronous launches.

Note: This function exhibits asynchronous behavior for most use cases.

Note: This function uses standard default stream semantics.

Note: Note that this function may also return *cudaErrorInitializationError*, *cudaErrorInsufficientDriver* or *cudaErrorNoDevice* if this call tries to initialize internal CUDA RT state.

Note: Note that as specified by *cudaStreamAddCallback* no CUDA function may be called from callback. *cudaErrorNotPermitted* may, but is not guaranteed to, be returned as a diagnostic in such case.

Parameters

- **`devPtr`** – - Pointer to be prefetched
- **`count`** – - Size in bytes

- **location** – - location to prefetch to
- **flags** – - flags for future use, must be zero now.
- **stream** – - Stream to enqueue prefetch operation

Returns

cudaSuccess, *cudaErrorInvalidValue*, *cudaErrorInvalidDevice*, *cudaErrorNotSupported*

__host__ *cudaError_t* **cudaMemPrefetchBatchAsync**(void **dptrs, size_t *sizes, size_t count, struct *cudaMemLocation* *prefetchLocs, size_t *prefetchLocIdxs, size_t numPrefetchLocs, unsigned long long flags, *cudaStream_t* stream)

Performs a batch of memory prefetches asynchronously.

Performs a batch of memory prefetches. The batch as a whole executes in stream order but operations within a batch are not guaranteed to execute in any specific order. All devices in the system must have a non-zero value for the device attribute *cudaDevAttrConcurrentManagedAccess* otherwise the API will return an error.

The semantics of the individual prefetch operations are as described in *cudaMemPrefetchAsync*.

Performs memory prefetch on address ranges specified in `dptrs` and `sizes`. Both arrays must be of the same length as specified by `count`. Each memory range specified must refer to managed memory allocated via *cudaMallocManaged* or declared via **managed** variables or it may also refer to system-allocated memory when all devices have a non-zero value for *cudaDevAttrPageableMemoryAccess*. The prefetch location for every operation in the batch is specified in the `prefetchLocs` array. Each entry in this array can apply to more than one operation. This can be done by specifying in the `prefetchLocIdxs` array, the index of the first prefetch operation that the corresponding entry in the `prefetchLocs` array applies to. Both `prefetchLocs` and `prefetchLocIdxs` must be of the same length as specified by `numPrefetchLocs`. For example, if a batch has 10 prefetches listed in dptrs/sizes, the first 4 of which are to be prefetched to one location and the remaining 6 are to be prefetched to another, then `numPrefetchLocs` will be 2, `prefetchLocIdxs` will be {0, 4} and `prefetchLocs` will contain the two locations. Note the first entry in `prefetchLocIdxs` must always be 0. Also, each entry must be greater than the previous entry and the last entry should be less than `count`. Furthermore, `numPrefetchLocs` must be lesser than or equal to `count`.

Parameters

- **dptrs** – - Array of pointers to be prefetched
- **sizes** – - Array of sizes for memory prefetch operations.
- **count** – - Size of `dptrs` and `sizes` arrays.
- **prefetchLocs** – - Array of locations to prefetch to.
- **prefetchLocIdxs** – - Array of indices to specify which operands each entry in the `prefetchLocs` array applies to. The locations specified in prefetchLocs[k] will be applied to copies starting from prefetchLocIdxs[k] through prefetchLocIdxs[k+1] - 1. Also prefetchLocs[numPrefetchLocs - 1] will apply to prefetches starting from prefetchLocIdxs[numPrefetchLocs - 1] through count - 1.
- **numPrefetchLocs** – - Size of `prefetchLocs` and `prefetchLocIdxs` arrays.
- **flags** – - Flags reserved for future use. Must be zero.

- **stream** – - The stream to enqueue the operations in. Must not be legacy NULL stream.

__host__ *cudaError_t* **cudaMemRangeGetAttribute**(void *data, size_t dataSize, enum *cudaMemRangeAttribute* attribute, const void *devPtr, size_t count)

Query an attribute of a given memory range.

Query an attribute about the memory range starting at `devPtr` with a size of `count` bytes. The memory range must refer to managed memory allocated via *cudaMallocManaged* or declared via **managed** variables.

The `attribute` parameter can take the following values:

- *cudaMemRangeAttributeReadMostly*: If this attribute is specified, `data` will be interpreted as a 32-bit integer, and `dataSize` must be 4. The result returned will be 1 if all pages in the given memory range have read-duplication enabled, or 0 otherwise.
- *cudaMemRangeAttributePreferredLocation*: If this attribute is specified, `data` will be interpreted as a 32-bit integer, and `dataSize` must be 4. The result returned will be a GPU device id if all pages in the memory range have that GPU as their preferred location, or it will be cudaCpuDeviceId if all pages in the memory range have the CPU as their preferred location, or it will be cudaInvalidDeviceId if either all the pages don't have the same preferred location or some of the pages don't have a preferred location at all. Note that the actual location of the pages in the memory range at the time of the query may be different from the preferred location.
- *cudaMemRangeAttributeAccessedBy*: If this attribute is specified, `data` will be interpreted as an array of 32-bit integers, and `dataSize` must be a non-zero multiple of 4. The result returned will be a list of device ids that had ::cudaMemAdviceSetAccessedBy set for that entire memory range. If any device does not have that advice set for the entire memory range, that device will not be included. If `data` is larger than the number of devices that have that advice set for that memory range, cudaInvalidDeviceId will be returned in all the extra space provided. For ex., if `dataSize` is 12 (i.e. `data` has 3 elements) and only device 0 has the advice set, then the result returned will be { 0, cudaInvalidDeviceId, cudaInvalidDeviceId }. If `data` is smaller than the number of devices that have that advice set, then only as many devices will be returned as can fit in the array. There is no guarantee on which specific devices will be returned, however.
- *cudaMemRangeAttributeLastPrefetchLocation*: If this attribute is specified, `data` will be interpreted as a 32-bit integer, and `dataSize` must be 4. The result returned will be the last location to which all pages in the memory range were prefetched explicitly via *cudaMemPrefetchAsync*. This will either be a GPU id or cudaCpuDeviceId depending on whether the last location for prefetch was a GPU or the CPU respectively. If any page in the memory range was never explicitly prefetched or if all pages were not prefetched to the same location, cudaInvalidDeviceId will be returned. Note that this simply returns the last location that the applicaton requested to prefetch the memory range to. It gives no indication as to whether the prefetch operation to that location has completed or even begun.
- *cudaMemRangeAttributePreferredLocationType*: If this attribute is specified, `data` will be interpreted as a *cudaMemLocationType*, and `dataSize` must be sizeof(cudaMemLocationType). The *cudaMemLocationType* returned will be *cudaMemLocationTypeDevice* if all pages in the memory range have the same GPU as their preferred location, or *cudaMemLocationType* will be *cudaMemLocationTypeHost* if all pages in the memory range have the CPU as their preferred location, or or it will be *cudaMemLocationTypeHostNuma* if all the pages in the memory range have the same host NUMA node ID as their preferred location or it will be cudaMemLocationTypeInvalid if either all the pages don't have the same preferred location or some of the pages don't have a preferred location

at all. Note that the actual location type of the pages in the memory range at the time of the query may be different from the preferred location type.

- *cudaMemRangeAttributePreferredLocationId*: If this attribute is specified, `data` will be interpreted as a 32-bit integer, and `dataSize` must be 4. If the *cudaMemRangeAttributePreferredLocationType* query for the same address range returns *cudaMemLocationTypeDevice*, it will be a valid device ordinal or if it returns *cudaMemLocationTypeHostNuma*, it will be a valid host NUMA node ID or if it returns any other location type, the id should be ignored.

- *cudaMemRangeAttributeLastPrefetchLocationType*: If this attribute is specified, `data` will be interpreted as a *cudaMemLocationType*, and `dataSize` must be sizeof(cudaMemLocationType). The result returned will be the last location type to which all pages in the memory range were prefetched explicitly via ::cuMemPrefetchAsync. The *cudaMemLocationType* returned will be *cudaMemLocationTypeDevice* if the last prefetch location was the GPU or *cudaMemLocationTypeHost* if it was the CPU or *cudaMemLocationTypeHostNuma* if the last prefetch location was a specific host NUMA node. If any page in the memory range was never explicitly prefetched or if all pages were not prefetched to the same location, ::CUmemLocationType will be cudaMemLocationTypeInvalid. Note that this simply returns the last location type that the application requested to prefetch the memory range to. It gives no indication as to whether the prefetch operation to that location has completed or even begun.

  - *cudaMemRangeAttributeLastPrefetchLocationId*: If this attribute is specified, `data` will be interpreted as a 32-bit integer, and `dataSize` must be 4. If the *cudaMemRangeAttributeLastPrefetchLocationType* query for the same address range returns *cudaMemLocationTypeDevice*, it will be a valid device ordinal or if it returns *cudaMemLocationTypeHostNuma*, it will be a valid host NUMA node ID or if it returns any other location type, the id should be ignored.

See also:

cudaMemRangeGetAttributes, *cudaMemPrefetchAsync*, *cudaMemAdvise*, ::cuMemRangeGetAttribute

---

Note: Note that this function may also return error codes from previous, asynchronous launches.

Note: This function exhibits asynchronous behavior for most use cases.

Note: This function uses standard default stream semantics.

Note: Note that this function may also return *cudaErrorInitializationError*, *cudaErrorInsufficientDriver* or *cudaErrorNoDevice* if this call tries to initialize internal CUDA RT state.

Note: Note that as specified by *cudaStreamAddCallback* no CUDA function may be called from callback. *cudaErrorNotPermitted* may, but is not guaranteed to, be returned as a diagnostic in such case.

Parameters

- **data** – - A pointers to a memory location where the result of each attribute query will be written to.
- **dataSize** – - Array containing the size of data
- **attribute** – - The attribute to query
- **devPtr** – - Start of the range to query
- **count** – - Size of the range to query

Returns

cudaSuccess, *cudaErrorInvalidValue*

__host__ *cudaError_t* **cudaMemRangeGetAttributes**(void **data, size_t *dataSizes, enum *cudaMemRangeAttribute* *attributes, size_t numAttributes, const void *devPtr, size_t count)

Query attributes of a given memory range.

Query attributes of the memory range starting at `devPtr` with a size of `count` bytes. The memory range must refer to managed memory allocated via *cudaMallocManaged* or declared via **managed** variables. The `attributes` array will be interpreted to have `numAttributes` entries. The `dataSizes` array will also be interpreted to have `numAttributes` entries. The results of the query will be stored in `data`.

The list of supported attributes are given below. Please refer to *cudaMemRangeGetAttribute* for attribute descriptions and restrictions.

- *cudaMemRangeAttributeReadMostly*
- *cudaMemRangeAttributePreferredLocation*
- *cudaMemRangeAttributeAccessedBy*
- *cudaMemRangeAttributeLastPrefetchLocation*
- :: cudaMemRangeAttributePreferredLocationType
- :: cudaMemRangeAttributePreferredLocationId
- :: cudaMemRangeAttributeLastPrefetchLocationType
- :: cudaMemRangeAttributeLastPrefetchLocationId

See also:

cudaMemRangeGetAttribute, *cudaMemAdvise*, *cudaMemPrefetchAsync*, ::cuMemRangeGetAttributes

---

Note: Note that this function may also return error codes from previous, asynchronous launches.

Note: Note that this function may also return *cudaErrorInitializationError*, *cudaErrorInsufficientDriver* or *cudaErrorNoDevice* if this call tries to initialize internal CUDA RT state.

Note: Note that as specified by *cudaStreamAddCallback* no CUDA function may be called from callback. *cudaErrorNotPermitted* may, but is not guaranteed to, be returned as a diagnostic in such case.

Parameters

- **data** – - A two-dimensional array containing pointers to memory locations where the result of each attribute query will be written to.
- **dataSizes** – - Array containing the sizes of each result
- **attributes** – - An array of attributes to query (numAttributes and the number of attributes in this array should match)
- **numAttributes** – - Number of attributes to query
- **devPtr** – - Start of the range to query
- **count** – - Size of the range to query

Returns

cudaSuccess, *cudaErrorInvalidValue*

__host__ *cudaError_t* **cudaMemcpy**(void *dst, const void *src, size_t count, enum *cudaMemcpyKind* kind)

Copies data between host and device.

Copies `count` bytes from the memory area pointed to by `src` to the memory area pointed to by `dst`, where `kind` specifies the direction of the copy, and must be one of *cudaMemcpyHostToHost*, *cudaMemcpyHostToDevice*, *cudaMemcpyDeviceToHost*, *cudaMemcpyDeviceToDevice*, or *cudaMemcpyDefault*. Passing *cudaMemcpyDefault* is recommended, in which case the type of transfer is inferred from the pointer values. However, *cudaMemcpyDefault* is only allowed on systems that support unified virtual addressing. Calling *cudaMemcpy()* with dst and src pointers that do not match the direction of the copy results in an undefined behavior.

See also:

cudaMemcpy2D, *cudaMemcpy2DToArray*, *cudaMemcpy2DFromArray*, *cudaMemcpy2DArrayToArray*, *cudaMemcpyToSymbol*, *cudaMemcpyFromSymbol*, *cudaMemcpyAsync*, *cudaMemcpy2DAsync*, *cudaMemcpy2DToArrayAsync*, *cudaMemcpy2DFromArrayAsync*, *cudaMemcpyToSymbolAsync*, *cudaMemcpyFromSymbolAsync*, ::cuMemcpyDtoH, ::cuMemcpyHtoD, ::cuMemcpyDtoD, ::cuMemcpy

Note: Note that this function may also return error codes from previous, asynchronous launches.

Note: Note that this function may also return *cudaErrorInitializationError*, *cudaErrorInsufficientDriver* or *cudaErrorNoDevice* if this call tries to initialize internal CUDA RT state.

Note: Note that as specified by *cudaStreamAddCallback* no CUDA function may be called from callback. *cudaErrorNotPermitted* may, but is not guaranteed to, be returned as a diagnostic in

such case.

Note: This function exhibits synchronous behavior for most use cases.

Note: Memory regions requested must be either entirely registered with CUDA, or in the case of host pageable transfers, not registered at all. Memory regions spanning over allocations that are both registered and not registered with CUDA are not supported and will return CUDA_ERROR_INVALID_VALUE.

Parameters

- **dst** – - Destination memory address
- **src** – - Source memory address
- **count** – - Size in bytes to copy
- **kind** – - Type of transfer

Returns

cudaSuccess, *cudaErrorInvalidValue*, *cudaErrorInvalidMemcpyDirection*

__host__ *cudaError_t* **cudaMemcpy2D**(void *dst, size_t dpitch, const void *src, size_t spitch, size_t width, size_t height, enum *cudaMemcpyKind* kind)

Copies data between host and device.

Copies a matrix (`height` rows of `width` bytes each) from the memory area pointed to by `src` to the memory area pointed to by `dst`, where `kind` specifies the direction of the copy, and must be one of *cudaMemcpyHostToHost*, *cudaMemcpyHostToDevice*, *cudaMemcpyDeviceToHost*, *cudaMemcpyDeviceToDevice*, or *cudaMemcpyDefault*. Passing *cudaMemcpyDefault* is recommended, in which case the type of transfer is inferred from the pointer values. However, *cudaMemcpyDefault* is only allowed on systems that support unified virtual addressing. `dpitch` and `spitch` are the widths in memory in bytes of the 2D arrays pointed to by `dst` and `src`, including any padding added to the end of each row. The memory areas may not overlap. `width` must not exceed either `dpitch` or `spitch`. Calling *cudaMemcpy2D()* with `dst` and `src` pointers that do not match the direction of the copy results in an undefined behavior. *cudaMemcpy2D()* returns an error if `dpitch` or `spitch` exceeds the maximum allowed.

See also:

cudaMemcpy, *cudaMemcpy2DToArray*, *cudaMemcpy2DFromArray*, *cudaMemcpy2DArrayToArray*, *cudaMemcpyToSymbol*, *cudaMemcpyFromSymbol*, *cudaMemcpyAsync*, *cudaMemcpy2DAsync*, *cudaMemcpy2DToArrayAsync*, *cudaMemcpy2DFromArrayAsync*, *cudaMemcpyToSymbolAsync*, *cudaMemcpyFromSymbolAsync*, ::cuMemcpy2D, ::cuMemcpy2DUnaligned

Note: Note that this function may also return error codes from previous, asynchronous launches.

Note: Note that this function may also return *cudaErrorInitializationError*, *cudaErrorInsufficientDriver* or *cudaErrorNoDevice* if this call tries to initialize internal CUDA RT state.

Note: Note that as specified by *cudaStreamAddCallback* no CUDA function may be called from callback. *cudaErrorNotPermitted* may, but is not guaranteed to, be returned as a diagnostic in such case.

Note: Memory regions requested must be either entirely registered with CUDA, or in the case of host pageable transfers, not registered at all. Memory regions spanning over allocations that are both registered and not registered with CUDA are not supported and will return CUDA_ERROR_INVALID_VALUE.

Parameters

- **dst** – - Destination memory address
- **dpitch** – - Pitch of destination memory
- **src** – - Source memory address
- **spitch** – - Pitch of source memory
- **width** – - Width of matrix transfer (columns in bytes)
- **height** – - Height of matrix transfer (rows)
- **kind** – - Type of transfer

Returns

cudaSuccess, *cudaErrorInvalidValue*, *cudaErrorInvalidPitchValue*, *cudaErrorInvalidMemcpyDirection*

__host__ *cudaError_t* **cudaMemcpy2DArrayToArray**(*cudaArray_t* dst, size_t wOffsetDst, size_t hOffsetDst, *cudaArray_const_t* src, size_t wOffsetSrc, size_t hOffsetSrc, size_t width, size_t height, enum *cudaMemcpyKind* kind = *cudaMemcpyDeviceToDevice*)

Copies data between host and device.

Copies a matrix (`height` rows of `width` bytes each) from the CUDA array `src` starting at `hOffsetSrc` rows and `wOffsetSrc` bytes from the upper left corner to the CUDA array `dst` starting at `hOffsetDst` rows and `wOffsetDst` bytes from the upper left corner, where `kind` specifies the direction of the copy, and must be one of *cudaMemcpyHostToHost*, *cudaMemcpyHostToDevice*, *cudaMemcpyDeviceToHost*, *cudaMemcpyDeviceToDevice*, or *cudaMemcpyDefault*. Passing *cudaMemcpyDefault* is recommended, in which case the type of transfer is inferred from the pointer values. However, *cudaMemcpyDefault* is only allowed on systems that support unified virtual addressing. `wOffsetDst` + `width` must not exceed the width of the CUDA array `dst`. `wOffsetSrc` + `width` must not exceed the width of the CUDA array `src`.

See also:

cudaMemcpy, *cudaMemcpy2D*, *cudaMemcpy2DToArray*, *cudaMemcpy2DFromArray*, *cudaMemcpyToSymbol*, *cudaMemcpyFromSymbol*, *cudaMemcpyAsync*, *cudaMemcpy2DAsync*, *cudaMemcpy2DToArrayAsync*, *cudaMemcpy2DFromArrayAsync*, *cudaMemcpyToSymbolAsync*, *cudaMemcpyFromSymbolAsync*, ::cuMemcpy2D, ::cuMemcpy2DUnaligned

---

Note: Note that this function may also return error codes from previous, asynchronous launches.

Note: This function exhibits synchronous behavior for most use cases.

Note: Note that this function may also return *cudaErrorInitializationError*, *cudaErrorInsufficientDriver* or *cudaErrorNoDevice* if this call tries to initialize internal CUDA RT state.

Note: Note that as specified by *cudaStreamAddCallback* no CUDA function may be called from callback. *cudaErrorNotPermitted* may, but is not guaranteed to, be returned as a diagnostic in such case.

Parameters

- **dst** – - Destination memory address
- **wOffsetDst** – - Destination starting X offset (columns in bytes)
- **hOffsetDst** – - Destination starting Y offset (rows)
- **src** – - Source memory address
- **wOffsetSrc** – - Source starting X offset (columns in bytes)
- **hOffsetSrc** – - Source starting Y offset (rows)
- **width** – - Width of matrix transfer (columns in bytes)
- **height** – - Height of matrix transfer (rows)
- **kind** – - Type of transfer

Returns

cudaSuccess, *cudaErrorInvalidValue*, *cudaErrorInvalidMemcpyDirection*

__host__ *cudaError_t* **cudaMemcpy2DAsync**(void *dst, size_t dpitch, const void *src, size_t spitch, size_t width, size_t height, enum *cudaMemcpyKind* kind, *cudaStream_t* stream = 0)

Copies data between host and device.

Copies a matrix (`height` rows of `width` bytes each) from the memory area pointed to by `src` to the memory area pointed to by `dst`, where `kind` specifies the direction of the copy, and must be one of *cudaMemcpyHostToHost*, *cudaMemcpyHostToDevice*, *cudaMemcpyDeviceToHost*, *cudaMemcpyDeviceToDevice*, or *cudaMemcpyDefault*. Passing *cudaMemcpyDefault* is recommended, in which case the type of transfer is inferred from the pointer values. However, *cudaMemcpyDefault* is only allowed on systems that support unified virtual addressing. `dpitch` and `spitch` are the widths in memory in bytes of the 2D arrays pointed to by `dst` and `src`, including any padding

added to the end of each row. The memory areas may not overlap. `width` must not exceed either `dpitch` or `spitch`.

Calling *cudaMemcpy2DAsync()* with `dst` and `src` pointers that do not match the direction of the copy results in an undefined behavior. *cudaMemcpy2DAsync()* returns an error if `dpitch` or `spitch` is greater than the maximum allowed.

cudaMemcpy2DAsync() is asynchronous with respect to the host, so the call may return before the copy is complete. The copy can optionally be associated to a stream by passing a non-zero `stream` argument. If `kind` is *cudaMemcpyHostToDevice* or *cudaMemcpyDeviceToHost* and `stream` is non-zero, the copy may overlap with operations in other streams.

The device version of this function only handles device to device copies and cannot be given local or shared pointers.

See also:

cudaMemcpy, *cudaMemcpy2D*, *cudaMemcpy2DToArray*, *cudaMemcpy2DFromArray*, *cudaMemcpy2DArrayToArray*, *cudaMemcpyToSymbol*, *cudaMemcpyFromSymbol*, *cudaMemcpyAsync*, *cudaMemcpy2DToArrayAsync*, *cudaMemcpy2DFromArrayAsync*, *cudaMemcpyToSymbolAsync*, *cudaMemcpyFromSymbolAsync*, ::cuMemcpy2DAsync

---

Note: Note that this function may also return error codes from previous, asynchronous launches.

Note: This function exhibits asynchronous behavior for most use cases.

Note: This function uses standard default stream semantics.

Note: Note that this function may also return *cudaErrorInitializationError*, *cudaErrorInsufficientDriver* or *cudaErrorNoDevice* if this call tries to initialize internal CUDA RT state.

Note: Note that as specified by *cudaStreamAddCallback* no CUDA function may be called from callback. *cudaErrorNotPermitted* may, but is not guaranteed to, be returned as a diagnostic in such case.

Note: Memory regions requested must be either entirely registered with CUDA, or in the case of host pageable transfers, not registered at all. Memory regions spanning over allocations that are both registered and not registered with CUDA are not supported and will return CUDA_ERROR_INVALID_VALUE.

Parameters

- **`dst`** – - Destination memory address
- **`dpitch`** – - Pitch of destination memory

- **src** – - Source memory address
- **spitch** – - Pitch of source memory
- **width** – - Width of matrix transfer (columns in bytes)
- **height** – - Height of matrix transfer (rows)
- **kind** – - Type of transfer
- **stream** – - Stream identifier

Returns

cudaSuccess, *cudaErrorInvalidValue*, *cudaErrorInvalidPitchValue*, *cudaErrorInvalidMemcpyDirection*

__host__ *cudaError_t* **cudaMemcpy2DFromArray**(void *dst, size_t dpitch, *cudaArray_const_t* src, size_t wOffset, size_t hOffset, size_t width, size_t height, enum *cudaMemcpyKind* kind)

Copies data between host and device.

Copies a matrix (`height` rows of `width` bytes each) from the CUDA array `src` starting at `hOffset` rows and `wOffset` bytes from the upper left corner to the memory area pointed to by `dst`, where `kind` specifies the direction of the copy, and must be one of *cudaMemcpyHostToHost*, *cudaMemcpyHostToDevice*, *cudaMemcpyDeviceToHost*, *cudaMemcpyDeviceToDevice*, or *cudaMemcpyDefault*. Passing *cudaMemcpyDefault* is recommended, in which case the type of transfer is inferred from the pointer values. However, *cudaMemcpyDefault* is only allowed on systems that support unified virtual addressing. `dpitch` is the width in memory in bytes of the 2D array pointed to by `dst`, including any padding added to the end of each row. `wOffset` + `width` must not exceed the width of the CUDA array `src`. `width` must not exceed `dpitch`. *cudaMemcpy2DFromArray()* returns an error if `dpitch` exceeds the maximum allowed.

See also:

cudaMemcpy, *cudaMemcpy2D*, *cudaMemcpy2DToArray*, *cudaMemcpy2DArrayToArray*, *cudaMemcpyToSymbol*, *cudaMemcpyFromSymbol*, *cudaMemcpyAsync*, *cudaMemcpy2DAsync*, *cudaMemcpy2DToArrayAsync*, *cudaMemcpy2DFromArrayAsync*, *cudaMemcpyToSymbolAsync*, *cudaMemcpyFromSymbolAsync*, ::cuMemcpy2D, ::cuMemcpy2DUnaligned

---

Note: Note that this function may also return error codes from previous, asynchronous launches.

Note: This function exhibits synchronous behavior for most use cases.

Note: Note that this function may also return *cudaErrorInitializationError*, *cudaErrorInsufficientDriver* or *cudaErrorNoDevice* if this call tries to initialize internal CUDA RT state.

Note: Note that as specified by *cudaStreamAddCallback* no CUDA function may be called from callback. *cudaErrorNotPermitted* may, but is not guaranteed to, be returned as a diagnostic in such case.

Note: Memory regions requested must be either entirely registered with CUDA, or in the case of host pageable transfers, not registered at all. Memory regions spanning over allocations that are both registered and not registered with CUDA are not supported and will return CUDA_ERROR_INVALID_VALUE.

Parameters

- **dst** – - Destination memory address
- **dpitch** – - Pitch of destination memory
- **src** – - Source memory address
- **wOffset** – - Source starting X offset (columns in bytes)
- **hOffset** – - Source starting Y offset (rows)
- **width** – - Width of matrix transfer (columns in bytes)
- **height** – - Height of matrix transfer (rows)
- **kind** – - Type of transfer

Returns

cudaSuccess, *cudaErrorInvalidValue*, *cudaErrorInvalidPitchValue*, *cudaErrorInvalidMemcpyDirection*

__host__ *cudaError_t* **cudaMemcpy2DFromArrayAsync**(void *dst, size_t dpitch, *cudaArray_const_t* src, size_t wOffset, size_t hOffset, size_t width, size_t height, enum *cudaMemcpyKind* kind, *cudaStream_t* stream = 0)

Copies data between host and device.

Copies a matrix (`height` rows of `width` bytes each) from the CUDA array `src` starting at `hOffset` rows and `wOffset` bytes from the upper left corner to the memory area pointed to by `dst`, where `kind` specifies the direction of the copy, and must be one of *cudaMemcpyHostToHost*, *cudaMemcpyHostToDevice*, *cudaMemcpyDeviceToHost*, *cudaMemcpyDeviceToDevice*, or *cudaMemcpyDefault*. Passing *cudaMemcpyDefault* is recommended, in which case the type of transfer is inferred from the pointer values. However, *cudaMemcpyDefault* is only allowed on systems that support unified virtual addressing. `dpitch` is the width in memory in bytes of the 2D array pointed to by `dst`, including any padding added to the end of each row. `wOffset` + `width` must not exceed the width of the CUDA array `src`. `width` must not exceed `dpitch`. *cudaMemcpy2DFromArrayAsync()* returns an error if `dpitch` exceeds the maximum allowed.

cudaMemcpy2DFromArrayAsync() is asynchronous with respect to the host, so the call may return before the copy is complete. The copy can optionally be associated to a stream by passing a non-zero `stream` argument. If `kind` is *cudaMemcpyHostToDevice* or *cudaMemcpyDeviceToHost* and `stream` is non-zero, the copy may overlap with operations in other streams.

cudaMemcpyToSymbolAsync, *cudaMemcpyFromSymbolAsync*, ::cuMemcpy2DAsync

See also:

cudaMemcpy, *cudaMemcpy2D*, *cudaMemcpy2DToArray*, *cudaMemcpy2DFromArray*, *cudaMemcpy2DArrayToArray*, *cudaMemcpyToSymbol*, *cudaMemcpyFromSymbol*, *cudaMemcpyAsync*, *cudaMemcpy2DAsync*, *cudaMemcpy2DToArrayAsync*,

Note: Note that this function may also return error codes from previous, asynchronous launches.

Note: This function exhibits asynchronous behavior for most use cases.

Note: This function uses standard default stream semantics.

Note: Note that this function may also return *cudaErrorInitializationError*, *cudaErrorInsufficientDriver* or *cudaErrorNoDevice* if this call tries to initialize internal CUDA RT state.

Note: Note that as specified by *cudaStreamAddCallback* no CUDA function may be called from callback. *cudaErrorNotPermitted* may, but is not guaranteed to, be returned as a diagnostic in such case.

Note: Memory regions requested must be either entirely registered with CUDA, or in the case of host pageable transfers, not registered at all. Memory regions spanning over allocations that are both registered and not registered with CUDA are not supported and will return CUDA_ERROR_INVALID_VALUE.

Parameters

- **dst** – - Destination memory address
- **dpitch** – - Pitch of destination memory
- **src** – - Source memory address
- **wOffset** – - Source starting X offset (columns in bytes)
- **hOffset** – - Source starting Y offset (rows)
- **width** – - Width of matrix transfer (columns in bytes)
- **height** – - Height of matrix transfer (rows)
- **kind** – - Type of transfer
- **stream** – - Stream identifier

Returns

cudaSuccess, *cudaErrorInvalidValue*, *cudaErrorInvalidPitchValue*, *cudaErrorInvalidMemcpyDirection*

__host__ *cudaError_t* **cudaMemcpy2DToArray**(*cudaArray_t* dst, size_t wOffset, size_t hOffset, const void *src, size_t spitch, size_t width, size_t height, enum *cudaMemcpyKind* kind)

Copies data between host and device.

Copies a matrix (`height` rows of `width` bytes each) from the memory area pointed to by `src` to the CUDA array `dst` starting at `hOffset` rows and `wOffset` bytes from the upper left corner,

where `kind` specifies the direction of the copy, and must be one of *cudaMemcpyHostToHost*, *cudaMemcpyHostToDevice*, *cudaMemcpyDeviceToHost*, *cudaMemcpyDeviceToDevice*, or *cudaMemcpyDefault*. Passing *cudaMemcpyDefault* is recommended, in which case the type of transfer is inferred from the pointer values. However, *cudaMemcpyDefault* is only allowed on systems that support unified virtual addressing. `spitch` is the width in memory in bytes of the 2D array pointed to by `src`, including any padding added to the end of each row. `wOffset` + `width` must not exceed the width of the CUDA array `dst`. `width` must not exceed `spitch`. *cudaMemcpy2DToArray()* returns an error if `spitch` exceeds the maximum allowed.

See also:

cudaMemcpy, *cudaMemcpy2D*, *cudaMemcpy2DFromArray*, *cudaMemcpy2DArrayToArray*, *cudaMemcpyToSymbol*, *cudaMemcpyFromSymbol*, *cudaMemcpyAsync*, *cudaMemcpy2DAsync*, *cudaMemcpy2DToArrayAsync*, *cudaMemcpy2DFromArrayAsync*, *cudaMemcpyToSymbolAsync*, *cudaMemcpyFromSymbolAsync*, ::cuMemcpy2D, ::cuMemcpy2DUnaligned

---

Note: Note that this function may also return error codes from previous, asynchronous launches.

Note: This function exhibits synchronous behavior for most use cases.

Note: Note that this function may also return *cudaErrorInitializationError*, *cudaErrorInsufficientDriver* or *cudaErrorNoDevice* if this call tries to initialize internal CUDA RT state.

Note: Note that as specified by *cudaStreamAddCallback* no CUDA function may be called from callback. *cudaErrorNotPermitted* may, but is not guaranteed to, be returned as a diagnostic in such case.

Note: Memory regions requested must be either entirely registered with CUDA, or in the case of host pageable transfers, not registered at all. Memory regions spanning over allocations that are both registered and not registered with CUDA are not supported and will return CUDA_ERROR_INVALID_VALUE.

Parameters

- **`dst`** – - Destination memory address
- **`wOffset`** – - Destination starting X offset (columns in bytes)
- **`hOffset`** – - Destination starting Y offset (rows)
- **`src`** – - Source memory address
- **`spitch`** – - Pitch of source memory
- **`width`** – - Width of matrix transfer (columns in bytes)
- **`height`** – - Height of matrix transfer (rows)
- **`kind`** – - Type of transfer

Returns

cudaSuccess, *cudaErrorInvalidValue*, *cudaErrorInvalidPitchValue*, *cudaErrorInvalidMemcpyDirection*

__host__ *cudaError_t* **cudaMemcpy2DToArrayAsync**(*cudaArray_t* dst, size_t wOffset, size_t hOffset, const void *src, size_t spitch, size_t width, size_t height, enum *cudaMemcpyKind* kind, *cudaStream_t* stream = 0)

Copies data between host and device.

Copies a matrix (`height` rows of `width` bytes each) from the memory area pointed to by `src` to the CUDA array `dst` starting at `hOffset` rows and `wOffset` bytes from the upper left corner, where `kind` specifies the direction of the copy, and must be one of *cudaMemcpyHostToHost*, *cudaMemcpyHostToDevice*, *cudaMemcpyDeviceToHost*, *cudaMemcpyDeviceToDevice*, or *cudaMemcpyDefault*. Passing *cudaMemcpyDefault* is recommended, in which case the type of transfer is inferred from the pointer values. However, *cudaMemcpyDefault* is only allowed on systems that support unified virtual addressing. `spitch` is the width in memory in bytes of the 2D array pointed to by `src`, including any padding added to the end of each row. `wOffset` + `width` must not exceed the width of the CUDA array `dst`. `width` must not exceed `spitch`. *cudaMemcpy2DToArrayAsync()* returns an error if `spitch` exceeds the maximum allowed.

cudaMemcpy2DToArrayAsync() is asynchronous with respect to the host, so the call may return before the copy is complete. The copy can optionally be associated to a stream by passing a non-zero `stream` argument. If `kind` is *cudaMemcpyHostToDevice* or *cudaMemcpyDeviceToHost* and `stream` is non-zero, the copy may overlap with operations in other streams.

cudaMemcpy2DFromArrayAsync, *cudaMemcpyToSymbolAsync*, *cudaMemcpyFromSymbolAsync*, ::cuMemcpy2DAsync

See also:

cudaMemcpy, *cudaMemcpy2D*, *cudaMemcpy2DToArray*, *cudaMemcpy2DFromArray*, *cudaMemcpy2DArrayToArray*, *cudaMemcpyToSymbol*, *cudaMemcpyFromSymbol*, *cudaMemcpyAsync*, *cudaMemcpy2DAsync*,

---

Note: Note that this function may also return error codes from previous, asynchronous launches.

Note: This function exhibits asynchronous behavior for most use cases.

Note: This function uses standard default stream semantics.

Note: Note that this function may also return *cudaErrorInitializationError*, *cudaErrorInsufficientDriver* or *cudaErrorNoDevice* if this call tries to initialize internal CUDA RT state.

Note: Note that as specified by *cudaStreamAddCallback* no CUDA function may be called from callback. *cudaErrorNotPermitted* may, but is not guaranteed to, be returned as a diagnostic in such case.

Note: Memory regions requested must be either entirely registered with CUDA, or in the case of host pageable transfers, not registered at all. Memory regions spanning over allocations that are both registered and not registered with CUDA are not supported and will return CUDA_ERROR_INVALID_VALUE.

Parameters

- **dst** – - Destination memory address
- **wOffset** – - Destination starting X offset (columns in bytes)
- **hOffset** – - Destination starting Y offset (rows)
- **src** – - Source memory address
- **spitch** – - Pitch of source memory
- **width** – - Width of matrix transfer (columns in bytes)
- **height** – - Height of matrix transfer (rows)
- **kind** – - Type of transfer
- **stream** – - Stream identifier

Returns

cudaSuccess, *cudaErrorInvalidValue*, *cudaErrorInvalidPitchValue*, *cudaErrorInvalidMemcpyDirection*

__host__ *cudaError_t* **cudaMemcpy3D**(const struct *cudaMemcpy3DParms* *p)

Copies data between 3D objects.

```
struct cudaExtent {
  size_t width;
  size_t height;
  size_t depth;
};
struct cudaExtent make_cudaExtent(size_t w, size_t h, size_t d);

struct cudaPos {
  size_t x;
  size_t y;
  size_t z;
};
struct cudaPos make_cudaPos(size_t x, size_t y, size_t z);

struct cudaMemcpy3DParms {
  cudaArray_t           srcArray;
  struct cudaPos        srcPos;
  struct cudaPitchedPtr srcPtr;
  cudaArray_t           dstArray;
  struct cudaPos        dstPos;
  struct cudaPitchedPtr dstPtr;
  struct cudaExtent     extent;
  enum cudaMemcpyKind   kind;
};
```

cudaMemcpy3D() copies data betwen two 3D objects. The source and destination objects may be in either host memory, device memory, or a CUDA array. The source, destination, extent, and kind of copy performed is specified by the *cudaMemcpy3DParms* struct which should be initialized to zero before use:

```
cudaMemcpy3DParms myParms = {0};
```

The struct passed to *cudaMemcpy3D()* must specify one of `srcArray` or `srcPtr` and one of `dstArray` or `dstPtr`. Passing more than one non-zero source or destination will cause *cudaMemcpy3D()* to return an error.

The `srcPos` and `dstPos` fields are optional offsets into the source and destination objects and are defined in units of each object's elements. The element for a host or device pointer is assumed to be **unsigned char**.

The `extent` field defines the dimensions of the transferred area in elements. If a CUDA array is participating in the copy, the extent is defined in terms of that array's elements. If no CUDA array is participating in the copy then the extents are defined in elements of **unsigned char**.

The `kind` field defines the direction of the copy. It must be one of *cudaMemcpyHostToHost*, *cudaMemcpyHostToDevice*, *cudaMemcpyDeviceToHost*, *cudaMemcpyDeviceToDevice*, or *cudaMemcpyDefault*. Passing *cudaMemcpyDefault* is recommended, in which case the type of transfer is inferred from the pointer values. However, *cudaMemcpyDefault* is only allowed on systems that support unified virtual addressing. For *cudaMemcpyHostToHost* or *cudaMemcpyHostToDevice* or *cudaMemcpyDeviceToHost* passed as kind and cudaArray type passed as source or destination, if the kind implies cudaArray type to be present on the host, *cudaMemcpy3D()* will disregard that implication and silently correct the kind based on the fact that cudaArray type can only be present on the device.

If the source and destination are both arrays, *cudaMemcpy3D()* will return an error if they do not have the same element size.

The source and destination object may not overlap. If overlapping source and destination objects are specified, undefined behavior will result.

The source object must entirely contain the region defined by `srcPos` and `extent`. The destination object must entirely contain the region defined by `dstPos` and `extent`.

cudaMemcpy3D() returns an error if the pitch of `srcPtr` or `dstPtr` exceeds the maximum allowed. The pitch of a *cudaPitchedPtr* allocated with *cudaMalloc3D()* will always be valid.

See also:

cudaMalloc3D, *cudaMalloc3DArray*, *cudaMemset3D*, *cudaMemcpy3DAsync*, *cudaMemcpy*, *cudaMemcpy2D*, *cudaMemcpy2DToArray*, *cudaMemcpy2DFromArray*, *cudaMemcpy2DArrayToArray*, *cudaMemcpyToSymbol*, *cudaMemcpyFromSymbol*, *cudaMemcpyAsync*, *cudaMemcpy2DAsync*, *cudaMemcpy2DToArrayAsync*, *cudaMemcpy2DFromArrayAsync*, *cudaMemcpyToSymbolAsync*, *cudaMemcpyFromSymbolAsync*, *make_cudaExtent*, *make_cudaPos*, ::cuMemcpy3D

Note: Note that this function may also return error codes from previous, asynchronous launches.

Note: This function exhibits synchronous behavior for most use cases.

Note: Note that this function may also return *cudaErrorInitializationError*, *cudaErrorInsufficientDriver* or *cudaErrorNoDevice* if this call tries to initialize internal CUDA RT state.

Note: Note that as specified by *cudaStreamAddCallback* no CUDA function may be called from callback. *cudaErrorNotPermitted* may, but is not guaranteed to, be returned as a diagnostic in such case.

Parameters

p – - 3D memory copy parameters

Returns

cudaSuccess, *cudaErrorInvalidValue*, *cudaErrorInvalidPitchValue*, *cudaErrorInvalidMemcpyDirection*

__host__ *cudaError_t* **cudaMemcpy3DAsync**(const struct *cudaMemcpy3DParms* *p, *cudaStream_t* stream = 0)

Copies data between 3D objects.

```
struct cudaExtent {
  size_t width;
  size_t height;
  size_t depth;
};
struct cudaExtent make_cudaExtent(size_t w, size_t h, size_t d);

struct cudaPos {
  size_t x;
  size_t y;
  size_t z;
};
struct cudaPos make_cudaPos(size_t x, size_t y, size_t z);

struct cudaMemcpy3DParms {
  cudaArray_t           srcArray;
  struct cudaPos        srcPos;
  struct cudaPitchedPtr srcPtr;
  cudaArray_t           dstArray;
  struct cudaPos        dstPos;
  struct cudaPitchedPtr dstPtr;
  struct cudaExtent     extent;
  enum cudaMemcpyKind   kind;
};
```

cudaMemcpy3DAsync() copies data betwen two 3D objects. The source and destination objects may be in either host memory, device memory, or a CUDA array. The source, destination, extent, and kind of copy performed is specified by the *cudaMemcpy3DParms* struct which should be initialized to zero before use:

```
cudaMemcpy3DParms myParms = {0};
```

The struct passed to *cudaMemcpy3DAsync()* must specify one of `srcArray` or `srcPtr` and one

of `dstArray` or `dstPtr`. Passing more than one non-zero source or destination will cause *cudaMemcpy3DAsync()* to return an error.

The `srcPos` and `dstPos` fields are optional offsets into the source and destination objects and are defined in units of each object's elements. The element for a host or device pointer is assumed to be **unsigned char**. For CUDA arrays, positions must be in the range [0, 2048) for any dimension.

The `extent` field defines the dimensions of the transferred area in elements. If a CUDA array is participating in the copy, the extent is defined in terms of that array's elements. If no CUDA array is participating in the copy then the extents are defined in elements of **unsigned char**.

The `kind` field defines the direction of the copy. It must be one of *cudaMemcpyHostToHost*, *cudaMemcpyHostToDevice*, *cudaMemcpyDeviceToHost*, *cudaMemcpyDeviceToDevice*, or *cudaMemcpyDefault*. Passing *cudaMemcpyDefault* is recommended, in which case the type of transfer is inferred from the pointer values. However, *cudaMemcpyDefault* is only allowed on systems that support unified virtual addressing. For *cudaMemcpyHostToHost* or *cudaMemcpyHostToDevice* or *cudaMemcpyDeviceToHost* passed as kind and cudaArray type passed as source or destination, if the kind implies cudaArray type to be present on the host, *cudaMemcpy3DAsync()* will disregard that implication and silently correct the kind based on the fact that cudaArray type can only be present on the device.

If the source and destination are both arrays, *cudaMemcpy3DAsync()* will return an error if they do not have the same element size.

The source and destination object may not overlap. If overlapping source and destination objects are specified, undefined behavior will result.

The source object must lie entirely within the region defined by `srcPos` and `extent`. The destination object must lie entirely within the region defined by `dstPos` and `extent`.

cudaMemcpy3DAsync() returns an error if the pitch of `srcPtr` or `dstPtr` exceeds the maximum allowed. The pitch of a *cudaPitchedPtr* allocated with *cudaMalloc3D()* will always be valid.

cudaMemcpy3DAsync() is asynchronous with respect to the host, so the call may return before the copy is complete. The copy can optionally be associated to a stream by passing a non-zero `stream` argument. If `kind` is *cudaMemcpyHostToDevice* or *cudaMemcpyDeviceToHost* and `stream` is non-zero, the copy may overlap with operations in other streams.

The device version of this function only handles device to device copies and cannot be given local or shared pointers.

See also:

cudaMalloc3D, *cudaMalloc3DArray*, *cudaMemset3D*, *cudaMemcpy3D*, *cudaMemcpy*, *cudaMemcpy2D*, *cudaMemcpy2DToArray*, :::*cudaMemcpy2DFromArray*, *cudaMemcpy2DArrayToArray*, *cudaMemcpyToSymbol*, *cudaMemcpyFromSymbol*, *cudaMemcpyAsync*, *cudaMemcpy2DAsync*, *cudaMemcpy2DToArrayAsync*, *cudaMemcpy2DFromArrayAsync*, *cudaMemcpyToSymbolAsync*, *cudaMemcpyFromSymbolAsync*, *make_cudaExtent*, *make_cudaPos*, ::cuMemcpy3DAsync

---

Note: Note that this function may also return error codes from previous, asynchronous launches.

Note: This function exhibits asynchronous behavior for most use cases.

Note: This function uses standard default stream semantics.

Note: Note that this function may also return *cudaErrorInitializationError*, *cudaErrorInsufficientDriver* or *cudaErrorNoDevice* if this call tries to initialize internal CUDA RT state.

Note: Note that as specified by *cudaStreamAddCallback* no CUDA function may be called from callback. *cudaErrorNotPermitted* may, but is not guaranteed to, be returned as a diagnostic in such case.

Parameters

- **p** – - 3D memory copy parameters
- **stream** – - Stream identifier

Returns

cudaSuccess, *cudaErrorInvalidValue*, *cudaErrorInvalidPitchValue*, *cudaErrorInvalidMemcpyDirection*

__host__ *cudaError_t* **cudaMemcpy3DBatchAsync**(size_t numOps, struct *cudaMemcpy3DBatchOp* *opList, unsigned long long flags, *cudaStream_t* stream)

Performs a batch of 3D memory copies asynchronously.

Performs a batch of memory copies. The batch as a whole executes in stream order but copies within a batch are not guaranteed to execute in any specific order. Note that this means specifying any dependent copies within a batch will result in undefined behavior.

Performs memory copies as specified in the `opList` array. The length of this array is specified in `numOps`. Each entry in this array describes a copy operation. This includes among other things, the source and destination operands for the copy as specified in *cudaMemcpy3DBatchOp::src* and *cudaMemcpy3DBatchOp::dst* respectively. The source and destination operands of a copy can either be a pointer or a CUDA array. The width, height and depth of a copy is specified in *cudaMemcpy3DBatchOp::extent*. The width, height and depth of a copy are specified in elements and must not be zero. For pointer-to-pointer copies, the element size is considered to be 1. For pointer to CUDA array or vice versa copies, the element size is determined by the CUDA array. For CUDA array to CUDA array copies, the element size of the two CUDA arrays must match.

For a given operand, if cudaMemcpy3DOperand::type is specified as *cudaMemcpyOperandTypePointer*, then ::cudaMemcpy3DOperand::op::ptr will be used. The ::cudaMemcpy3DOperand::op::ptr::ptr field must contain the pointer where the copy should begin. The ::cudaMemcpy3DOperand::op::ptr::rowLength field specifies the length of each row in elements and must either be zero or be greater than or equal to the width of the copy specified in ::cudaMemcpy3DBatchOp::extent::width. The ::cudaMemcpy3DOperand::op::ptr::layerHeight field specifies the height of each layer and must either be zero or be greater than or equal to the height of the copy specified in ::cudaMemcpy3DBatchOp::extent::height. When either of these values is zero, that aspect of the operand is considered to be tightly packed according to the copy extent. For managed memory pointers on devices where *cudaDevAttrConcurrentManagedAccess* is true or system-allocated pageable memory on devices where *cudaDevAttrPageableMemoryAccess* is true, the ::cudaMemcpy3DOperand::op::ptr::locHint field can be used to hint the location of the operand.

If an operand's type is specified as *cudaMemcpyOperandTypeArray*, then ::cudaMemcpy3DOperand::op::array will be used. The ::cudaMemcpy3DOperand::op::array::array field specifies the CUDA array and ::cudaMemcpy3DOperand::op::array::offset specifies the 3D offset into that array where the copy begins.

The *cudaMemcpyAttributes::srcAccessOrder* indicates the source access ordering to be observed for copies associated with the attribute. If the source access order is set to *cudaMemcpySrcAccessOrderStream*, then the source will be accessed in stream order. If the source access order is set to *cudaMemcpySrcAccessOrderDuringApiCall* then it indicates that access to the source pointer can be out of stream order and all accesses must be complete before the API call returns. This flag is suited for ephemeral sources (ex., stack variables) when it's known that no prior operations in the stream can be accessing the memory and also that the lifetime of the memory is limited to the scope that the source variable was declared in. Specifying this flag allows the driver to optimize the copy and removes the need for the user to synchronize the stream after the API call. If the source access order is set to *cudaMemcpySrcAccessOrderAny* then it indicates that access to the source pointer can be out of stream order and the accesses can happen even after the API call returns. This flag is suited for host pointers allocated outside CUDA (ex., via malloc) when it's known that no prior operations in the stream can be accessing the memory. Specifying this flag allows the driver to optimize the copy on certain platforms. Each memcopy operation in `opList` must have a valid srcAccessOrder setting, otherwise this API will return *cudaErrorInvalidValue*.

The *cudaMemcpyAttributes::flags* field can be used to specify certain flags for copies. Setting the *cudaMemcpyFlagPreferOverlapWithCompute* flag indicates that the associated copies should preferably overlap with any compute work. Note that this flag is a hint and can be ignored depending on the platform and other parameters of the copy.

Note: Note that this function may also return error codes from previous, asynchronous launches.

Note: This function exhibits asynchronous behavior for most use cases.

Note: Memory regions requested must be either entirely registered with CUDA, or in the case of host pageable transfers, not registered at all. Memory regions spanning over allocations that are both registered and not registered with CUDA are not supported and will return CUDA_ERROR_INVALID_VALUE.

Parameters

- **`numOps`** – - Total number of memcpy operations.
- **`opList`** – - Array of size `numOps` containing the actual memcpy operations.
- **`flags`** – - Flags for future use, must be zero now.
- **`stream`** – - The stream to enqueue the operations in. Must not be default NULL stream.

Returns

cudaSuccess cudaErrorInvalidValue

__host__ *cudaError_t* **`cudaMemcpy3DPeer`**(const struct *cudaMemcpy3DPeerParms* *p)

Copies memory between devices.

Perform a 3D memory copy according to the parameters specified in p. See the definition of the *cudaMemcpy3DPeerParms* structure for documentation of its parameters.

Note that this function is synchronous with respect to the host only if the source or destination of the transfer is host memory. Note also that this copy is serialized with respect to all pending and future asynchronous work in to the current device, the copy's source device, and the copy's destination device (use *cudaMemcpy3DPeerAsync* to avoid this synchronization).

See also:

cudaMemcpy, *cudaMemcpyPeer*, *cudaMemcpyAsync*, *cudaMemcpyPeerAsync*, *cudaMemcpy3DPeerAsync*, ::cuMemcpy3DPeer

Note: Note that this function may also return error codes from previous, asynchronous launches.

Note: This function exhibits synchronous behavior for most use cases.

Note: Note that this function may also return *cudaErrorInitializationError*, *cudaErrorInsufficientDriver* or *cudaErrorNoDevice* if this call tries to initialize internal CUDA RT state.

Note: Note that as specified by *cudaStreamAddCallback* no CUDA function may be called from callback. *cudaErrorNotPermitted* may, but is not guaranteed to, be returned as a diagnostic in such case.

Parameters

p – - Parameters for the memory copy

Returns

cudaSuccess, *cudaErrorInvalidValue*, *cudaErrorInvalidDevice*, *cudaErrorInvalidPitchValue*

__host__ *cudaError_t* **cudaMemcpy3DPeerAsync**(const struct *cudaMemcpy3DPeerParms* *p, *cudaStream_t* stream = 0)

Copies memory between devices asynchronously.

Perform a 3D memory copy according to the parameters specified in p. See the definition of the *cudaMemcpy3DPeerParms* structure for documentation of its parameters.

See also:

cudaMemcpy, *cudaMemcpyPeer*, *cudaMemcpyAsync*, *cudaMemcpyPeerAsync*, *cudaMemcpy3DPeerAsync*, ::cuMemcpy3DPeerAsync

Note: Note that this function may also return error codes from previous, asynchronous launches.

Note: This function exhibits asynchronous behavior for most use cases.

Note: This function uses standard default stream semantics.

Note: Note that this function may also return *cudaErrorInitializationError*, *cudaErrorInsufficientDriver* or *cudaErrorNoDevice* if this call tries to initialize internal CUDA RT state.

Note: Note that as specified by *cudaStreamAddCallback* no CUDA function may be called from callback. *cudaErrorNotPermitted* may, but is not guaranteed to, be returned as a diagnostic in such case.

Parameters

- **p** – - Parameters for the memory copy
- **stream** – - Stream identifier

Returns

cudaSuccess, *cudaErrorInvalidValue*, *cudaErrorInvalidDevice*, *cudaErrorInvalidPitchValue*

__host__ *cudaError_t* **cudaMemcpy3DWithAttributesAsync**(struct *cudaMemcpy3DBatchOp* *op, unsigned long long flags, *cudaStream_t* stream)

Performs 3D asynchronous memory copy with the specified attributes.

Performs the copy operation specified in `op`. `flags` specifies the flags for the copy and `stream` specifies the stream to enqueue the operation in.

For more information regarding the operation, please refer to *cudaMemcpy3DBatchOp* and it's usage desciption in::cudaMemcpy3DBatchAsync

See also:

cudaMemcpy3DBatchAsync

Note: Note that this function may also return error codes from previous, asynchronous launches.

Note: This function exhibits asynchronous behavior for most use cases.

Note: Memory regions requested must be either entirely registered with CUDA, or in the case of host pageable transfers, not registered at all. Memory regions spanning over allocations that are both registered and not registered with CUDA are not supported and will return CUDA_ERROR_INVALID_VALUE.

Parameters

- **op** – - Operation to perform
- **flags** – - Flags for the copy, must be zero now.
- **stream** – - Stream to enqueue the operation in

Returns

cudaSuccess, *cudaErrorInvalidValue*

__host__ *cudaError_t* **cudaMemcpyAsync**(void *dst, const void *src, size_t count, enum *cudaMemcpyKind* kind, *cudaStream_t* stream = 0)

Copies data between host and device.

Copies `count` bytes from the memory area pointed to by `src` to the memory area pointed to by `dst`, where `kind` specifies the direction of the copy, and must be one of *cudaMemcpyHostToHost*, *cudaMemcpyHostToDevice*, *cudaMemcpyDeviceToHost*, *cudaMemcpyDeviceToDevice*, or *cudaMemcpyDefault*. Passing *cudaMemcpyDefault* is recommended, in which case the type of transfer is inferred from the pointer values. However, *cudaMemcpyDefault* is only allowed on systems that support unified virtual addressing.

The memory areas may not overlap. Calling *cudaMemcpyAsync()* with `dst` and `src` pointers that do not match the direction of the copy results in an undefined behavior.

cudaMemcpyAsync() is asynchronous with respect to the host, so the call may return before the copy is complete. The copy can optionally be associated to a stream by passing a non-zero `stream` argument. If `kind` is *cudaMemcpyHostToDevice* or *cudaMemcpyDeviceToHost* and the `stream` is non-zero, the copy may overlap with operations in other streams.

The device version of this function only handles device to device copies and cannot be given local or shared pointers.

See also:

cudaMemcpy, *cudaMemcpy2D*, *cudaMemcpy2DToArray*, *cudaMemcpy2DFromArray*, *cudaMemcpy2DArrayToArray*, *cudaMemcpyToSymbol*, *cudaMemcpyFromSymbol*, *cudaMemcpy2DAsync*, *cudaMemcpy2DToArrayAsync*, *cudaMemcpy2DFromArrayAsync*, *cudaMemcpyToSymbolAsync*, *cudaMemcpyFromSymbolAsync*, ::cuMemcpyAsync, ::cuMemcpyDtoHAsync, ::cuMemcpyHtoDAsync, ::cuMemcpyDtoDAsync

---

Note: Note that this function may also return error codes from previous, asynchronous launches.

Note: This function exhibits asynchronous behavior for most use cases.

Note: This function uses standard default stream semantics.

Note: Note that this function may also return *cudaErrorInitializationError*, *cudaErrorInsufficientDriver* or *cudaErrorNoDevice* if this call tries to initialize internal CUDA RT state.

Note: Note that as specified by *cudaStreamAddCallback* no CUDA function may be called from callback. *cudaErrorNotPermitted* may, but is not guaranteed to, be returned as a diagnostic in such case.

Note: Memory regions requested must be either entirely registered with CUDA, or in the case of host pageable transfers, not registered at all. Memory regions spanning over allocations that are both registered and not registered with CUDA are not supported and will return CUDA_ERROR_INVALID_VALUE.

Parameters

- **dst** – - Destination memory address
- **src** – - Source memory address
- **count** – - Size in bytes to copy
- **kind** – - Type of transfer
- **stream** – - Stream identifier

Returns

cudaSuccess, *cudaErrorInvalidValue*, *cudaErrorInvalidMemcpyDirection*

__host__ *cudaError_t* **cudaMemcpyBatchAsync**(void *const *dsts, const void *const *srcs, const size_t *sizes, size_t count, struct *cudaMemcpyAttributes* *attrs, size_t *attrsIdxs, size_t numAttrs, *cudaStream_t* stream)

Performs a batch of memory copies asynchronously.

Performs a batch of memory copies. The batch as a whole executes in stream order but copies within a batch are not guaranteed to execute in any specific order. This API only supports pointer-to-pointer copies. For copies involving CUDA arrays, please see *cudaMemcpy3DBatchAsync*.

Performs memory copies from source buffers specified in `srcs` to destination buffers specified in `dsts`. The size of each copy is specified in `sizes`. All three arrays must be of the same length as specified by `count`. Since there are no ordering guarantees for copies within a batch, specifying any dependent copies within a batch will result in undefined behavior.

Every copy in the batch has to be associated with a set of attributes specified in the `attrs` array. Each entry in this array can apply to more than one copy. This can be done by specifying in the `attrsIdxs` array, the index of the first copy that the corresponding entry in the `attrs` array applies to. Both `attrs` and `attrsIdxs` must be of the same length as specified by `numAttrs`. For example, if a batch has 10 copies listed in dst/src/sizes, the first 6 of which have one set of attributes and the remaining 4 another, then `numAttrs` will be 2, `attrsIdxs` will be {0, 6} and `attrs` will contains the two sets of attributes. Note that the first entry in `attrsIdxs` must always be 0. Also, each entry must be greater than the previous entry and the last entry should be less than `count`. Furthermore, `numAttrs` must be lesser than or equal to `count`.

The *cudaMemcpyAttributes::srcAccessOrder* indicates the source access ordering to be observed for copies associated with the attribute. If the source access order is set to *cudaMemcpySrcAccessOrderStream*, then the source will be accessed in stream order. If the source access order is set to *cudaMemcpySrcAccessOrderDuringApiCall* then it indicates that access to the source pointer can be out of stream order and all accesses must be complete before the API call returns.

This flag is suited for ephemeral sources (ex., stack variables) when it's known that no prior operations in the stream can be accessing the memory and also that the lifetime of the memory is limited to the scope that the source variable was declared in. Specifying this flag allows the driver to optimize the copy and removes the need for the user to synchronize the stream after the API call. If the source access order is set to *cudaMemcpySrcAccessOrderAny* then it indicates that access to the source pointer can be out of stream order and the accesses can happen even after the API call returns. This flag is suited for host pointers allocated outside CUDA (ex., via malloc) when it's known that no prior operations in the stream can be accessing the memory. Specifying this flag allows the driver to optimize the copy on certain platforms. Each memcpy operation in the batch must have a valid *cudaMemcpyAttributes* corresponding to it including the appropriate srcAccessOrder setting, otherwise the API will return *cudaErrorInvalidValue*.

The *cudaMemcpyAttributes::srcLocHint* and *cudaMemcpyAttributes::dstLocHint* allows applications to specify hint locations for operands of a copy when the operand doesn't have a fixed location. That is, these hints are only applicable for managed memory pointers on devices where *cudaDevAttrConcurrentManagedAccess* is true or system-allocated pageable memory on devices where *cudaDevAttrPageableMemoryAccess* is true. For other cases, these hints are ignored.

The *cudaMemcpyAttributes::flags* field can be used to specify certain flags for copies. Setting the *cudaMemcpyFlagPreferOverlapWithCompute* flag indicates that the associated copies should preferably overlap with any compute work. Note that this flag is a hint and can be ignored depending on the platform and other parameters of the copy.

Note: Note that this function may also return error codes from previous, asynchronous launches.

Note: This function exhibits asynchronous behavior for most use cases.

Note: Memory regions requested must be either entirely registered with CUDA, or in the case of host pageable transfers, not registered at all. Memory regions spanning over allocations that are both registered and not registered with CUDA are not supported and will return CUDA_ERROR_INVALID_VALUE.

Parameters

- **`dsts`** – - Array of destination pointers.
- **`srcs`** – - Array of memcpy source pointers.
- **`sizes`** – - Array of sizes for memcpy operations.
- **`count`** – - Size of `dsts`, `srcs` and `sizes` arrays
- **`attrs`** – - Array of memcpy attributes.
- **`attrsIdxs`** – - Array of indices to specify which copies each entry in the `attrs` array applies to. The attributes specified in attrs[k] will be applied to copies starting from attrsIdxs[k] through attrsIdxs[k+1] - 1. Also attrs[numAttrs-1] will apply to copies starting from attrsIdxs[numAttrs-1] through count - 1.
- **`numAttrs`** – - Size of `attrs` and `attrsIdxs` arrays.
- **`stream`** – - The stream to enqueue the operations in. Must not be legacy NULL stream.

Returns

cudaSuccess *cudaErrorInvalidValue*

__host__ *cudaError_t* **cudaMemcpyFromSymbol**(void *dst, const void *symbol, size_t count, size_t offset = 0, enum *cudaMemcpyKind* kind = *cudaMemcpyDeviceToHost*)

Copies data from the given symbol on the device.

Copies `count` bytes from the memory area pointed to by `offset` bytes from the start of symbol `symbol` to the memory area pointed to by `dst`. The memory areas may not overlap. `symbol` is a variable that resides in global or constant memory space. `kind` can be either *cudaMemcpyDeviceToHost*, *cudaMemcpyDeviceToDevice*, or *cudaMemcpyDefault*. Passing *cudaMemcpyDefault* is recommended, in which case the type of transfer is inferred from the pointer values. However, *cudaMemcpyDefault* is only allowed on systems that support unified virtual addressing.

See also:

cudaMemcpy, *cudaMemcpy2D*, *cudaMemcpy2DToArray*, *cudaMemcpy2DFromArray*, *cudaMemcpy2DArrayToArray*, *cudaMemcpyToSymbol*, *cudaMemcpyAsync*, *cudaMemcpy2DAsync*, *cudaMemcpy2DToArrayAsync*, *cudaMemcpy2DFromArrayAsync*, *cudaMemcpyToSymbolAsync*, *cudaMemcpyFromSymbolAsync*, ::cuMemcpy, ::cuMemcpyDtoH, ::cuMemcpyDtoD

Note: Note that this function may also return error codes from previous, asynchronous launches.

Note: This function exhibits synchronous behavior for most use cases.

Note: Use of a string naming a variable as the `symbol` parameter was deprecated in CUDA 4.1 and removed in CUDA 5.0.

Note: Note that this function may also return *cudaErrorInitializationError*, *cudaErrorInsufficientDriver* or *cudaErrorNoDevice* if this call tries to initialize internal CUDA RT state.

Note: Note that as specified by *cudaStreamAddCallback* no CUDA function may be called from callback. *cudaErrorNotPermitted* may, but is not guaranteed to, be returned as a diagnostic in such case.

Parameters

- **dst** – - Destination memory address
- **symbol** – - Device symbol address
- **count** – - Size in bytes to copy
- **offset** – - Offset from start of symbol in bytes
- **kind** – - Type of transfer

Returns

cudaSuccess, *cudaErrorInvalidValue*, *cudaErrorInvalidSymbol*, *cudaErrorInvalidMemcpyDirection*, *cudaErrorNoKernelImageForDevice*

__host__ *cudaError_t* **cudaMemcpyFromSymbolAsync**(void *dst, const void *symbol, size_t count, size_t offset, enum *cudaMemcpyKind* kind, *cudaStream_t* stream = 0)

Copies data from the given symbol on the device.

Copies `count` bytes from the memory area pointed to by `offset` bytes from the start of symbol `symbol` to the memory area pointed to by `dst`. The memory areas may not overlap. `symbol` is a variable that resides in global or constant memory space. `kind` can be either *cudaMemcpyDeviceToHost*, *cudaMemcpyDeviceToDevice*, or *cudaMemcpyDefault*. Passing *cudaMemcpyDefault* is recommended, in which case the type of transfer is inferred from the pointer values. However, *cudaMemcpyDefault* is only allowed on systems that support unified virtual addressing.

cudaMemcpyFromSymbolAsync() is asynchronous with respect to the host, so the call may return before the copy is complete. The copy can optionally be associated to a stream by passing a non-zero `stream` argument. If `kind` is *cudaMemcpyDeviceToHost* and `stream` is non-zero, the copy may overlap with operations in other streams.

See also:

cudaMemcpy, *cudaMemcpy2D*, *cudaMemcpy2DToArray*, *cudaMemcpy2DFromArray*, *cudaMemcpy2DArrayToArray*, *cudaMemcpyToSymbol*, *cudaMemcpyFromSymbol*, *cudaMemcpyAsync*, *cudaMemcpy2DAsync*, *cudaMemcpy2DToArrayAsync*, *cudaMemcpy2DFromArrayAsync*, *cudaMemcpyToSymbolAsync*, ::cuMemcpyAsync, ::cuMemcpyDtoHAsync, ::cuMemcpyDtoDAsync

---

Note: Note that this function may also return error codes from previous, asynchronous launches.

Note: This function exhibits asynchronous behavior for most use cases.

Note: This function uses standard default stream semantics.

Note: Use of a string naming a variable as the `symbol` parameter was deprecated in CUDA 4.1 and removed in CUDA 5.0.

Note: Note that this function may also return *cudaErrorInitializationError*, *cudaErrorInsufficientDriver* or *cudaErrorNoDevice* if this call tries to initialize internal CUDA RT state.

Note: Note that as specified by *cudaStreamAddCallback* no CUDA function may be called from callback. *cudaErrorNotPermitted* may, but is not guaranteed to, be returned as a diagnostic in such case.

Parameters

- **dst** – - Destination memory address
- **symbol** – - Device symbol address
- **count** – - Size in bytes to copy
- **offset** – - Offset from start of symbol in bytes
- **kind** – - Type of transfer
- **stream** – - Stream identifier

Returns

cudaSuccess, *cudaErrorInvalidValue*, *cudaErrorInvalidSymbol*, *cudaErrorInvalidMemcpyDirection*, *cudaErrorNoKernelImageForDevice*

__host__ *cudaError_t* **cudaMemcpyPeer**(void *dst, int dstDevice, const void *src, int srcDevice, size_t count)

Copies memory between two devices.

Copies memory from one device to memory on another device. `dst` is the base device pointer of the destination memory and `dstDevice` is the destination device. `src` is the base device pointer of the source memory and `srcDevice` is the source device. `count` specifies the number of bytes to copy.

Note that this function is asynchronous with respect to the host, but serialized with respect all pending and future asynchronous work in to the current device, `srcDevice`, and `dstDevice` (use *cudaMemcpyPeerAsync* to avoid this synchronization).

See also:

cudaMemcpy, *cudaMemcpyAsync*, *cudaMemcpyPeerAsync*, *cudaMemcpy3DPeerAsync*, ::cuMemcpyPeer

---

Note: Note that this function may also return error codes from previous, asynchronous launches.

Note: This function exhibits synchronous behavior for most use cases.

Note: Note that this function may also return *cudaErrorInitializationError*, *cudaErrorInsufficientDriver* or *cudaErrorNoDevice* if this call tries to initialize internal CUDA RT state.

Note: Note that as specified by *cudaStreamAddCallback* no CUDA function may be called from callback. *cudaErrorNotPermitted* may, but is not guaranteed to, be returned as a diagnostic in such case.

Parameters

- **dst** – - Destination device pointer

- **dstDevice** – - Destination device
- **src** – - Source device pointer
- **srcDevice** – - Source device
- **count** – - Size of memory copy in bytes

Returns

cudaSuccess, *cudaErrorInvalidValue*, *cudaErrorInvalidDevice*

__host__ *cudaError_t* **cudaMemcpyPeerAsync**(void *dst, int dstDevice, const void *src, int srcDevice, size_t count, *cudaStream_t* stream = 0)

Copies memory between two devices asynchronously.

Copies memory from one device to memory on another device. `dst` is the base device pointer of the destination memory and `dstDevice` is the destination device. `src` is the base device pointer of the source memory and `srcDevice` is the source device. `count` specifies the number of bytes to copy.

Note that this function is asynchronous with respect to the host and all work on other devices.

See also:

cudaMemcpy, *cudaMemcpyPeer*, *cudaMemcpyAsync*, *cudaMemcpy3DPeerAsync*, ::cuMemcpyPeerAsync

---

Note: Note that this function may also return error codes from previous, asynchronous launches.

Note: This function exhibits asynchronous behavior for most use cases.

Note: This function uses standard default stream semantics.

Note: Note that this function may also return *cudaErrorInitializationError*, *cudaErrorInsufficientDriver* or *cudaErrorNoDevice* if this call tries to initialize internal CUDA RT state.

Note: Note that as specified by *cudaStreamAddCallback* no CUDA function may be called from callback. *cudaErrorNotPermitted* may, but is not guaranteed to, be returned as a diagnostic in such case.

Parameters

- **dst** – - Destination device pointer
- **dstDevice** – - Destination device
- **src** – - Source device pointer
- **srcDevice** – - Source device

- **count** – - Size of memory copy in bytes
- **stream** – - Stream identifier

Returns

cudaSuccess, *cudaErrorInvalidValue*, *cudaErrorInvalidDevice*

__host__ *cudaError_t* **cudaMemcpyToSymbol**(const void *symbol, const void *src, size_t count, size_t offset = 0, enum *cudaMemcpyKind* kind = *cudaMemcpyHostToDevice*)

Copies data to the given symbol on the device.

Copies `count` bytes from the memory area pointed to by `src` to the memory area pointed to by `offset` bytes from the start of symbol `symbol`. The memory areas may not overlap. `symbol` is a variable that resides in global or constant memory space. `kind` can be either *cudaMemcpyHostToDevice*, *cudaMemcpyDeviceToDevice*, or *cudaMemcpyDefault*. Passing *cudaMemcpyDefault* is recommended, in which case the type of transfer is inferred from the pointer values. However, *cudaMemcpyDefault* is only allowed on systems that support unified virtual addressing.

See also:

cudaMemcpy, *cudaMemcpy2D*, *cudaMemcpy2DToArray*, *cudaMemcpy2DFromArray*, *cudaMemcpy2DArrayToArray*, *cudaMemcpyFromSymbol*, *cudaMemcpyAsync*, *cudaMemcpy2DAsync*, *cudaMemcpy2DToArrayAsync*, *cudaMemcpy2DFromArrayAsync*, *cudaMemcpyToSymbolAsync*, *cudaMemcpyFromSymbolAsync*, ::cuMemcpy, ::cuMemcpyHtoD, ::cuMemcpyDtoD

Note: Note that this function may also return error codes from previous, asynchronous launches.

Note: This function exhibits synchronous behavior for most use cases.

Note: Use of a string naming a variable as the `symbol` parameter was deprecated in CUDA 4.1 and removed in CUDA 5.0.

Note: Note that this function may also return *cudaErrorInitializationError*, *cudaErrorInsufficientDriver* or *cudaErrorNoDevice* if this call tries to initialize internal CUDA RT state.

Note: Note that as specified by *cudaStreamAddCallback* no CUDA function may be called from callback. *cudaErrorNotPermitted* may, but is not guaranteed to, be returned as a diagnostic in such case.

Parameters

- **symbol** – - Device symbol address
- **src** – - Source memory address
- **count** – - Size in bytes to copy

- **offset** – - Offset from start of symbol in bytes
- **kind** – - Type of transfer

Returns

cudaSuccess, *cudaErrorInvalidValue*, *cudaErrorInvalidSymbol*, *cudaErrorInvalidMemcpyDirection*, *cudaErrorNoKernelImageForDevice*

__host__ *cudaError_t* **cudaMemcpyToSymbolAsync**(const void *symbol, const void *src, size_t count, size_t offset, enum *cudaMemcpyKind* kind, *cudaStream_t* stream = 0)

Copies data to the given symbol on the device.

Copies `count` bytes from the memory area pointed to by `src` to the memory area pointed to by `offset` bytes from the start of symbol `symbol`. The memory areas may not overlap. `symbol` is a variable that resides in global or constant memory space. `kind` can be either *cudaMemcpyHostToDevice*, *cudaMemcpyDeviceToDevice*, or *cudaMemcpyDefault*. Passing *cudaMemcpyDefault* is recommended, in which case the type of transfer is inferred from the pointer values. However, *cudaMemcpyDefault* is only allowed on systems that support unified virtual addressing.

cudaMemcpyToSymbolAsync() is asynchronous with respect to the host, so the call may return before the copy is complete. The copy can optionally be associated to a stream by passing a non-zero `stream` argument. If `kind` is *cudaMemcpyHostToDevice* and `stream` is non-zero, the copy may overlap with operations in other streams.

See also:

cudaMemcpy, *cudaMemcpy2D*, *cudaMemcpy2DToArray*, *cudaMemcpy2DFromArray*, *cudaMemcpy2DArrayToArray*, *cudaMemcpyToSymbol*, *cudaMemcpyFromSymbol*, *cudaMemcpyAsync*, *cudaMemcpy2DAsync*, *cudaMemcpy2DToArrayAsync*, *cudaMemcpy2DFromArrayAsync*, *cudaMemcpyFromSymbolAsync*, ::cuMemcpyAsync, ::cuMemcpyHtoDAsync, ::cuMemcpyDtoDAsync

---

Note: Note that this function may also return error codes from previous, asynchronous launches.

Note: This function exhibits asynchronous behavior for most use cases.

Note: This function uses standard default stream semantics.

Note: Use of a string naming a variable as the `symbol` parameter was deprecated in CUDA 4.1 and removed in CUDA 5.0.

Note: Note that this function may also return *cudaErrorInitializationError*, *cudaErrorInsufficientDriver* or *cudaErrorNoDevice* if this call tries to initialize internal CUDA RT state.

Note: Note that as specified by *cudaStreamAddCallback* no CUDA function may be called from callback. *cudaErrorNotPermitted* may, but is not guaranteed to, be returned as a diagnostic in

such case.

Parameters

- **symbol** – - Device symbol address
- **src** – - Source memory address
- **count** – - Size in bytes to copy
- **offset** – - Offset from start of symbol in bytes
- **kind** – - Type of transfer
- **stream** – - Stream identifier

Returns

cudaSuccess, *cudaErrorInvalidValue*, *cudaErrorInvalidSymbol*, *cudaErrorInvalidMemcpyDirection*, *cudaErrorNoKernelImageForDevice*

__host__ *cudaError_t* **cudaMemcpyWithAttributesAsync**(void *dst, const void *src, size_t size, struct *cudaMemcpyAttributes* *attr, *cudaStream_t* stream)

Performs asynchronous memory copy operation with the specified attributes.

Performs asynchronous memory copy operation where `dst` and `src` are the destination and source pointers respectively. `size` specifies the number of bytes to copy. `attr` specifies the attributes for the copy and `stream` specifies the stream to enqueue the operation in.

For more information regarding the attributes, please refer to *cudaMemcpyAttributes* and it's usage desciption in::cudaMemcpyBatchAsync

See also:

cudaMemcpyBatchAsync

Note: Note that this function may also return error codes from previous, asynchronous launches.

Note: This function exhibits asynchronous behavior for most use cases.

Note: Memory regions requested must be either entirely registered with CUDA, or in the case of host pageable transfers, not registered at all. Memory regions spanning over allocations that are both registered and not registered with CUDA are not supported and will return CUDA_ERROR_INVALID_VALUE.

Parameters

- **dst** – - Destination device pointer
- **src** – - Source device pointer
- **size** – - Number of bytes to copy

- **attr** – - Attributes for the copy
- **stream** – - Stream to enqueue the operation in

Returns
cudaSuccess, *cudaErrorInvalidValue*

__host__ *cudaError_t* **cudaMemset**(void *devPtr, int value, size_t count)

Initializes or sets device memory to a value.

Fills the first `count` bytes of the memory area pointed to by `devPtr` with the constant byte value `value`.

Note that this function is asynchronous with respect to the host unless `devPtr` refers to pinned host memory.

See also:

::cuMemsetD8, ::cuMemsetD16, ::cuMemsetD32

Note: Note that this function may also return error codes from previous, asynchronous launches.

Note: See also memset synchronization details.

Note: Note that this function may also return *cudaErrorInitializationError*, *cudaErrorInsufficient-Driver* or *cudaErrorNoDevice* if this call tries to initialize internal CUDA RT state.

Note: Note that as specified by *cudaStreamAddCallback* no CUDA function may be called from callback. *cudaErrorNotPermitted* may, but is not guaranteed to, be returned as a diagnostic in such case.

Parameters

- **devPtr** – - Pointer to device memory
- **value** – - Value to set for each byte of specified memory
- **count** – - Size in bytes to set

Returns
cudaSuccess, *cudaErrorInvalidValue*,

__host__ *cudaError_t* **cudaMemset2D**(void *devPtr, size_t pitch, int value, size_t width, size_t height)

Initializes or sets device memory to a value.

Sets to the specified value `value` a matrix (`height` rows of `width` bytes each) pointed to by `dstPtr`. `pitch` is the width in bytes of the 2D array pointed to by `dstPtr`, including any padding added to the end of each row. This function performs fastest when the pitch is one that has been passed back by *cudaMallocPitch()*.

Note that this function is asynchronous with respect to the host unless `devPtr` refers to pinned host memory.

See also:

cudaMemset, *cudaMemset3D*, *cudaMemsetAsync*, *cudaMemset2DAsync*, *cudaMemset3DAsync*, ::cuMemsetD2D8, ::cuMemsetD2D16, ::cuMemsetD2D32

Note: Note that this function may also return error codes from previous, asynchronous launches.

Note: See also memset synchronization details.

Note: Note that this function may also return *cudaErrorInitializationError*, *cudaErrorInsufficientDriver* or *cudaErrorNoDevice* if this call tries to initialize internal CUDA RT state.

Note: Note that as specified by *cudaStreamAddCallback* no CUDA function may be called from callback. *cudaErrorNotPermitted* may, but is not guaranteed to, be returned as a diagnostic in such case.

Parameters

- **`devPtr`** – - Pointer to 2D device memory
- **`pitch`** – - Pitch in bytes of 2D device memory(Unused if `height` is 1)
- **`value`** – - Value to set for each byte of specified memory
- **`width`** – - Width of matrix set (columns in bytes)
- **`height`** – - Height of matrix set (rows)

Returns

cudaSuccess, *cudaErrorInvalidValue*,

__host__ *cudaError_t* **`cudaMemset2DAsync`**(void *devPtr, size_t pitch, int value, size_t width, size_t height, *cudaStream_t* stream = 0)

Initializes or sets device memory to a value.

Sets to the specified value `value` a matrix (`height` rows of `width` bytes each) pointed to by `dstPtr`. `pitch` is the width in bytes of the 2D array pointed to by `dstPtr`, including any padding added to the end of each row. This function performs fastest when the pitch is one that has been passed back by *cudaMallocPitch()*.

cudaMemset2DAsync() is asynchronous with respect to the host, so the call may return before the memset is complete. The operation can optionally be associated to a stream by passing a non-zero `stream` argument. If `stream` is non-zero, the operation may overlap with operations in other streams.

The device version of this function only handles device to device copies and cannot be given local or shared pointers.

See also:

cudaMemset, *cudaMemset2D*, *cudaMemset3D*, *cudaMemsetAsync*, *cudaMemset3DAsync*, ::cuMemsetD2D8Async, ::cuMemsetD2D16Async, ::cuMemsetD2D32Async

---

Note: Note that this function may also return error codes from previous, asynchronous launches.

Note: See also memset synchronization details.

Note: This function uses standard default stream semantics.

Note: Note that this function may also return *cudaErrorInitializationError*, *cudaErrorInsufficientDriver* or *cudaErrorNoDevice* if this call tries to initialize internal CUDA RT state.

Note: Note that as specified by *cudaStreamAddCallback* no CUDA function may be called from callback. *cudaErrorNotPermitted* may, but is not guaranteed to, be returned as a diagnostic in such case.

Parameters

- **devPtr** – - Pointer to 2D device memory
- **pitch** – - Pitch in bytes of 2D device memory(Unused if `height` is 1)
- **value** – - Value to set for each byte of specified memory
- **width** – - Width of matrix set (columns in bytes)
- **height** – - Height of matrix set (rows)
- **stream** – - Stream identifier

Returns

cudaSuccess, *cudaErrorInvalidValue*,

__host__ *cudaError_t* **cudaMemset3D**(struct *cudaPitchedPtr* pitchedDevPtr, int value, struct *cudaExtent* extent)

Initializes or sets device memory to a value.

Initializes each element of a 3D array to the specified value `value`. The object to initialize is defined by `pitchedDevPtr`. The `pitch` field of `pitchedDevPtr` is the width in memory in bytes of the 3D array pointed to by `pitchedDevPtr`, including any padding added to the end of each row. The `xsize` field specifies the logical width of each row in bytes, while the `ysize` field specifies the height of each 2D slice in rows. The `pitch` field of `pitchedDevPtr` is ignored when `height` and `depth` are both equal to 1.

The extents of the initialized region are specified as a `width` in bytes, a `height` in rows, and a `depth` in slices.

Extents with `width` greater than or equal to the `xsize` of `pitchedDevPtr` may perform significantly faster than extents narrower than the `xsize`. Secondarily, extents with `height` equal to the `ysize` of `pitchedDevPtr` will perform faster than when the `height` is shorter than the `ysize`.

This function performs fastest when the `pitchedDevPtr` has been allocated by *cudaMalloc3D()*.

Note that this function is asynchronous with respect to the host unless `pitchedDevPtr` refers to pinned host memory.

See also:

cudaMemset, *cudaMemset2D*, *cudaMemsetAsync*, *cudaMemset2DAsync*, *cudaMemset3DAsync*, *cudaMalloc3D*, *make_cudaPitchedPtr*, *make_cudaExtent*

Note: Note that this function may also return error codes from previous, asynchronous launches.

Note: See also memset synchronization details.

Note: Note that this function may also return *cudaErrorInitializationError*, *cudaErrorInsufficientDriver* or *cudaErrorNoDevice* if this call tries to initialize internal CUDA RT state.

Note: Note that as specified by *cudaStreamAddCallback* no CUDA function may be called from callback. *cudaErrorNotPermitted* may, but is not guaranteed to, be returned as a diagnostic in such case.

Parameters

- **`pitchedDevPtr`** – - Pointer to pitched device memory
- **`value`** – - Value to set for each byte of specified memory
- **`extent`** – - Size parameters for where to set device memory (`width` field in bytes)

Returns

cudaSuccess, *cudaErrorInvalidValue*,

__host__ *cudaError_t* **`cudaMemset3DAsync`**(struct *cudaPitchedPtr* pitchedDevPtr, int value, struct *cudaExtent* extent, *cudaStream_t* stream = 0)

Initializes or sets device memory to a value.

Initializes each element of a 3D array to the specified value `value`. The object to initialize is defined by `pitchedDevPtr`. The `pitch` field of `pitchedDevPtr` is the width in memory in bytes of the 3D array pointed to by `pitchedDevPtr`, including any padding added to the end of each row. The `xsize` field specifies the logical width of each row in bytes, while the `ysize` field specifies the height of each 2D slice in rows. The `pitch` field of `pitchedDevPtr` is ignored when `height` and `depth` are both equal to 1.

The extents of the initialized region are specified as a `width` in bytes, a `height` in rows, and a `depth` in slices.

Extents with `width` greater than or equal to the `xsize` of `pitchedDevPtr` may perform significantly faster than extents narrower than the `xsize`. Secondarily, extents with `height` equal to the `ysize` of `pitchedDevPtr` will perform faster than when the `height` is shorter than the `ysize`.

This function performs fastest when the `pitchedDevPtr` has been allocated by *cudaMalloc3D()*.

cudaMemset3DAsync() is asynchronous with respect to the host, so the call may return before the memset is complete. The operation can optionally be associated to a stream by passing a non-zero `stream` argument. If `stream` is non-zero, the operation may overlap with operations in other streams.

The device version of this function only handles device to device copies and cannot be given local or shared pointers.

See also:

cudaMemset, *cudaMemset2D*, *cudaMemset3D*, *cudaMemsetAsync*, *cudaMemset2DAsync*, *cudaMalloc3D*, *make_cudaPitchedPtr*, *make_cudaExtent*

Note: Note that this function may also return error codes from previous, asynchronous launches.

Note: See also memset synchronization details.

Note: This function uses standard default stream semantics.

Note: Note that this function may also return *cudaErrorInitializationError*, *cudaErrorInsufficientDriver* or *cudaErrorNoDevice* if this call tries to initialize internal CUDA RT state.

Note: Note that as specified by *cudaStreamAddCallback* no CUDA function may be called from callback. *cudaErrorNotPermitted* may, but is not guaranteed to, be returned as a diagnostic in such case.

Parameters

- **`pitchedDevPtr`** – - Pointer to pitched device memory
- **`value`** – - Value to set for each byte of specified memory
- **`extent`** – - Size parameters for where to set device memory (`width` field in bytes)
- **`stream`** – - Stream identifier

Returns

cudaSuccess, *cudaErrorInvalidValue*,

__host__ *cudaError_t* **cudaMemsetAsync**(void *devPtr, int value, size_t count, *cudaStream_t* stream = 0)

Initializes or sets device memory to a value.

Fills the first `count` bytes of the memory area pointed to by `devPtr` with the constant byte value `value`.

cudaMemsetAsync() is asynchronous with respect to the host, so the call may return before the memset is complete. The operation can optionally be associated to a stream by passing a non-zero `stream` argument. If `stream` is non-zero, the operation may overlap with operations in other streams.

The device version of this function only handles device to device copies and cannot be given local or shared pointers.

See also:

cudaMemset, *cudaMemset2D*, *cudaMemset3D*, *cudaMemset2DAsync*, *cudaMemset3DAsync*, ::cuMemsetD8Async, ::cuMemsetD16Async, ::cuMemsetD32Async

---

Note: Note that this function may also return error codes from previous, asynchronous launches.

Note: See also memset synchronization details.

Note: This function uses standard default stream semantics.

Note: Note that this function may also return *cudaErrorInitializationError*, *cudaErrorInsufficientDriver* or *cudaErrorNoDevice* if this call tries to initialize internal CUDA RT state.

Note: Note that as specified by *cudaStreamAddCallback* no CUDA function may be called from callback. *cudaErrorNotPermitted* may, but is not guaranteed to, be returned as a diagnostic in such case.

Parameters

- **devPtr** – - Pointer to device memory
- **value** – - Value to set for each byte of specified memory
- **count** – - Size in bytes to set
- **stream** – - Stream identifier

Returns

cudaSuccess, *cudaErrorInvalidValue*,

__host__ *cudaError_t* **cudaMipmappedArrayGetMemoryRequirements**(struct *cudaArrayMemoryRequirements* *memoryRequirements, *cudaMipmappedArray_t* mipmap, int device)

Returns the memory requirements of a CUDA mipmapped array.

Returns the memory requirements of a CUDA mipmapped array in `memoryRequirements` If the CUDA mipmapped array is not allocated with flag *cudaArrayDeferredMapping* *cudaErrorInvalidValue* will be returned.

The returned value in *cudaArrayMemoryRequirements::size* represents the total size of the CUDA mipmapped array. The returned value in *cudaArrayMemoryRequirements::alignment* represents the alignment necessary for mapping the CUDA mipmapped array.

See also:

cudaArrayGetMemoryRequirements

Parameters

- **memoryRequirements** – **[out]** - Pointer to *cudaArrayMemoryRequirements*
- **mipmap** – **[in]** - CUDA mipmapped array to get the memory requirements of
- **device** – **[in]** - Device to get the memory requirements for

Returns
cudaSuccess *cudaErrorInvalidValue*

__host__ *cudaError_t* **cudaMipmappedArrayGetSparseProperties**(struct *cudaArraySparseProperties* *sparseProperties, *cudaMipmappedArray_t* mipmap)

Returns the layout properties of a sparse CUDA mipmapped array.

Returns the sparse array layout properties in `sparseProperties`. If the CUDA mipmapped array is not allocated with flag *cudaArraySparse* *cudaErrorInvalidValue* will be returned.

For non-layered CUDA mipmapped arrays, *cudaArraySparseProperties::miptailSize* returns the size of the mip tail region. The mip tail region includes all mip levels whose width, height or depth is less than that of the tile. For layered CUDA mipmapped arrays, if *cudaArraySparseProperties::flags* contains *cudaArraySparsePropertiesSingleMipTail*, then *cudaArraySparseProperties::miptailSize* specifies the size of the mip tail of all layers combined. Otherwise, *cudaArraySparseProperties::miptailSize* specifies mip tail size per layer. The returned value of *cudaArraySparseProperties::miptailFirstLevel* is valid only if *cudaArraySparseProperties::miptailSize* is non-zero.

See also:

cudaArrayGetSparseProperties, ::cuMemMapArrayAsync

Parameters

- **sparseProperties** – **[out]** - Pointer to return *cudaArraySparseProperties*
- **mipmap** – **[in]** - The CUDA mipmapped array to get the sparse properties of

Returns
cudaSuccess *cudaErrorInvalidValue*

__host__ struct *cudaExtent* **make_cudaExtent**(size_t w, size_t h, size_t d)

Returns a *cudaExtent* based on input parameters.

Returns a *cudaExtent* based on the specified input parameters w, h, and d.

See also:

make_cudaPitchedPtr, *make_cudaPos*

Parameters

- **w** – - Width in elements when referring to array memory, in bytes when referring to linear memory
- **h** – - Height in elements
- **d** – - Depth in elements

Returns
cudaExtent specified by w, h, and d

__host__ struct *cudaPitchedPtr* **make_cudaPitchedPtr**(void *d, size_t p, size_t xsz, size_t ysz)

Returns a *cudaPitchedPtr* based on input parameters.

Returns a *cudaPitchedPtr* based on the specified input parameters d, p, xsz, and ysz.

See also:

make_cudaExtent, *make_cudaPos*

Parameters

- **d** – - Pointer to allocated memory
- **p** – - Pitch of allocated memory in bytes
- **xsz** – - Logical width of allocation in elements
- **ysz** – - Logical height of allocation in elements

Returns
cudaPitchedPtr specified by d, p, xsz, and ysz

__host__ struct *cudaPos* **make_cudaPos**(size_t x, size_t y, size_t z)

Returns a *cudaPos* based on input parameters.

Returns a *cudaPos* based on the specified input parameters x, y, and z.

See also:

make_cudaExtent, *make_cudaPitchedPtr*

Parameters

- **x** – - X position
- **y** – - Y position
- **z** – - Z position

Returns

cudaPos specified by x, y, and z

6.26. Memory Management [DEPRECATED]

This section describes deprecated memory management functions of the CUDA runtime application programming interface.

Some functions have overloaded C++ API template versions documented separately in the *C++ API Routines* module.

Functions

__host__ cudaError_t *cudaMemcpyArrayToArray*(cudaArray_t dst, size_t wOffsetDst, size_t hOffsetDst, cudaArray_const_t src, size_t wOffsetSrc, size_t hOffsetSrc, size_t count, enum cudaMemcpyKind kind=cudaMemcpyDeviceToDevice)
Copies data between host and device.

__host__ cudaError_t *cudaMemcpyFromArray*(void *dst, cudaArray_const_t src, size_t wOffset, size_t hOffset, size_t count, enum cudaMemcpyKind kind)
Copies data between host and device.

__host__ cudaError_t *cudaMemcpyFromArrayAsync*(void *dst, cudaArray_const_t src, size_t wOffset, size_t hOffset, size_t count, enum cudaMemcpyKind kind, cudaStream_t stream=0)
Copies data between host and device.

__host__ cudaError_t *cudaMemcpyToArray*(cudaArray_t dst, size_t wOffset, size_t hOffset, const void *src, size_t count, enum cudaMemcpyKind kind)
Copies data between host and device.

__host__ cudaError_t *cudaMemcpyToArrayAsync*(cudaArray_t dst, size_t wOffset, size_t hOffset, const void *src, size_t count, enum cudaMemcpyKind kind, cudaStream_t stream=0)
Copies data between host and device.

6.26.1. Functions

__host__ *cudaError_t* **cudaMemcpyArrayToArray**(*cudaArray_t* dst, size_t wOffsetDst, size_t hOffsetDst, *cudaArray_const_t* src, size_t wOffsetSrc, size_t hOffsetSrc, size_t count, enum *cudaMemcpyKind* kind = *cudaMemcpyDeviceToDevice*)

Copies data between host and device.

Deprecated:

Copies `count` bytes from the CUDA array `src` starting at `hOffsetSrc` rows and `wOffsetSrc` bytes from the upper left corner to the CUDA array `dst` starting at `hOffsetDst` rows and `wOffsetDst` bytes from the upper left corner, where `kind` specifies the direction of the copy, and must be one of *cudaMemcpyHostToHost*, *cudaMemcpyHostToDevice*, *cudaMemcpyDeviceToHost*, *cudaMemcpyDeviceToDevice*, or *cudaMemcpyDefault*. Passing *cudaMemcpyDefault* is recommended, in which case the type of transfer is inferred from the pointer values. However, *cudaMemcpyDefault* is only allowed on systems that support unified virtual addressing.

See also:

cudaMemcpy, *cudaMemcpy2D*, *cudaMemcpyToArray*, *cudaMemcpy2DToArray*, *cudaMemcpyFromArray*, *cudaMemcpy2DFromArray*, *cudaMemcpy2DArrayToArray*, *cudaMemcpyToSymbol*, *cudaMemcpyFromSymbol*, *cudaMemcpyAsync*, *cudaMemcpy2DAsync*, *cudaMemcpyToArrayAsync*, *cudaMemcpy2DToArrayAsync*, *cudaMemcpyFromArrayAsync*, *cudaMemcpy2DFromArrayAsync*, *cudaMemcpyToSymbolAsync*, *cudaMemcpyFromSymbolAsync*, ::cuMemcpyAtoA

Note: Note that this function may also return error codes from previous, asynchronous launches.

Note: Note that this function may also return *cudaErrorInitializationError*, *cudaErrorInsufficientDriver* or *cudaErrorNoDevice* if this call tries to initialize internal CUDA RT state.

Note: Note that as specified by *cudaStreamAddCallback* no CUDA function may be called from callback. *cudaErrorNotPermitted* may, but is not guaranteed to, be returned as a diagnostic in such case.

Parameters

- **dst** – - Destination memory address
- **wOffsetDst** – - Destination starting X offset (columns in bytes)
- **hOffsetDst** – - Destination starting Y offset (rows)
- **src** – - Source memory address
- **wOffsetSrc** – - Source starting X offset (columns in bytes)
- **hOffsetSrc** – - Source starting Y offset (rows)

- **count** – - Size in bytes to copy
- **kind** – - Type of transfer

Returns

cudaSuccess, *cudaErrorInvalidValue*, *cudaErrorInvalidMemcpyDirection*

__host__ *cudaError_t* **cudaMemcpyFromArray**(void *dst, *cudaArray_const_t* src, size_t wOffset, size_t hOffset, size_t count, enum *cudaMemcpyKind* kind)

Copies data between host and device.

Deprecated:

Copies `count` bytes from the CUDA array `src` starting at `hOffset` rows and `wOffset` bytes from the upper left corner to the memory area pointed to by `dst`, where `kind` specifies the direction of the copy, and must be one of *cudaMemcpyHostToHost*, *cudaMemcpyHostToDevice*, *cudaMemcpyDeviceToHost*, *cudaMemcpyDeviceToDevice*, or *cudaMemcpyDefault*. Passing *cudaMemcpyDefault* is recommended, in which case the type of transfer is inferred from the pointer values. However, *cudaMemcpyDefault* is only allowed on systems that support unified virtual addressing.

See also:

cudaMemcpy, *cudaMemcpy2D*, *cudaMemcpyToArray*, *cudaMemcpy2DToArray*, *cudaMemcpy2DFromArray*, *cudaMemcpyArrayToArray*, *cudaMemcpy2DArrayToArray*, *cudaMemcpyToSymbol*, *cudaMemcpyFromSymbol*, *cudaMemcpyAsync*, *cudaMemcpy2DAsync*, *cudaMemcpyToArrayAsync*, *cudaMemcpy2DToArrayAsync*, *cudaMemcpyFromArrayAsync*, *cudaMemcpy2DFromArrayAsync*, *cudaMemcpyToSymbolAsync*, *cudaMemcpyFromSymbolAsync*, ::cuMemcpyAtoH, ::cuMemcpyAtoD

Note: Note that this function may also return error codes from previous, asynchronous launches.

Note: This function exhibits synchronous behavior for most use cases.

Note: Note that this function may also return *cudaErrorInitializationError*, *cudaErrorInsufficientDriver* or *cudaErrorNoDevice* if this call tries to initialize internal CUDA RT state.

Note: Note that as specified by *cudaStreamAddCallback* no CUDA function may be called from callback. *cudaErrorNotPermitted* may, but is not guaranteed to, be returned as a diagnostic in such case.

Parameters

- **dst** – - Destination memory address
- **src** – - Source memory address
- **wOffset** – - Source starting X offset (columns in bytes)

- **hOffset** – - Source starting Y offset (rows)
- **count** – - Size in bytes to copy
- **kind** – - Type of transfer

Returns

cudaSuccess, *cudaErrorInvalidValue*, *cudaErrorInvalidMemcpyDirection*

__host__ *cudaError_t* **cudaMemcpyFromArrayAsync**(void *dst, *cudaArray_const_t* src, size_t wOffset, size_t hOffset, size_t count, enum *cudaMemcpyKind* kind, *cudaStream_t* stream = 0)

Copies data between host and device.

Deprecated:

Copies `count` bytes from the CUDA array `src` starting at `hOffset` rows and `wOffset` bytes from the upper left corner to the memory area pointed to by `dst`, where `kind` specifies the direction of the copy, and must be one of *cudaMemcpyHostToHost*, *cudaMemcpyHostToDevice*, *cudaMemcpyDeviceToHost*, *cudaMemcpyDeviceToDevice*, or *cudaMemcpyDefault*. Passing *cudaMemcpyDefault* is recommended, in which case the type of transfer is inferred from the pointer values. However, *cudaMemcpyDefault* is only allowed on systems that support unified virtual addressing.

cudaMemcpyFromArrayAsync() is asynchronous with respect to the host, so the call may return before the copy is complete. The copy can optionally be associated to a stream by passing a non-zero `stream` argument. If `kind` is *cudaMemcpyHostToDevice* or *cudaMemcpyDeviceToHost* and `stream` is non-zero, the copy may overlap with operations in other streams.

See also:

cudaMemcpy, *cudaMemcpy2D*, *cudaMemcpyToArray*, *cudaMemcpy2DToArray*, *cudaMemcpyFromArray*, *cudaMemcpy2DFromArray*, *cudaMemcpyArrayToArray*, *cudaMemcpy2DArrayToArray*, *cudaMemcpyToSymbol*, *cudaMemcpyFromSymbol*, *cudaMemcpyAsync*, *cudaMemcpy2DAsync*, *cudaMemcpyToArrayAsync*, *cudaMemcpy2DToArrayAsync*, *cudaMemcpy2DFromArrayAsync*, *cudaMemcpyToSymbolAsync*, *cudaMemcpyFromSymbolAsync*, ::cuMemcpyAtoHAsync, ::cuMemcpy2DAsync

Note: Note that this function may also return error codes from previous, asynchronous launches.

Note: This function exhibits asynchronous behavior for most use cases.

Note: This function uses standard default stream semantics.

Note: Note that this function may also return *cudaErrorInitializationError*, *cudaErrorInsufficientDriver* or *cudaErrorNoDevice* if this call tries to initialize internal CUDA RT state.

Note: Note that as specified by *cudaStreamAddCallback* no CUDA function may be called from callback. *cudaErrorNotPermitted* may, but is not guaranteed to, be returned as a diagnostic in such case.

Parameters

- **dst** – - Destination memory address
- **src** – - Source memory address
- **wOffset** – - Source starting X offset (columns in bytes)
- **hOffset** – - Source starting Y offset (rows)
- **count** – - Size in bytes to copy
- **kind** – - Type of transfer
- **stream** – - Stream identifier

Returns

cudaSuccess, *cudaErrorInvalidValue*, *cudaErrorInvalidMemcpyDirection*

__host__ *cudaError_t* **cudaMemcpyToArray**(*cudaArray_t* dst, size_t wOffset, size_t hOffset, const void *src, size_t count, enum *cudaMemcpyKind* kind)

Copies data between host and device.

Deprecated:

Copies `count` bytes from the memory area pointed to by `src` to the CUDA array `dst` starting at `hOffset` rows and `wOffset` bytes from the upper left corner, where `kind` specifies the direction of the copy, and must be one of *cudaMemcpyHostToHost*, *cudaMemcpyHostToDevice*, *cudaMemcpyDeviceToHost*, *cudaMemcpyDeviceToDevice*, or *cudaMemcpyDefault*. Passing *cudaMemcpyDefault* is recommended, in which case the type of transfer is inferred from the pointer values. However, *cudaMemcpyDefault* is only allowed on systems that support unified virtual addressing.

See also:

cudaMemcpy, *cudaMemcpy2D*, *cudaMemcpy2DToArray*, *cudaMemcpyFromArray*, *cudaMemcpy2DFromArray*, *cudaMemcpyArrayToArray*, *cudaMemcpy2DArrayToArray*, *cudaMemcpyToSymbol*, *cudaMemcpyFromSymbol*, *cudaMemcpyAsync*, *cudaMemcpy2DAsync*, *cudaMemcpyToArrayAsync*, *cudaMemcpy2DToArrayAsync*, *cudaMemcpyFromArrayAsync*, *cudaMemcpy2DFromArrayAsync*, *cudaMemcpyToSymbolAsync*, *cudaMemcpyFromSymbolAsync*, ::cuMemcpyHtoA, ::cuMemcpyDtoA

Note: Note that this function may also return error codes from previous, asynchronous launches.

Note: This function exhibits synchronous behavior for most use cases.

Note: Note that this function may also return *cudaErrorInitializationError*, *cudaErrorInsufficientDriver* or *cudaErrorNoDevice* if this call tries to initialize internal CUDA RT state.

Note: Note that as specified by *cudaStreamAddCallback* no CUDA function may be called from callback. *cudaErrorNotPermitted* may, but is not guaranteed to, be returned as a diagnostic in such case.

Parameters

- **dst** – - Destination memory address
- **wOffset** – - Destination starting X offset (columns in bytes)
- **hOffset** – - Destination starting Y offset (rows)
- **src** – - Source memory address
- **count** – - Size in bytes to copy
- **kind** – - Type of transfer

Returns

cudaSuccess, *cudaErrorInvalidValue*, *cudaErrorInvalidMemcpyDirection*

__host__ *cudaError_t* **cudaMemcpyToArrayAsync**(*cudaArray_t* dst, size_t wOffset, size_t hOffset, const void *src, size_t count, enum *cudaMemcpyKind* kind, *cudaStream_t* stream = 0)

Copies data between host and device.

Deprecated:

Copies `count` bytes from the memory area pointed to by `src` to the CUDA array `dst` starting at `hOffset` rows and `wOffset` bytes from the upper left corner, where `kind` specifies the direction of the copy, and must be one of *cudaMemcpyHostToHost*, *cudaMemcpyHostToDevice*, *cudaMemcpyDeviceToHost*, *cudaMemcpyDeviceToDevice*, or *cudaMemcpyDefault*. Passing *cudaMemcpyDefault* is recommended, in which case the type of transfer is inferred from the pointer values. However, *cudaMemcpyDefault* is only allowed on systems that support unified virtual addressing.

cudaMemcpyToArrayAsync() is asynchronous with respect to the host, so the call may return before the copy is complete. The copy can optionally be associated to a stream by passing a non-zero `stream` argument. If `kind` is *cudaMemcpyHostToDevice* or *cudaMemcpyDeviceToHost* and `stream` is non-zero, the copy may overlap with operations in other streams.

See also:

cudaMemcpy, *cudaMemcpy2D*, *cudaMemcpyToArray*, *cudaMemcpy2DToArray*, *cudaMemcpyFromArray*, *cudaMemcpy2DFromArray*, *cudaMemcpyArrayToArray*, *cudaMemcpy2DArrayToArray*, *cudaMemcpyToSymbol*, *cudaMemcpyFromSymbol*, *cudaMemcpyAsync*, *cudaMemcpy2DAsync*, *cudaMemcpy2DToArrayAsync*, *cudaMemcpyFromArrayAsync*, *cudaMemcpy2DFromArrayAsync*, *cudaMemcpyToSymbolAsync*, *cudaMemcpyFromSymbolAsync*, ::cuMemcpyHtoAAsync, ::cuMemcpy2DAsync

Note: Note that this function may also return error codes from previous, asynchronous launches.

Note: This function exhibits asynchronous behavior for most use cases.

Note: This function uses standard default stream semantics.

Note: Note that this function may also return *cudaErrorInitializationError*, *cudaErrorInsufficientDriver* or *cudaErrorNoDevice* if this call tries to initialize internal CUDA RT state.

Note: Note that as specified by *cudaStreamAddCallback* no CUDA function may be called from callback. *cudaErrorNotPermitted* may, but is not guaranteed to, be returned as a diagnostic in such case.

Parameters

- **dst** – - Destination memory address
- **wOffset** – - Destination starting X offset (columns in bytes)
- **hOffset** – - Destination starting Y offset (rows)
- **src** – - Source memory address
- **count** – - Size in bytes to copy
- **kind** – - Type of transfer
- **stream** – - Stream identifier

Returns

cudaSuccess, *cudaErrorInvalidValue*, *cudaErrorInvalidMemcpyDirection*

6.27. Occupancy

This section describes the occupancy calculation functions of the CUDA runtime application programming interface.

Besides the occupancy calculator functions (*cudaOccupancyMaxActiveBlocksPerMultiprocessor* and *cudaOccupancyMaxActiveBlocksPerMultiprocessorWithFlags*), there are also C++ only occupancy-based launch configuration functions documented in *C++ API Routines* module.

See *cudaOccupancyMaxPotentialBlockSize (C++ API)*, *cudaOccupancyMaxPotentialBlockSize (C++ API)*, *cudaOccupancyMaxPotentialBlockSizeVariableSMem (C++ API)*, *cudaOccupancyMaxPotentialBlockSizeVariableSMem (C++ API)* cudaOccupancyAvailableDynamicSMemPerBlock (C++ API),

Functions

__host__ cudaError_t *cudaOccupancyAvailableDynamicSMemPerBlock*(size_t *dynamicSmemSize, const void *func, int numBlocks, int blockSize)
Returns dynamic shared memory available per block when launching `numBlocks` blocks on SM.

__host__ cudaError_t *cudaOccupancyMaxActiveBlocksPerMultiprocessor*(int *numBlocks, const void *func, int blockSize, size_t dynamicSMemSize)
Returns occupancy for a device function.

__host__ cudaError_t *cudaOccupancyMaxActiveBlocksPerMultiprocessorWithFlags*(int *numBlocks, const void *func, int blockSize, size_t dynamicSMemSize, unsigned int flags)
Returns occupancy for a device function with the specified flags.

__host__ cudaError_t *cudaOccupancyMaxActiveClusters*(int *numClusters, const void *func, const cudaLaunchConfig_t *launchConfig)
Given the kernel function (`func`) and launch configuration (`config`), return the maximum number of clusters that could co-exist on the target device in `*numClusters` .

__host__ cudaError_t *cudaOccupancyMaxPotentialClusterSize*(int *clusterSize, const void *func, const cudaLaunchConfig_t *launchConfig)
Given the kernel function (`func`) and launch configuration (`config`), return the maximum cluster size in `*clusterSize` .

6.27.1. Functions

__host__ *cudaError_t* **cudaOccupancyAvailableDynamicSMemPerBlock**(size_t *dynamicSmemSize, const void *func, int numBlocks, int blockSize)

Returns dynamic shared memory available per block when launching `numBlocks` blocks on SM.

Returns in `*dynamicSmemSize` the maximum size of dynamic shared memory to allow `numBlocks` blocks per SM.

See also:

cudaOccupancyMaxActiveBlocksPerMultiprocessorWithFlags, *cudaOccupancyMaxPotentialBlockSize (C++ API)*, *cudaOccupancyMaxPotentialBlockSizeWithFlags (C++ API)*, *cudaOccupancyMaxPotentialBlockSizeVariableSMem (C++ API)*, *cudaOccupancyMaxPotentialBlockSizeVariableSMemWithFlags (C++ API)*, *cudaOccupancyAvailableDynamicSMemPerBlock*

Note: Note that this function may also return error codes from previous, asynchronous launches.

Note: Note that this function may also return *cudaErrorInitializationError*, *cudaErrorInsufficientDriver* or *cudaErrorNoDevice* if this call tries to initialize internal CUDA RT state.

Note: Note that as specified by *cudaStreamAddCallback* no CUDA function may be called from callback. *cudaErrorNotPermitted* may, but is not guaranteed to, be returned as a diagnostic in

such case.

Note: The API can also be used with a kernel *cudaKernel_t* by querying the handle using *cudaLibraryGetKernel()* or *cudaGetKernel* and then passing it to the API by casting to void*. The symbol `entryFuncAddr` passed to *cudaGetKernel* should be a symbol that is registered with the same CUDA Runtime instance.

Note: Passing a symbol that belongs that belongs to a different runtime instance will result in undefined behavior. The only type that can be reliably passed to a different runtime instance is *cudaKernel_t*

Parameters

- **dynamicSmemSize** – - Returned maximum dynamic shared memory
- **func** – - Kernel function for which occupancy is calculated
- **numBlocks** – - Number of blocks to fit on SM
- **blockSize** – - Size of the block

Returns

cudaSuccess, *cudaErrorInvalidDevice*, *cudaErrorInvalidDeviceFunction*, *cudaErrorInvalidValue*, *cudaErrorUnknown*,

__host__ *cudaError_t* **cudaOccupancyMaxActiveBlocksPerMultiprocessor**(int *numBlocks, const void *func, int blockSize, size_t dynamicSMemSize)

Returns occupancy for a device function.

Returns in `*numBlocks` the maximum number of active blocks per streaming multiprocessor for the device function.

See also:

cudaOccupancyMaxActiveBlocksPerMultiprocessorWithFlags, *cudaOccupancyMaxPotentialBlockSize (C++ API)*, *cudaOccupancyMaxPotentialBlockSizeWithFlags (C++ API)*, *cudaOccupancyMaxPotentialBlockSizeVariableSMem (C++ API)*, *cudaOccupancyMaxPotentialBlockSizeVariableSMemWithFlags (C++ API)*, cudaOccupancyAvailableDynamicSMemPerBlock (C++ API), ::cuOccupancyMaxActiveBlocksPerMultiprocessor

Note: Note that this function may also return error codes from previous, asynchronous launches.

Note: Note that this function may also return *cudaErrorInitializationError*, *cudaErrorInsufficientDriver* or *cudaErrorNoDevice* if this call tries to initialize internal CUDA RT state.

Note: Note that as specified by *cudaStreamAddCallback* no CUDA function may be called from callback. *cudaErrorNotPermitted* may, but is not guaranteed to, be returned as a diagnostic in such case.

Note: The API can also be used with a kernel *cudaKernel_t* by querying the handle using *cudaLibraryGetKernel()* or *cudaGetKernel* and then passing it to the API by casting to void*. The symbol `entryFuncAddr` passed to *cudaGetKernel* should be a symbol that is registered with the same CUDA Runtime instance.

Note: Passing a symbol that belongs that belongs to a different runtime instance will result in undefined behavior. The only type that can be reliably passed to a different runtime instance is *cudaKernel_t*

Parameters

- **numBlocks** – - Returned occupancy
- **func** – - Kernel function for which occupancy is calculated
- **blockSize** – - Block size the kernel is intended to be launched with
- **dynamicSMemSize** – - Per-block dynamic shared memory usage intended, in bytes

Returns

cudaSuccess, *cudaErrorInvalidDevice*, *cudaErrorInvalidDeviceFunction*, *cudaErrorInvalidValue*, *cudaErrorUnknown*,

__host__ *cudaError_t* **cudaOccupancyMaxActiveBlocksPerMultiprocessorWithFlags**(int *numBlocks, const void *func, int blockSize, size_t dynamicSMemSize, unsigned int flags)

Returns occupancy for a device function with the specified flags.

Returns in `*numBlocks` the maximum number of active blocks per streaming multiprocessor for the device function.

The `flags` parameter controls how special cases are handled. Valid flags include:

- *cudaOccupancyDefault*: keeps the default behavior as *cudaOccupancyMaxActiveBlocksPerMultiprocessor*
- *cudaOccupancyDisableCachingOverride*: This flag suppresses the default behavior on platform where global caching affects occupancy. On such platforms, if caching is enabled, but per-block SM resource usage would result in zero occupancy, the occupancy calculator will calculate the occupancy as if caching is disabled. Setting this flag makes the occupancy calculator to return 0 in such cases. More information can be found about this feature in the "Unified L1/Texture Cache" section of the Maxwell tuning guide.

See also:

cudaOccupancyMaxActiveBlocksPerMultiprocessor, *cudaOccupancyMaxPotentialBlockSize (C++ API)*, *cudaOccupancyMaxPotentialBlockSizeWithFlags (C++ API)*, *cudaOccupancyMaxPotentialBlockSizeVariableSMem (C++ API)*, *cudaOccupancyMaxPotentialBlockSizeVariableSMemWithFlags (C++ API)*, cudaOccupancyAvailableDynamicSMemPerBlock (C++ API), ::cuOccupancyMaxActiveBlocksPerMultiprocessorWithFlags

---

Note: Note that this function may also return error codes from previous, asynchronous launches.

Note: Note that this function may also return *cudaErrorInitializationError*, *cudaErrorInsufficientDriver* or *cudaErrorNoDevice* if this call tries to initialize internal CUDA RT state.

Note: Note that as specified by *cudaStreamAddCallback* no CUDA function may be called from callback. *cudaErrorNotPermitted* may, but is not guaranteed to, be returned as a diagnostic in such case.

Note: The API can also be used with a kernel *cudaKernel_t* by querying the handle using *cudaLibraryGetKernel()* or *cudaGetKernel* and then passing it to the API by casting to void*. The symbol `entryFuncAddr` passed to *cudaGetKernel* should be a symbol that is registered with the same CUDA Runtime instance.

Note: Passing a symbol that belongs that belongs to a different runtime instance will result in undefined behavior. The only type that can be reliably passed to a different runtime instance is *cudaKernel_t*

Parameters

- **`numBlocks`** – - Returned occupancy
- **`func`** – - Kernel function for which occupancy is calculated
- **`blockSize`** – - Block size the kernel is intended to be launched with

- **dynamicSMemSize** – - Per-block dynamic shared memory usage intended, in bytes
- **flags** – - Requested behavior for the occupancy calculator

Returns

cudaSuccess, *cudaErrorInvalidDevice*, *cudaErrorInvalidDeviceFunction*, *cudaErrorInvalidValue*, *cudaErrorUnknown*,

__host__ *cudaError_t* **cudaOccupancyMaxActiveClusters**(int *numClusters, const void *func, const *cudaLaunchConfig_t* *launchConfig)

Given the kernel function (`func`) and launch configuration (`config`), return the maximum number of clusters that could co-exist on the target device in `*numClusters`.

If the function has required cluster size already set (see *cudaFuncGetAttributes*), the cluster size from config must either be unspecified or match the required size. Without required sizes, the cluster size must be specified in config, else the function will return an error.

Note that various attributes of the kernel function may affect occupancy calculation. Runtime environment may affect how the hardware schedules the clusters, so the calculated occupancy is not guaranteed to be achievable.

See also:

cudaFuncGetAttributes cudaOccupancyMaxActiveClusters (C++ API), ::cuOccupancyMaxActiveClusters

---

Note: Note that this function may also return error codes from previous, asynchronous launches.

Note: Note that this function may also return *cudaErrorInitializationError*, *cudaErrorInsufficientDriver* or *cudaErrorNoDevice* if this call tries to initialize internal CUDA RT state.

Note: Note that as specified by *cudaStreamAddCallback* no CUDA function may be called from callback. *cudaErrorNotPermitted* may, but is not guaranteed to, be returned as a diagnostic in such case.

Note: The API can also be used with a kernel *cudaKernel_t* by querying the handle using *cudaLibraryGetKernel()* or *cudaGetKernel* and then passing it to the API by casting to void*. The symbol `entryFuncAddr` passed to *cudaGetKernel* should be a symbol that is registered with the same CUDA Runtime instance.

Note: Passing a symbol that belongs that belongs to a different runtime instance will result in undefined behavior. The only type that can be reliably passed to a different runtime instance is *cudaKernel_t*

Parameters

- **numClusters** – - Returned maximum number of clusters that could co-exist on the target device
- **func** – - Kernel function for which maximum number of clusters are calculated
- **config** – - Launch configuration for the given kernel function

Returns

cudaSuccess, *cudaErrorInvalidDeviceFunction*, *cudaErrorInvalidValue*, *cudaErrorInvalidClusterSize*, *cudaErrorUnknown*,

__host__ *cudaError_t* **cudaOccupancyMaxPotentialClusterSize**(int *clusterSize, const void *func, const *cudaLaunchConfig_t* *launchConfig)

Given the kernel function (`func`) and launch configuration (`config`), return the maximum cluster size in `*clusterSize`.

The cluster dimensions in `config` are ignored. If func has a required cluster size set (see *cudaFuncGetAttributes*),`*clusterSize` will reflect the required cluster size.

By default this function will always return a value that's portable on future hardware. A higher value may be returned if the kernel function allows non-portable cluster sizes.

This function will respect the compile time launch bounds.

See also:

cudaFuncGetAttributes cudaOccupancyMaxPotentialClusterSize (C++ API), ::cuOccupancyMaxPotentialClusterSize

---

Note: Note that this function may also return error codes from previous, asynchronous launches.

Note: Note that this function may also return *cudaErrorInitializationError*, *cudaErrorInsufficientDriver* or *cudaErrorNoDevice* if this call tries to initialize internal CUDA RT state.

Note: Note that as specified by *cudaStreamAddCallback* no CUDA function may be called from callback. *cudaErrorNotPermitted* may, but is not guaranteed to, be returned as a diagnostic in such case.

Note: The API can also be used with a kernel *cudaKernel_t* by querying the handle using *cudaLibraryGetKernel()* or *cudaGetKernel* and then passing it to the API by casting to void*. The symbol `entryFuncAddr` passed to *cudaGetKernel* should be a symbol that is registered with the same CUDA Runtime instance.

Note: Passing a symbol that belongs that belongs to a different runtime instance will result in undefined behavior. The only type that can be reliably passed to a different runtime instance is *cudaKernel_t*

Parameters

- **clusterSize** – - Returned maximum cluster size that can be launched for the given kernel function and launch configuration
- **func** – - Kernel function for which maximum cluster size is calculated
- **config** – - Launch configuration for the given kernel function

Returns

cudaSuccess, *cudaErrorInvalidDeviceFunction*, *cudaErrorInvalidValue*, *cudaErrorUnknown*,

6.28. OpenGL Interoperability

impl_private

This section describes the OpenGL interoperability functions of the CUDA runtime application programming interface. Note that mapping of OpenGL resources is performed with the graphics API agnostic, resource mapping interface described in *Graphics Interopability*.

Enumerations

cudaGLDeviceList

CUDA devices corresponding to the current OpenGL context.

Functions

__host__ cudaError_t *cudaGLGetDevices*(unsigned int *pCudaDeviceCount, int *pCudaDevices, unsigned int cudaDeviceCount, enum cudaGLDeviceList deviceList)

Gets the CUDA devices associated with the current OpenGL context.

__host__ cudaError_t *cudaGraphicsGLRegisterBuffer*(struct cudaGraphicsResource **resource, GLuint buffer, unsigned int flags)

Registers an OpenGL buffer object.

__host__ cudaError_t *cudaGraphicsGLRegisterImage*(struct cudaGraphicsResource **resource, GLuint image, GLenum target, unsigned int flags)

Register an OpenGL texture or renderbuffer object.

__host__ cudaError_t *cudaWGLGetDevice*(int *device, HGPUNV hGpu)

Gets the CUDA device associated with hGpu.

Typedefs

HGPUNV

6.28.1. Enumerations

enum **cudaGLDeviceList**

CUDA devices corresponding to the current OpenGL context.

Values:

enumerator **cudaGLDeviceListAll** = 1

The CUDA devices for all GPUs used by the current OpenGL context.

enumerator **cudaGLDeviceListCurrentFrame** = 2

The CUDA devices for the GPUs used by the current OpenGL context in its currently rendering frame.

enumerator **cudaGLDeviceListNextFrame** = 3

The CUDA devices for the GPUs to be used by the current OpenGL context in the next frame

6.28.2. Functions

__host__ *cudaError_t* **cudaGLGetDevices**(unsigned int *pCudaDeviceCount, int *pCudaDevices, unsigned int cudaDeviceCount, enum *cudaGLDeviceList* deviceList)

Gets the CUDA devices associated with the current OpenGL context.

Returns in `*pCudaDeviceCount` the number of CUDA-compatible devices corresponding to the current OpenGL context. Also returns in `*pCudaDevices` at most `cudaDeviceCount` of the CUDA-compatible devices corresponding to the current OpenGL context. If any of the GPUs being used by the current OpenGL context are not CUDA capable then the call will return *cudaErrorNoDevice*.

See also:

cudaGraphicsUnregisterResource, *cudaGraphicsMapResources*, *cudaGraphicsSubResourceGetMappedArray*, *cudaGraphicsResourceGetMappedPointer*, ::cuGLGetDevices

Note: This function is not supported on Mac OS X.

Note: Note that this function may also return error codes from previous, asynchronous launches.

Parameters

- **pCudaDeviceCount** – - Returned number of CUDA devices corresponding to the current OpenGL context
- **pCudaDevices** – - Returned CUDA devices corresponding to the current OpenGL context
- **cudaDeviceCount** – - The size of the output device array pCudaDevices
- **deviceList** – - The set of devices to return. This set may be *cudaGLDeviceListAll* for all devices, *cudaGLDeviceListCurrentFrame* for the devices used to render the current frame (in SLI), or *cudaGLDeviceListNextFrame* for the devices used to render the next frame (in SLI).

Returns

cudaSuccess, *cudaErrorNoDevice*, *cudaErrorInvalidGraphicsContext*, *cudaErrorOperatingSystem*, *cudaErrorUnknown*

__host__ *cudaError_t* **cudaGraphicsGLRegisterBuffer**(struct cudaGraphicsResource **resource, GLuint buffer, unsigned int flags)

Registers an OpenGL buffer object.

Registers the buffer object specified by buffer for access by CUDA. A handle to the registered object is returned as resource. The register flags flags specify the intended usage, as follows:

- *cudaGraphicsRegisterFlagsNone*: Specifies no hints about how this resource will be used. It is therefore assumed that this resource will be read from and written to by CUDA. This is the default value.
- *cudaGraphicsRegisterFlagsReadOnly*: Specifies that CUDA will not write to this resource.
- *cudaGraphicsRegisterFlagsWriteDiscard*: Specifies that CUDA will not read from this resource and will write over the entire contents of the resource, so none of the data previously stored in the resource will be preserved.

See also:

cudaGraphicsUnregisterResource, *cudaGraphicsMapResources*, *cudaGraphicsResourceGetMappedPointer*, ::cuGraphicsGLRegisterBuffer

---

Note: Note that this function may also return error codes from previous, asynchronous launches.

Parameters

- **resource** – - Pointer to the returned object handle
- **buffer** – - name of buffer object to be registered
- **flags** – - Register flags

Returns

cudaSuccess, *cudaErrorInvalidDevice*, *cudaErrorInvalidValue*, *cudaErrorInvalidResourceHandle*, *cudaErrorOperatingSystem*, *cudaErrorUnknown*

__host__ *cudaError_t* **cudaGraphicsGLRegisterImage**(struct cudaGraphicsResource **resource, GLuint image, GLenum target, unsigned int flags)

Register an OpenGL texture or renderbuffer object.

Registers the texture or renderbuffer object specified by `image` for access by CUDA. A handle to the registered object is returned as `resource`.

`target` must match the type of the object, and must be one of ::GL_TEXTURE_2D, ::GL_TEXTURE_RECTANGLE, ::GL_TEXTURE_CUBE_MAP, ::GL_TEXTURE_3D, ::GL_TEXTURE_2D_ARRAY, or ::GL_RENDERBUFFER.

The register flags `flags` specify the intended usage, as follows:

- *cudaGraphicsRegisterFlagsNone*: Specifies no hints about how this resource will be used. It is therefore assumed that this resource will be read from and written to by CUDA. This is the default value.
- *cudaGraphicsRegisterFlagsReadOnly*: Specifies that CUDA will not write to this resource.
- *cudaGraphicsRegisterFlagsWriteDiscard*: Specifies that CUDA will not read from this resource and will write over the entire contents of the resource, so none of the data previously stored in the resource will be preserved.
- *cudaGraphicsRegisterFlagsSurfaceLoadStore*: Specifies that CUDA will bind this resource to a surface reference.
- *cudaGraphicsRegisterFlagsTextureGather*: Specifies that CUDA will perform texture gather operations on this resource.

The following image formats are supported. For brevity's sake, the list is abbreviated. For ex., {GL_R, GL_RG} X {8, 16} would expand to the following 4 formats {GL_R8, GL_R16, GL_RG8, GL_RG16} :

- GL_RED, GL_RG, GL_RGBA, GL_LUMINANCE, GL_ALPHA, GL_LUMINANCE_ALPHA, GL_INTENSITY
- {GL_R, GL_RG, GL_RGBA} X {8, 16, 16F, 32F, 8UI, 16UI, 32UI, 8I, 16I, 32I}
- {GL_LUMINANCE, GL_ALPHA, GL_LUMINANCE_ALPHA, GL_INTENSITY} X {8, 16, 16F_ARB, 32F_ARB, 8UI_EXT, 16UI_EXT, 32UI_EXT, 8I_EXT, 16I_EXT, 32I_EXT}

The following image classes are currently disallowed:

- Textures with borders
- Multisampled renderbuffers

See also:

cudaGraphicsUnregisterResource, *cudaGraphicsMapResources*, *cudaGraphicsSubResourceGetMappedArray*, ::cuGraphicsGLRegisterImage

---

Note: Note that this function may also return error codes from previous, asynchronous launches.

Parameters

- **resource** – - Pointer to the returned object handle

- **image** – - name of texture or renderbuffer object to be registered
- **target** – - Identifies the type of object specified by image
- **flags** – - Register flags

Returns

cudaSuccess, *cudaErrorInvalidDevice*, *cudaErrorInvalidValue*, *cudaErrorInvalidResourceHandle*, *cudaErrorOperatingSystem*, *cudaErrorUnknown*

__host__ *cudaError_t* **cudaWGLGetDevice**(int *device, *HGPUNV* hGpu)

Gets the CUDA device associated with hGpu.

Returns the CUDA device associated with a hGpu, if applicable.

See also:

::WGL_NV_gpu_affinity, ::cuWGLGetDevice

Note: Note that this function may also return error codes from previous, asynchronous launches.

Parameters

- **device** – - Returns the device associated with hGpu, or -1 if hGpu is not a compute device.
- **hGpu** – - Handle to a GPU, as queried via WGL_NV_gpu_affinity

Returns

cudaSuccess

6.28.3. Typedefs

typedef void ***HGPUNV**

6.29. OpenGL Interoperability [DEPRECATED]

This section describes deprecated OpenGL interoperability functionality.

Enumerations

cudaGLMapFlags
CUDA GL Map Flags.

Functions

__host__ cudaError_t *cudaGLMapBufferObject*(void **devPtr, GLuint bufObj)
Maps a buffer object for access by CUDA.

__host__ cudaError_t *cudaGLMapBufferObjectAsync*(void **devPtr, GLuint bufObj, cudaStream_t stream)
Maps a buffer object for access by CUDA.

__host__ cudaError_t *cudaGLRegisterBufferObject*(GLuint bufObj)
Registers a buffer object for access by CUDA.

__host__ cudaError_t *cudaGLSetBufferObjectMapFlags*(GLuint bufObj, unsigned int flags)
Set usage flags for mapping an OpenGL buffer.

__host__ cudaError_t *cudaGLSetGLDevice*(int device)
Sets a CUDA device to use OpenGL interoperability.

__host__ cudaError_t *cudaGLUnmapBufferObject*(GLuint bufObj)
Unmaps a buffer object for access by CUDA.

__host__ cudaError_t *cudaGLUnmapBufferObjectAsync*(GLuint bufObj, cudaStream_t stream)
Unmaps a buffer object for access by CUDA.

__host__ cudaError_t *cudaGLUnregisterBufferObject*(GLuint bufObj)
Unregisters a buffer object for access by CUDA.

6.29.1. Enumerations

enum **cudaGLMapFlags**
CUDA GL Map Flags.

Values:

enumerator **cudaGLMapFlagsNone** = 0
Default; Assume resource can be read/written.

enumerator **cudaGLMapFlagsReadOnly** = 1
CUDA kernels will not write to this resource.

enumerator **cudaGLMapFlagsWriteDiscard** = 2
CUDA kernels will only write to and will not read from this resource.

6.29.2. Functions

__host__ *cudaError_t* **cudaGLMapBufferObject**(void **devPtr, GLuint bufObj)

Maps a buffer object for access by CUDA.

Deprecated:

This function is deprecated as of CUDA 3.0.

Maps the buffer object of ID `bufObj` into the address space of CUDA and returns in `*devPtr` the base pointer of the resulting mapping. The buffer must have previously been registered by calling *cudaGLRegisterBufferObject()*. While a buffer is mapped by CUDA, any OpenGL operation which references the buffer will result in undefined behavior. The OpenGL context used to create the buffer, or another context from the same share group, must be bound to the current thread when this is called.

All streams in the current thread are synchronized with the current GL context.

See also:

cudaGraphicsMapResources

Note: Note that this function may also return error codes from previous, asynchronous launches.

Parameters

- **devPtr** – - Returned device pointer to CUDA object
- **bufObj** – - Buffer object ID to map

Returns

cudaSuccess, *cudaErrorMapBufferObjectFailed*

__host__ *cudaError_t* **cudaGLMapBufferObjectAsync**(void **devPtr, GLuint bufObj, *cudaStream_t* stream)

Maps a buffer object for access by CUDA.

Deprecated:

This function is deprecated as of CUDA 3.0.

Maps the buffer object of ID `bufObj` into the address space of CUDA and returns in `*devPtr` the base pointer of the resulting mapping. The buffer must have previously been registered by calling *cudaGLRegisterBufferObject()*. While a buffer is mapped by CUDA, any OpenGL operation which references the buffer will result in undefined behavior. The OpenGL context used to create the buffer, or another context from the same share group, must be bound to the current thread when this is called.

Stream /p stream is synchronized with the current GL context.

See also:

cudaGraphicsMapResources

Note: Note that this function may also return error codes from previous, asynchronous launches.

Parameters

- **devPtr** – - Returned device pointer to CUDA object
- **bufObj** – - Buffer object ID to map
- **stream** – - Stream to synchronize

Returns

cudaSuccess, *cudaErrorMapBufferObjectFailed*

__host__ *cudaError_t* **cudaGLRegisterBufferObject**(GLuint bufObj)

Registers a buffer object for access by CUDA.

Deprecated:

This function is deprecated as of CUDA 3.0.

Registers the buffer object of ID `bufObj` for access by CUDA. This function must be called before CUDA can map the buffer object. The OpenGL context used to create the buffer, or another context from the same share group, must be bound to the current thread when this is called.

See also:

cudaGraphicsGLRegisterBuffer

Note: Note that this function may also return error codes from previous, asynchronous launches.

Parameters

bufObj – - Buffer object ID to register

Returns

cudaSuccess, *cudaErrorInitializationError*

__host__ *cudaError_t* **cudaGLSetBufferObjectMapFlags**(GLuint bufObj, unsigned int flags)

Set usage flags for mapping an OpenGL buffer.

Deprecated:

This function is deprecated as of CUDA 3.0.

Set flags for mapping the OpenGL buffer `bufObj`

Changes to flags will take effect the next time `bufObj` is mapped. The `flags` argument may be any of the following:

- *cudaGLMapFlagsNone*: Specifies no hints about how this buffer will be used. It is therefore assumed that this buffer will be read from and written to by CUDA kernels. This is the default value.
- *cudaGLMapFlagsReadOnly*: Specifies that CUDA kernels which access this buffer will not write to the buffer.
- *cudaGLMapFlagsWriteDiscard*: Specifies that CUDA kernels which access this buffer will not read from the buffer and will write over the entire contents of the buffer, so none of the data previously stored in the buffer will be preserved.

If `bufObj` has not been registered for use with CUDA, then *cudaErrorInvalidResourceHandle* is returned. If `bufObj` is presently mapped for access by CUDA, then *cudaErrorUnknown* is returned.

See also:

cudaGraphicsResourceSetMapFlags

Note: Note that this function may also return error codes from previous, asynchronous launches.

Parameters

- **`bufObj`** – - Registered buffer object to set flags for
- **`flags`** – - Parameters for buffer mapping

Returns

cudaSuccess, *cudaErrorInvalidValue*, *cudaErrorInvalidResourceHandle*, *cudaErrorUnknown*

__host__ *cudaError_t* **`cudaGLSetGLDevice`**(int device)

Sets a CUDA device to use OpenGL interoperability.

Deprecated:

This function is deprecated as of CUDA 5.0.

This function is deprecated and should no longer be used. It is no longer necessary to associate a CUDA device with an OpenGL context in order to achieve maximum interoperability performance.

This function will immediately initialize the primary context on `device` if needed.

See also:

cudaGraphicsGLRegisterBuffer, *cudaGraphicsGLRegisterImage*

Note: Note that this function may also return error codes from previous, asynchronous launches.

Parameters

`device` – - Device to use for OpenGL interoperability

Returns

cudaSuccess, *cudaErrorInvalidDevice*, *cudaErrorSetOnActiveProcess*

__host__ *cudaError_t* **cudaGLUnmapBufferObject**(GLuint bufObj)

Unmaps a buffer object for access by CUDA.

Deprecated:

This function is deprecated as of CUDA 3.0.

Unmaps the buffer object of ID `bufObj` for access by CUDA. When a buffer is unmapped, the base address returned by *cudaGLMapBufferObject()* is invalid and subsequent references to the address result in undefined behavior. The OpenGL context used to create the buffer, or another context from the same share group, must be bound to the current thread when this is called.

All streams in the current thread are synchronized with the current GL context.

See also:

cudaGraphicsUnmapResources

Note: Note that this function may also return error codes from previous, asynchronous launches.

Parameters

`bufObj` – - Buffer object to unmap

Returns

cudaSuccess, *cudaErrorUnmapBufferObjectFailed*

__host__ *cudaError_t* **cudaGLUnmapBufferObjectAsync**(GLuint bufObj, *cudaStream_t* stream)

Unmaps a buffer object for access by CUDA.

Deprecated:

This function is deprecated as of CUDA 3.0.

Unmaps the buffer object of ID `bufObj` for access by CUDA. When a buffer is unmapped, the base address returned by *cudaGLMapBufferObject()* is invalid and subsequent references to the address result in undefined behavior. The OpenGL context used to create the buffer, or another context from the same share group, must be bound to the current thread when this is called.

Stream /p stream is synchronized with the current GL context.

See also:

cudaGraphicsUnmapResources

Note: Note that this function may also return error codes from previous, asynchronous launches.

Parameters

- **`bufObj`** – - Buffer object to unmap
- **`stream`** – - Stream to synchronize

Returns
cudaSuccess, *cudaErrorUnmapBufferObjectFailed*

__host__ *cudaError_t* **cudaGLUnregisterBufferObject**(GLuint bufObj)
Unregisters a buffer object for access by CUDA.

Deprecated:
This function is deprecated as of CUDA 3.0.

Unregisters the buffer object of ID `bufObj` for access by CUDA and releases any CUDA resources associated with the buffer. Once a buffer is unregistered, it may no longer be mapped by CUDA. The GL context used to create the buffer, or another context from the same share group, must be bound to the current thread when this is called.

See also:
cudaGraphicsUnregisterResource

Note: Note that this function may also return error codes from previous, asynchronous launches.

Parameters
bufObj – - Buffer object to unregister

Returns
cudaSuccess

6.30. Peer Device Memory Access

This section describes the peer device memory access functions of the CUDA runtime application programming interface.

Functions

__host__ cudaError_t *cudaDeviceCanAccessPeer*(int *canAccessPeer, int device, int peerDevice)
Queries if a device may directly access a peer device's memory.

__host__ cudaError_t *cudaDeviceDisablePeerAccess*(int peerDevice)
Disables direct access to memory allocations on a peer device.

__host__ cudaError_t *cudaDeviceEnablePeerAccess*(int peerDevice, unsigned int flags)
Enables direct access to memory allocations on a peer device.

6.30.1. Functions

__host__ *cudaError_t* **cudaDeviceCanAccessPeer**(int *canAccessPeer, int device, int peerDevice)

Queries if a device may directly access a peer device's memory.

Returns in `*canAccessPeer` a value of 1 if device `device` is capable of directly accessing memory from `peerDevice` and 0 otherwise. If direct access of `peerDevice` from `device` is possible, then access may be enabled by calling *cudaDeviceEnablePeerAccess()*.

See also:

cudaDeviceEnablePeerAccess, *cudaDeviceDisablePeerAccess*, ::cuDeviceCanAccessPeer

---

Note: Note that this function may also return error codes from previous, asynchronous launches.

Note: Note that this function may also return *cudaErrorInitializationError*, *cudaErrorInsufficientDriver* or *cudaErrorNoDevice* if this call tries to initialize internal CUDA RT state.

Note: Note that as specified by *cudaStreamAddCallback* no CUDA function may be called from callback. *cudaErrorNotPermitted* may, but is not guaranteed to, be returned as a diagnostic in such case.

Parameters

- **canAccessPeer** – - Returned access capability
- **device** – - Device from which allocations on `peerDevice` are to be directly accessed.
- **peerDevice** – - Device on which the allocations to be directly accessed by `device` reside.

Returns

cudaSuccess, *cudaErrorInvalidDevice*

__host__ *cudaError_t* **cudaDeviceDisablePeerAccess**(int peerDevice)

Disables direct access to memory allocations on a peer device.

Returns *cudaErrorPeerAccessNotEnabled* if direct access to memory on `peerDevice` has not yet been enabled from the current device.

See also:

cudaDeviceCanAccessPeer, *cudaDeviceEnablePeerAccess*, ::cuCtxDisablePeerAccess

---

Note: Note that this function may also return error codes from previous, asynchronous launches.

Note: Note that this function may also return *cudaErrorInitializationError*, *cudaErrorInsufficientDriver* or *cudaErrorNoDevice* if this call tries to initialize internal CUDA RT state.

Note: Note that as specified by *cudaStreamAddCallback* no CUDA function may be called from callback. *cudaErrorNotPermitted* may, but is not guaranteed to, be returned as a diagnostic in such case.

Parameters

peerDevice – - Peer device to disable direct access to

Returns

cudaSuccess, *cudaErrorPeerAccessNotEnabled*, *cudaErrorInvalidDevice*

__host__ *cudaError_t* **cudaDeviceEnablePeerAccess**(int peerDevice, unsigned int flags)

Enables direct access to memory allocations on a peer device.

On success, all allocations from `peerDevice` will immediately be accessible by the current device. They will remain accessible until access is explicitly disabled using *cudaDeviceDisablePeerAccess()* or either device is reset using *cudaDeviceReset()*.

Note that access granted by this call is unidirectional and that in order to access memory on the current device from `peerDevice`, a separate symmetric call to *cudaDeviceEnablePeerAccess()* is required.

Note that there are both device-wide and system-wide limitations per system configuration, as noted in the CUDA Programming Guide under the section “Peer-to-Peer Memory Access”.

Returns *cudaErrorInvalidDevice* if *cudaDeviceCanAccessPeer()* indicates that the current device cannot directly access memory from `peerDevice`.

Returns *cudaErrorPeerAccessAlreadyEnabled* if direct access of `peerDevice` from the current device has already been enabled.

Returns *cudaErrorInvalidValue* if `flags` is not 0.

See also:

cudaDeviceCanAccessPeer, *cudaDeviceDisablePeerAccess*, ::cuCtxEnablePeerAccess

Note: Note that this function may also return error codes from previous, asynchronous launches.

Note: Note that this function may also return *cudaErrorInitializationError*, *cudaErrorInsufficientDriver* or *cudaErrorNoDevice* if this call tries to initialize internal CUDA RT state.

Note: Note that as specified by *cudaStreamAddCallback* no CUDA function may be called from callback. *cudaErrorNotPermitted* may, but is not guaranteed to, be returned as a diagnostic in such case.

Parameters

- **peerDevice** – - Peer device to enable direct access to from the current device
- **flags** – - Reserved for future use and must be set to 0

Returns

cudaSuccess, *cudaErrorInvalidDevice*, *cudaErrorPeerAccessAlreadyEnabled*, *cudaErrorInvalidValue*

6.31. Profiler Control

This section describes the profiler control functions of the CUDA runtime application programming interface.

Functions

__host__ cudaError_t *cudaProfilerStart*(void)
Enable profiling.

__host__ cudaError_t *cudaProfilerStop*(void)
Disable profiling.

6.31.1. Functions

__host__ *cudaError_t* **cudaProfilerStart**(void)

Enable profiling.

Enables profile collection by the active profiling tool for the current context. If profiling is already enabled, then *cudaProfilerStart()* has no effect.

cudaProfilerStart and cudaProfilerStop APIs are used to programmatically control the profiling granularity by allowing profiling to be done only on selective pieces of code.

See also:

cudaProfilerStop, ::cuProfilerStart

Note: Note that this function may also return error codes from previous, asynchronous launches.

Returns

cudaSuccess

__host__ *cudaError_t* **cudaProfilerStop**(void)

Disable profiling.

Disables profile collection by the active profiling tool for the current context. If profiling is already disabled, then *cudaProfilerStop()* has no effect.

cudaProfilerStart and cudaProfilerStop APIs are used to programmatically control the profiling granularity by allowing profiling to be done only on selective pieces of code.

See also:

cudaProfilerStart, ::cuProfilerStop

Note: Note that this function may also return error codes from previous, asynchronous launches.

Returns

cudaSuccess

6.32. Stream Management

This section describes the stream management functions of the CUDA runtime application programming interface.

Functions

__host__ cudaError_t *cudaCtxResetPersistingL2Cache*(void)
Resets all persisting lines in cache to normal status.

__host__ cudaError_t *cudaStreamAddCallback*(cudaStream_t stream, cudaStreamCallback_t callback, void *userData, unsigned int flags)
Add a callback to a compute stream.

__host__ cudaError_t *cudaStreamAttachMemAsync*(cudaStream_t stream, void *devPtr, size_t length=0, unsigned int flags=0x04)
Attach memory to a stream asynchronously.

__host__ cudaError_t *cudaStreamBeginCapture*(cudaStream_t stream, enum cudaStreamCaptureMode mode)
Begins graph capture on a stream.

__host__ cudaError_t *cudaStreamBeginCaptureToGraph*(cudaStream_t stream, cudaGraph_t graph, const cudaGraphNode_t *dependencies, const cudaGraphEdgeData *dependencyData, size_t numDependencies, enum cudaStreamCaptureMode mode)
Begins graph capture on a stream to an existing graph.

__host__ cudaError_t *cudaStreamBeginRecaptureToGraph*(cudaStream_t stream, enum cudaStreamCaptureMode mode, cudaGraph_t graph, struct cudaGraphRecaptureCallbackData *callbackData=0)
Begin graph capture on a stream to an existing graph.

__host__ cudaError_t *cudaStreamCopyAttributes*(cudaStream_t dst, cudaStream_t src)
Copies attributes from source stream to destination stream.

__host__ cudaError_t *cudaStreamCreate*(cudaStream_t *pStream)
Create an asynchronous stream.

__host__ cudaError_t *cudaStreamCreateWithFlags*(cudaStream_t *pStream, unsigned int flags)
Create an asynchronous stream.

__host__ cudaError_t *cudaStreamCreateWithPriority*(cudaStream_t *pStream, unsigned int flags, int priority)
Create an asynchronous stream with the specified priority.

__host__ cudaError_t *cudaStreamDestroy*(cudaStream_t stream)
Destroys and cleans up an asynchronous stream.

__host__ cudaError_t *cudaStreamEndCapture*(cudaStream_t stream, cudaGraph_t *pGraph)
Ends capture on a stream, returning the captured graph.

__host__ cudaError_t *cudaStreamGetAttribute*(cudaStream_t hStream, cudaLaunchAttributeID attr, cudaLaunchAttributeValue *value_out)
Queries stream attribute.

__host__ cudaError_t *cudaStreamGetCaptureInfo*(cudaStream_t stream, enum cudaStreamCaptureStatus *captureStatus_out, unsigned long long *id_out=0, cudaGraph_t *graph_out=0, const cudaGraphNode_t **dependencies_out=0, const cudaGraphEdgeData **edgeData_out=0, size_t *numDependencies_out=0)
Query a stream's capture state.

__host__ cudaError_t *cudaStreamGetDevice*(cudaStream_t hStream, int *device)
Query the device of a stream.

__host__ cudaError_t *cudaStreamGetFlags*(cudaStream_t hStream, unsigned int *flags)
Query the flags of a stream.

__host__ cudaError_t *cudaStreamGetId*(cudaStream_t hStream, unsigned long long *streamId)
Query the Id of a stream.

__host__ cudaError_t *cudaStreamGetPriority*(cudaStream_t hStream, int *priority)
Query the priority of a stream.

__host__ cudaError_t *cudaStreamIsCapturing*(cudaStream_t stream, enum cudaStreamCaptureStatus *pCaptureStatus)
Returns a stream's capture status.

__host__ cudaError_t *cudaStreamQuery*(cudaStream_t stream)
Queries an asynchronous stream for completion status.

__host__ cudaError_t *cudaStreamSetAttribute*(cudaStream_t hStream, cudaLaunchAttributeID attr, const cudaLaunchAttributeValue *value)
Sets stream attribute.

__host__ cudaError_t *cudaStreamSynchronize*(cudaStream_t stream)
Waits for stream tasks to complete.

__host__ cudaError_t *cudaStreamUpdateCaptureDependencies*(cudaStream_t stream, cudaGraphNode_t *dependencies, const cudaGraphEdgeData *dependencyData, size_t numDependencies, unsigned int flags=0)
Update the set of dependencies in a capturing stream.

__host__ cudaError_t *cudaStreamWaitEvent*(cudaStream_t stream, cudaEvent_t event, unsigned int flags=0)
Make a compute stream wait on an event.

__host__ cudaError_t *cudaThreadExchangeStreamCaptureMode*(enum cudaStreamCaptureMode *mode)
Swaps the stream capture interaction mode for a thread.

Structs

cudaGraphRecaptureCallbackData
Struct of user callback data that is invoked when node parameter mismatches are detected while recapturing to an existing graph.

Typedefs

cudaGraphRecaptureCallback_t
Callback function invoked when node parameter mismatches are detected while recapturing to an existing graph.

cudaStreamCallback_t
Type of stream callback functions.

6.32.1. Functions

__host__ *cudaError_t* **cudaCtxResetPersistingL2Cache**(void)

Resets all persisting lines in cache to normal status.

Resets all persisting lines in cache to normal status. Takes effect on function return.

See also:

cudaAccessPolicyWindow

Note: Note that this function may also return error codes from previous, asynchronous launches.

Returns
cudaSuccess,

__host__ *cudaError_t* **cudaStreamAddCallback**(*cudaStream_t* stream, cudaStreamCallback_t callback, void *userData, unsigned int flags)

Add a callback to a compute stream.

Adds a callback to be called on the host after all currently enqueued items in the stream have completed. For each cudaStreamAddCallback call, a callback will be executed exactly once. The callback will block later work in the stream until it is finished.

The callback may be passed *cudaSuccess* or an error code. In the event of a device error, all subsequently executed callbacks will receive an appropriate *cudaError_t*.

Callbacks must not make any CUDA API calls. Attempting to use CUDA APIs may result in *cudaErrorNotPermitted*. Callbacks must not perform any synchronization that may depend on outstanding device work or other callbacks that are not mandated to run earlier. Callbacks without a mandated order (in independent streams) execute in undefined order and may be serialized.

For the purposes of Unified Memory, callback execution makes a number of guarantees:

- The callback stream is considered idle for the duration of the callback. Thus, for example, a callback may always use memory attached to the callback stream.
- The start of execution of a callback has the same effect as synchronizing an event recorded in the same stream immediately prior to the callback. It thus synchronizes streams which have been "joined" prior to the callback.
- Adding device work to any stream does not have the effect of making the stream active until all preceding callbacks have executed. Thus, for example, a callback might use global attached memory even if work has been added to another stream, if it has been properly ordered with an event.
- Completion of a callback does not cause a stream to become active except as described above. The callback stream will remain idle if no device work follows the callback, and will remain idle across consecutive callbacks without device work in between. Thus, for example, stream synchronization can be done by signaling from a callback at the end of the stream.

See also:

cudaStreamCreate, *cudaStreamCreateWithFlags*, *cudaStreamQuery*, *cudaStreamSynchronize*, *cudaStreamWaitEvent*, *cudaStreamDestroy*, *cudaMallocManaged*, *cudaStreamAttachMemAsync*, *cudaLaunchHostFunc*, ::cuStreamAddCallback

---

Note: This function is slated for eventual deprecation and removal. If you do not require the callback to execute in case of a device error, consider using *cudaLaunchHostFunc*. Additionally, this function is not supported with *cudaStreamBeginCapture* and *cudaStreamEndCapture*, unlike *cudaLaunchHostFunc*.

Note: This function uses standard default stream semantics.

Note: Note that this function may also return error codes from previous, asynchronous launches.

Note: Note that this function may also return *cudaErrorInitializationError*, *cudaErrorInsufficientDriver* or *cudaErrorNoDevice* if this call tries to initialize internal CUDA RT state.

Note: Note that as specified by *cudaStreamAddCallback* no CUDA function may be called from callback. *cudaErrorNotPermitted* may, but is not guaranteed to, be returned as a diagnostic in such case.

Parameters

- **`stream`** – - Stream to add callback to
- **`callback`** – - The function to call once preceding stream operations are complete
- **`userData`** – - User specified data to be passed to the callback function

- **flags** – - Reserved for future use, must be 0

Returns

cudaSuccess, *cudaErrorInvalidResourceHandle*, *cudaErrorInvalidValue*, *cudaErrorNotSupported*

__host__ *cudaError_t* **cudaStreamAttachMemAsync**(*cudaStream_t* stream, void *devPtr, size_t length = 0, unsigned int flags = 0x04)

Attach memory to a stream asynchronously.

Enqueues an operation in `stream` to specify stream association of `length` bytes of memory starting from `devPtr`. This function is a stream-ordered operation, meaning that it is dependent on, and will only take effect when, previous work in stream has completed. Any previous association is automatically replaced.

`devPtr` must point to an one of the following types of memories:

- managed memory declared using the **managed** keyword or allocated with *cudaMallocManaged*.
- a valid host-accessible region of system-allocated pageable memory. This type of memory may only be specified if the device associated with the stream reports a non-zero value for the device attribute *cudaDevAttrPageableMemoryAccess*.

For managed allocations, `length` must be either zero or the entire allocation's size. Both indicate that the entire allocation's stream association is being changed. Currently, it is not possible to change stream association for a portion of a managed allocation.

For pageable allocations, `length` must be non-zero.

The stream association is specified using `flags` which must be one of *cudaMemAttachGlobal*, *cudaMemAttachHost* or *cudaMemAttachSingle*. The default value for `flags` is *cudaMemAttachSingle* If the *cudaMemAttachGlobal* flag is specified, the memory can be accessed by any stream on any device. If the *cudaMemAttachHost* flag is specified, the program makes a guarantee that it won't access the memory on the device from any stream on a device that has a zero value for the device attribute *cudaDevAttrConcurrentManagedAccess*. If the *cudaMemAttachSingle* flag is specified and `stream` is associated with a device that has a zero value for the device attribute *cudaDevAttrConcurrentManagedAccess*, the program makes a guarantee that it will only access the memory on the device from `stream`. It is illegal to attach singly to the NULL stream, because the NULL stream is a virtual global stream and not a specific stream. An error will be returned in this case.

When memory is associated with a single stream, the Unified Memory system will allow CPU access to this memory region so long as all operations in `stream` have completed, regardless of whether other streams are active. In effect, this constrains exclusive ownership of the managed memory region by an active GPU to per-stream activity instead of whole-GPU activity.

Accessing memory on the device from streams that are not associated with it will produce undefined results. No error checking is performed by the Unified Memory system to ensure that kernels launched into other streams do not access this region.

It is a program's responsibility to order calls to *cudaStreamAttachMemAsync* via events, synchronization or other means to ensure legal access to memory at all times. Data visibility and coherency will be changed appropriately for all kernels which follow a stream-association change.

If `stream` is destroyed while data is associated with it, the association is removed and the association reverts to the default visibility of the allocation as specified at *cudaMallocManaged*. For **managed** variables, the default association is always *cudaMemAttachGlobal*. Note that destroying a stream is an asynchronous operation, and as a result, the change to default association won't happen until all work in the stream has completed.

See also:

cudaStreamCreate, *cudaStreamCreateWithFlags*, *cudaStreamWaitEvent*, *cudaStreamSynchronize*, *cudaStreamAddCallback*, *cudaStreamDestroy*, *cudaMallocManaged*, ::cuStreamAttachMemAsync

---

Note: Note that this function may also return error codes from previous, asynchronous launches.

Note: Note that this function may also return *cudaErrorInitializationError*, *cudaErrorInsufficientDriver* or *cudaErrorNoDevice* if this call tries to initialize internal CUDA RT state.

Note: Note that as specified by *cudaStreamAddCallback* no CUDA function may be called from callback. *cudaErrorNotPermitted* may, but is not guaranteed to, be returned as a diagnostic in such case.

Parameters

- **flags** – Memory can only be accessed by a single stream on the associated device
- **stream** – - Stream in which to enqueue the attach operation
- **devPtr** – - Pointer to memory (must be a pointer to managed memory or to a valid host-accessible region of system-allocated memory)
- **length** – - Length of memory (defaults to zero)
- **flags** – - Must be one of *cudaMemAttachGlobal*, *cudaMemAttachHost* or *cudaMemAttachSingle* (defaults to *cudaMemAttachSingle*)

Returns

cudaSuccess, *cudaErrorNotReady*, *cudaErrorInvalidValue*, *cudaErrorInvalidResourceHandle*

__host__ *cudaError_t* **cudaStreamBeginCapture**(*cudaStream_t* stream, enum *cudaStreamCaptureMode* mode)

Begins graph capture on a stream.

Begin graph capture on `stream`. When a stream is in capture mode, all operations pushed into the stream will not be executed, but will instead be captured into a graph, which will be returned via *cudaStreamEndCapture*. Capture may not be initiated if `stream` is *cudaStreamLegacy*. Capture must be ended on the same stream in which it was initiated, and it may only be initiated if the stream is not already in capture mode. The capture mode may be queried via *cudaStreamIsCapturing*. A unique id representing the capture sequence may be queried via *cudaStreamGetCaptureInfo*.

If `mode` is not cudaStreamCaptureModeRelaxed, *cudaStreamEndCapture* must be called on this stream from the same thread.

See also:

cudaStreamCreate, *cudaStreamIsCapturing*, *cudaStreamEndCapture*, *cudaThreadExchangeStreamCaptureMode*

Note: Kernels captured using this API must not use texture and surface references. Reading or writing through any texture or surface reference is undefined behavior. This restriction does not apply to texture and surface objects.

Note: Note that this function may also return error codes from previous, asynchronous launches.

Parameters

- **stream** – - Stream in which to initiate capture
- **mode** – - Controls the interaction of this capture sequence with other API calls that are potentially unsafe. For more details see *cudaThreadExchangeStreamCaptureMode*.

Returns

cudaSuccess, *cudaErrorInvalidValue*

__host__ *cudaError_t* **cudaStreamBeginCaptureToGraph**(*cudaStream_t* stream, *cudaGraph_t* graph, const *cudaGraphNode_t* *dependencies, const *cudaGraphEdgeData* *dependencyData, size_t numDependencies, enum *cudaStreamCaptureMode* mode)

Begins graph capture on a stream to an existing graph.

Begin graph capture on `stream`. When a stream is in capture mode, all operations pushed into the stream will not be executed, but will instead be captured into `graph`, which will be returned via *cudaStreamEndCapture*.

Capture may not be initiated if `stream` is *cudaStreamLegacy*. Capture must be ended on the same stream in which it was initiated, and it may only be initiated if the stream is not already in capture mode. The capture mode may be queried via *cudaStreamIsCapturing*. A unique id representing the capture sequence may be queried via *cudaStreamGetCaptureInfo*.

If `mode` is not cudaStreamCaptureModeRelaxed, *cudaStreamEndCapture* must be called on this stream from the same thread.

See also:

cudaStreamCreate, *cudaStreamIsCapturing*, *cudaStreamEndCapture*, *cudaThreadExchangeStreamCaptureMode*

Note: Kernels captured using this API must not use texture and surface references. Reading or writing through any texture or surface reference is undefined behavior. This restriction does not apply to texture and surface objects.

Note: Note that this function may also return error codes from previous, asynchronous launches.

Parameters

- **stream** – - Stream in which to initiate capture.
- **graph** – - Graph to capture into.
- **dependencies** – - Dependencies of the first node captured in the stream. Can be NULL if numDependencies is 0.
- **dependencyData** – - Optional array of data associated with each dependency.
- **numDependencies** – - Number of dependencies.
- **mode** – - Controls the interaction of this capture sequence with other API calls that are potentially unsafe. For more details see *cudaThreadExchangeStreamCaptureMode*.

Returns

cudaSuccess, *cudaErrorInvalidValue*

__host__ *cudaError_t* **cudaStreamBeginRecaptureToGraph**(*cudaStream_t* stream, enum *cudaStreamCaptureMode* mode, *cudaGraph_t* graph, struct *cudaGraphRecaptureCallbackData* *callbackData = 0)

Begin graph capture on a stream to an existing graph.

Begin graph capture on `stream` to the existing `graph`. The node creation order while recapturing the graph must be identical to the original graph. The recapture will fail immediately for:

- Topology mismatches between the existing graph and the recaptured graph
- Parameter mismatches for memory allocation or free nodes

Any other node parameter mismatches during recapture can be configured to call the function provided in `callbackFunc`. The recapture will fail immediately if the callback returns anything other than cudaSuccess.

If the recapture fails for any reason, the `graph` will be in an undefined state and should be destroyed.

See cudaStreamBeginCapture for additional detail on beginning the capture.

See also:

cudaStreamCreate, *cudaStreamBeginCapture*, *cudaStreamIsCapturing*, *cudaStreamEndCapture*, *cudaThreadExchangeStreamCaptureMode*

Note: Any user objects associated with `graph` will be released prior to the recapture.

Note: Note that this function may also return error codes from previous, asynchronous launches.

Parameters

- **stream** – - Stream in which to initiate capture
- **mode** – - Controls the interaction of this capture sequence with other API calls that are potentially unsafe. For more details see *cudaThreadExchangeStreamCaptureMode*.
- **graph** – - Existing CUDA graph to be captured into
- **callbackData** – - Optional struct of callback data that will be invoked for all parameter mismatches from the original graph

Returns

cudaSuccess, ::cudaErrorDeinitialized, ::cudaErrorNotInitialized, *cudaErrorInvalidValue*,

__host__ *cudaError_t* **cudaStreamCopyAttributes**(*cudaStream_t* dst, *cudaStream_t* src)

Copies attributes from source stream to destination stream.

Copies attributes from source stream `src` to destination stream `dst`. Both streams must have the same context.

See also:

cudaAccessPolicyWindow

Note: Note that this function may also return error codes from previous, asynchronous launches.

Parameters

- **dst** – **[out]** Destination stream
- **src** – **[in]** Source stream For attributes see cudaStreamAttrID

Returns

cudaSuccess, *cudaErrorNotSupported*

__host__ *cudaError_t* **cudaStreamCreate**(*cudaStream_t* *pStream)

Create an asynchronous stream.

Creates a new asynchronous stream on the context that is current to the calling host thread. If no context is current to the calling host thread, then the primary context for a device is selected, made current to the calling thread, and initialized before creating a stream on it.

See also:

cudaStreamCreateWithPriority, *cudaStreamCreateWithFlags*, *cudaStreamGetPriority*, *cudaStreamGetFlags*, *cudaStreamGetDevice*, *cudaStreamGetDevResource*, *cudaStreamQuery*, *cudaStreamSynchronize*, *cudaStreamWaitEvent*, *cudaStreamAddCallback*, *cudaSetDevice*, *cudaStreamDestroy*, ::cuStreamCreate

Note: Note that this function may also return error codes from previous, asynchronous launches.

Note: Note that this function may also return *cudaErrorInitializationError*, *cudaErrorInsufficientDriver* or *cudaErrorNoDevice* if this call tries to initialize internal CUDA RT state.

Note: Note that as specified by *cudaStreamAddCallback* no CUDA function may be called from callback. *cudaErrorNotPermitted* may, but is not guaranteed to, be returned as a diagnostic in such case.

Parameters

pStream – - Pointer to new stream identifier

Returns

cudaSuccess, *cudaErrorInvalidValue* *cudaErrorExternalDevice*

__host__ *cudaError_t* **cudaStreamCreateWithFlags**(*cudaStream_t* *pStream, unsigned int flags)

Create an asynchronous stream.

Creates a new asynchronous stream on the context that is current to the calling host thread. If no context is current to the calling host thread, then the primary context for a device is selected, made current to the calling thread, and initialized before creating a stream on it. The `flags` argument determines the behaviors of the stream. Valid values for `flags` are

- *cudaStreamDefault*: Default stream creation flag.
- *cudaStreamNonBlocking*: Specifies that work running in the created stream may run concurrently with work in stream 0 (the NULL stream), and that the created stream should perform no implicit synchronization with stream 0.

See also:

cudaStreamCreate, *cudaStreamCreateWithPriority*, *cudaStreamGetFlags*, *cudaStreamGetDevice*, *cudaStreamGetDevResource*, *cudaStreamQuery*, *cudaStreamSynchronize*, *cudaStreamWaitEvent*, *cudaStreamAddCallback*, *cudaSetDevice*, *cudaStreamDestroy*, ::cuStreamCreate

Note: Note that this function may also return error codes from previous, asynchronous launches.

Note: Note that this function may also return *cudaErrorInitializationError*, *cudaErrorInsufficientDriver* or *cudaErrorNoDevice* if this call tries to initialize internal CUDA RT state.

Note: Note that as specified by *cudaStreamAddCallback* no CUDA function may be called from callback. *cudaErrorNotPermitted* may, but is not guaranteed to, be returned as a diagnostic in such case.

Parameters

- **pStream** – - Pointer to new stream identifier
- **flags** – - Parameters for stream creation

Returns

cudaSuccess, *cudaErrorInvalidValue* *cudaErrorExternalDevice*

__host__ *cudaError_t* **cudaStreamCreateWithPriority**(*cudaStream_t* *pStream, unsigned int flags, int priority)

Create an asynchronous stream with the specified priority.

Creates a stream with the specified priority and returns a handle in `pStream`. The stream is created on the context that is current to the calling host thread. If no context is current to the calling host thread, then the primary context for a device is selected, made current to the calling thread, and initialized before creating a stream on it. This affects the scheduling priority of work in the stream. Priorities provide a hint to preferentially run work with higher priority when possible, but do not preempt already-running work or provide any other functional guarantee on execution order.

`priority` follows a convention where lower numbers represent higher priorities. '0' represents default priority. The range of meaningful numerical priorities can be queried using *cudaDeviceGetStreamPriorityRange*. If the specified priority is outside the numerical range returned by *cudaDeviceGetStreamPriorityRange*, it will automatically be clamped to the lowest or the highest number in the range.

See also:

cudaStreamCreate, *cudaStreamCreateWithFlags*, *cudaDeviceGetStreamPriorityRange*, *cudaStreamGetPriority*, *cudaStreamQuery*, *cudaStreamWaitEvent*, *cudaStreamAddCallback*, *cudaStreamSynchronize*, *cudaSetDevice*, *cudaStreamDestroy*, ::cuStreamCreateWithPriority

---

Note: Note that this function may also return error codes from previous, asynchronous launches.

Note: Note that this function may also return *cudaErrorInitializationError*, *cudaErrorInsufficientDriver* or *cudaErrorNoDevice* if this call tries to initialize internal CUDA RT state.

Note: Note that as specified by *cudaStreamAddCallback* no CUDA function may be called from callback. *cudaErrorNotPermitted* may, but is not guaranteed to, be returned as a diagnostic in such case.

Note: Stream priorities are supported only on GPUs with compute capability 3.5 or higher.

Note: In the current implementation, only compute kernels launched in priority streams are affected by the stream's priority. Stream priorities have no effect on host-to-device and device-to-host memory operations.

Parameters

- **pStream** – - Pointer to new stream identifier

- **flags** – - Flags for stream creation. See *cudaStreamCreateWithFlags* for a list of valid flags that can be passed
- **priority** – - Priority of the stream. Lower numbers represent higher priorities. See *cudaDeviceGetStreamPriorityRange* for more information about the meaningful stream priorities that can be passed.

Returns

cudaSuccess, *cudaErrorInvalidValue* *cudaErrorExternalDevice*

__host__ *cudaError_t* **cudaStreamDestroy**(*cudaStream_t* stream)

Destroys and cleans up an asynchronous stream.

Destroys and cleans up the asynchronous stream specified by `stream`.

In case the device is still doing work in the stream `stream` when *cudaStreamDestroy()* is called, the function will return immediately and the resources associated with `stream` will be released automatically once the device has completed all work in `stream`.

See also:

cudaStreamCreate, *cudaStreamCreateWithFlags*, *cudaStreamQuery*, *cudaStreamWaitEvent*, *cudaStreamSynchronize*, *cudaStreamAddCallback*, ::cuStreamDestroy

---

Note: This function uses standard default stream semantics.

Note: Note that this function may also return error codes from previous, asynchronous launches.

Note: Note that this function may also return *cudaErrorInitializationError*, *cudaErrorInsufficientDriver* or *cudaErrorNoDevice* if this call tries to initialize internal CUDA RT state.

Note: Note that as specified by *cudaStreamAddCallback* no CUDA function may be called from callback. *cudaErrorNotPermitted* may, but is not guaranteed to, be returned as a diagnostic in such case.

Note: Use of the handle after this call is undefined behavior.

Parameters

stream – - Stream identifier

Returns

cudaSuccess, *cudaErrorInvalidValue*, *cudaErrorInvalidResourceHandle*

__host__ *cudaError_t* **cudaStreamEndCapture**(*cudaStream_t* stream, *cudaGraph_t* *pGraph)

Ends capture on a stream, returning the captured graph.

End capture on `stream`, returning the captured graph via `pGraph`. Capture must have been initiated on `stream` via a call to *cudaStreamBeginCapture*. If capture was invalidated, due to a violation of the rules of stream capture, then a NULL graph will be returned.

If the `mode` argument to *cudaStreamBeginCapture* was not cudaStreamCaptureModeRelaxed, this call must be from the same thread as *cudaStreamBeginCapture*.

See also:

cudaStreamCreate, *cudaStreamBeginCapture*, *cudaStreamIsCapturing*, *cudaGraphDestroy*

Note: Note that this function may also return error codes from previous, asynchronous launches.

Parameters

- **`stream`** – - Stream to query
- **`pGraph`** – - The captured graph

Returns

cudaSuccess, *cudaErrorInvalidValue*, *cudaErrorStreamCaptureWrongThread*

__host__ *cudaError_t* **`cudaStreamGetAttribute`**(*cudaStream_t* hStream, *cudaLaunchAttributeID* attr, cudaLaunchAttributeValue *value_out)

Queries stream attribute.

Queries attribute `attr` from `hStream` and stores it in corresponding member of `value_out`.

See also:

cudaAccessPolicyWindow

Note: Note that this function may also return error codes from previous, asynchronous launches.

Parameters

- **`hStream`** – **[in]**
- **`attr`** – **[in]**
- **`value_out`** –

Returns

cudaSuccess, *cudaErrorInvalidValue*, *cudaErrorInvalidResourceHandle*

__host__ *cudaError_t* **`cudaStreamGetCaptureInfo`**(*cudaStream_t* stream, enum *cudaStreamCaptureStatus* *captureStatus_out, unsigned long long *id_out = 0, *cudaGraph_t* *graph_out = 0, const *cudaGraphNode_t* **dependencies_out = 0, const *cudaGraphEdgeData* **edgeData_out = 0, size_t *numDependencies_out = 0)

Query a stream's capture state.

Query stream state related to stream capture.

If called on *cudaStreamLegacy* (the "null stream") while a stream not created with *cudaStreamNonBlocking* is capturing, returns *cudaErrorStreamCaptureImplicit*.

Valid data (other than capture status) is returned only if both of the following are true:

- the call returns cudaSuccess
- the returned capture status is *cudaStreamCaptureStatusActive*

If `edgeData_out` is non-NULL then `dependencies_out` must be as well. If `dependencies_out` is non-NULL and `edgeData_out` is NULL, but there is non-zero edge data for one or more of the current stream dependencies, the call will return *cudaErrorLossyQuery*.

See also:

cudaStreamBeginCapture, *cudaStreamIsCapturing*, *cudaStreamUpdateCaptureDependencies*

Note: Graph objects are not threadsafe. More here.

Note: Note that this function may also return error codes from previous, asynchronous launches.

Parameters

- **`stream`** – - The stream to query
- **`captureStatus_out`** – - Location to return the capture status of the stream; required
- **`id_out`** – - Optional location to return an id for the capture sequence, which is unique over the lifetime of the process
- **`graph_out`** – - Optional location to return the graph being captured into. All operations other than destroy and node removal are permitted on the graph while the capture sequence is in progress. This API does not transfer ownership of the graph, which is transferred or destroyed at *cudaStreamEndCapture*. Note that the graph handle may be invalidated before end of capture for certain errors. Nodes that are or become unreachable from the original stream at *cudaStreamEndCapture* due to direct actions on the graph do not trigger *cudaErrorStreamCaptureUnjoined*.
- **`dependencies_out`** – - Optional location to store a pointer to an array of nodes. The next node to be captured in the stream will depend on this set of nodes, absent operations such as event wait which modify this set. The array pointer is valid until the next API call which operates on the stream or until the capture is terminated. The node handles may be copied out and are valid until they or the graph is destroyed. The driver-owned array may also be passed directly to APIs that operate on the graph (not the stream) without copying.
- **`edgeData_out`** – - Optional location to store a pointer to an array of graph edge data. This array parallels `dependencies_out`; the next node to be added has an edge to `dependencies_out`[i] with annotation `edgeData_out`[i] for each `i`. The array pointer is valid until the next API call which operates on the stream or until the capture is terminated.

- **numDependencies_out** – - Optional location to store the size of the array returned in dependencies_out.

Returns

cudaSuccess, *cudaErrorInvalidValue*, *cudaErrorStreamCaptureImplicit*, *cudaErrorLossyQuery*

__host__ *cudaError_t* **cudaStreamGetDevice**(*cudaStream_t* hStream, int *device)

Query the device of a stream.

Returns in `*device` the device of the stream.

See also:

cudaSetDevice, *cudaGetDevice*, *cudaStreamCreate*, *cudaStreamGetPriority*, *cudaStreamGetFlags*, ::cuStreamGetId

---

Note: This function uses standard default stream semantics.

Note: Note that this function may also return error codes from previous, asynchronous launches.

Note: Note that this function may also return *cudaErrorInitializationError*, *cudaErrorInsufficientDriver* or *cudaErrorNoDevice* if this call tries to initialize internal CUDA RT state.

Note: Note that as specified by *cudaStreamAddCallback* no CUDA function may be called from callback. *cudaErrorNotPermitted* may, but is not guaranteed to, be returned as a diagnostic in such case.

Parameters

- **hStream** – - Handle to the stream to be queried
- **device** – - Returns the device to which the stream belongs

Returns

cudaSuccess, *cudaErrorInvalidValue*, ::cudaErrorDeviceUnavailable,

__host__ *cudaError_t* **cudaStreamGetFlags**(*cudaStream_t* hStream, unsigned int *flags)

Query the flags of a stream.

Query the flags of a stream. The flags are returned in `flags`. See *cudaStreamCreateWithFlags* for a list of valid flags.

See also:

cudaStreamCreateWithPriority, *cudaStreamCreateWithFlags*, *cudaStreamGetPriority*, *cudaStreamGetDevice*, ::cuStreamGetFlags

Note: This function uses standard default stream semantics.

Note: Note that this function may also return error codes from previous, asynchronous launches.

Note: Note that this function may also return *cudaErrorInitializationError*, *cudaErrorInsufficientDriver* or *cudaErrorNoDevice* if this call tries to initialize internal CUDA RT state.

Note: Note that as specified by *cudaStreamAddCallback* no CUDA function may be called from callback. *cudaErrorNotPermitted* may, but is not guaranteed to, be returned as a diagnostic in such case.

Parameters

- **hStream** – - Handle to the stream to be queried
- **flags** – - Pointer to an unsigned integer in which the stream's flags are returned

Returns

cudaSuccess, *cudaErrorInvalidValue*, *cudaErrorInvalidResourceHandle*

__host__ *cudaError_t* **cudaStreamGetId**(*cudaStream_t* hStream, unsigned long long *streamId)

Query the Id of a stream.

Query the Id of a stream. The Id is returned in `streamId`. The Id is unique for the life of the program.

The stream handle `hStream` can refer to any of the following:

- a stream created via any of the CUDA runtime APIs such as *cudaStreamCreate*, *cudaStreamCreateWithFlags* and *cudaStreamCreateWithPriority*, or their driver API equivalents such as ::cuStreamCreate or ::cuStreamCreateWithPriority. Passing an invalid handle will result in undefined behavior.
- any of the special streams such as the NULL stream, *cudaStreamLegacy* and *cudaStreamPerThread* respectively. The driver API equivalents of these are also accepted which are NULL, ::CU_STREAM_LEGACY and ::CU_STREAM_PER_THREAD.

See also:

cudaStreamCreateWithPriority, *cudaStreamCreateWithFlags*, *cudaStreamGetPriority*, *cudaStreamGetFlags*, ::cuStreamGetId

Note: This function uses standard default stream semantics.

Note: Note that this function may also return error codes from previous, asynchronous launches.

Note: Note that this function may also return *cudaErrorInitializationError*, *cudaErrorInsufficientDriver* or *cudaErrorNoDevice* if this call tries to initialize internal CUDA RT state.

Note: Note that as specified by *cudaStreamAddCallback* no CUDA function may be called from callback. *cudaErrorNotPermitted* may, but is not guaranteed to, be returned as a diagnostic in such case.

Parameters

- **hStream** – - Handle to the stream to be queried
- **streamId** – - Pointer to an unsigned long long in which the stream Id is returned

Returns

cudaSuccess, *cudaErrorInvalidValue*, *cudaErrorInvalidResourceHandle*

__host__ *cudaError_t* **cudaStreamGetPriority**(*cudaStream_t* hStream, int *priority)

Query the priority of a stream.

Query the priority of a stream. The priority is returned in in `priority`. Note that if the stream was created with a priority outside the meaningful numerical range returned by *cudaDeviceGetStreamPriorityRange*, this function returns the clamped priority. See *cudaStreamCreateWithPriority* for details about priority clamping.

See also:

cudaStreamCreateWithPriority, *cudaDeviceGetStreamPriorityRange*, *cudaStreamGetFlags*, *cudaStreamGetDevice*, *cudaStreamGetDevResource*, ::cuStreamGetPriority

Note: Note that this function may also return error codes from previous, asynchronous launches.

Note: Note that this function may also return *cudaErrorInitializationError*, *cudaErrorInsufficientDriver* or *cudaErrorNoDevice* if this call tries to initialize internal CUDA RT state.

Note: Note that as specified by *cudaStreamAddCallback* no CUDA function may be called from callback. *cudaErrorNotPermitted* may, but is not guaranteed to, be returned as a diagnostic in such case.

Parameters

- **hStream** – - Handle to the stream to be queried
- **priority** – - Pointer to a signed integer in which the stream's priority is returned

Returns

cudaSuccess, *cudaErrorInvalidValue*, *cudaErrorInvalidResourceHandle*

__host__ *cudaError_t* **cudaStreamIsCapturing**(*cudaStream_t* stream, enum *cudaStreamCaptureStatus* *pCaptureStatus)

Returns a stream's capture status.

Return the capture status of `stream` via `pCaptureStatus`. After a successful call, `*pCaptureStatus` will contain one of the following:

- *cudaStreamCaptureStatusNone*: The stream is not capturing.
- *cudaStreamCaptureStatusActive*: The stream is capturing.
- *cudaStreamCaptureStatusInvalidated*: The stream was capturing but an error has invalidated the capture sequence. The capture sequence must be terminated with *cudaStreamEndCapture* on the stream where it was initiated in order to continue using `stream`.

Note that, if this is called on *cudaStreamLegacy* (the "null stream") while a blocking stream on the same device is capturing, it will return *cudaErrorStreamCaptureImplicit* and `*pCaptureStatus` is unspecified after the call. The blocking stream capture is not invalidated.

When a blocking stream is capturing, the legacy stream is in an unusable state until the blocking stream capture is terminated. The legacy stream is not supported for stream capture, but attempted use would have an implicit dependency on the capturing stream(s).

See also:

cudaStreamCreate, *cudaStreamBeginCapture*, *cudaStreamEndCapture*

Note: Note that this function may also return error codes from previous, asynchronous launches.

Parameters

- **stream** – - Stream to query
- **pCaptureStatus** – - Returns the stream's capture status

Returns

cudaSuccess, *cudaErrorInvalidValue*, *cudaErrorStreamCaptureImplicit*

__host__ *cudaError_t* **cudaStreamQuery**(*cudaStream_t* stream)

Queries an asynchronous stream for completion status.

Returns *cudaSuccess* if all operations in `stream` have completed, or *cudaErrorNotReady* if not.

For the purposes of Unified Memory, a return value of *cudaSuccess* is equivalent to having called *cudaStreamSynchronize()*.

See also:

cudaStreamCreate, *cudaStreamCreateWithFlags*, *cudaStreamWaitEvent*, *cudaStreamSynchronize*, *cudaStreamAddCallback*, *cudaStreamDestroy*, ::cuStreamQuery

---

Note: This function uses standard default stream semantics.

Note: Note that this function may also return error codes from previous, asynchronous launches.

Note: Note that this function may also return *cudaErrorInitializationError*, *cudaErrorInsufficientDriver* or *cudaErrorNoDevice* if this call tries to initialize internal CUDA RT state.

Note: Note that as specified by *cudaStreamAddCallback* no CUDA function may be called from callback. *cudaErrorNotPermitted* may, but is not guaranteed to, be returned as a diagnostic in such case.

Parameters

stream – - Stream identifier

Returns

cudaSuccess, *cudaErrorNotReady*, *cudaErrorInvalidResourceHandle*

__host__ *cudaError_t* **cudaStreamSetAttribute**(*cudaStream_t* hStream, *cudaLaunchAttributeID* attr, const cudaLaunchAttributeValue *value)

Sets stream attribute.

Sets attribute `attr` on `hStream` from corresponding attribute of `value`. The updated attribute will be applied to subsequent work submitted to the stream. It will not affect previously submitted work.

See also:

cudaAccessPolicyWindow

Note: Note that this function may also return error codes from previous, asynchronous launches.

Parameters

- **hStream** – **[out]**
- **attr** – **[in]**
- **value** –

Returns

cudaSuccess, *cudaErrorInvalidValue*, *cudaErrorInvalidResourceHandle*

__host__ *cudaError_t* **cudaStreamSynchronize**(*cudaStream_t* stream)

Waits for stream tasks to complete.

Blocks until `stream` has completed all operations. If the *cudaDeviceScheduleBlockingSync* flag was set for this device, the host thread will block until the stream is finished with all of its tasks.

See also:

cudaStreamCreate, *cudaStreamCreateWithFlags*, *cudaStreamQuery*, *cudaStreamWaitEvent*, *cudaStreamAddCallback*, *cudaStreamDestroy*, ::cuStreamSynchronize

---

Note: This function uses standard default stream semantics.

Note: Note that this function may also return error codes from previous, asynchronous launches.

Note: Note that this function may also return *cudaErrorInitializationError*, *cudaErrorInsufficientDriver* or *cudaErrorNoDevice* if this call tries to initialize internal CUDA RT state.

Note: Note that as specified by *cudaStreamAddCallback* no CUDA function may be called from callback. *cudaErrorNotPermitted* may, but is not guaranteed to, be returned as a diagnostic in such case.

Parameters

`stream` – - Stream identifier

Returns

cudaSuccess, *cudaErrorInvalidResourceHandle*

__host__ *cudaError_t* **`cudaStreamUpdateCaptureDependencies`**(*cudaStream_t* stream, *cudaGraphNode_t* *dependencies, const *cudaGraphEdgeData* *dependencyData, size_t numDependencies, unsigned int flags = 0)

Update the set of dependencies in a capturing stream.

Modifies the dependency set of a capturing stream. The dependency set is the set of nodes that the next captured node in the stream will depend on.

Valid flags are *cudaStreamAddCaptureDependencies* and *cudaStreamSetCaptureDependencies*. These control whether the set passed to the API is added to the existing set or replaces it. A flags value of 0 defaults to *cudaStreamAddCaptureDependencies*.

Nodes that are removed from the dependency set via this API do not result in *cudaErrorStreamCaptureUnjoined* if they are unreachable from the stream at *cudaStreamEndCapture*.

Returns *cudaErrorIllegalState* if the stream is not capturing.

See also:

cudaStreamBeginCapture, *cudaStreamGetCaptureInfo*,

Note: Note that this function may also return error codes from previous, asynchronous launches.

Parameters

- **stream** – - The stream to update
- **dependencies** – - The set of dependencies to add
- **dependencyData** – - Optional array of data associated with each dependency.
- **numDependencies** – - The size of the dependencies array
- **flags** – - See above

Returns

cudaSuccess, *cudaErrorInvalidValue*, *cudaErrorIllegalState*

__host__ *cudaError_t* **cudaStreamWaitEvent**(*cudaStream_t* stream, *cudaEvent_t* event, unsigned int flags = 0)

Make a compute stream wait on an event.

Makes all future work submitted to `stream` wait for all work captured in `event`. See *cudaEventRecord()* for details on what is captured by an event. The synchronization will be performed efficiently on the device when applicable. `event` may be from a different device than `stream`.

flags include:

- *cudaEventWaitDefault*: Default event creation flag.
- *cudaEventWaitExternal*: Event is captured in the graph as an external event node when performing stream capture.

See also:

cudaStreamCreate, *cudaStreamCreateWithFlags*, *cudaStreamQuery*, *cudaStreamSynchronize*, *cudaStreamAddCallback*, *cudaStreamDestroy*, ::cuStreamWaitEvent

---

Note: This function uses standard default stream semantics.

Note: Note that this function may also return error codes from previous, asynchronous launches.

Note: Note that this function may also return *cudaErrorInitializationError*, *cudaErrorInsufficientDriver* or *cudaErrorNoDevice* if this call tries to initialize internal CUDA RT state.

Note: Note that as specified by *cudaStreamAddCallback* no CUDA function may be called from callback. *cudaErrorNotPermitted* may, but is not guaranteed to, be returned as a diagnostic in such case.

Parameters

- **stream** – - Stream to wait
- **event** – - Event to wait on

- **flags** – - Parameters for the operation(See above)

Returns
cudaSuccess, *cudaErrorInvalidValue*, *cudaErrorInvalidResourceHandle*

__host__ *cudaError_t* **cudaThreadExchangeStreamCaptureMode**(enum *cudaStreamCaptureMode* *mode)

Swaps the stream capture interaction mode for a thread.

Sets the calling thread's stream capture interaction mode to the value contained in `*mode`, and overwrites `*mode` with the previous mode for the thread. To facilitate deterministic behavior across function or module boundaries, callers are encouraged to use this API in a push-pop fashion:

```
cudaStreamCaptureMode mode = desiredMode;
cudaThreadExchangeStreamCaptureMode(&mode);
...
cudaThreadExchangeStreamCaptureMode(&mode); // restore previous mode
```

During stream capture (see *cudaStreamBeginCapture*), some actions, such as a call to *cudaMalloc*, may be unsafe. In the case of *cudaMalloc*, the operation is not enqueued asynchronously to a stream, and is not observed by stream capture. Therefore, if the sequence of operations captured via *cudaStreamBeginCapture* depended on the allocation being replayed whenever the graph is launched, the captured graph would be invalid.

Therefore, stream capture places restrictions on API calls that can be made within or concurrently to a *cudaStreamBeginCapture*-*cudaStreamEndCapture* sequence. This behavior can be controlled via this API and flags to *cudaStreamBeginCapture*.

A thread's mode is one of the following:

- `cudaStreamCaptureModeGlobal:` This is the default mode. If the local thread has an ongoing capture sequence that was not initiated with `cudaStreamCaptureModeRelaxed` at `cuStreamBeginCapture`, or if any other thread has a concurrent capture sequence initiated with `cudaStreamCaptureModeGlobal`, this thread is prohibited from potentially unsafe API calls.
- `cudaStreamCaptureModeThreadLocal:` If the local thread has an ongoing capture sequence not initiated with `cudaStreamCaptureModeRelaxed`, it is prohibited from potentially unsafe API calls. Concurrent capture sequences in other threads are ignored.
- `cudaStreamCaptureModeRelaxed:` The local thread is not prohibited from potentially unsafe API calls. Note that the thread is still prohibited from API calls which necessarily conflict with stream capture, for example, attempting *cudaEventQuery* on an event that was last recorded inside a capture sequence.

See also:

cudaStreamBeginCapture

Note: Note that this function may also return error codes from previous, asynchronous launches.

Parameters
mode – - Pointer to mode value to swap with the current mode

Returns
cudaSuccess, *cudaErrorInvalidValue*

6.32.2. Typedefs

cudaError_t(__stdcall * cudaGraphRecaptureCallback_t)(void *data, cudaGraphNode_t node, const struct cudaGraphNodeParams *originalParams, const struct cudaGraphNodeParams *recaptureParams, enum cudaGraphRecaptureStatus status)

Callback function invoked when node parameter mismatches are detected while recapturing to an existing graph.

Parameter struct pointers are only valid within the callback.

Param data
User parameter provided at beginning of recapture

Param node
Graph node with the parameter mismatch

Param originalParams
The original node parameters from the graph

Param recaptureParams
The node parameters received during the recapture

Param status
status Status code for the recapture of the node

Return
Error code for the callback. Anything other than cudaSuccess will cause the recapture to fail immediately.

void(__stdcall * cudaStreamCallback_t)(cudaStream_t stream, cudaError_t status, void *userData)

Type of stream callback functions.

Param stream
The stream as passed to *cudaStreamAddCallback*, may be NULL.

Param status
cudaSuccess or any persistent error on the stream.

Param userData
User parameter provided at registration.

6.33. Stream Ordered Memory Allocator

```
Functions for controlling the behavior of the underlying allocator.
```

6.33.1. overview

The asynchronous allocator allows the user to allocate and free in stream order. All asynchronous accesses of the allocation must happen between the stream executions of the allocation and the free. If the memory is accessed outside of the promised stream order, a use before allocation / use after free error will cause undefined behavior.

The allocator is free to reallocate the memory as long as it can guarantee that compliant memory accesses will not overlap temporally. The allocator may refer to internal stream ordering as well as inter-stream dependencies (such as CUDA events and null stream dependencies) when establishing the temporal guarantee. The allocator may also insert inter-stream dependencies to establish the temporal guarantee.

6.33.2. Supported Platforms

Whether or not a device supports the integrated stream ordered memory allocator may be queried by calling *cudaDeviceGetAttribute()* with the device attribute *cudaDevAttrMemoryPoolsSupported*.

Functions

__host__ cudaError_t *cudaFreeAsync*(void *devPtr, cudaStream_t hStream)
Frees memory with stream ordered semantics.

__host__ cudaError_t *cudaMallocAsync*(void **devPtr, size_t size, cudaStream_t hStream)
Allocates memory with stream ordered semantics.

__host__ cudaError_t *cudaMallocFromPoolAsync*(void **ptr, size_t size, cudaMemPool_t memPool, cudaStream_t stream)
Allocates memory from a specified pool with stream ordered semantics.

__host__ cudaError_t *cudaMemGetDefaultMemPool*(cudaMemPool_t *memPool, struct cudaMemLocation *location, enum cudaMemAllocationType type)
Returns the default memory pool for a given location and allocation type.

__host__ cudaError_t *cudaMemGetMemPool*(cudaMemPool_t *memPool, struct cudaMemLocation *location, enum cudaMemAllocationType type)
Gets the current memory pool for a given memory location and allocation type.

__host__ cudaError_t *cudaMemPoolCreate*(cudaMemPool_t *memPool, const struct cudaMemPoolProps *poolProps)
Creates a memory pool.

__host__ cudaError_t *cudaMemPoolDestroy*(cudaMemPool_t memPool)
Destroys the specified memory pool.

__host__ cudaError_t *cudaMemPoolExportPointer*(struct cudaMemPoolPtrExportData *exportData, void *ptr)
Export data to share a memory pool allocation between processes.

__host__ cudaError_t *cudaMemPoolExportToShareableHandle*(void *shareableHandle, cudaMemPool_t memPool, enum cudaMemAllocationHandleType handleType, unsigned int flags)
Exports a memory pool to the requested handle type.

__host__ cudaError_t *cudaMemPoolGetAccess*(enum cudaMemAccessFlags *flags, cudaMemPool_t memPool, struct cudaMemLocation *location)
Returns the accessibility of a pool from a device.

__host__ cudaError_t *cudaMemPoolGetAttribute*(cudaMemPool_t memPool, enum cudaMemPoolAttr attr, void *value)
Gets attributes of a memory pool.

__host__ cudaError_t *cudaMemPoolImportFromShareableHandle*(cudaMemPool_t *memPool, void *shareableHandle, enum cudaMemAllocationHandleType handleType, unsigned int flags)
imports a memory pool from a shared handle.

__host__ cudaError_t *cudaMemPoolImportPointer*(void **ptr, cudaMemPool_t memPool, struct cudaMemPoolPtrExportData *exportData)
Import a memory pool allocation from another process.

__host__ cudaError_t *cudaMemPoolSetAccess*(cudaMemPool_t memPool, const struct cudaMemAccessDesc *descList, size_t count)
Controls visibility of pools between devices.

__host__ cudaError_t *cudaMemPoolSetAttribute*(cudaMemPool_t memPool, enum cudaMemPoolAttr attr, void *value)
Sets attributes of a memory pool.

__host__ cudaError_t *cudaMemPoolTrimTo*(cudaMemPool_t memPool, size_t minBytesToKeep)
Tries to release memory back to the OS.

__host__ cudaError_t *cudaMemSetMemPool*(struct cudaMemLocation *location, enum cudaMemAllocationType type, cudaMemPool_t memPool)
Sets the current memory pool for a memory location and allocation type.

6.33.3. Functions

__host__ *cudaError_t* **cudaFreeAsync**(void *devPtr, *cudaStream_t* hStream)

Frees memory with stream ordered semantics.

Inserts a free operation into `hStream`. The allocation must not be accessed after stream execution reaches the free. After this API returns, accessing the memory from any subsequent work launched on the GPU or querying its pointer attributes results in undefined behavior.

See also:

::cuMemFreeAsync, *cudaMallocAsync*

Note: During stream capture, this function results in the creation of a free node and must therefore be passed the address of a graph allocation.

Note: Note that this function may also return error codes from previous, asynchronous launches.

Note: This function uses standard default stream semantics.

Note: Note that this function may also return *cudaErrorInitializationError*, *cudaErrorInsufficientDriver* or *cudaErrorNoDevice* if this call tries to initialize internal CUDA RT state.

Note: Note that as specified by *cudaStreamAddCallback* no CUDA function may be called from callback. *cudaErrorNotPermitted* may, but is not guaranteed to, be returned as a diagnostic in such case.

Parameters

- **dptr** – - memory to free
- **hStream** – - The stream establishing the stream ordering promise

Returns

cudaSuccess, *cudaErrorInvalidValue*, *cudaErrorNotSupported*

__host__ *cudaError_t* **cudaMallocAsync**(void **devPtr, size_t size, *cudaStream_t* hStream)

Allocates memory with stream ordered semantics.

Inserts an allocation operation into `hStream`. A pointer to the allocated memory is returned immediately in *dptr. The allocation must not be accessed until the the allocation operation completes. The allocation comes from the memory pool associated with the stream's device.

See also:

::cuMemAllocAsync, *cudaMallocAsync (C++ API)*, *cudaMallocFromPoolAsync*, *cudaFreeAsync*, *cudaDeviceSetMemPool*, *cudaDeviceGetDefaultMemPool*, *cudaDeviceGetMemPool*, *cudaMemPoolSetAccess*, *cudaMemPoolSetAttribute*, *cudaMemPoolGetAttribute*

Note: The default memory pool of a device contains device memory from that device.

Note: Basic stream ordering allows future work submitted into the same stream to use the allocation. Stream query, stream synchronize, and CUDA events can be used to guarantee that the allocation operation completes before work submitted in a separate stream runs.

Note: During stream capture, this function results in the creation of an allocation node. In this case, the allocation is owned by the graph instead of the memory pool. The memory pool's properties are used to set the node's creation parameters.

Note: Note that this function may also return error codes from previous, asynchronous launches.

Note: This function uses standard default stream semantics.

Note: Note that this function may also return *cudaErrorInitializationError*, *cudaErrorInsufficientDriver* or *cudaErrorNoDevice* if this call tries to initialize internal CUDA RT state.

Note: Note that as specified by *cudaStreamAddCallback* no CUDA function may be called from callback. *cudaErrorNotPermitted* may, but is not guaranteed to, be returned as a diagnostic in such case.

Parameters

- **devPtr** – **[out]** - Returned device pointer
- **size** – **[in]** - Number of bytes to allocate
- **hStream** – **[in]** - The stream establishing the stream ordering contract and the memory pool to allocate from

Returns

cudaSuccess, *cudaErrorInvalidValue*, *cudaErrorNotSupported*, ::cudaErrorOutOfMemory,

__host__ *cudaError_t* **cudaMallocFromPoolAsync**(void **ptr, size_t size, *cudaMemPool_t* memPool, *cudaStream_t* stream)

Allocates memory from a specified pool with stream ordered semantics.

Inserts an allocation operation into `stream`. A pointer to the allocated memory is returned immediately in *dptr. The allocation must not be accessed until the the allocation operation completes. The allocation comes from the specified memory pool.

See also:

::cuMemAllocFromPoolAsync, *cudaMallocAsync (C++ API)*, *cudaMallocAsync*, *cudaFreeAsync*, *cudaDeviceGetDefaultMemPool*, *cudaMemPoolCreate*, *cudaMemPoolSetAccess*, *cudaMemPoolSetAttribute*

Note:

- The specified memory pool may be from a device different than that of the specified `stream`.
- Basic stream ordering allows future work submitted into the same stream to use the allocation. Stream query, stream synchronize, and CUDA events can be used to guarantee that the allocation operation completes before work submitted in a separate stream runs.

Note: During stream capture, this function results in the creation of an allocation node. In this case, the allocation is owned by the graph instead of the memory pool. The memory pool's properties are used to set the node's creation parameters.

Parameters

- **ptr** – **[out]** - Returned device pointer
- **bytesize** – **[in]** - Number of bytes to allocate
- **memPool** – **[in]** - The pool to allocate from
- **stream** – **[in]** - The stream establishing the stream ordering semantic

Returns

cudaSuccess, *cudaErrorInvalidValue*, *cudaErrorNotSupported*, ::cudaErrorOutOfMemory

__host__ *cudaError_t* **cudaMemGetDefaultMemPool**(*cudaMemPool_t* *memPool, struct *cudaMemLocation* *location, enum *cudaMemAllocationType* type)

Returns the default memory pool for a given location and allocation type.

The memory location can be of one of *cudaMemLocationTypeDevice*, *cudaMemLocationTypeHost*, *cudaMemLocationTypeHostNuma*, or *cudaMemLocationTypeDeviceLocalityDomain*. The allocation type can be one of *cudaMemAllocationTypePinned* or *cudaMemAllocationTypeManaged*. When the allocation type is *cudaMemAllocationTypeManaged*, the location type can also be *cudaMemLocationTypeNone* to indicate no preferred location for the managed memory pool. *cudaMemAllocationTypeManaged* can not be used with *cudaMemLocationTypeDeviceLocalityDomain*. In all other cases, the call return *cudaErrorInvalidValue*

See also:

::cuMemAllocAsync, ::cuMemPoolTrimTo, ::cuMemPoolGetAttribute, ::cuMemPoolSetAttribute, cuMemPoolSetAccess, ::cuMemGetMemPool, ::cuMemPoolCreate

Returns

cudaSuccess, *cudaErrorInvalidValue*, *cudaErrorNotSupported*,

__host__ *cudaError_t* **cudaMemGetMemPool**(*cudaMemPool_t* *memPool, struct *cudaMemLocation* *location, enum *cudaMemAllocationType* type)

Gets the current memory pool for a given memory location and allocation type.

The memory location can be of one of *cudaMemLocationTypeDevice*, *cudaMemLocationTypeHost*, *cudaMemLocationTypeHostNuma*, or *cudaMemLocationTypeDeviceLocalityDomain*. The allocation type can be one of *cudaMemAllocationTypePinned* or *cudaMemAllocationTypeManaged*. When the allocation type is *cudaMemAllocationTypeManaged*, the location type can also be *cudaMemLocationTypeNone* to indicate no preferred location for the managed memory pool. *cudaMemAllocationTypeManaged* can not be used with *cudaMemLocationTypeDeviceLocalityDomain*. In all other cases, the call return *cudaErrorInvalidValue*

Returns the last pool provided to *cudaMemSetMemPool* or *cudaDeviceSetMemPool* for this location and allocation type or the location's default memory pool if *cudaMemSetMemPool* or *cudaDeviceSetMemPool* for that allocType and location has never been called. By default the current

mempool of a location is the default mempool for a device that can be obtained via cudaMemGetDefaultMemPool Otherwise the returned pool must have been set with *cudaDeviceSetMemPool*.

See also:

::cuDeviceGetDefaultMemPool, ::cuMemPoolCreate, ::cuDeviceSetMemPool, ::cuMemSetMemPool

Returns

cudaSuccess, *cudaErrorInvalidValue*, *cudaErrorNotSupported*

__host__ *cudaError_t* **cudaMemPoolCreate**(*cudaMemPool_t* *memPool, const struct *cudaMemPoolProps* *poolProps)

Creates a memory pool.

Creates a CUDA memory pool and returns the handle in `pool`. The `poolProps` determines the properties of the pool such as the backing device and IPC capabilities.

To create a memory pool for host memory not targeting a specific NUMA node, applications must set set ::cudaMemPoolProps::cudaMemLocation::type to *cudaMemLocationTypeHost*. ::cudaMemPoolProps::cudaMemLocation::id is ignored for such pools. Pools created with the type *cudaMemLocationTypeHost* are not IPC capable and *cudaMemPoolProps::handleTypes* must be 0, any other values will result in *cudaErrorInvalidValue*. To create a memory pool targeting a specific host NUMA node, applications must set ::cudaMemPoolProps::cudaMemLocation::type to *cudaMemLocationTypeHostNuma* and ::cudaMemPoolProps::cudaMemLocation::id must specify the NUMA ID of the host memory node. Specifying *cudaMemLocationTypeHostNumaCurrent* as the ::cudaMemPoolProps::cudaMemLocation::type will result in *cudaErrorInvalidValue*. To create a memory pool targeting a specific device locality domain, applications must set ::cudaMemPoolProps::cudaMemLocation::type to *cudaMemLocationTypeDeviceLocalityDomain*, ::cudaMemPoolProps::cudaMemLocation::localized.deviceId must specify the device ID, and ::cudaMemPoolProps::cudaMemLocation::localized.localityDomainId must specify the locality domain ID. The locality domain ID must be a valid locality domain ID for the specified device. See also *cudaDeviceGetAttribute* with the attribute *cudaDevAttrLocalityDomainCount*. By default, the pool's memory will be accessible from the device it is allocated on. In the case of pools created with *cudaMemLocationTypeHostNuma* or *cudaMemLocationTypeHost*, their default accessibility will be from the host CPU. Applications can control the maximum size of the pool by specifying a non-zero value for *cudaMemPoolProps::maxSize*. If set to 0, the maximum size of the pool will default to a system dependent value.

Applications that intend to use ::CU_MEM_HANDLE_TYPE_FABRIC based memory sharing must ensure: (1) `nvidia-caps-imex-channels` character device is created by the driver and is listed under /proc/devices (2) have at least one IMEX channel file accessible by the user launching the application.

When exporter and importer CUDA processes have been granted access to the same IMEX channel, they can securely share memory.

The IMEX channel security model works on a per user basis. Which means all processes under a user can share memory if the user has access to a valid IMEX channel. When multi-user isolation is desired, a separate IMEX channel is required for each user.

These channel files exist in /dev/nvidia-caps-imex-channels/channel* and can be created using standard OS native calls like mknod on Linux. For example: To create channel0 with the major number from /proc/devices users can execute the following command: `mknod /dev/nvidia-caps-imex-channels/channel0 c <major number> 0`

To create a managed memory pool, applications must set *cudaMemPoolProps*::cudaMemAllocationType to *cudaMemAllocationTypeManaged*. ::cudaMemPoolProps::cudaMemAllocationHandleType must also be set to *cudaMemHandleTypeNone* since IPC is not supported. For managed memory pools, ::cudaMemPoolProps::cudaMemLocation will be treated as the preferred location for all allocations created from the pool. An application can also set *cudaMemLocationTypeNone* to indicate no preferred location. *cudaMemPoolProps::maxSize* must be set to zero for managed memory pools. *cudaMemPoolProps::usage* should be zero as decompress for managed memory is not supported. For managed memory pools, all devices on the system must have non-zero ::concurrentManagedAccess. If not, this call returns *cudaErrorNotSupported*

See also:

::cuMemPoolCreate, *cudaDeviceSetMemPool*, *cudaMallocFromPoolAsync*, *cudaMemPoolExportToShareableHandle*, *cudaDeviceGetDefaultMemPool*, *cudaDeviceGetMemPool*

Note: Specifying *cudaMemHandleTypeNone* creates a memory pool that will not support IPC.

Returns

cudaSuccess, *cudaErrorInvalidValue*, *cudaErrorNotSupported*

__host__ *cudaError_t* **cudaMemPoolDestroy**(*cudaMemPool_t* memPool)

Destroys the specified memory pool.

If any pointers obtained from this pool haven't been freed or the pool has free operations that haven't completed when *cudaMemPoolDestroy* is invoked, the function will return immediately and the resources associated with the pool will be released automatically once there are no more outstanding allocations.

Destroying the current mempool of a device sets the default mempool of that device as the current mempool for that device.

See also:

cuMemPoolDestroy, *cudaFreeAsync*, *cudaDeviceSetMemPool*, *cudaDeviceGetDefaultMemPool*, *cudaDeviceGetMemPool*, *cudaMemPoolCreate*

Note: A device's default memory pool cannot be destroyed.

Returns

cudaSuccess, *cudaErrorInvalidValue*

__host__ *cudaError_t* **cudaMemPoolExportPointer**(struct *cudaMemPoolPtrExportData* *exportData, void *ptr)

Export data to share a memory pool allocation between processes.

Constructs `shareData_out` for sharing a specific allocation from an already shared memory pool. The recipient process can import the allocation with the *cudaMemPoolImportPointer* api. The data is not a handle and may be shared through any IPC mechanism.

See also:

::cuMemPoolExportPointer, *cudaMemPoolExportToShareableHandle*, *cudaMemPoolImportFromShareableHandle*, *cudaMemPoolImportPointer*

Parameters

- **shareData_out** – **[out]** - Returned export data
- **ptr** – **[in]** - pointer to memory being exported

Returns

cudaSuccess, *cudaErrorInvalidValue*, ::cudaErrorOutOfMemory

__host__ *cudaError_t* **cudaMemPoolExportToShareableHandle**(void *shareableHandle, *cudaMemPool_t* memPool, enum *cudaMemAllocationHandleType* handleType, unsigned int flags)

Exports a memory pool to the requested handle type.

Given an IPC capable mempool, create an OS handle to share the pool with another process. A recipient process can convert the shareable handle into a mempool with *cudaMemPoolImportFromShareableHandle*. Individual pointers can then be shared with the *cudaMemPoolExportPointer* and *cudaMemPoolImportPointer* APIs. The implementation of what the shareable handle is and how it can be transferred is defined by the requested handle type.

See also:

::cuMemPoolExportToShareableHandle, *cudaMemPoolImportFromShareableHandle*, *cudaMemPoolExportPointer*, *cudaMemPoolImportPointer*

Note: : To create an IPC capable mempool, create a mempool with a CUmemAllocationHandleType other than cudaMemHandleTypeNone.

Parameters

- **handle_out** – **[out]** - pointer to the location in which to store the requested handle
- **pool** – **[in]** - pool to export
- **handleType** – **[in]** - the type of handle to create
- **flags** – **[in]** - must be 0

Returns

cudaSuccess, *cudaErrorInvalidValue*, ::cudaErrorOutOfMemory

__host__ *cudaError_t* **cudaMemPoolGetAccess**(enum *cudaMemAccessFlags* *flags, *cudaMemPool_t* memPool, struct *cudaMemLocation* *location)

Returns the accessibility of a pool from a device.

Returns the accessibility of the pool's memory from the specified location.

See also:

::cuMemPoolGetAccess, *cudaMemPoolSetAccess*

Parameters

- **flags** – **[out]** - the accessibility of the pool from the specified location
- **memPool** – **[in]** - the pool being queried
- **location** – **[in]** - the location accessing the pool

__host__ *cudaError_t* **cudaMemPoolGetAttribute**(*cudaMemPool_t* memPool, enum *cudaMemPoolAttr* attr, void *value)

Gets attributes of a memory pool.

Supported attributes are:

- *cudaMemPoolAttrReleaseThreshold*: (value type = cuuint64_t) Amount of reserved memory in bytes to hold onto before trying to release memory back to the OS. When more than the release threshold bytes of memory are held by the memory pool, the allocator will try to release memory back to the OS on the next call to stream, event or context synchronize. (default 0)
- *cudaMemPoolReuseFollowEventDependencies*: (value type = int) Allow *cudaMallocAsync* to use memory asynchronously freed in another stream as long as a stream ordering dependency of the allocating stream on the free action exists. Cuda events and null stream interactions can create the required stream ordered dependencies. (default enabled)
- *cudaMemPoolReuseAllowOpportunistic*: (value type = int) Allow reuse of already completed frees when there is no dependency between the free and allocation. (default enabled)
- *cudaMemPoolReuseAllowInternalDependencies*: (value type = int) Allow *cudaMallocAsync* to insert new stream dependencies in order to establish the stream ordering required to reuse a piece of memory released by *cudaFreeAsync* (default enabled).
- *cudaMemPoolAttrReservedMemCurrent*: (value type = cuuint64_t) Amount of backing memory currently allocated for the mempool.
- *cudaMemPoolAttrReservedMemHigh*: (value type = cuuint64_t) High watermark of backing memory allocated for the mempool since the last time it was reset.
- *cudaMemPoolAttrUsedMemCurrent*: (value type = cuuint64_t) Amount of memory from the pool that is currently in use by the application.
- *cudaMemPoolAttrUsedMemHigh*: (value type = cuuint64_t) High watermark of the amount of memory from the pool that was in use by the application since the last time it was reset.

The following properties can be also be queried on imported and default pools:

- *cudaMemPoolAttrAllocationType*: (value type = cudaMemAllocationType) The allocation type of the mempool
- *cudaMemPoolAttrExportHandleTypes*: (value type = cudaMemAllocationHandleType) Available export handle types for the mempool. For imported pools this value is always cudaMemHandleTypeNone as an imported pool cannot be re-exported
- *cudaMemPoolAttrLocationId*: (value type = int) The location id for the mempool. If the location type for this pool is cudaMemLocationTypeInvisible then ID will be cudaInvalidDeviceId.

- *cudaMemPoolAttrLocationType*: (value type = cudaMemLocationType) The location type for the mempool. For imported memory pools where the device is not directly visible to the importing process or pools imported via fabric handles across nodes this will be cudaMemlocataionTypeInvisible.
- *cudaMemPoolAttrLocalityDomainId*: (value type = int) The locality domain id for the mempool, if the mempool is localized to a locality domain. A value of -1 indicates that the mempool is not localized to a locality domain.
- *cudaMemPoolAttrMaxPoolSize*: (value type = cuuint64_t) Maximum size of the pool in bytes, this value may be higher than what was initially passed to cuMemPoolCreate due to alignment requirements. A value of 0 indicates no maximum size. For cudaMemAllocationTypeManaged and IPC imported pools this value will be system dependent.
- *cudaMemPoolAttrHwDecompressEnabled*: (value type = int) Indicates whether the pool has hardware compresssion enabled

See also:

::cuMemPoolGetAttribute, *cudaMallocAsync*, *cudaFreeAsync*, *cudaDeviceGetDefaultMemPool*, *cudaDeviceGetMemPool*, *cudaMemPoolCreate*

Note: Note that as specified by *cudaStreamAddCallback* no CUDA function may be called from callback. *cudaErrorNotPermitted* may, but is not guaranteed to, be returned as a diagnostic in such case.

Parameters

- **pool** – **[in]** - The memory pool to get attributes of
- **attr** – **[in]** - The attribute to get
- **value** – **[in]** - Retrieved value

Returns
cudaSuccess, *cudaErrorInvalidValue*

__host__ *cudaError_t* **cudaMemPoolImportFromShareableHandle**(*cudaMemPool_t* *memPool, void *shareableHandle, enum *cudaMemAllocationHandleType* handleType, unsigned int flags)

imports a memory pool from a shared handle.

Specific allocations can be imported from the imported pool with *cudaMemPoolImportPointer*.

See also:

::cuMemPoolImportFromShareableHandle, *cudaMemPoolExportToShareableHandle*, *cudaMemPoolExportPointer*, *cudaMemPoolImportPointer*

Note: Imported memory pools do not support creating new allocations. As such imported memory pools may not be used in *cudaDeviceSetMemPool* or *cudaMallocFromPoolAsync* calls.

Parameters

- **pool_out** – **[out]** - Returned memory pool
- **handle** – **[in]** - OS handle of the pool to open
- **handleType** – **[in]** - The type of handle being imported
- **flags** – **[in]** - must be 0

Returns
cudaSuccess, *cudaErrorInvalidValue*, ::cudaErrorOutOfMemory

__host__ *cudaError_t* **cudaMemPoolImportPointer**(void **ptr, *cudaMemPool_t* memPool, struct *cudaMemPoolPtrExportData* *exportData)

Import a memory pool allocation from another process.

Returns in `ptr_out` a pointer to the imported memory. The imported memory must not be accessed before the allocation operation completes in the exporting process. The imported memory must be freed from all importing processes before being freed in the exporting process. The pointer may be freed with cudaFree or cudaFreeAsync. If *cudaFreeAsync* is used, the free must be completed on the importing process before the free operation on the exporting process.

See also:

::cuMemPoolImportPointer, *cudaMemPoolExportToShareableHandle*, *cudaMemPoolImportFromShareableHandle*, *cudaMemPoolExportPointer*

Note: The *cudaFreeAsync* api may be used in the exporting process before the *cudaFreeAsync* operation completes in its stream as long as the *cudaFreeAsync* in the exporting process specifies a stream with a stream dependency on the importing process's *cudaFreeAsync*.

Parameters

- **ptr_out** – **[out]** - pointer to imported memory
- **pool** – **[in]** - pool from which to import
- **shareData** – **[in]** - data specifying the memory to import

Returns
::CUDA_SUCCESS, ::CUDA_ERROR_INVALID_VALUE, ::CUDA_ERROR_NOT_INITIALIZED, ::CUDA_ERROR_OUT_OF_MEMORY

__host__ *cudaError_t* **cudaMemPoolSetAccess**(*cudaMemPool_t* memPool, const struct *cudaMemAccessDesc* *descList, size_t count)

Controls visibility of pools between devices.

See also:

::cuMemPoolSetAccess, *cudaMemPoolGetAccess*, *cudaMallocAsync*, *cudaFreeAsync*

Parameters

- **pool** – **[in]** - The pool being modified
- **map** – **[in]** - Array of access descriptors. Each descriptor instructs the access to enable for a single gpu
- **count** – **[in]** - Number of descriptors in the map array.

Returns

cudaSuccess, *cudaErrorInvalidValue*, *cudaErrorNotSupported*

__host__ *cudaError_t* **cudaMemPoolSetAttribute**(*cudaMemPool_t* memPool, enum *cudaMemPoolAttr* attr, void *value)

Sets attributes of a memory pool.

Supported attributes are:

- *cudaMemPoolAttrReleaseThreshold*: (value type = cuuint64_t) Amount of reserved memory in bytes to hold onto before trying to release memory back to the OS. When more than the release threshold bytes of memory are held by the memory pool, the allocator will try to release memory back to the OS on the next call to stream, event or context synchronize. (default 0)
- *cudaMemPoolReuseFollowEventDependencies*: (value type = int) Allow *cudaMallocAsync* to use memory asynchronously freed in another stream as long as a stream ordering dependency of the allocating stream on the free action exists. Cuda events and null stream interactions can create the required stream ordered dependencies. (default enabled)
- *cudaMemPoolReuseAllowOpportunistic*: (value type = int) Allow reuse of already completed frees when there is no dependency between the free and allocation. (default enabled)
- *cudaMemPoolReuseAllowInternalDependencies*: (value type = int) Allow *cudaMallocAsync* to insert new stream dependencies in order to establish the stream ordering required to reuse a piece of memory released by *cudaFreeAsync* (default enabled).
- *cudaMemPoolAttrReservedMemHigh*: (value type = cuuint64_t) Reset the high watermark that tracks the amount of backing memory that was allocated for the memory pool. It is illegal to set this attribute to a non-zero value.
- *cudaMemPoolAttrUsedMemHigh*: (value type = cuuint64_t) Reset the high watermark that tracks the amount of used memory that was allocated for the memory pool. It is illegal to set this attribute to a non-zero value.

See also:

::cuMemPoolSetAttribute, *cudaMallocAsync*, *cudaFreeAsync*, *cudaDeviceGetDefaultMemPool*, *cudaDeviceGetMemPool*, *cudaMemPoolCreate*

Note: Note that as specified by *cudaStreamAddCallback* no CUDA function may be called from callback. *cudaErrorNotPermitted* may, but is not guaranteed to, be returned as a diagnostic in such case.

Parameters

- **pool** – **[in]** - The memory pool to modify
- **attr** – **[in]** - The attribute to modify

- **value** – **[in]** - Pointer to the value to assign

Returns
cudaSuccess, *cudaErrorInvalidValue*

__host__ *cudaError_t* **cudaMemPoolTrimTo**(*cudaMemPool_t* memPool, size_t minBytesToKeep)

Tries to release memory back to the OS.

Releases memory back to the OS until the pool contains fewer than minBytesToKeep reserved bytes, or there is no more memory that the allocator can safely release. The allocator cannot release OS allocations that back outstanding asynchronous allocations. The OS allocations may happen at different granularity from the user allocations.

See also:

::cuMemPoolTrimTo, *cudaMallocAsync*, *cudaFreeAsync*, *cudaDeviceGetDefaultMemPool*, *cudaDeviceGetMemPool*, *cudaMemPoolCreate*

Note: : Allocations that have not been freed count as outstanding.

Note: : Allocations that have been asynchronously freed but whose completion has not been observed on the host (eg. by a synchronize) can count as outstanding.

Note: Note that as specified by *cudaStreamAddCallback* no CUDA function may be called from callback. *cudaErrorNotPermitted* may, but is not guaranteed to, be returned as a diagnostic in such case.

Parameters

- **pool** – **[in]** - The memory pool to trim
- **minBytesToKeep** – **[in]** - If the pool has less than minBytesToKeep reserved, the TrimTo operation is a no-op. Otherwise the pool will be guaranteed to have at least minBytesToKeep bytes reserved after the operation.

Returns
cudaSuccess, *cudaErrorInvalidValue*

__host__ *cudaError_t* **cudaMemSetMemPool**(struct *cudaMemLocation* *location, enum *cudaMemAllocationType* type, *cudaMemPool_t* memPool)

Sets the current memory pool for a memory location and allocation type.

The memory location can be of one of *cudaMemLocationTypeDevice*, *cudaMemLocationTypeHost* *cudaMemLocationTypeHostNuma*, or *cudaMemLocationTypeDeviceLocalityDomain*. The allocation type can be one of *cudaMemAllocationTypePinned* or *cudaMemAllocationTypeManaged*. When the allocation type is *cudaMemAllocationTypeManaged*, the location type can also be *cudaMemLocationTypeNone* to indicate no preferred location for the managed memory pool. *cudaMemAllocationTypeManaged* can not be used with *cudaMemLocationTypeDeviceLocalityDomain*. In all other cases, the call return *cudaErrorInvalidValue*

When a memory pool is set as the current memory pool, the location parameter should be the same as the location of the pool. If the location type or index don't match, the call returns *cudaErrorInvalidValue*. The type of memory pool should also match the parameter allocType. Else the call returns *cudaErrorInvalidValue* .

By default, a memory location's current memory pool is its default memory pool. If the location type is *cudaMemLocationTypeDevice* and the allocation type is *cudaMemAllocationTypePinned*, then this API is the equivalent of calling *cudaDeviceSetMemPool* with the location id as the device. For further details on the implications, please refer to the documentation for *cudaDeviceSetMemPool*.

See also:

::cuDeviceGetDefaultMemPool, ::cuDeviceGetMemPool, ::cuMemGetMemPool, ::cuMemPoolCreate, ::cuMemPoolDestroy, ::cuMemAllocFromPoolAsync

---

Note: Use *cudaMallocFromPoolAsync* to specify asynchronous allocations from a device different than the one the stream runs on.

Returns

cudaSuccess, *cudaErrorInvalidValue*, *cudaErrorNotSupported*

6.34. Surface Object Management

This section describes the low level texture object management functions of the CUDA runtime application programming interface.

The surface object API is only supported on devices of compute capability 3.0 or higher.

Functions

__host__ cudaError_t *cudaCreateSurfaceObject*(cudaSurfaceObject_t *pSurfObject, const struct cudaResourceDesc *pResDesc)
Creates a surface object.

__host__ cudaError_t *cudaDestroySurfaceObject*(cudaSurfaceObject_t surfObject)
Destroys a surface object.

__host__ cudaError_t *cudaGetSurfaceObjectResourceDesc*(struct cudaResourceDesc *pResDesc, cudaSurfaceObject_t surfObject)
Returns a surface object's resource descriptor Returns the resource descriptor for the surface object specified by `surfObject` .

6.34.1. Functions

__host__ *cudaError_t* **cudaCreateSurfaceObject**(*cudaSurfaceObject_t* *pSurfObject, const struct *cudaResourceDesc* *pResDesc)

Creates a surface object.

Creates a surface object and returns it in `pSurfObject`. `pResDesc` describes the data to perform surface load/stores on. *cudaResourceDesc::resType* must be *cudaResourceTypeArray* and ::cudaResourceDesc::res::array::array must be set to a valid CUDA array handle.

Surface objects are only supported on devices of compute capability 3.0 or higher. Additionally, a surface object is an opaque value, and, as such, should only be accessed through CUDA API calls.

See also:

cudaDestroySurfaceObject, ::cuSurfObjectCreate

Note: Note that this function may also return *cudaErrorInitializationError*, *cudaErrorInsufficientDriver* or *cudaErrorNoDevice* if this call tries to initialize internal CUDA RT state.

Note: Note that as specified by *cudaStreamAddCallback* no CUDA function may be called from callback. *cudaErrorNotPermitted* may, but is not guaranteed to, be returned as a diagnostic in such case.

Parameters

- **pSurfObject** – - Surface object to create
- **pResDesc** – - Resource descriptor

Returns

cudaSuccess, *cudaErrorInvalidValue*, *cudaErrorInvalidChannelDescriptor*, *cudaErrorInvalidResourceHandle*

__host__ *cudaError_t* **cudaDestroySurfaceObject**(*cudaSurfaceObject_t* surfObject)

Destroys a surface object.

Destroys the surface object specified by `surfObject`.

See also:

cudaCreateSurfaceObject, ::cuSurfObjectDestroy

Note: Note that this function may also return *cudaErrorInitializationError*, *cudaErrorInsufficientDriver* or *cudaErrorNoDevice* if this call tries to initialize internal CUDA RT state.

Note: Note that as specified by *cudaStreamAddCallback* no CUDA function may be called from callback. *cudaErrorNotPermitted* may, but is not guaranteed to, be returned as a diagnostic in

such case.

Note: Use of the handle after this call is undefined behavior.

Parameters

surfObject – - Surface object to destroy

Returns

cudaSuccess, *cudaErrorInvalidValue*

__host__ *cudaError_t* **cudaGetSurfaceObjectResourceDesc**(struct *cudaResourceDesc* *pResDesc, *cudaSurfaceObject_t* surfObject)

Returns a surface object's resource descriptor Returns the resource descriptor for the surface object specified by `surfObject`.

See also:

cudaCreateSurfaceObject, ::cuSurfObjectGetResourceDesc

Note: Note that this function may also return *cudaErrorInitializationError*, *cudaErrorInsufficientDriver* or *cudaErrorNoDevice* if this call tries to initialize internal CUDA RT state.

Note: Note that as specified by *cudaStreamAddCallback* no CUDA function may be called from callback. *cudaErrorNotPermitted* may, but is not guaranteed to, be returned as a diagnostic in such case.

Parameters

- **pResDesc** – - Resource descriptor
- **surfObject** – - Surface object

Returns

cudaSuccess, *cudaErrorInvalidValue*

6.35. Texture Object Management

This section describes the low level texture object management functions of the CUDA runtime application programming interface.

The texture object API is only supported on devices of compute capability 3.0 or higher.

Functions

__host__ struct cudaChannelFormatDesc *cudaCreateChannelDesc*(int x, int y, int z, int w, enum cudaChannelFormatKind f)
Returns a channel descriptor using the specified format.

__host__ cudaError_t *cudaCreateTextureObject*(cudaTextureObject_t *pTexObject, const struct cudaResourceDesc *pResDesc, const struct cudaTextureDesc *pTexDesc, const struct cudaResourceViewDesc *pResViewDesc)
Creates a texture object.

__host__ cudaError_t *cudaDestroyTextureObject*(cudaTextureObject_t texObject)
Destroys a texture object.

__host__ cudaError_t *cudaGetChannelDesc*(struct cudaChannelFormatDesc *desc, cudaArray_const_t array)
Get the channel descriptor of an array.

__host__ cudaError_t *cudaGetTextureObjectResourceDesc*(struct cudaResourceDesc *pResDesc, cudaTextureObject_t texObject)
Returns a texture object's resource descriptor.

__host__ cudaError_t *cudaGetTextureObjectResourceViewDesc*(struct cudaResourceViewDesc *pResViewDesc, cudaTextureObject_t texObject)
Returns a texture object's resource view descriptor.

__host__ cudaError_t *cudaGetTextureObjectTextureDesc*(struct cudaTextureDesc *pTexDesc, cudaTextureObject_t texObject)
Returns a texture object's texture descriptor.

6.35.1. Functions

__host__ struct *cudaChannelFormatDesc* **cudaCreateChannelDesc**(int x, int y, int z, int w, enum *cudaChannelFormatKind* f)

Returns a channel descriptor using the specified format.

Returns a channel descriptor with format f and number of bits of each component x, y, z, and w. The *cudaChannelFormatDesc* is defined as:

```
struct cudaChannelFormatDesc {
  int x, y, z, w;
  enum cudaChannelFormatKind f;
};
```

where *cudaChannelFormatKind* is one of *cudaChannelFormatKindSigned*, *cudaChannelFormatKindUnsigned*, or *cudaChannelFormatKindFloat*.

See also:

cudaCreateChannelDesc (C++ API), *cudaGetChannelDesc*, *cudaCreateTextureObject*, *cudaCreateSurfaceObject*

Parameters

- **x** - - X component
- **y** - - Y component
- **z** - - Z component
- **w** - - W component
- **f** - - Channel format

Returns
Channel descriptor with format f

__host__ *cudaError_t* **cudaCreateTextureObject**(*cudaTextureObject_t* *pTexObject, const struct *cudaResourceDesc* *pResDesc, const struct *cudaTextureDesc* *pTexDesc, const struct *cudaResourceViewDesc* *pResViewDesc)

Creates a texture object.

Creates a texture object and returns it in `pTexObject`. `pResDesc` describes the data to texture from. `pTexDesc` describes how the data should be sampled. `pResViewDesc` is an optional argument that specifies an alternate format for the data described by `pResDesc`, and also describes the subresource region to restrict access to when texturing. `pResViewDesc` can only be specified if the type of resource is a CUDA array or a CUDA mipmapped array not in a block compressed format.

Texture objects are only supported on devices of compute capability 3.0 or higher. Additionally, a texture object is an opaque value, and, as such, should only be accessed through CUDA API calls.

The *cudaResourceDesc* structure is defined as:

```
struct cudaResourceDesc {
    enum cudaResourceType resType;

    union {
        struct {
            cudaArray_t array;
        } array;
        struct {
            cudaMipmappedArray_t mipmap;
        } mipmap;
        struct {
            void *devPtr;
            struct cudaChannelFormatDesc desc;
            size_t sizeInBytes;
        } linear;
        struct {
            void *devPtr;
            struct cudaChannelFormatDesc desc;
            size_t width;
            size_t height;
            size_t pitchInBytes;
        } pitch2D;
    } res;
};
```

where:

- *cudaResourceDesc::resType* specifies the type of resource to texture from. CUresourceType is defined as:

```
enum cudaResourceType {
    cudaResourceTypeArray          = 0x00,
    cudaResourceTypeMipmappedArray = 0x01,
    cudaResourceTypeLinear         = 0x02,
    cudaResourceTypePitch2D        = 0x03
};
```

The *cudaTextureDesc* struct is defined as

```
struct cudaTextureDesc {
    enum cudaTextureAddressMode addressMode[3];
    enum cudaTextureFilterMode  filterMode;
    enum cudaTextureReadMode    readMode;
    int                         sRGB;
    float                       borderColor[4];
    int                         normalizedCoords;
    unsigned int                maxAnisotropy;
    enum cudaTextureFilterMode  mipmapFilterMode;
    float                       mipmapLevelBias;
    float                       minMipmapLevelClamp;
    float                       maxMipmapLevelClamp;
    int                         disableTrilinearOptimization;
    int                         seamlessCubemap;
};
```

where

- *cudaTextureDesc::addressMode* specifies the addressing mode for each dimension of the texture data. *cudaTextureAddressMode* is defined as:

  ```
  enum cudaTextureAddressMode {
      cudaAddressModeWrap   = 0,
      cudaAddressModeClamp  = 1,
      cudaAddressModeMirror = 2,
      cudaAddressModeBorder = 3
  };
  ```

  This is ignored if *cudaResourceDesc::resType* is *cudaResourceTypeLinear*. Also, if *cudaTextureDesc::normalizedCoords* is set to zero, *cudaAddressModeWrap* and *cudaAddressModeMirror* won't be supported and will be switched to *cudaAddressModeClamp*.

- *cudaTextureDesc::filterMode* specifies the filtering mode to be used when fetching from the texture. *cudaTextureFilterMode* is defined as:

  ```
  enum cudaTextureFilterMode {
      cudaFilterModePoint  = 0,
      cudaFilterModeLinear = 1
  };
  ```

  This is ignored if *cudaResourceDesc::resType* is *cudaResourceTypeLinear*.

- *cudaTextureDesc::readMode* specifies whether integer data should be converted to floating point or not. *cudaTextureReadMode* is defined as:

  ```
  enum cudaTextureReadMode {
      cudaReadModeElementType     = 0,
  ```

(continues on next page)

(continued from previous page)

```
    cudaReadModeNormalizedFloat = 1
};
```

Note that this applies only to 8-bit and 16-bit integer formats. 32-bit integer format would not be promoted, regardless of whether or not this *cudaTextureDesc::readMode* is set *cudaReadModeNormalizedFloat* is specified.

- *cudaTextureDesc::sRGB* specifies whether sRGB to linear conversion should be performed during texture fetch.
- *cudaTextureDesc::borderColor* specifies the float values of color. where: *cudaTextureDesc::borderColor*[0] contains value of 'R', *cudaTextureDesc::borderColor*[1] contains value of 'G', *cudaTextureDesc::borderColor*[2] contains value of 'B', *cudaTextureDesc::borderColor*[3] contains value of 'A' Note that application using integer border color values will need to <reinterpret_cast> these values to float. The values are set only when the addressing mode specified by *cudaTextureDesc::addressMode* is cudaAddressModeBorder.
- *cudaTextureDesc::normalizedCoords* specifies whether the texture coordinates will be normalized or not.
- *cudaTextureDesc::maxAnisotropy* specifies the maximum anistropy ratio to be used when doing anisotropic filtering. This value will be clamped to the range [1,16].
- *cudaTextureDesc::mipmapFilterMode* specifies the filter mode when the calculated mipmap level lies between two defined mipmap levels.
- *cudaTextureDesc::mipmapLevelBias* specifies the offset to be applied to the calculated mipmap level.
- *cudaTextureDesc::minMipmapLevelClamp* specifies the lower end of the mipmap level range to clamp access to.
- *cudaTextureDesc::maxMipmapLevelClamp* specifies the upper end of the mipmap level range to clamp access to.
- *cudaTextureDesc::disableTrilinearOptimization* specifies whether the trilinear filtering optimizations will be disabled.
- *cudaTextureDesc::seamlessCubemap* specifies whether seamless cube map filtering is enabled. This flag can only be specified if the underlying resource is a CUDA array or a CUDA mipmapped array that was created with the flag *cudaArrayCubemap*. When seamless cube map filtering is enabled, texture address modes specified by *cudaTextureDesc::addressMode* are ignored. Instead, if the *cudaTextureDesc::filterMode* is set to *cudaFilterModePoint* the address mode *cudaAddressModeClamp* will be applied for all dimensions. If the *cudaTextureDesc::filterMode* is set to *cudaFilterModeLinear* seamless cube map filtering will be performed when sampling along the cube face borders.

  If *cudaResourceDesc::resType* is set to *cudaResourceTypeArray*, ::cudaResourceDesc::res::array::array must be set to a valid CUDA array handle.

  If *cudaResourceDesc::resType* is set to *cudaResourceTypeMipmappedArray*, ::cudaResourceDesc::res::mipmap::mipmap must be set to a valid CUDA mipmapped array handle and *cudaTextureDesc::normalizedCoords* must be set to true.

  If *cudaResourceDesc::resType* is set to *cudaResourceTypeLinear*, ::cudaResourceDesc::res::linear::devPtr must be set to a valid device pointer, that is aligned to *cudaDeviceProp::textureAlignment*. ::cudaResourceDesc::res::linear::desc describes the format and the number of components per array element. ::cudaResourceDesc::res::linear::sizeInBytes specifies the size of the array in bytes. The to-

tal number of elements in the linear address range cannot exceed *cudaDeviceGetTexture1DLinearMaxWidth()*. The number of elements is computed as (sizeInBytes / sizeof(desc)).

If *cudaResourceDesc::resType* is set to *cudaResourceTypePitch2D*, ::cudaResourceDesc::res::pitch2D::devPtr must be set to a valid device pointer, that is aligned to *cudaDeviceProp::textureAlignment*. ::cudaResourceDesc::res::pitch2D::desc describes the format and the number of components per array element. ::cudaResourceDesc::res::pitch2D::width and ::cudaResourceDesc::res::pitch2D::height specify the width and height of the array in elements, and cannot exceed *cudaDeviceProp::maxTexture2DLinear*[0] and *cudaDeviceProp::maxTexture2DLinear*[1] respectively. ::cudaResourceDesc::res::pitch2D::pitchInBytes specifies the pitch between two rows in bytes and has to be aligned to *cudaDeviceProp::texturePitchAlignment*. Pitch cannot exceed *cudaDeviceProp::maxTexture2DLinear*[2].

The *cudaResourceViewDesc* struct is defined as

```
struct cudaResourceViewDesc {
    enum cudaResourceViewFormat format;
    size_t                      width;
    size_t                      height;
    size_t                      depth;
    unsigned int                firstMipmapLevel;
    unsigned int                lastMipmapLevel;
    unsigned int                firstLayer;
    unsigned int                lastLayer;
};
```

where:

- *cudaResourceViewDesc::format* specifies how the data contained in the CUDA array or CUDA mipmapped array should be interpreted. Note that this can incur a change in size of the texture data. If the resource view format is a block compressed format, then the underlying CUDA array or CUDA mipmapped array has to have a 32-bit unsigned integer format with 2 or 4 channels, depending on the block compressed format. For ex., BC1 and BC4 require the underlying CUDA array to have a 32-bit unsigned int with 2 channels. The other BC formats require the underlying resource to have the same 32-bit unsigned int format but with 4 channels.
- *cudaResourceViewDesc::width* specifies the new width of the texture data. If the resource view format is a block compressed format, this value has to be 4 times the original width of the resource. For non block compressed formats, this value has to be equal to that of the original resource.
- *cudaResourceViewDesc::height* specifies the new height of the texture data. If the resource view format is a block compressed format, this value has to be 4 times the original height of the resource. For non block compressed formats, this value has to be equal to that of the original resource.
- *cudaResourceViewDesc::depth* specifies the new depth of the texture data. This value has to be equal to that of the original resource.
- *cudaResourceViewDesc::firstMipmapLevel* specifies the most detailed mipmap level. This will be the new mipmap level zero. For non-mipmapped resources, this value has to be zero.*cudaTextureDesc::minMipmapLevelClamp* and *cudaTextureDesc::maxMipmapLevelClamp* will be relative to this value. For ex., if the firstMipmapLevel is set to 2, and a minMipmapLevelClamp of 1.2 is specified, then the actual minimum mipmap level clamp will be 3.2.

- *cudaResourceViewDesc::lastMipmapLevel* specifies the least detailed mipmap level. For non-mipmapped resources, this value has to be zero.
- *cudaResourceViewDesc::firstLayer* specifies the first layer index for layered textures. This will be the new layer zero. For non-layered resources, this value has to be zero.
- *cudaResourceViewDesc::lastLayer* specifies the last layer index for layered textures. For non-layered resources, this value has to be zero.

See also:

cudaDestroyTextureObject, ::cuTexObjectCreate

Note: Note that this function may also return *cudaErrorInitializationError*, *cudaErrorInsufficientDriver* or *cudaErrorNoDevice* if this call tries to initialize internal CUDA RT state.

Note: Note that as specified by *cudaStreamAddCallback* no CUDA function may be called from callback. *cudaErrorNotPermitted* may, but is not guaranteed to, be returned as a diagnostic in such case.

Parameters

- **pTexObject** – - Texture object to create
- **pResDesc** – - Resource descriptor
- **pTexDesc** – - Texture descriptor
- **pResViewDesc** – - Resource view descriptor

Returns

cudaSuccess, *cudaErrorInvalidValue*

__host__ *cudaError_t* **cudaDestroyTextureObject**(*cudaTextureObject_t* texObject)

Destroys a texture object.

Destroys the texture object specified by `texObject`.

See also:

cudaCreateTextureObject, ::cuTexObjectDestroy

Note: Note that this function may also return *cudaErrorInitializationError*, *cudaErrorInsufficientDriver* or *cudaErrorNoDevice* if this call tries to initialize internal CUDA RT state.

Note: Note that as specified by *cudaStreamAddCallback* no CUDA function may be called from callback. *cudaErrorNotPermitted* may, but is not guaranteed to, be returned as a diagnostic in such case.

Note: Use of the handle after this call is undefined behavior.

Parameters
texObject – - Texture object to destroy

Returns
cudaSuccess, *cudaErrorInvalidValue*

__host__ *cudaError_t* **cudaGetChannelDesc**(struct *cudaChannelFormatDesc* *desc, *cudaArray_const_t* array)

Get the channel descriptor of an array.

Returns in *desc the channel descriptor of the CUDA array array.

See also:

cudaCreateChannelDesc (C API), *cudaCreateTextureObject*, *cudaCreateSurfaceObject*

Note: Note that this function may also return error codes from previous, asynchronous launches.

Note: Note that this function may also return *cudaErrorInitializationError*, *cudaErrorInsufficientDriver* or *cudaErrorNoDevice* if this call tries to initialize internal CUDA RT state.

Note: Note that as specified by *cudaStreamAddCallback* no CUDA function may be called from callback. *cudaErrorNotPermitted* may, but is not guaranteed to, be returned as a diagnostic in such case.

Parameters

- **desc** – - Channel format
- **array** – - Memory array on device

Returns
cudaSuccess, *cudaErrorInvalidValue*

__host__ *cudaError_t* **cudaGetTextureObjectResourceDesc**(struct *cudaResourceDesc* *pResDesc, *cudaTextureObject_t* texObject)

Returns a texture object's resource descriptor.

Returns the resource descriptor for the texture object specified by texObject.

See also:

cudaCreateTextureObject, ::cuTexObjectGetResourceDesc

Note: Note that this function may also return *cudaErrorInitializationError*, *cudaErrorInsufficientDriver* or *cudaErrorNoDevice* if this call tries to initialize internal CUDA RT state.

Note: Note that as specified by *cudaStreamAddCallback* no CUDA function may be called from callback. *cudaErrorNotPermitted* may, but is not guaranteed to, be returned as a diagnostic in such case.

Parameters

- **pResDesc** – - Resource descriptor
- **texObject** – - Texture object

Returns
cudaSuccess, *cudaErrorInvalidValue*

__host__ *cudaError_t* **cudaGetTextureObjectResourceViewDesc**(struct *cudaResourceViewDesc* *pResViewDesc, *cudaTextureObject_t* texObject)

Returns a texture object's resource view descriptor.

Returns the resource view descriptor for the texture object specified by `texObject`. If no resource view was specified, *cudaErrorInvalidValue* is returned.

See also:

cudaCreateTextureObject, ::cuTexObjectGetResourceViewDesc

Note: Note that this function may also return *cudaErrorInitializationError*, *cudaErrorInsufficientDriver* or *cudaErrorNoDevice* if this call tries to initialize internal CUDA RT state.

Note: Note that as specified by *cudaStreamAddCallback* no CUDA function may be called from callback. *cudaErrorNotPermitted* may, but is not guaranteed to, be returned as a diagnostic in such case.

Parameters

- **pResViewDesc** – - Resource view descriptor
- **texObject** – - Texture object

Returns
cudaSuccess, *cudaErrorInvalidValue*

__host__ *cudaError_t* **cudaGetTextureObjectTextureDesc**(struct *cudaTextureDesc* *pTexDesc, *cudaTextureObject_t* texObject)

Returns a texture object's texture descriptor.

Returns the texture descriptor for the texture object specified by `texObject`.

See also:

cudaCreateTextureObject, ::cuTexObjectGetTextureDesc

Note: Note that this function may also return *cudaErrorInitializationError*, *cudaErrorInsufficient-Driver* or *cudaErrorNoDevice* if this call tries to initialize internal CUDA RT state.

Note: Note that as specified by *cudaStreamAddCallback* no CUDA function may be called from callback. *cudaErrorNotPermitted* may, but is not guaranteed to, be returned as a diagnostic in such case.

Parameters

- **pTexDesc** – - Texture descriptor
- **texObject** – - Texture object

Returns

cudaSuccess, *cudaErrorInvalidValue*

6.36. Unified Addressing

This section describes the unified addressing functions of the CUDA runtime application programming interface.

6.36.1. Overview

CUDA devices can share a unified address space with the host.

For these devices there is no distinction between a device pointer and a host pointer — the same pointer value may be used to access memory from the host program and from a kernel running on the device (with exceptions enumerated below).

6.36.2. Supported Platforms

Whether or not a device supports unified addressing may be queried by calling *cudaGetDeviceProperties()* with the device property *cudaDeviceProp::unifiedAddressing*.

Unified addressing is automatically enabled in 64-bit processes .

6.36.3. Looking Up Information from Pointer Values

It is possible to look up information about the memory which backs a pointer value. For instance, one may want to know if a pointer points to host or device memory. As another example, in the case of device memory, one may want to know on which CUDA device the memory resides. These properties may be queried using the function *cudaPointerGetAttributes()*

Since pointers are unique, it is not necessary to specify information about the pointers specified to *cudaMemcpy()* and other copy functions.

The copy direction *cudaMemcpyDefault* may be used to specify that the CUDA runtime should infer the location of the pointer from its value.

6.36.4. Automatic Mapping of Host Allocated Host Memory

All host memory allocated through all devices using *cudaMallocHost()* and *cudaHostAlloc()* is always directly accessible from all devices that support unified addressing. This is the case regardless of whether or not the flags *cudaHostAllocPortable* and *cudaHostAllocMapped* are specified.

The pointer value through which allocated host memory may be accessed in kernels on all devices that support unified addressing is the same as the pointer value through which that memory is accessed on the host. It is not necessary to call *cudaHostGetDevicePointer()* to get the device pointer for these allocations.

Note that this is not the case for memory allocated using the flag *cudaHostAllocWriteCombined*, as discussed below.

6.36.5. Direct Access of Peer Memory

Upon enabling direct access from a device that supports unified addressing to another peer device that supports unified addressing using *cudaDeviceEnablePeerAccess()* all memory allocated in the peer device using *cudaMalloc()* and *cudaMallocPitch()* will immediately be accessible by the current device. The device pointer value through which any peer's memory may be accessed in the current device is the same pointer value through which that memory may be accessed from the peer device.

6.36.6. Exceptions, Disjoint Addressing

Not all memory may be accessed on devices through the same pointer value through which they are accessed on the host. These exceptions are host memory registered using *cudaHostRegister()* and host memory allocated using the flag *cudaHostAllocWriteCombined*. For these exceptions, there exists a distinct host and device address for the memory. The device address is guaranteed to not overlap any valid host pointer range and is guaranteed to have the same value across all devices that support unified addressing.

This device address may be queried using *cudaHostGetDevicePointer()* when a device using unified addressing is current. Either the host or the unified device pointer value may be used to refer to this memory in *cudaMemcpy()* and similar functions using the *cudaMemcpyDefault* memory direction.

Functions

__host__ cudaError_t *cudaPointerGetAttributes*(struct cudaPointerAttributes *attributes, const void *ptr)
Returns attributes about a specified pointer.

6.36.7. Functions

__host__ *cudaError_t* **cudaPointerGetAttributes**(struct *cudaPointerAttributes* *attributes, const void *ptr)

Returns attributes about a specified pointer.

Returns in `*attributes` the attributes of the pointer `ptr`. If pointer was not allocated in, mapped by or registered with context supporting unified addressing *cudaErrorInvalidValue* is returned.

The *cudaPointerAttributes* structure is defined as:

```
struct cudaPointerAttributes {
    enum cudaMemoryType type;
    int device;
    void *devicePointer;
    void *hostPointer;
    int localityDomainOrdinal;
}
```

In this structure, the individual fields mean

- *cudaPointerAttributes::type* identifies type of memory. It can be *cudaMemoryTypeUnregistered* for unregistered host memory, *cudaMemoryTypeHost* for registered host memory, *cudaMemoryTypeDevice* for device memory or *cudaMemoryTypeManaged* for managed memory.
- *device* is the device against which `ptr` was allocated. If `ptr` has memory type *cudaMemoryTypeDevice* then this identifies the device on which the memory referred to by `ptr` physically resides. If `ptr` has memory type *cudaMemoryTypeHost* then this identifies the device which

was current when the allocation was made (and if that device is deinitialized then this allocation will vanish with that device's state).

- *devicePointer* is the device pointer alias through which the memory referred to by `ptr` may be accessed on the current device. If the memory referred to by `ptr` cannot be accessed directly by the current device then this is NULL.
- *hostPointer* is the host pointer alias through which the memory referred to by `ptr` may be accessed on the host. If the memory referred to by `ptr` cannot be accessed directly by the host then this is NULL.
- localityDomainOrdinal is the locality domain ordinal for device allocations localized to a locality domain, or -1 when the allocation is not localized to a locality domain.

See also:

cudaGetDeviceCount, *cudaGetDevice*, *cudaSetDevice*, *cudaChooseDevice*, *cudaInitDevice*, ::cuPointerGetAttributes

---

Note: In CUDA 11.0 forward passing host pointer will return *cudaMemoryTypeUnregistered* in *cudaPointerAttributes::type* and call will return *cudaSuccess*.

Note: Note that this function may also return *cudaErrorInitializationError*, *cudaErrorInsufficientDriver* or *cudaErrorNoDevice* if this call tries to initialize internal CUDA RT state.

Note: Note that as specified by *cudaStreamAddCallback* no CUDA function may be called from callback. *cudaErrorNotPermitted* may, but is not guaranteed to, be returned as a diagnostic in such case.

Parameters

- **`attributes`** – - Attributes for the specified pointer
- **`ptr`** – - Pointer to get attributes for

Returns

cudaSuccess, *cudaErrorInvalidDevice*, *cudaErrorInvalidValue*

6.37. VDPAU Interoperability

This section describes the VDPAU interoperability functions of the CUDA runtime application programming interface.

Functions

__host__ cudaError_t *cudaGraphicsVDPAURegisterOutputSurface*(struct cudaGraphicsResource **resource, VdpOutputSurface vdpSurface, unsigned int flags)
Register a VdpOutputSurface object.

__host__ cudaError_t *cudaGraphicsVDPAURegisterVideoSurface*(struct cudaGraphicsResource **resource, VdpVideoSurface vdpSurface, unsigned int flags)
Register a VdpVideoSurface object.

__host__ cudaError_t *cudaVDPAUGetDevice*(int *device, VdpDevice vdpDevice, VdpGetProcAddress *vdpGetProcAddress)
Gets the CUDA device associated with a VdpDevice.

__host__ cudaError_t *cudaVDPAUSetVDPAUDevice*(int device, VdpDevice vdpDevice, VdpGetProcAddress *vdpGetProcAddress)
Sets a CUDA device to use VDPAU interoperability.

6.37.1. Functions

__host__ *cudaError_t* **cudaGraphicsVDPAURegisterOutputSurface**(struct cudaGraphicsResource **resource, VdpOutputSurface vdpSurface, unsigned int flags)

Register a VdpOutputSurface object.

Registers the VdpOutputSurface specified by `vdpSurface` for access by CUDA. A handle to the registered object is returned as `resource`. The surface's intended usage is specified using `flags`, as follows:

- *cudaGraphicsMapFlagsNone*: Specifies no hints about how this resource will be used. It is therefore assumed that this resource will be read from and written to by CUDA. This is the default value.
- *cudaGraphicsMapFlagsReadOnly*: Specifies that CUDA will not write to this resource.
- *cudaGraphicsMapFlagsWriteDiscard*: Specifies that CUDA will not read from this resource and will write over the entire contents of the resource, so none of the data previously stored in the resource will be preserved.

See also:

cudaVDPAUSetVDPAUDevice, *cudaGraphicsUnregisterResource*, *cudaGraphicsSubResourceGetMappedArray*, ::cuGraphicsVDPAURegisterOutputSurface

Note: Note that this function may also return error codes from previous, asynchronous launches.

Parameters

- **resource** – - Pointer to the returned object handle
- **vdpSurface** – - VDPAU object to be registered
- **flags** – - Map flags

Returns

cudaSuccess, *cudaErrorInvalidDevice*, *cudaErrorInvalidValue*, *cudaErrorInvalidResourceHandle*, *cudaErrorUnknown*

__host__ *cudaError_t* **cudaGraphicsVDPAURegisterVideoSurface**(struct cudaGraphicsResource **resource, VdpVideoSurface vdpSurface, unsigned int flags)

Register a VdpVideoSurface object.

Registers the VdpVideoSurface specified by `vdpSurface` for access by CUDA. A handle to the registered object is returned as `resource`. The surface's intended usage is specified using `flags`, as follows:

- *cudaGraphicsMapFlagsNone*: Specifies no hints about how this resource will be used. It is therefore assumed that this resource will be read from and written to by CUDA. This is the default value.
- *cudaGraphicsMapFlagsReadOnly*: Specifies that CUDA will not write to this resource.
- *cudaGraphicsMapFlagsWriteDiscard*: Specifies that CUDA will not read from this resource and will write over the entire contents of the resource, so none of the data previously stored in the resource will be preserved.

See also:

cudaVDPAUSetVDPAUDevice, *cudaGraphicsUnregisterResource*, *cudaGraphicsSubResourceGetMappedArray*, ::cuGraphicsVDPAURegisterVideoSurface

---

Note: Note that this function may also return error codes from previous, asynchronous launches.

Parameters

- **resource** – - Pointer to the returned object handle
- **vdpSurface** – - VDPAU object to be registered
- **flags** – - Map flags

Returns

cudaSuccess, *cudaErrorInvalidDevice*, *cudaErrorInvalidValue*, *cudaErrorInvalidResourceHandle*, *cudaErrorUnknown*

__host__ *cudaError_t* **cudaVDPAUGetDevice**(int *device, VdpDevice vdpDevice, VdpGetProcAddress *vdpGetProcAddress)

Gets the CUDA device associated with a VdpDevice.

Returns the CUDA device associated with a VdpDevice, if applicable.

See also:

cudaVDPAUSetVDPAUDevice, ::cuVDPAUGetDevice

Note: Note that this function may also return error codes from previous, asynchronous launches.

Parameters

- **device** – - Returns the device associated with vdpDevice, or -1 if the device associated with vdpDevice is not a compute device.
- **vdpDevice** – - A VdpDevice handle
- **vdpGetProcAddress** – - VDPAU's VdpGetProcAddress function pointer

Returns

cudaSuccess

__host__ *cudaError_t* **cudaVDPAUSetVDPAUDevice**(int device, VdpDevice vdpDevice, VdpGetProcAddress *vdpGetProcAddress)

Sets a CUDA device to use VDPAU interoperability.

Records `vdpDevice` as the VdpDevice for VDPAU interoperability with the CUDA device `device` and sets `device` as the current device for the calling host thread.

This function will immediately initialize the primary context on `device` if needed.

If `device` has already been initialized then this call will fail with the error *cudaErrorSetOnActiveProcess*. In this case it is necessary to reset `device` using *cudaDeviceReset()* before VDPAU interoperability on `device` may be enabled.

See also:

cudaGraphicsVDPAURegisterVideoSurface, *cudaGraphicsVDPAURegisterOutputSurface*, *cudaDeviceReset*

Note: Note that this function may also return error codes from previous, asynchronous launches.

Parameters

- **device** – - Device to use for VDPAU interoperability
- **vdpDevice** – - The VdpDevice to interoperate with
- **vdpGetProcAddress** – - VDPAU's VdpGetProcAddress function pointer

Returns

cudaSuccess, *cudaErrorInvalidDevice*, *cudaErrorSetOnActiveProcess*

6.38. Version Management

Functions

__host__ cudaError_t *cudaDriverGetVersion*(int *driverVersion)
Returns the latest version of CUDA supported by the driver.

__host__ cudaError_t *cudaRuntimeGetVersion*(int *runtimeVersion)
Returns the CUDA Runtime version.

6.38.1. Functions

__host__ *cudaError_t* **cudaDriverGetVersion**(int *driverVersion)

Returns the latest version of CUDA supported by the driver.

Returns in `*driverVersion` the latest version of CUDA supported by the driver. The version is returned as (1000 major + 10 minor). For example, CUDA 9.2 would be represented by 9020. If no driver is installed, then 0 is returned as the driver version.

This function automatically returns *cudaErrorInvalidValue* if `driverVersion` is NULL.

See also:

cudaRuntimeGetVersion, ::cuDriverGetVersion

Note: Note that this function may also return error codes from previous, asynchronous launches.

Note: Note that this function may also return *cudaErrorInitializationError*, *cudaErrorInsufficientDriver* or *cudaErrorNoDevice* if this call tries to initialize internal CUDA RT state.

Note: Note that as specified by *cudaStreamAddCallback* no CUDA function may be called from callback. *cudaErrorNotPermitted* may, but is not guaranteed to, be returned as a diagnostic in such case.

Parameters
`driverVersion` – - Returns the CUDA driver version.

Returns
cudaSuccess, *cudaErrorInvalidValue*

__host__ *cudaError_t* **cudaRuntimeGetVersion**(int *runtimeVersion)

Returns the CUDA Runtime version.

Returns in `*runtimeVersion` the version number of the current CUDA Runtime instance. The version is returned as (1000 major + 10 minor). For example, CUDA 9.2 would be represented by 9020.

As of CUDA 12.0, this function no longer initializes CUDA. The purpose of this API is solely to return a compile-time constant stating the CUDA Toolkit version in the above format.

As of CUDA 13.0, on Windows, the `runtimeVersion` may not be the same as the CUDA Toolkit version the app was built with. Windows will use the CUDA Runtime packaged with the display driver, and that version will be reported.

This function automatically returns *cudaErrorInvalidValue* if the `runtimeVersion` argument is NULL.

See also:

cudaDriverGetVersion, ::cuDriverGetVersion

Note: Note that this function may also return *cudaErrorInitializationError*, *cudaErrorInsufficientDriver* or *cudaErrorNoDevice* if this call tries to initialize internal CUDA RT state.

Note: Note that as specified by *cudaStreamAddCallback* no CUDA function may be called from callback. *cudaErrorNotPermitted* may, but is not guaranteed to, be returned as a diagnostic in such case.

Parameters

`runtimeVersion` – - Returns the CUDA Runtime version.

Returns

cudaSuccess, *cudaErrorInvalidValue*

C++ API Routines

impl_private

Data types used by CUDA Runtime

Device Management

impl_private

Device Management [DEPRECATED]

This section describes the deprecated device management functions of the CUDA runtime application programming interface.

Direct3D 10 Interoperability

impl_private

Direct3D 10 Interoperability [DEPRECATED]

This section describes deprecated Direct3D 10 interoperability functions.

Direct3D 11 Interoperability

impl_private

Direct3D 11 Interoperability [DEPRECATED]

This section describes deprecated Direct3D 11 interoperability functions.

Direct3D 9 Interoperability

impl_private

Direct3D 9 Interoperability [DEPRECATED]

This section describes deprecated Direct3D 9 interoperability functions.

Driver Entry Point Access
This section describes the driver entry point access functions of CUDA runtime application programming interface.

EGL Interoperability
This section describes the EGL interoperability functions of the CUDA runtime application programming interface.

Error Handling
This section describes the error handling functions of the CUDA runtime application programming interface.

Error Log Management Functions
This section describes the error log management functions of the CUDA runtime application programming interface.

Event Management
This section describes the event management functions of the CUDA runtime application programming interface.

Execution Context Management
This section describes the execution context management functions of the CUDA runtime application programming interface.

Execution Control
This section describes the execution control functions of the CUDA runtime application programming interface.

Execution Control [DEPRECATED]
This section describes the deprecated execution control functions of the CUDA runtime application programming interface.

External Resource Interoperability
This section describes the external resource interoperability functions of the CUDA runtime application programming interface.

Graph Management
This section describes the graph management functions of CUDA runtime application programming interface.

Graphics Interoperability
This section describes the graphics interoperability functions of the CUDA runtime application programming interface.

Interactions with the CUDA Driver API
This section describes the interactions between the CUDA Driver API and the CUDA Runtime API.

Library Management
This section describes the library management functions of the CUDA runtime application programming interface.

Logical Endpoint
This section describes the functions for Logical Endpoint operations management.

Memory Management
This section describes the memory management functions of the CUDA runtime application programming interface.

Memory Management [DEPRECATED]
This section describes deprecated memory management functions of the CUDA runtime application programming interface.

Occupancy
This section describes the occupancy calculation functions of the CUDA runtime application programming interface.

OpenGL Interoperability
impl_private

OpenGL Interoperability [DEPRECATED]
This section describes deprecated OpenGL interoperability functionality.

Peer Device Memory Access
This section describes the peer device memory access functions of the CUDA runtime application programming interface.

Profiler Control
This section describes the profiler control functions of the CUDA runtime application programming interface.

Stream Management
This section describes the stream management functions of the CUDA runtime application programming interface.

Stream Ordered Memory Allocator

Surface Object Management
This section describes the low level texture object management functions of the CUDA runtime application programming interface.

Texture Object Management
This section describes the low level texture object management functions of the CUDA runtime application programming interface.

Unified Addressing
This section describes the unified addressing functions of the CUDA runtime application programming interface.

VDPAU Interoperability
This section describes the VDPAU interoperability functions of the CUDA runtime application programming interface.

Version Management

Chapter 7. Structs

7.1. CUuuid_st

struct **CUuuid_st**

Public Members

char **bytes**[16]

< CUDA definition of UUID

7.2. cudaAccessPolicyWindow

struct **cudaAccessPolicyWindow**

Specifies an access policy for a window, a contiguous extent of memory beginning at base_ptr and ending at base_ptr + num_bytes.

Partition into many segments and assign segments such that. sum of “hit segments” / window == approx. ratio. sum of “miss segments” / window == approx 1-ratio. Segments and ratio specifications are fitted to the capabilities of the architecture. Accesses in a hit segment apply the hitProp access policy. Accesses in a miss segment apply the missProp access policy.

Public Members

void ***base_ptr**

Starting address of the access policy window.

CUDA driver may align it.

enum *cudaAccessProperty* **hitProp**

::CUaccessProperty set for hit.

float **hitRatio**

hitRatio specifies percentage of lines assigned hitProp, rest are assigned missProp.

enum *cudaAccessProperty* **missProp**

::CUaccessProperty set for miss.

Must be either NORMAL or STREAMING.

size_t **num_bytes**

Size in bytes of the window policy.

CUDA driver may restrict the maximum size and alignment.

7.3. cudaArrayMemoryRequirements

struct **cudaArrayMemoryRequirements**

CUDA array and CUDA mipmapped array memory requirements.

Public Members

size_t **alignment**

Alignment necessary for mapping the array.

unsigned int **reserved**[4]

size_t **size**

Total size of the array.

7.4. cudaArraySparseProperties

struct **cudaArraySparseProperties**

Sparse CUDA array and CUDA mipmapped array properties.

Public Members

unsigned int **depth**
Tile depth in elements.

unsigned int **flags**
Flags will either be zero or *cudaArraySparsePropertiesSingleMipTail*.

unsigned int **height**
Tile height in elements.

unsigned int **miptailFirstLevel**
First mip level at which the mip tail begins.

unsigned long long **miptailSize**
Total size of the mip tail.

unsigned int **reserved**[4]

struct *cudaArraySparseProperties*::[anonymous] **tileExtent**

unsigned int **width**
Tile width in elements.

7.5. cudaAsyncNotificationInfo_t

struct **cudaAsyncNotificationInfo_t**
Information describing an async notification event.

Public Members

unsigned long long **bytesOverBudget**
The number of bytes that the process has allocated above its device memory budget.

union *cudaAsyncNotificationInfo_t*::[anonymous] **info**
Information about the notification.

`type` must be checked in order to interpret this field.

struct *cudaAsyncNotificationInfo_t*::[anonymous]::[anonymous] **overBudget**
Information about notifications of type `cudaAsyncNotificationTypeOverBudget`.

cudaAsyncNotificationType **type**

The type of notification being sent.

7.6. cudaChannelFormatDesc

struct **cudaChannelFormatDesc**

CUDA Channel format descriptor.

Public Members

enum *cudaChannelFormatKind* **f**

Channel format kind.

int **w**

w

int **x**

x

int **y**

y

int **z**

z

7.7. cudaChildGraphNodeParams

struct **cudaChildGraphNodeParams**

Child graph node parameters.

Public Members

cudaGraph_t **graph**

The child graph to clone into the node for node creation, or a handle to the graph owned by the node for node query.

The graph must not contain conditional nodes. Graphs containing memory allocation or memory free nodes must set the ownership to be moved to the parent.

enum *cudaGraphChildGraphNodeOwnership* **ownership**

The ownership relationship of the child graph node.

7.8. cudaConditionalNodeParams

struct **cudaConditionalNodeParams**

CUDA conditional node parameters.

Public Members

cudaExecutionContext_t **ctx**

CUDA Execution Context.

cudaGraphConditionalHandle **handle**

Conditional node handle.

Handles must be created in advance of creating the node using *cudaGraphConditionalHandleCreate*.

cudaGraph_t ***phGraph_out**

CUDA-owned array populated with conditional node child graphs during creation of the node.

Valid for the lifetime of the conditional node. The contents of the graph(s) are subject to the following constraints:

- Allowed node types are kernel nodes, empty nodes, child graphs, memsets, memcopies, and conditionals. This applies recursively to child graphs and conditional bodies.
- All kernels, including kernels in nested conditionals or child graphs at any level, must belong to the same device context.

These graphs may be populated using graph node creation APIs or *cudaStreamBeginCaptureToGraph*. cudaGraphCondTypeIf: phGraph_out[0] is executed when the condition is non-zero. If `size` == 2, phGraph_out[1] will be executed when the condition is zero. cudaGraphCondTypeWhile: phGraph_out[0] is executed as long as the condition is non-zero. cudaGraphCondTypeSwitch: phGraph_out[n] is executed when the condition is equal to n. If the condition >= `size`, no body graph is executed.

unsigned int **size**

Size of graph output array.

Allowed values are 1 for cudaGraphCondTypeWhile, 1 or 2 for cudaGraphCondTypeIf, or any value greater than zero for cudaGraphCondTypeSwitch.

enum *cudaGraphConditionalNodeType* **type**

Type of conditional node.

7.9. cudaDevResource

struct **cudaDevResource**

A tagged union describing different resources identified by the type field.

This structure should not be directly modified outside of the API that created it.

```
struct {
    enum cudaDevResourceType type;
    union {
        struct cudaDevSmResource sm;
        struct cudaDevWorkqueueConfigResource wqConfig;
        struct cudaDevWorkqueueResource wq;
    };
};
```

- If `type` is `cudaDevResourceTypeInvalid`, this resoure is not valid and cannot be further accessed.
- If `type` is `cudaDevResourceTypeSm`, the *cudaDevSmResource* structure `sm` is filled in. For example, `sm.smCount` will reflect the amount of streaming multiprocessors available in this resource.
- If `type` is `cudaDevResourceTypeWorkqueueConfig`, the *cudaDevWorkqueueConfigResource* structure `wqConfig` is filled in.
- If `type` is `cudaDevResourceTypeWorkqueue`, the *cudaDevWorkqueueResource* structure `wq` is filled in.

Public Members

union *cudaDevResource*::[anonymous] **[anonymous]**

unsigned char **_internal_padding**[92]

unsigned char **_oversize**[40]

struct cudaDevResource_st ***nextResource**

struct *cudaDevSmResource* **sm**

Resource corresponding to cudaDevResourceTypeSm `type`.

enum *cudaDevResourceType* **type**

Type of resource, dictates which union field was last set.

struct *cudaDevWorkqueueResource* **wq**

Resource corresponding to cudaDevResourceTypeWorkqueue `type`.

struct *cudaDevWorkqueueConfigResource* **wqConfig**

Resource corresponding to cudaDevResourceTypeWorkqueueConfig `type`.

7.10. cudaDevSmResource

struct **cudaDevSmResource**

Data for SM-related resources All parameters in this structure are OUTPUT only.

Do not write to any of the fields in this structure.

Public Members

unsigned int **flags**

The flags set on this SM resource.

For available flags see *cudaDevSmResourceGroup_flags*.

unsigned int **localityDomainId**

Locality domain that the SM must be located on.

Only valid if cudaDevSmResourceConstraintTypeLocalityDomainId is set in flags

unsigned int **minSmPartitionSize**

The minimum number of streaming multiprocessors required to partition this resource.

unsigned int **smCoscheduledAlignment**

The number of streaming multiprocessors in this resource that are guaranteed to be co-scheduled on the same GPU processing cluster.

smCount will be a multiple of this value, unless the backfill flag is set.

unsigned int **smCount**

The amount of streaming multiprocessors available in this resource.

7.11. cudaDevSmResourceGroupParams

struct **cudaDevSmResourceGroupParams**

Input data for splitting SMs.

Public Members

unsigned int **coscheduledSmCount**
: The amount of co-scheduled SMs grouped together for locality purposes.

unsigned int **flags**
: Combination of `cudaDevSmResourceGroup_flags` values to indicate this this group is created.

unsigned int **localityDomainId**
: Locality domain that the SM must be located on.

  Only valid if cudaDevSmResourceGroupLocalityDomainId is set in flags

unsigned int **preferredCoscheduledSmCount**
: When possible, combine co-scheduled groups together into larger groups of this size.

unsigned int **reserved**[11]

unsigned int **smCount**
: The amount of SMs available in this resource.

7.12. cudaDevWorkqueueConfigResource

struct **cudaDevWorkqueueConfigResource**
: Data for workqueue configuration related resources.

Public Members

int **device**
: The device on which the workqueue resources are available.

enum *cudaDevWorkqueueConfigScope* **sharingScope**
: The sharing scope for the workqueue resources.

unsigned int **wqConcurrencyLimit**
: The expected maximum number of concurrent stream-ordered workloads.

7.13. cudaDevWorkqueueResource

struct **cudaDevWorkqueueResource**

Handle to a pre-existing workqueue related resource.

Public Members

unsigned char **reserved**[40]

Reserved for future use.

7.14. cudaDeviceProp

struct **cudaDeviceProp**

CUDA device properties.

Public Members

int **accessPolicyMaxWindowSize**

The maximum value of *cudaAccessPolicyWindow::num_bytes*.

int **asyncEngineCount**

Number of asynchronous engines.

int **canMapHostMemory**

Device can map host memory with cudaHostAlloc/cudaHostGetDevicePointer.

int **canUseHostPointerForRegisteredMem**

Device can access host registered memory at the same virtual address as the CPU.

int **clusterLaunch**

Indicates device supports cluster launch.

int **computePreemptionSupported**

Device supports Compute Preemption.

int **concurrentKernels**

Device can possibly execute multiple kernels concurrently.

int **concurrentManagedAccess**

Device can coherently access managed memory concurrently with the CPU.

int **cooperativeLaunch**
Device supports launching cooperative kernels via *cudaLaunchCooperativeKernel*.

int **deferredMappingCudaArraySupported**
1 if the device supports deferred mapping CUDA arrays and CUDA mipmapped arrays

int **deviceNumaConfig**
NUMA configuration of a device: value is of type *cudaDeviceNumaConfig* enum.

int **deviceNumaId**
NUMA node ID of the GPU memory.

int **directManagedMemAccessFromHost**
Host can directly access managed memory on the device without migration.

int **ECCEnabled**
Device has ECC support enabled.

int **globalL1CacheSupported**
Device supports caching globals in L1.

unsigned int **gpuDirectRDMAFlushWritesOptions**
Bitmask to be interpreted according to the *cudaFlushGPUDirectRDMAWritesOptions* enum.

int **gpuDirectRDMASupported**
1 if the device supports GPUDirect RDMA APIs, 0 otherwise

int **gpuDirectRDMAWritesOrdering**
See the *cudaGPUDirectRDMAWritesOrdering* enum for numerical values.

unsigned int **gpuPciDeviceID**
The combined 16-bit PCI device ID and 16-bit PCI vendor ID.

unsigned int **gpuPciSubsystemID**
The combined 16-bit PCI subsystem ID and 16-bit PCI subsystem vendor ID.

int **hostNativeAtomicSupported**
Link between the device and the host supports native atomic operations.

int **hostNumaId**
NUMA ID of the host node closest to the device or -1 when system does not support NUMA.

int **hostNumaMultinodeIpcSupported**
1 if the device supports HostNuma location IPC between nodes in a multi-node system.

int **hostRegisterReadOnlySupported**

Device supports using the *cudaHostRegister* flag cudaHostRegisterReadOnly to register memory that must be mapped as read-only to the GPU.

int **hostRegisterSupported**

Device supports host memory registration via *cudaHostRegister*.

int **integrated**

Device is integrated as opposed to discrete.

int **ipcEventSupported**

Device supports IPC Events.

int **isMultiGpuBoard**

Device is on a multi-GPU board.

int **l2CacheSize**

Size of L2 cache in bytes.

int **localL1CacheSupported**

Device supports caching locals in L1.

char **luid**[8]

8-byte locally unique identifier.

Value is undefined on TCC and non-Windows platforms

unsigned int **luidDeviceNodeMask**

LUID device node mask.

Value is undefined on TCC and non-Windows platforms

int **major**

Major compute capability.

int **managedMemory**

Device supports allocating managed memory on this system.

int **maxBlocksPerMultiProcessor**

Maximum number of resident blocks per multiprocessor.

int **maxGridSize**[3]

Maximum size of each dimension of a grid.

int **maxSurface1D**

Maximum 1D surface size.

int **maxSurface1DLayered**[2]
Maximum 1D layered surface dimensions.

int **maxSurface2D**[2]
Maximum 2D surface dimensions.

int **maxSurface2DLayered**[3]
Maximum 2D layered surface dimensions.

int **maxSurface3D**[3]
Maximum 3D surface dimensions.

int **maxSurfaceCubemap**
Maximum Cubemap surface dimensions.

int **maxSurfaceCubemapLayered**[2]
Maximum Cubemap layered surface dimensions.

int **maxTexture1D**
Maximum 1D texture size.

int **maxTexture1DLayered**[2]
Maximum 1D layered texture dimensions.

int **maxTexture1DMipmap**
Maximum 1D mipmapped texture size.

int **maxTexture2D**[2]
Maximum 2D texture dimensions.

int **maxTexture2DGather**[2]
Maximum 2D texture dimensions if texture gather operations have to be performed.

int **maxTexture2DLayered**[3]
Maximum 2D layered texture dimensions.

int **maxTexture2DLinear**[3]
Maximum dimensions (width, height, pitch) for 2D textures bound to pitched memory.

int **maxTexture2DMipmap**[2]
Maximum 2D mipmapped texture dimensions.

int **maxTexture3D**[3]
Maximum 3D texture dimensions.

int **maxTexture3DAlt**[3]

Maximum alternate 3D texture dimensions.

int **maxTextureCubemap**

Maximum Cubemap texture dimensions.

int **maxTextureCubemapLayered**[2]

Maximum Cubemap layered texture dimensions.

int **maxThreadsDim**[3]

Maximum size of each dimension of a block.

int **maxThreadsPerBlock**

Maximum number of threads per block.

int **maxThreadsPerMultiProcessor**

Maximum resident threads per multiprocessor.

int **memoryBusWidth**

Global memory bus width in bits.

int **memoryPoolsSupported**

1 if the device supports using the cudaMallocAsync and cudaMemPool family of APIs, 0 otherwise

unsigned int **memoryPoolSupportedHandleTypes**

Bitmask of handle types supported with mempool-based IPC.

size_t **memPitch**

Maximum pitch in bytes allowed by memory copies.

int **minor**

Minor compute capability.

int **mpsEnabled**

Indicates if contexts created on this device will be shared via MPS.

int **multiGpuBoardGroupID**

Unique identifier for a group of devices on the same multi-GPU board.

int **multiProcessorCount**

Number of multiprocessors on device.

char **name**[256]

ASCII string identifying device.

int **pageableMemoryAccess**
: Device supports coherently accessing pageable memory without calling cudaHostRegister on it.

int **pageableMemoryAccessUsesHostPageTables**
: Device accesses pageable memory via the host's page tables.

int **pciBusID**
: PCI bus ID of the device.

int **pciDeviceID**
: PCI device ID of the device.

int **pciDomainID**
: PCI domain ID of the device.

int **persistingL2CacheMaxSize**
: Device's maximum l2 persisting lines capacity setting in bytes.

int **regsPerBlock**
: 32-bit registers available per block

int **regsPerMultiprocessor**
: 32-bit registers available per multiprocessor

int **reserved**[56]
: Reserved for future use.

size_t **reservedSharedMemPerBlock**
: Shared memory reserved by CUDA driver per block in bytes.

size_t **sharedMemPerBlock**
: Shared memory available per block in bytes.

size_t **sharedMemPerBlockOptin**
: Per device maximum shared memory per block usable by special opt in.

size_t **sharedMemPerMultiprocessor**
: Shared memory available per multiprocessor in bytes.

int **sparseCudaArraySupported**
: 1 if the device supports sparse CUDA arrays and sparse CUDA mipmapped arrays, 0 otherwise

int **streamPrioritiesSupported**
: Device supports stream priorities.

size_t **surfaceAlignment**
: Alignment requirements for surfaces.

int **tccDriver**
: 1 if device is a Tesla device using TCC driver, 0 otherwise

size_t **textureAlignment**
: Alignment requirement for textures.

size_t **texturePitchAlignment**
: Pitch alignment requirement for texture references bound to pitched memory.

int **timelineSemaphoreInteropSupported**
: External timeline semaphore interop is supported on the device.

size_t **totalConstMem**
: Constant memory available on device in bytes.

size_t **totalGlobalMem**
: Global memory available on device in bytes.

int **unifiedAddressing**
: Device shares a unified address space with the host.

int **unifiedFunctionPointers**
: Indicates device supports unified pointers.

cudaUUID_t **uuid**
: 16-byte unique identifier

int **warpSize**
: Warp size in threads.

7.15. cudaEglFrame

struct **cudaEglFrame**
: CUDA EGLFrame Descriptor - structure defining one frame of EGL.

  Each frame may contain one or more planes depending on whether the surface is Multiplanar or not. Each plane of EGLFrame is represented by *cudaEglPlaneDesc* which is defined as:

```
typedef struct cudaEglPlaneDesc_st {
    unsigned int width;
    unsigned int height;
```

(continues on next page)

(continued from previous page)

```
    unsigned int depth;
    unsigned int pitch;
    unsigned int numChannels;
    struct cudaChannelFormatDesc channelDesc;
    unsigned int reserved[4];
} cudaEglPlaneDesc;
```

Public Members

cudaEglColorFormat **eglColorFormat**

CUDA EGL Color Format.

union *cudaEglFrame*::[anonymous] **frame**

cudaEglFrameType **frameType**

Array or Pitch.

cudaArray_t **pArray**[3]

Array of CUDA arrays corresponding to each plane.

unsigned int **planeCount**

Number of planes.

cudaEglPlaneDesc **planeDesc**[3]

CUDA EGL Plane Descriptor *cudaEglPlaneDesc*.

struct *cudaPitchedPtr* **pPitch**[3]

Array of Pointers corresponding to each plane.

7.16. cudaEglPlaneDesc

struct **cudaEglPlaneDesc**

CUDA EGL Plane Descriptor - structure defining each plane of a CUDA EGLFrame.

Public Members

struct *cudaChannelFormatDesc* **channelDesc**
Channel Format Descriptor.

unsigned int **depth**
Depth of plane.

unsigned int **height**
Height of plane.

unsigned int **numChannels**
Number of channels for the plane.

unsigned int **pitch**
Pitch of plane.

unsigned int **reserved**[4]
Reserved for future use.

unsigned int **width**
Width of plane.

7.17. cudaEventRecordNodeParams

struct **cudaEventRecordNodeParams**
Event record node parameters.

Public Members

cudaExecutionContext_t **ctx**
CUDA Execution Context.

cudaEvent_t **event**
The event to record when the node executes.

7.18. cudaEventWaitNodeParams

struct **cudaEventWaitNodeParams**

Event wait node parameters.

Public Members

cudaEvent_t **event**

The event to wait on from the node.

7.19. cudaExtent

struct **cudaExtent**

CUDA extent.

See also:

make_cudaExtent

Public Members

size_t **depth**

Depth in elements.

size_t **height**

Height in elements.

size_t **width**

Width in elements when referring to array memory, in bytes when referring to linear memory.

7.20. cudaExternalMemoryBufferDesc

struct **cudaExternalMemoryBufferDesc**

External memory buffer descriptor.

Public Members

unsigned int **flags**
: Flags reserved for future use.

  Must be zero.

unsigned long long **offset**
: Offset into the memory object where the buffer's base is.

unsigned int **reserved**[16]
: Must be zero.

unsigned long long **size**
: Size of the buffer.

7.21. cudaExternalMemoryHandleDesc

struct **cudaExternalMemoryHandleDesc**
: External memory handle descriptor.

Public Members

int **fd**
: File descriptor referencing the memory object.

  Valid when type is *cudaExternalMemoryHandleTypeOpaqueFd*

unsigned int **flags**
: Flags must either be zero or *cudaExternalMemoryDedicated*.

void ***handle**
: Valid NT handle.

  Must be NULL if 'name' is non-NULL

union *cudaExternalMemoryHandleDesc*::[anonymous] **handle**

const void ***name**
: Name of a valid memory object.

  Must be NULL if 'handle' is non-NULL.

const void ***nvSciBufObject**

A handle representing NvSciBuf Object.

Valid when type is *cudaExternalMemoryHandleTypeNvSciBuf*

unsigned int **reserved**[16]

Must be zero.

unsigned long long **size**

Size of the memory allocation.

enum *cudaExternalMemoryHandleType* **type**

Type of the handle.

struct *cudaExternalMemoryHandleDesc*::[anonymous]::[anonymous] **win32**

Win32 handle referencing the semaphore object.

Valid when type is one of the following:

- *cudaExternalMemoryHandleTypeOpaqueWin32*
- *cudaExternalMemoryHandleTypeOpaqueWin32Kmt*
- *cudaExternalMemoryHandleTypeD3D12Heap*
- *cudaExternalMemoryHandleTypeD3D12Resource*
- *cudaExternalMemoryHandleTypeD3D11Resource*
- *cudaExternalMemoryHandleTypeD3D11ResourceKmt* Exactly one of 'handle' and 'name' must be non-NULL. If type is one of the following: *cudaExternalMemoryHandleTypeOpaqueWin32Kmt* *cudaExternalMemoryHandleTypeD3D11ResourceKmt* then 'name' must be NULL.

7.22. cudaExternalMemoryMipmappedArrayDesc

struct **cudaExternalMemoryMipmappedArrayDesc**

External memory mipmap descriptor.

Public Members

struct *cudaExtent* **extent**

Dimensions of base level of the mipmap chain.

unsigned int **flags**

Flags associated with CUDA mipmapped arrays.

See *cudaMallocMipmappedArray*

struct *cudaChannelFormatDesc* **formatDesc**
: Format of base level of the mipmap chain.

unsigned int **numLevels**
: Total number of levels in the mipmap chain.

unsigned long long **offset**
: Offset into the memory object where the base level of the mipmap chain is.

unsigned int **reserved**[16]
: Must be zero.

7.23. cudaExternalSemaphoreHandleDesc

struct **cudaExternalSemaphoreHandleDesc**
: External semaphore handle descriptor.

Public Members

int **fd**
: File descriptor referencing the semaphore object.

Valid when type is one of the following:

- *cudaExternalSemaphoreHandleTypeOpaqueFd*
- *cudaExternalSemaphoreHandleTypeTimelineSemaphoreFd*

unsigned int **flags**
: Flags reserved for the future.

Must be zero.

void ***handle**
: Valid NT handle.

Must be NULL if 'name' is non-NULL

union *cudaExternalSemaphoreHandleDesc*::[anonymous] **handle**

const void ***name**
: Name of a valid synchronization primitive.

Must be NULL if 'handle' is non-NULL.

const void ***nvSciSyncObj**

Valid NvSciSyncObj.

Must be non NULL

unsigned int **reserved**[16]

Must be zero.

enum *cudaExternalSemaphoreHandleType* **type**

Type of the handle.

struct *cudaExternalSemaphoreHandleDesc*::[anonymous]::[anonymous] **win32**

Win32 handle referencing the semaphore object.

Valid when type is one of the following:

- *cudaExternalSemaphoreHandleTypeOpaqueWin32*
- *cudaExternalSemaphoreHandleTypeOpaqueWin32Kmt*
- *cudaExternalSemaphoreHandleTypeD3D12Fence*
- *cudaExternalSemaphoreHandleTypeD3D11Fence*
- *cudaExternalSemaphoreHandleTypeKeyedMutex*
- *cudaExternalSemaphoreHandleTypeTimelineSemaphoreWin32* Exactly one of 'handle' and 'name' must be non-NULL. If type is one of the following: *cudaExternalSemaphore-HandleTypeOpaqueWin32Kmt cudaExternalSemaphoreHandleTypeKeyedMutexKmt* then 'name' must be NULL.

7.24. cudaExternalSemaphoreSignalNodeParams

struct **cudaExternalSemaphoreSignalNodeParams**

External semaphore signal node parameters.

Public Members

cudaExternalSemaphore_t ***extSemArray**

Array of external semaphore handles.

unsigned int **numExtSems**

Number of handles and parameters supplied in extSemArray and paramsArray.

const struct *cudaExternalSemaphoreSignalParams* ***paramsArray**

Array of external semaphore signal parameters.

7.25. cudaExternalSemaphoreSignalNodeParamsV2

struct **cudaExternalSemaphoreSignalNodeParamsV2**

External semaphore signal node parameters.

Public Members

cudaExecutionContext_t **ctx**

CUDA Execution Context.

cudaExternalSemaphore_t ***extSemArray**

Array of external semaphore handles.

unsigned int **numExtSems**

Number of handles and parameters supplied in extSemArray and paramsArray.

const struct *cudaExternalSemaphoreSignalParams* ***paramsArray**

Array of external semaphore signal parameters.

7.26. cudaExternalSemaphoreSignalParams

struct **cudaExternalSemaphoreSignalParams**

External semaphore signal parameters, compatible with driver type.

Public Members

struct *cudaExternalSemaphoreSignalParams*::[anonymous]::[anonymous] **fence**

Parameters for fence objects.

void ***fence**

Pointer to NvSciSyncFence.

Valid if *cudaExternalSemaphoreHandleType* is of type *cudaExternalSemaphoreHandleTypeNvSciSync*.

unsigned int **flags**

Only when *cudaExternalSemaphoreSignalParams* is used to signal a *cudaExternalSemaphore_t* of type *cudaExternalSemaphoreHandleTypeNvSciSync*, the valid flag is *cudaExternalSemaphoreSignalSkipNvSciBufMemSync*: which indicates that while signaling the *cudaExternalSemaphore_t*, no memory synchronization operations should be performed for any external memory object imported as *cudaExternalMemoryHandleTypeNvSciBuf*.

For all other types of *cudaExternalSemaphore_t*, flags must be zero.

unsigned long long **key**

struct *cudaExternalSemaphoreSignalParams*::[anonymous]::[anonymous] **keyedMutex**
Parameters for keyed mutex objects.

union *cudaExternalSemaphoreSignalParams*::[anonymous]::[anonymous] **nvSciSync**

struct *cudaExternalSemaphoreSignalParams*::[anonymous] **params**

unsigned long long **reserved**

unsigned int **reserved**[12]

unsigned long long **value**
Value of fence to be signaled.

7.27. cudaExternalSemaphoreWaitNodeParams

struct **cudaExternalSemaphoreWaitNodeParams**
External semaphore wait node parameters.

Public Members

cudaExternalSemaphore_t ***extSemArray**
Array of external semaphore handles.

unsigned int **numExtSems**
Number of handles and parameters supplied in extSemArray and paramsArray.

const struct *cudaExternalSemaphoreWaitParams* ***paramsArray**
Array of external semaphore wait parameters.

7.28. cudaExternalSemaphoreWaitNodeParamsV2

struct **cudaExternalSemaphoreWaitNodeParamsV2**
External semaphore wait node parameters.

Public Members

cudaExecutionContext_t **ctx**

CUDA Execution Context.

cudaExternalSemaphore_t ***extSemArray**

Array of external semaphore handles.

unsigned int **numExtSems**

Number of handles and parameters supplied in extSemArray and paramsArray.

const struct *cudaExternalSemaphoreWaitParams* ***paramsArray**

Array of external semaphore wait parameters.

7.29. cudaExternalSemaphoreWaitParams

struct **cudaExternalSemaphoreWaitParams**

External semaphore wait parameters, compatible with driver type.

Public Members

struct *cudaExternalSemaphoreWaitParams*::[anonymous]::[anonymous] **fence**

Parameters for fence objects.

void ***fence**

Pointer to NvSciSyncFence.

Valid if *cudaExternalSemaphoreHandleType* is of type *cudaExternalSemaphoreHandleTypeNvSciSync*.

unsigned int **flags**

Only when *cudaExternalSemaphoreSignalParams* is used to signal a *cudaExternalSemaphore_t* of type *cudaExternalSemaphoreHandleTypeNvSciSync*, the valid flag is *cudaExternalSemaphoreSignalSkipNvSciBufMemSync*: which indicates that while waiting for the *cudaExternalSemaphore_t*, no memory synchronization operations should be performed for any external memory object imported as *cudaExternalMemoryHandleTypeNvSciBuf*.

For all other types of *cudaExternalSemaphore_t*, flags must be zero.

unsigned long long **key**

Value of key to acquire the mutex with.

struct *cudaExternalSemaphoreWaitParams*::[anonymous]::[anonymous] **keyedMutex**

Parameters for keyed mutex objects.

union *cudaExternalSemaphoreWaitParams*::[anonymous]::[anonymous] **nvSciSync**

struct *cudaExternalSemaphoreWaitParams*::[anonymous] **params**

unsigned long long **reserved**

unsigned int **reserved**[10]

unsigned int **timeoutMs**

Timeout in milliseconds to wait to acquire the mutex.

unsigned long long **value**

Value of fence to be waited on.

7.30. cudaFuncAttributes

struct **cudaFuncAttributes**

CUDA function attributes.

Public Members

int **binaryVersion**

The binary architecture version for which the function was compiled.

This value is the major binary version * 10 + the minor binary version, so a binary version 1.3 function would return the value 13.

int **cacheModeCA**

The attribute to indicate whether the function has been compiled with user specified option "-Xptxas –dlcm=ca" set.

int **clusterDimMustBeSet**

If this attribute is set, the kernel must launch with a valid cluster dimension specified.

int **clusterSchedulingPolicyPreference**

The block scheduling policy of a function.

See *cudaFuncSetAttribute*

size_t **constSizeBytes**

The size in bytes of user-allocated constant memory required by this function.

int **deviceNodeUpdateStatus**

Whether the function can be updated on device.

1 means device node update is supported, 0 is unsupported or driver is too old to check the value.

size_t **localSizeBytes**

The size in bytes of local memory used by each thread of this function.

int **maxDynamicSharedSizeBytes**

The maximum size in bytes of dynamic shared memory per block for this function.

Any launch must have a dynamic shared memory size smaller than this value.

This attribute is ignored if the sharedMemoryMode function or launch attribute is set.

This attribute cannot be used to access oversized shared memory. Oversized shared memory can only be accessed by setting the shared memory mode.

See *cudaFuncSetAttribute*

int **maxThreadsPerBlock**

The maximum number of threads per block, beyond which a launch of the function would fail.

This number depends on both the function and the device on which the function is currently loaded.

int **nonPortableClusterSizeAllowed**

Whether the function can be launched with non-portable cluster size.

1 is allowed, 0 is disallowed. A non-portable cluster size may only function on the specific SKUs the program is tested on. The launch might fail if the program is run on a different hardware platform.

CUDA API provides *cudaOccupancyMaxActiveClusters* to assist with checking whether the desired size can be launched on the current device.

Portable Cluster Size

A portable cluster size is guaranteed to be functional on all compute capabilities higher than the target compute capability. The portable cluster size for sm_90 is 8 blocks per cluster. This value may increase for future compute capabilities.

The specific hardware unit may support higher cluster sizes that's not guaranteed to be portable.

See *cudaFuncSetAttribute*

int **numRegs**

The number of registers used by each thread of this function.

int **preferredShmemCarveout**

On devices where the L1 cache and shared memory use the same hardware resources, this sets the shared memory carveout preference, in percent of the maximum shared memory.

Refer to *cudaDevAttrMaxSharedMemoryPerMultiprocessor*. This is only a hint, and the driver can choose a different ratio if required to execute the function.

See *cudaFuncSetAttribute*

int **ptxVersion**

The PTX virtual architecture version for which the function was compiled.

This value is the major PTX version * 10 + the minor PTX version, so a PTX version 1.3 function would return the value 13.

int **requiredClusterDepth**

int **requiredClusterHeight**

int **requiredClusterWidth**

The required cluster width/height/depth in blocks.

The values must either all be 0 or all be positive. The validity of the cluster dimensions is otherwise checked at launch time.

If the value is set during compile time, it cannot be set at runtime. Setting it at runtime should return cudaErrorNotPermitted.

See *cudaFuncSetAttribute*

int **reserved**[14]

Reserved for future use.

cudaSharedMemoryMode **sharedMemoryMode**

This controls a kernel's use of non-portable or oversized shared memory configurations.

See *cudaFuncSetAttribute*

size_t **sharedSizeBytes**

The size in bytes of statically-allocated shared memory per block required by this function.

This does not include dynamically-allocated shared memory requested by the user at run-time.

7.31. cudaGraphEdgeData

struct **cudaGraphEdgeData**

Optional annotation for edges in a CUDA graph.

Note, all edges implicitly have annotations and default to a zero-initialized value if not specified. A zero-initialized struct indicates a standard full serialization of two nodes with memory visibility.

Public Members

unsigned char **from_port**

This indicates when the dependency is triggered from the upstream node on the edge.

The meaning is specfic to the node type. A value of 0 in all cases means full completion of the upstream node, with memory visibility to the downstream node or portion thereof (indicated by `to_port`).

Only kernel nodes define non-zero ports. A kernel node can use the following output port types: *cudaGraphKernelNodePortDefault*, *cudaGraphKernelNodePortProgrammatic*, or *cudaGraphKernelNodePortLaunchCompletion*.

unsigned char **reserved**[5]

These bytes are unused and must be zeroed.

This ensures compatibility if additional fields are added in the future.

unsigned char **to_port**

This indicates what portion of the downstream node is dependent on the upstream node or portion thereof (indicated by `from_port`).

The meaning is specific to the node type. A value of 0 in all cases means the entirety of the downstream node is dependent on the upstream work.

Currently no node types define non-zero ports. Accordingly, this field must be set to zero.

unsigned char **type**

This should be populated with a value from *cudaGraphDependencyType*.

(It is typed as char due to compiler-specific layout of bitfields.) See *cudaGraphDependencyType*.

7.32. cudaGraphExecUpdateResultInfo

struct **cudaGraphExecUpdateResultInfo**

Result information returned by cudaGraphExecUpdate.

Public Members

cudaGraphNode_t **errorFromNode**

The from node of error edge when the topologies do not match.

Otherwise NULL.

cudaGraphNode_t **errorNode**

The “to node” of the error edge when the topologies do not match.

The error node when the error is associated with a specific node. NULL when the error is generic.

enum *cudaGraphExecUpdateResult* **result**

Gives more specific detail when a cuda graph update fails.

7.33. cudaGraphInstantiateParams

struct **cudaGraphInstantiateParams**

Graph instantiation parameters.

Public Members

cudaGraphNode_t **errNode_out**

The node which caused instantiation to fail, if any.

unsigned long long **flags**

Instantiation flags.

cudaGraphInstantiateResult **result_out**

Whether instantiation was successful.

If it failed, the reason why

cudaStream_t **uploadStream**

Upload stream.

7.34. cudaGraphKernelNodeUpdate

struct **cudaGraphKernelNodeUpdate**

Struct to specify a single node update to pass as part of a larger array to *cudaGraphKernelNodeUpdatesApply*.

Public Members

enum *cudaGraphKernelNodeField* **field**

Which type of update to apply.

Determines how updateData is interpreted

uint3 **gridDim**

Grid dimensions.

unsigned int **isEnabled**

Node enable/disable data.

Nonzero if the node should be enabled, 0 if it should be disabled

cudaGraphDeviceNode_t **node**

Node to update.

size_t **offset**

Offset into the parameter buffer at which to apply the update.

struct *cudaGraphKernelNodeUpdate*::[anonymous]::[anonymous] **param**

Kernel parameter data.

const void ***pValue**

Kernel parameter data to write in.

size_t **size**

Number of bytes to update.

union *cudaGraphKernelNodeUpdate*::[anonymous] **updateData**

Update data to apply.

Which field is used depends on field's value

7.35. cudaGraphNodeParams

struct **cudaGraphNodeParams**

Graph node parameters.

See *cudaGraphAddNode*.

Public Members

union *cudaGraphNodeParams*::[anonymous] **[anonymous]**

struct *cudaMemAllocNodeParamsV2* **alloc**

Memory allocation node parameters.

struct *cudaConditionalNodeParams* **conditional**

Conditional node parameters.

struct *cudaEventRecordNodeParams* **eventRecord**

Event record node parameters.

struct *cudaEventWaitNodeParams* **eventWait**

Event wait node parameters.

struct *cudaExternalSemaphoreSignalNodeParamsV2* **extSemSignal**

External semaphore signal node parameters.

struct *cudaExternalSemaphoreWaitNodeParamsV2* **extSemWait**

External semaphore wait node parameters.

struct *cudaMemFreeNodeParams* **free**

Memory free node parameters.

struct *cudaChildGraphNodeParams* **graph**

Child graph node parameters.

struct *cudaHostNodeParamsV2* **host**

Host node parameters.

struct *cudaKernelNodeParamsV2* **kernel**

Kernel node parameters.

struct *cudaMemcpyNodeParams* **memcpy**

Memcpy node parameters.

struct *cudaMemcpyBatchNodeParams* **memcpyBatch**

Batch memcpy node parameters.

struct *cudaMemsetParamsV2* **memset**

Memset node parameters.

int **reserved0**[3]

Reserved.

Must be zero.

long long **reserved1**[29]

Padding.

Unused bytes must be zero.

long long **reserved2**

Reserved bytes.

Must be zero.

enum *cudaGraphNodeType* **type**

Type of the node.

7.36. cudaGraphRecaptureCallbackData

struct **cudaGraphRecaptureCallbackData**

Struct of user callback data that is invoked when node parameter mismatches are detected while recapturing to an existing graph.

Public Members

cudaGraphRecaptureCallback_t **callbackFunc**

Callback function that will be invoked.

void ***userData**

Generic pointer that is passed to the callback function.

7.37. cudaHostNodeParams

struct **cudaHostNodeParams**

CUDA host node parameters.

Public Members

cudaHostFn_t **fn**

The function to call when the node executes.

void ***userData**

Argument to pass to the function.

7.38. cudaHostNodeParamsV2

struct **cudaHostNodeParamsV2**

CUDA host node parameters.

Public Members

cudaExecutionContext_t **ctx**

CUDA Execution Context.

cudaHostFn_t **fn**

The function to call when the node executes.

unsigned int **syncMode**

The synchronization mode to use for the host task.

void ***userData**

Argument to pass to the function.

7.39. cudaIpcEventHandle_t

struct **cudaIpcEventHandle_t**

CUDA IPC event handle.

Public Members

char **reserved**[64]

7.40. cudaIpcMemHandle_t

struct **cudaIpcMemHandle_t**

CUDA IPC memory handle.

Public Members

char **reserved**[64]

7.41. cudaKernelNodeParams

struct **cudaKernelNodeParams**

CUDA GPU kernel node parameters.

Public Members

dim3 **blockDim**

Block dimensions.

void ****extra**

Pointer to kernel arguments in the “extra” format.

void ***func**

Kernel to launch.

dim3 **gridDim**

Grid dimensions.

void ****kernelParams**

Array of pointers to individual kernel arguments.

unsigned int **sharedMemBytes**

Dynamic shared-memory size per thread block in bytes.

7.42. cudaKernelNodeParamsV2

struct **cudaKernelNodeParamsV2**

CUDA GPU kernel node parameters.

Public Members

union *cudaKernelNodeParamsV2*::[anonymous] **[anonymous]**

uint3 **blockDim**

Block dimensions.

cudaExecutionContext_t **ctx**

Context in which to run the kernel.

If NULL will try to use the current context.

cudaFunction_t **cuFunc**

functionType = cudaKernelFucntionTypeFunction

void ****extra**

Pointer to kernel arguments in the “extra” format.

void ***func**

functionType = cudaKernelFucntionTypeDevice

enum *cudaKernelFunctionType* **functionType**

Type of handle passed in the func/kern/cuFunc union above.

uint3 **gridDim**

Grid dimensions.

cudaKernel_t **kern**

functionType = cudaKernelFucntionTypeKernel

void ****kernelParams**
Array of pointers to individual kernel arguments.

unsigned int **sharedMemBytes**
Dynamic shared-memory size per thread block in bytes.

7.43. cudaLaunchAttribute

struct **cudaLaunchAttribute**
Launch attribute.

Public Members

cudaLaunchAttributeID **id**
Attribute to set.

char **pad**[8 - sizeof(*cudaLaunchAttributeID*)]

cudaLaunchAttributeValue **val**
Value of the attribute.

7.44. cudaLaunchConfig_t

struct **cudaLaunchConfig_t**
CUDA extensible launch configuration.

Public Members

cudaLaunchAttribute ***attrs**
List of attributes; nullable if *cudaLaunchConfig_t::numAttrs* == 0.

dim3 **blockDim**
Block dimensions.

size_t **dynamicSmemBytes**
Dynamic shared-memory size per thread block in bytes.

dim3 **gridDim**
Grid dimensions.

unsigned int **numAttrs**

Number of attributes populated in *cudaLaunchConfig_t::attrs*.

cudaStream_t **stream**

Stream identifier.

7.45. cudaLaunchMemSyncDomainMap

struct **cudaLaunchMemSyncDomainMap**

Memory Synchronization Domain map.

See *cudaLaunchMemSyncDomain*.

By default, kernels are launched in domain 0. Kernel launched with *cudaLaunchMemSyncDomainRemote* will have a different domain ID. User may also alter the domain ID with *cudaLaunchMemSyncDomainMap* for a specific stream / graph node / kernel launch. See *cudaLaunchAttributeMemSyncDomainMap*.

Domain ID range is available through *cudaDevAttrMemSyncDomainCount*.

Public Members

unsigned char **default_**

The default domain ID to use for designated kernels.

unsigned char **remote**

The remote domain ID to use for designated kernels.

7.46. cudaMemAccessDesc

struct **cudaMemAccessDesc**

Memory access descriptor.

Public Members

enum *cudaMemAccessFlags* **flags**

::CUmemProt accessibility flags to set on the request

struct *cudaMemLocation* **location**

Location on which the request is to change it's accessibility.

7.47. cudaMemAllocNodeParams

struct **cudaMemAllocNodeParams**

Memory allocation node parameters.

Public Members

size_t **accessDescCount**

in: Number of accessDescss

const struct *cudaMemAccessDesc* ***accessDescs**

in: number of memory access descriptors.

Must not exceed the number of GPUs.

size_t **bytesize**

in: size in bytes of the requested allocation

void ***dptr**

out: address of the allocation returned by CUDA

struct *cudaMemPoolProps* **poolProps**

in: location where the allocation should reside (specified in ::location).

::handleTypes must be *cudaMemHandleTypeNone*. IPC is not supported. in: array of memory access descriptors. Used to describe peer GPU access

7.48. cudaMemAllocNodeParamsV2

struct **cudaMemAllocNodeParamsV2**

Memory allocation node parameters.

Public Members

size_t **accessDescCount**

in: Number of accessDescss

const struct *cudaMemAccessDesc* ***accessDescs**

in: number of memory access descriptors.

Must not exceed the number of GPUs.

size_t **bytesize**

in: size in bytes of the requested allocation

void ***dptr**

out: address of the allocation returned by CUDA

struct *cudaMemPoolProps* **poolProps**

in: location where the allocation should reside (specified in ::location).

::handleTypes must be *cudaMemHandleTypeNone*. IPC is not supported. in: array of memory access descriptors. Used to describe peer GPU access

7.49. cudaMemFabricHandle_t

struct **cudaMemFabricHandle_t**

Public Members

char **reserved**[64]

7.50. cudaMemFreeNodeParams

struct **cudaMemFreeNodeParams**

Memory free node parameters.

Public Members

void ***dptr**

in: the pointer to free

7.51. cudaMemLocation

struct **cudaMemLocation**

Specifies a memory location.

To specify a gpu, set type = *cudaMemLocationTypeDevice* and set id = the gpu's device ordinal. To specify a cpu NUMA node, set type = *cudaMemLocationTypeHostNuma* and set id = host NUMA node id.

Public Members

union *cudaMemLocation*::[anonymous] **[anonymous]**

unsigned char **deviceId**

Device ID.

int **id**

Identifier for cudaMemLocationType::cudaMemLocationTypeDevice, cudaMemLocationType::cudaMemLocationTypeHost, or cudaMemLocationType::cudaMemLocationTypeHostNuma.

unsigned char **localityDomainId**

Locality domain ID.

struct *cudaMemLocation*::[anonymous]::[anonymous] **localized**

Identifier for cudaMemLocationType::cudaMemLocationTypeDeviceLocalityDomain.

enum *cudaMemLocationType* **type**

Specifies the location type, which modifies the meaning of id.

7.52. cudaMemPoolProps

struct **cudaMemPoolProps**

Specifies the properties of allocations made from the pool.

Public Members

enum *cudaMemAllocationType* **allocType**

Allocation type.

Currently must be specified as cudaMemAllocationTypePinned

enum *cudaMemAllocationHandleType* **handleTypes**

Handle types that will be supported by allocations from the pool.

struct *cudaMemLocation* **location**

Location allocations should reside.

size_t **maxSize**

Maximum pool size.

When set to 0, defaults to a system dependent value.

unsigned char **reserved**[53]

reserved for future use, must be 0

unsigned char **reserved1**

reserved for future use, must be 0

unsigned short **usage**

Bitmask indicating intended usage for the pool.

void ***win32SecurityAttributes**

Windows-specific LPSECURITYATTRIBUTES required when *cudaMemHandleTypeWin32* is specified.

This security attribute defines the scope of which exported allocations may be tranferred to other processes. In all other cases, this field is required to be zero.

7.53. cudaMemPoolPtrExportData

struct **cudaMemPoolPtrExportData**

Opaque data for exporting a pool allocation.

Public Members

unsigned char **reserved**[64]

7.54. cudaMemcpy3DBatchOp

struct **cudaMemcpy3DBatchOp**

Public Members

struct *cudaMemcpy3DOperand* **dst**

Destination memcpy operand.

struct *cudaExtent* **extent**

Extents of the memcpy between src and dst.

The width, height and depth components must not be 0.

unsigned int **flags**

Additional flags for copy from src to dst.

See *cudaMemcpyFlags*.

struct *cudaMemcpy3DOperand* **src**

Source memcpy operand.

enum *cudaMemcpySrcAccessOrder* **srcAccessOrder**

Source access ordering to be observed for copy from src to dst.

7.55. cudaMemcpy3DOperand

struct **cudaMemcpy3DOperand**

Struct representing an operand for copy with *cudaMemcpy3DBatchAsync*.

Public Members

cudaArray_t **array**

struct *cudaMemcpy3DOperand*::[anonymous]::[anonymous] **array**

Struct representing an operand when cudaMemcpy3DOperand::type is *cudaMemcpyOperandTypeArray*.

size_t **layerHeight**

Height of each layer in elements.

struct *cudaMemLocation* **locHint**

Hint location for the operand.

Ignored when the pointers are not managed memory or memory allocated outside CUDA.

struct *cudaOffset3D* **offset**

union *cudaMemcpy3DOperand*::[anonymous] **op**

void ***ptr**

struct *cudaMemcpy3DOperand*::[anonymous]::[anonymous] **ptr**

Struct representing an operand when cudaMemcpy3DOperand::type is *cudaMemcpyOperandTypePointer*.

size_t **rowLength**
: Length of each row in elements.

enum *cudaMemcpy3DOperandType* **type**

7.56. cudaMemcpy3DParms

struct **cudaMemcpy3DParms**
: CUDA 3D memory copying parameters.

Public Members

cudaArray_t **dstArray**
: Destination memory address.

struct *cudaPos* **dstPos**
: Destination position offset.

struct *cudaPitchedPtr* **dstPtr**
: Pitched destination memory address.

struct *cudaExtent* **extent**
: Requested memory copy size.

enum *cudaMemcpyKind* **kind**
: Type of transfer.

cudaArray_t **srcArray**
: Source memory address.

struct *cudaPos* **srcPos**
: Source position offset.

struct *cudaPitchedPtr* **srcPtr**
: Pitched source memory address.

7.57. cudaMemcpy3DPeerParms

struct **cudaMemcpy3DPeerParms**

CUDA 3D cross-device memory copying parameters.

Public Members

cudaArray_t **dstArray**

Destination memory address.

int **dstDevice**

Destination device.

struct *cudaPos* **dstPos**

Destination position offset.

struct *cudaPitchedPtr* **dstPtr**

Pitched destination memory address.

struct *cudaExtent* **extent**

Requested memory copy size.

cudaArray_t **srcArray**

Source memory address.

int **srcDevice**

Source device.

struct *cudaPos* **srcPos**

Source position offset.

struct *cudaPitchedPtr* **srcPtr**

Pitched source memory address.

7.58. cudaMemcpyAttributes

struct **cudaMemcpyAttributes**

Attributes specific to copies within a batch.

For more details on usage see *cudaMemcpyBatchAsync*.

Public Members

struct *cudaMemLocation* **dstLocHint**

Hint location for the destination operand.

Ignored when the pointers are not managed memory or memory allocated outside CUDA.

unsigned int **flags**

Additional flags for copies with this attribute.

See *cudaMemcpyFlags*.

enum *cudaMemcpySrcAccessOrder* **srcAccessOrder**

Source access ordering to be observed for copies with this attribute.

struct *cudaMemLocation* **srcLocHint**

Hint location for the source operand.

Ignored when the pointers are not managed memory or memory allocated outside CUDA.

7.59. cudaMemcpyBatchNodeParams

struct **cudaMemcpyBatchNodeParams**

CUDA Batch Memcpy Node Parameters.

Public Members

struct *cudaMemcpyAttributes* ***attrs**

Array of memcpy attributes.

This field cannot be updated in an instantiated graph.

size_t ***attrsIdxs**

Array of indices specifying which copies each entry in `attrs` applies to.

The array contents cannot be updated in an instantiated graph.

size_t **count**

Size of `dsts`, `srcs` and `sizes` arrays.

This field cannot be updated in an instantiated graph.

cudaExecutionContext_t **ctx**

Preferred CUDA Execution Context to use for the copies.

This field cannot be updated in an instantiated graph.

void ****dsts**

Array of destination pointers.

This field can be updated in an instantiated graph.

size_t **numAttrs**

Size of `attrs` and `attrsIdxs` arrays.

The array contents cannot be updated in an instantiated graph.

size_t ***sizes**

Array of sizes for memcpy operations.

This field can be updated in an instantiated graph. A size of 0 disables the individual copy operation.

void ****srcs**

Array of source pointers.

This field can be updated in an instantiated graph.

7.60. cudaMemcpyNodeParams

struct **cudaMemcpyNodeParams**

Memcpy node parameters.

Public Members

struct *cudaMemcpy3DParms* **copyParams**

Parameters for the memory copy.

cudaExecutionContext_t **ctx**

Context in which to run the memcpy.

If NULL will try to use the current context.

int **flags**

Must be zero.

int **reserved**

Must be zero.

7.61. cudaMemsetParams

struct **cudaMemsetParams**

CUDA Memset node parameters.

Public Members

void ***dst**

Destination device pointer.

unsigned int **elementSize**

Size of each element in bytes.

Must be 1, 2, or 4.

size_t **height**

Number of rows.

size_t **pitch**

Pitch of destination device pointer.

Unused if height is 1

unsigned int **value**

Value to be set.

size_t **width**

Width of the row in elements.

7.62. cudaMemsetParamsV2

struct **cudaMemsetParamsV2**

CUDA Memset node parameters.

Public Members

cudaExecutionContext_t **ctx**

Context in which to run the memset.

If NULL will try to use the current context.

void ***dst**

Destination device pointer.

unsigned int **elementSize**

Size of each element in bytes.

Must be 1, 2, or 4.

size_t **height**

Number of rows.

size_t **pitch**

Pitch of destination device pointer.

Unused if height is 1

unsigned int **value**

Value to be set.

size_t **width**

Width of the row in elements.

7.63. cudaOffset3D

struct **cudaOffset3D**

Struct representing offset into a *cudaArray_t* in elements.

Public Members

size_t **x**

size_t **y**

size_t **z**

7.64. cudaPitchedPtr

struct **cudaPitchedPtr**

CUDA Pitched memory pointer.

See also:

make_cudaPitchedPtr

Public Members

size_t **pitch**

Pitch of allocated memory in bytes.

void ***ptr**

Pointer to allocated memory.

size_t **xsize**

Logical width of allocation in elements.

size_t **ysize**

Logical height of allocation in elements.

7.65. cudaPointerAttributes

struct **cudaPointerAttributes**

CUDA pointer attributes.

Public Members

union *cudaPointerAttributes*::[anonymous] **[anonymous]**

Locality domain ordinal for device allocations localized to a locality domain, or -1 when the allocation is not localized to a locality domain.

int **device**

The device against which the memory was allocated or registered.

If the memory type is *cudaMemoryTypeDevice* then this identifies the device on which the memory referred physically resides. If the memory type is *cudaMemoryTypeHost* or::cudaMemoryTypeManaged then this identifies the device which was current when the memory was allocated or registered (and if that device is deinitialized then this allocation will vanish with that device's state).

void ***devicePointer**

The address which may be dereferenced on the current device to access the memory or NULL if no such address exists.

void ***hostPointer**

The address which may be dereferenced on the host to access the memory or NULL if no such address exists.

Note: CUDA doesn't check if unregistered memory is allocated so this field may contain invalid pointer if an invalid pointer has been passed to CUDA.

---

int **localityDomainOrdinal**

long **reserved**[7]

Must be zero.

enum *cudaMemoryType* **type**

The type of memory - *cudaMemoryTypeUnregistered*, *cudaMemoryTypeHost*, *cudaMemoryTypeDevice* or *cudaMemoryTypeManaged*.

long **unused**

7.66. cudaPos

struct **cudaPos**

CUDA 3D position.

See also:

make_cudaPos

Public Members

size_t **x**

x

size_t **y**

y

size_t **z**

z

7.67. cudaResourceDesc

struct **cudaResourceDesc**

CUDA resource descriptor.

Public Members

cudaArray_t **array**

CUDA array.

struct *cudaResourceDesc*::[anonymous]::[anonymous] **array**

struct *cudaChannelFormatDesc* **desc**

Channel descriptor.

void ***devPtr**

Device pointer.

unsigned int **flags**

Flags (must be zero)

size_t **height**

Height of the array in elements.

struct *cudaResourceDesc*::[anonymous]::[anonymous] **linear**

cudaMipmappedArray_t **mipmap**

CUDA mipmapped array.

struct *cudaResourceDesc*::[anonymous]::[anonymous] **mipmap**

struct *cudaResourceDesc*::[anonymous]::[anonymous] **pitch2D**

size_t **pitchInBytes**

Pitch between two rows in bytes.

union *cudaResourceDesc*::[anonymous] **res**

int **reserved**[32]

struct *cudaResourceDesc*::[anonymous]::[anonymous] **reserved**

enum *cudaResourceType* **resType**
Resource type.

size_t **sizeInBytes**
Size in bytes.

size_t **width**
Width of the array in elements.

7.68. cudaResourceViewDesc

struct **cudaResourceViewDesc**
CUDA resource view descriptor.

Public Members

size_t **depth**
Depth of the resource view.

unsigned int **firstLayer**
First layer index.

unsigned int **firstMipmapLevel**
First defined mipmap level.

enum *cudaResourceViewFormat* **format**
Resource view format.

size_t **height**
Height of the resource view.

unsigned int **lastLayer**
Last layer index.

unsigned int **lastMipmapLevel**
Last defined mipmap level.

unsigned int **reserved**[16]
Must be zero.

size_t **width**
Width of the resource view.

7.69. cudaTextureDesc

struct **cudaTextureDesc**

CUDA texture descriptor.

Public Members

enum *cudaTextureAddressMode* **addressMode**[3]

Texture address mode for up to 3 dimensions.

float **borderColor**[4]

Texture Border Color.

int **disableTrilinearOptimization**

Disable any trilinear filtering optimizations.

enum *cudaTextureFilterMode* **filterMode**

Texture filter mode.

unsigned int **maxAnisotropy**

Limit to the anisotropy ratio.

float **maxMipmapLevelClamp**

Upper end of the mipmap level range to clamp access to.

float **minMipmapLevelClamp**

Lower end of the mipmap level range to clamp access to.

enum *cudaTextureFilterMode* **mipmapFilterMode**

Mipmap filter mode.

float **mipmapLevelBias**

Offset applied to the supplied mipmap level.

int **normalizedCoords**

Indicates whether texture reads are normalized or not.

enum *cudaTextureReadMode* **readMode**

Texture read mode.

int **seamlessCubemap**

Enable seamless cube map filtering.

int **sRGB**

Perform sRGB->linear conversion during texture read.

7.70. cudalibraryHostUniversalFunctionAndDataTable

struct **cudalibraryHostUniversalFunctionAndDataTable**

Public Members

void ***dataTable**

size_t **dataWindowSize**

void ***functionTable**

size_t **functionWindowSize**

CUuuid_st

cudaAccessPolicyWindow
Specifies an access policy for a window, a contiguous extent of memory beginning at base_ptr and ending at base_ptr + num_bytes.

cudaArrayMemoryRequirements
CUDA array and CUDA mipmapped array memory requirements.

cudaArraySparseProperties
Sparse CUDA array and CUDA mipmapped array properties.

cudaAsyncNotificationInfo_t
Information describing an async notification event.

cudaChannelFormatDesc
CUDA Channel format descriptor.

cudaChildGraphNodeParams
Child graph node parameters.

cudaConditionalNodeParams
CUDA conditional node parameters.

cudaDevResource
A tagged union describing different resources identified by the type field.

cudaDevSmResource
Data for SM-related resources All parameters in this structure are OUTPUT only.

cudaDevSmResourceGroupParams
Input data for splitting SMs.

cudaDevWorkqueueConfigResource
Data for workqueue configuration related resources.

cudaDevWorkqueueResource
Handle to a pre-existing workqueue related resource.

cudaDeviceProp
CUDA device properties.

cudaEglFrame
CUDA EGLFrame Descriptor - structure defining one frame of EGL.

cudaEglPlaneDesc
CUDA EGL Plane Descriptor - structure defining each plane of a CUDA EGLFrame.

cudaEventRecordNodeParams
Event record node parameters.

cudaEventWaitNodeParams
Event wait node parameters.

cudaExtent
CUDA extent.

cudaExternalMemoryBufferDesc
External memory buffer descriptor.

cudaExternalMemoryHandleDesc
External memory handle descriptor.

cudaExternalMemoryMipmappedArrayDesc
External memory mipmap descriptor.

cudaExternalSemaphoreHandleDesc
External semaphore handle descriptor.

cudaExternalSemaphoreSignalNodeParams
External semaphore signal node parameters.

cudaExternalSemaphoreSignalNodeParamsV2
External semaphore signal node parameters.

cudaExternalSemaphoreSignalParams
External semaphore signal parameters, compatible with driver type.

cudaExternalSemaphoreWaitNodeParams
External semaphore wait node parameters.

cudaExternalSemaphoreWaitNodeParamsV2
External semaphore wait node parameters.

cudaExternalSemaphoreWaitParams
External semaphore wait parameters, compatible with driver type.

cudaFuncAttributes
CUDA function attributes.

cudaGraphEdgeData
Optional annotation for edges in a CUDA graph.

cudaGraphExecUpdateResultInfo
Result information returned by cudaGraphExecUpdate.

cudaGraphInstantiateParams
Graph instantiation parameters.

cudaGraphKernelNodeUpdate
Struct to specify a single node update to pass as part of a larger array to *cudaGraphKernelNode-UpdatesApply* .

cudaGraphNodeParams
Graph node parameters.

cudaGraphRecaptureCallbackData
Struct of user callback data that is invoked when node parameter mismatches are detected while recapturing to an existing graph.

cudaHostNodeParams
CUDA host node parameters.

cudaHostNodeParamsV2
CUDA host node parameters.

cudaIpcEventHandle_t
CUDA IPC event handle.

cudaIpcMemHandle_t
CUDA IPC memory handle.

cudaKernelNodeParams
CUDA GPU kernel node parameters.

cudaKernelNodeParamsV2
CUDA GPU kernel node parameters.

cudaLaunchAttribute
Launch attribute.

cudaLaunchConfig_t
CUDA extensible launch configuration.

cudaLaunchMemSyncDomainMap
Memory Synchronization Domain map.

cudaMemAccessDesc
Memory access descriptor.

cudaMemAllocNodeParams
Memory allocation node parameters.

cudaMemAllocNodeParamsV2
Memory allocation node parameters.

cudaMemFabricHandle_t

cudaMemFreeNodeParams
Memory free node parameters.

cudaMemLocation
Specifies a memory location.

cudaMemPoolProps
Specifies the properties of allocations made from the pool.

cudaMemPoolPtrExportData
Opaque data for exporting a pool allocation.

cudaMemcpy3DBatchOp

cudaMemcpy3DOperand
Struct representing an operand for copy with *cudaMemcpy3DBatchAsync* .

cudaMemcpy3DParms
CUDA 3D memory copying parameters.

cudaMemcpy3DPeerParms
CUDA 3D cross-device memory copying parameters.

cudaMemcpyAttributes
Attributes specific to copies within a batch.

cudaMemcpyBatchNodeParams
CUDA Batch Memcpy Node Parameters.

cudaMemcpyNodeParams
Memcpy node parameters.

cudaMemsetParams
CUDA Memset node parameters.

cudaMemsetParamsV2
CUDA Memset node parameters.

cudaOffset3D
Struct representing offset into a *cudaArray_t* in elements.

cudaPitchedPtr
CUDA Pitched memory pointer.

cudaPointerAttributes
CUDA pointer attributes.

cudaPos
CUDA 3D position.

cudaResourceDesc
CUDA resource descriptor.

cudaResourceViewDesc
CUDA resource view descriptor.

cudaTextureDesc
CUDA texture descriptor.

cudalibraryHostUniversalFunctionAndDataTable

Chapter 8. Deprecated List

Member *cudaD3D10GetDirect3DDevice* (ID3D10Device **ppD3D10Device)

This function is deprecated as of CUDA 5.0.

Member *cudaD3D10MapResources* (int count, ID3D10Resource **ppResources)

This function is deprecated as of CUDA 3.0.

Member *cudaD3D10RegisterResource* (ID3D10Resource *pResource, unsigned int flags)

This function is deprecated as of CUDA 3.0.

Member *cudaD3D10ResourceGetMappedArray* (cudaArray **ppArray, ID3D10Resource *pResource, unsigned int subResource)

This function is deprecated as of CUDA 3.0.

Member *cudaD3D10ResourceGetMappedPitch* (size_t *pPitch, size_t *pPitchSlice, ID3D10Resource *pResource, unsigned int subResource)

This function is deprecated as of CUDA 3.0.

Member *cudaD3D10ResourceGetMappedPointer* (void **pPointer, ID3D10Resource *pResource, unsigned int subResource)

This function is deprecated as of CUDA 3.0.

Member *cudaD3D10ResourceGetMappedSize* (size_t *pSize, ID3D10Resource *pResource, unsigned int subResource)

This function is deprecated as of CUDA 3.0.

Member *cudaD3D10ResourceGetSurfaceDimensions* (size_t *pWidth, size_t *pHeight, size_t *pDepth, ID3D10Resource *pResource, unsigned int subResource)

This function is deprecated as of CUDA 3.0.

Member *cudaD3D10ResourceSetMapFlags* (ID3D10Resource *pResource, unsigned int flags)

This function is deprecated as of CUDA 3.0.

Member *cudaD3D10SetDirect3DDevice* (ID3D10Device *pD3D10Device, int device=-1)

This function is deprecated as of CUDA 5.0.

Member *cudaD3D10UnmapResources* (int count, ID3D10Resource **ppResources)

This function is deprecated as of CUDA 3.0.

Member *cudaD3D10UnregisterResource* (ID3D10Resource *pResource)

This function is deprecated as of CUDA 3.0.

Member *cudaD3D11GetDirect3DDevice* (ID3D11Device **ppD3D11Device)

This function is deprecated as of CUDA 5.0.

Member *cudaD3D11SetDirect3DDevice* (ID3D11Device *pD3D11Device, int device=-1)

This function is deprecated as of CUDA 5.0.

Member *cudaD3D9MapResources* (int count, IDirect3DResource9 **ppResources)

This function is deprecated as of CUDA 3.0.

Member *cudaD3D9RegisterResource* (IDirect3DResource9 *pResource, unsigned int flags)

This function is deprecated as of CUDA 3.0.

Member *cudaD3D9ResourceGetMappedArray* (cudaArray **ppArray, IDirect3DResource9 *pResource, unsigned int face, unsigned int level)

This function is deprecated as of CUDA 3.0.

Member *cudaD3D9ResourceGetMappedPitch* (size_t *pPitch, size_t *pPitchSlice, IDirect3DResource9 *pResource, unsigned int face, unsigned int level)

This function is deprecated as of CUDA 3.0.

Member *cudaD3D9ResourceGetMappedPointer* (void **pPointer, IDirect3DResource9 *pResource, unsigned int face, unsigned int level)

This function is deprecated as of CUDA 3.0.

Member *cudaD3D9ResourceGetMappedSize* (size_t *pSize, IDirect3DResource9 *pResource, unsigned int face, unsigned int level)

This function is deprecated as of CUDA 3.0.

Member *cudaD3D9ResourceGetSurfaceDimensions* (size_t *pWidth, size_t *pHeight, size_t *pDepth, IDirect3DResource9 *pResource, unsigned int face, unsigned int level)

This function is deprecated as of CUDA 3.0.

Member *cudaD3D9ResourceSetMapFlags* (IDirect3DResource9 *pResource, unsigned int flags)

This function is deprecated as of CUDA 3.0.

Member *cudaD3D9UnmapResources* (int count, IDirect3DResource9 **ppResources)

This function is deprecated as of CUDA 3.0.

Member *cudaD3D9UnregisterResource* (IDirect3DResource9 *pResource)

This function is deprecated as of CUDA 3.0.

Member *cudaDeviceBlockingSync*

This flag was deprecated as of CUDA 4.0 and replaced with *cudaDeviceScheduleBlockingSync*.

Member *cudaDeviceGetSharedMemConfig* (enum cudaSharedMemConfig *pConfig)

Member *cudaDeviceSetSharedMemConfig* (enum cudaSharedMemConfig config)

Member *cudaErrorAddressOfConstant*

This error return is deprecated as of CUDA 3.1. Variables in constant memory may now have their address taken by the runtime via *cudaGetSymbolAddress()*.

Member *cudaErrorInvalidDevicePointer*

This error return is deprecated as of CUDA 10.1.

Member *cudaErrorInvalidHostPointer*

This error return is deprecated as of CUDA 10.1.

Member *cudaErrorMemoryValueTooLarge*

This error return is deprecated as of CUDA 3.1. Device emulation mode was removed with the CUDA 3.1 release.

Member *cudaErrorMixedDeviceExecution*

This error return is deprecated as of CUDA 3.1. Device emulation mode was removed with the CUDA 3.1 release.

Member *cudaErrorNotYetImplemented*

This error return is deprecated as of CUDA 4.1.

Member *cudaErrorPriorLaunchFailure*

This error return is deprecated as of CUDA 3.1. Device emulation mode was removed with the CUDA 3.1 release.

Member *cudaErrorProfilerAlreadyStarted*

This error return is deprecated as of CUDA 5.0. It is no longer an error to call *cudaProfilerStart()* when profiling is already enabled.

Member *cudaErrorProfilerAlreadyStopped*

This error return is deprecated as of CUDA 5.0. It is no longer an error to call *cudaProfilerStop()* when profiling is already disabled.

Member *cudaErrorProfilerNotInitialized*

This error return is deprecated as of CUDA 5.0. It is no longer an error to attempt to enable/disable the profiling via *cudaProfilerStart* or *cudaProfilerStop* without initialization.

Member *cudaErrorSynchronizationError*

This error return is deprecated as of CUDA 3.1. Device emulation mode was removed with the CUDA 3.1 release.

Member *cudaErrorTextureFetchFailed*

This error return is deprecated as of CUDA 3.1. Device emulation mode was removed with the CUDA 3.1 release.

Member *cudaErrorTextureNotBound*

This error return is deprecated as of CUDA 3.1. Device emulation mode was removed with the CUDA 3.1 release.

Member *cudaFuncSetSharedMemConfig* (const void *func, enum cudaSharedMemConfig config)

Member *cudaGetDriverEntryPoint* (const char *symbol, void **funcPtr, unsigned long long flags, enum cudaDriverEntryPointQueryResult *driverStatus=NULL)

This function is deprecated as of CUDA 13.0

Member *cudaGLMapBufferObject* (void **devPtr, GLuint bufObj)

This function is deprecated as of CUDA 3.0.

Member *cudaGLMapBufferObjectAsync* (void **devPtr, GLuint bufObj, cudaStream_t stream)

This function is deprecated as of CUDA 3.0.

Member *cudaGLRegisterBufferObject* (GLuint bufObj)

This function is deprecated as of CUDA 3.0.

Member *cudaGLSetBufferObjectMapFlags* (GLuint bufObj, unsigned int flags)

This function is deprecated as of CUDA 3.0.

Member *cudaGLSetGLDevice* (int device)

This function is deprecated as of CUDA 5.0.

Member *cudaGLUnmapBufferObject* (GLuint bufObj)

This function is deprecated as of CUDA 3.0.

Member *cudaGLUnmapBufferObjectAsync* (GLuint bufObj, cudaStream_t stream)

This function is deprecated as of CUDA 3.0.

Member *cudaGLUnregisterBufferObject* (GLuint bufObj)

This function is deprecated as of CUDA 3.0.

Member *cudaMemcpyArrayToArray* (cudaArray_t dst, size_t wOffsetDst, size_t hOffsetDst, cudaArray_const_t src, size_t wOffsetSrc, size_t hOffsetSrc, size_t count, enum cudaMemcpyKind kind=cudaMemcpyDeviceToDevice)

Member *cudaMemcpyFromArray* (void *dst, cudaArray_const_t src, size_t wOffset, size_t hOffset, size_t count, enum cudaMemcpyKind kind)

Member *cudaMemcpyFromArrayAsync* (void *dst, cudaArray_const_t src, size_t wOffset, size_t hOffset, size_t count, enum cudaMemcpyKind kind, cudaStream_t stream=0)

Member *cudaMemcpyToArray* (cudaArray_t dst, size_t wOffset, size_t hOffset, const void *src, size_t count, enum cudaMemcpyKind kind)

Member *cudaMemcpyToArrayAsync* (cudaArray_t dst, size_t wOffset, size_t hOffset, const void *src, size_t count, enum cudaMemcpyKind kind, cudaStream_t stream=0)

Member *cudaSharedMemConfig*

Chapter 9. Notices

9.1. Notice

This document is provided for information purposes only and shall not be regarded as a warranty of a certain functionality, condition, or quality of a product. NVIDIA Corporation (“NVIDIA”) makes no representations or warranties, expressed or implied, as to the accuracy or completeness of the information contained in this document and assumes no responsibility for any errors contained herein. NVIDIA shall have no liability for the consequences or use of such information or for any infringement of patents or other rights of third parties that may result from its use. This document is not a commitment to develop, release, or deliver any Material (defined below), code, or functionality.

NVIDIA reserves the right to make corrections, modifications, enhancements, improvements, and any other changes to this document, at any time without notice.

Customer should obtain the latest relevant information before placing orders and should verify that such information is current and complete.

NVIDIA products are sold subject to the NVIDIA standard terms and conditions of sale supplied at the time of order acknowledgement, unless otherwise agreed in an individual sales agreement signed by authorized representatives of NVIDIA and customer (“Terms of Sale”). NVIDIA hereby expressly objects to applying any customer general terms and conditions with regards to the purchase of the NVIDIA product referenced in this document. No contractual obligations are formed either directly or indirectly by this document.

NVIDIA products are not designed, authorized, or warranted to be suitable for use in medical, military, aircraft, space, or life support equipment, nor in applications where failure or malfunction of the NVIDIA product can reasonably be expected to result in personal injury, death, or property or environmental damage. NVIDIA accepts no liability for inclusion and/or use of NVIDIA products in such equipment or applications and therefore such inclusion and/or use is at customer’s own risk.

NVIDIA makes no representation or warranty that products based on this document will be suitable for any specified use. Testing of all parameters of each product is not necessarily performed by NVIDIA. It is customer’s sole responsibility to evaluate and determine the applicability of any information contained in this document, ensure the product is suitable and fit for the application planned by customer, and perform the necessary testing for the application in order to avoid a default of the application or the product. Weaknesses in customer’s product designs may affect the quality and reliability of the NVIDIA product and may result in additional or different conditions and/or requirements beyond those contained in this document. NVIDIA accepts no liability related to any default, damage, costs, or problem which may be based on or attributable to: (i) the use of the NVIDIA product in any manner that is contrary to this document or (ii) customer product designs.

No license, either expressed or implied, is granted under any NVIDIA patent right, copyright, or other NVIDIA intellectual property right under this document. Information published by NVIDIA regarding third-party products or services does not constitute a license from NVIDIA to use such products or

services or a warranty or endorsement thereof. Use of such information may require a license from a third party under the patents or other intellectual property rights of the third party, or a license from NVIDIA under the patents or other intellectual property rights of NVIDIA.

Reproduction of information in this document is permissible only if approved in advance by NVIDIA in writing, reproduced without alteration and in full compliance with all applicable export laws and regulations, and accompanied by all associated conditions, limitations, and notices.

THIS DOCUMENT AND ALL NVIDIA DESIGN SPECIFICATIONS, REFERENCE BOARDS, FILES, DRAWINGS, DIAGNOSTICS, LISTS, AND OTHER DOCUMENTS (TOGETHER AND SEPARATELY, "MATERIALS") ARE BEING PROVIDED "AS IS." NVIDIA MAKES NO WARRANTIES, EXPRESSED, IMPLIED, STATUTORY, OR OTHERWISE WITH RESPECT TO THE MATERIALS, AND EXPRESSLY DISCLAIMS ALL IMPLIED WARRANTIES OF NONINFRINGEMENT, MERCHANTABILITY, AND FITNESS FOR A PARTICULAR PURPOSE. TO THE EXTENT NOT PROHIBITED BY LAW, IN NO EVENT WILL NVIDIA BE LIABLE FOR ANY DAMAGES, INCLUDING WITHOUT LIMITATION ANY DIRECT, INDIRECT, SPECIAL, INCIDENTAL, PUNITIVE, OR CONSEQUENTIAL DAMAGES, HOWEVER CAUSED AND REGARDLESS OF THE THEORY OF LIABILITY, ARISING OUT OF ANY USE OF THIS DOCUMENT, EVEN IF NVIDIA HAS BEEN ADVISED OF THE POSSIBILITY OF SUCH DAMAGES. Notwithstanding any damages that customer might incur for any reason whatsoever, NVIDIA's aggregate and cumulative liability towards customer for the products described herein shall be limited in accordance with the Terms of Sale for the product.

9.2. OpenCL

OpenCL is a trademark of Apple Inc. used under license to the Khronos Group Inc.

9.3. Trademarks

NVIDIA and the NVIDIA logo are trademarks or registered trademarks of NVIDIA Corporation in the U.S. and other countries. Other company and product names may be trademarks of the respective companies with which they are associated.

Copyright

©2007-2026, NVIDIA Corporation & affiliates. All rights reserved