CUDA DRIVER API

TRM-06703-001 _v6.5 | August 2014

API Reference Manual

TABLE OF CONTENTS

Chapter 1.
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. In the reference documentation, each memcpy function is categorized as synchronous or asynchronous, corresponding to the definitions below.

Memcpy

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

Synchronous

1. All transfers involving Unified Memory regions are fully synchronous with respect to the host.

2. 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.

3. For transfers from pinned host memory to device memory, the function is synchronous with respect to the host.

4. For transfers from device to either pageable or pinned host memory, the function returns only once the copy has completed.

5. For transfers from device memory to device memory, no host-side synchronization is performed.

6. For transfers from any host memory to any host memory, the function is fully synchronous with respect to the host.

Asynchronous

1. For transfers from pageable host memory to device memory, host memory is copied to a staging buffer immediately (no device synchronization is performed). The function will return once the pageable buffer has been copied to the staging memory. The DMA transfer to final destination may not have completed.

2. For transfers between pinned host memory and device memory, the function is fully asynchronous.

3. For transfers from device memory to pageable host memory, the function will return only once the copy has completed.

4. For all other transfers, the function is fully asynchronous. If pageable memory must first be staged to pinned memory, this will be handled asynchronously with a worker thread.

5. For transfers from any host memory to any host memory, the function is fully synchronous with respect to the host.

Memset

The synchronous memset functions are asynchronous with respect to the host except when the target is pinned host memory or a Unified Memory region, in which case they are fully synchronous. 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 2.
STREAM SYNCHRONIZATION BEHAVIOR

NULL stream

The NULL stream or stream 0 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 NULL stream such as a kernel launch or **cudaStreamWaitEvent()**, the NULL stream first waits on all blocking streams, the action is queued in the NULL stream, and then all blocking streams wait on the NULL stream.

For example, the following code launches a kernel **k_1** in stream **s**, then **k_2** in the NULL 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**.

Actions are added in the NULL stream by passing **0** as the **cudaStream_t** parameter of applicable APIs, or by calling the corresponding synchronous APIs which do not take the **cudaStream_t** parameter. The NULL stream is also the default for kernel launches if a stream is not specified.

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

Chapter 3.
MODULES

Here is a list of all modules:

- ▸ Data types used by CUDA driver
- ▸ Error Handling
- ▸ Initialization
- ▸ Version Management
- ▸ Device Management
- ▸ Device Management [DEPRECATED]
- ▸ Context Management
- ▸ Context Management [DEPRECATED]
- ▸ Module Management
- ▸ Memory Management
- ▸ Unified Addressing
- ▸ Stream Management
- ▸ Event Management
- ▸ Execution Control
- ▸ Execution Control [DEPRECATED]
- ▸ Occupancy
- ▸ Texture Reference Management
- ▸ Texture Reference Management [DEPRECATED]
- ▸ Surface Reference Management
- ▸ Texture Object Management
- ▸ Surface Object Management
- ▸ Peer Context Memory Access
- ▸ Graphics Interoperability
- ▸ Profiler Control
- ▸ OpenGL Interoperability

    - ▸ OpenGL Interoperability [DEPRECATED]

▸ Direct3D 9 Interoperability

    ▸ Direct3D 9 Interoperability [DEPRECATED]

▸ Direct3D 10 Interoperability

    ▸ Direct3D 10 Interoperability [DEPRECATED]

▸ Direct3D 11 Interoperability

    ▸ Direct3D 11 Interoperability [DEPRECATED]

▸ VDPAU Interoperability

3.1. Data types used by CUDA driver

struct CUDA_ARRAY3D_DESCRIPTOR

struct CUDA_ARRAY_DESCRIPTOR

struct CUDA_MEMCPY2D

struct CUDA_MEMCPY3D

struct CUDA_MEMCPY3D_PEER

struct CUDA_POINTER_ATTRIBUTE_P2P_TOKENS

struct CUDA_RESOURCE_DESC

struct CUDA_RESOURCE_VIEW_DESC

struct CUDA_TEXTURE_DESC

struct CUdevprop

struct CUipcEventHandle

struct CUipcMemHandle

enum CUaddress_mode

Texture reference addressing modes

Values

CU_TR_ADDRESS_MODE_WRAP = 0
   Wrapping address mode
CU_TR_ADDRESS_MODE_CLAMP = 1
   Clamp to edge address mode
CU_TR_ADDRESS_MODE_MIRROR = 2
   Mirror address mode
CU_TR_ADDRESS_MODE_BORDER = 3
   Border address mode

enum CUarray_cubemap_face

Array indices for cube faces

Values

CU_CUBEMAP_FACE_POSITIVE_X = 0x00
　　Positive X face of cubemap
CU_CUBEMAP_FACE_NEGATIVE_X = 0x01
　　Negative X face of cubemap
CU_CUBEMAP_FACE_POSITIVE_Y = 0x02
　　Positive Y face of cubemap
CU_CUBEMAP_FACE_NEGATIVE_Y = 0x03
　　Negative Y face of cubemap
CU_CUBEMAP_FACE_POSITIVE_Z = 0x04
　　Positive Z face of cubemap
CU_CUBEMAP_FACE_NEGATIVE_Z = 0x05
　　Negative Z face of cubemap

enum CUarray_format

Array formats

Values

CU_AD_FORMAT_UNSIGNED_INT8 = 0x01
　　Unsigned 8-bit integers
CU_AD_FORMAT_UNSIGNED_INT16 = 0x02
　　Unsigned 16-bit integers
CU_AD_FORMAT_UNSIGNED_INT32 = 0x03
　　Unsigned 32-bit integers
CU_AD_FORMAT_SIGNED_INT8 = 0x08
　　Signed 8-bit integers
CU_AD_FORMAT_SIGNED_INT16 = 0x09
　　Signed 16-bit integers
CU_AD_FORMAT_SIGNED_INT32 = 0x0a
　　Signed 32-bit integers
CU_AD_FORMAT_HALF = 0x10
　　16-bit floating point
CU_AD_FORMAT_FLOAT = 0x20
　　32-bit floating point

enum CUcomputemode

Compute Modes

Values

CU_COMPUTEMODE_DEFAULT = 0
Default compute mode (Multiple contexts allowed per device)
CU_COMPUTEMODE_EXCLUSIVE = 1
Compute-exclusive-thread mode (Only one context used by a single thread can be present on this device at a time)
CU_COMPUTEMODE_PROHIBITED = 2
Compute-prohibited mode (No contexts can be created on this device at this time)
CU_COMPUTEMODE_EXCLUSIVE_PROCESS = 3
Compute-exclusive-process mode (Only one context used by a single process can be present on this device at a time)

enum CUctx_flags

Context creation flags

Values

CU_CTX_SCHED_AUTO = 0x00
Automatic scheduling
CU_CTX_SCHED_SPIN = 0x01
Set spin as default scheduling
CU_CTX_SCHED_YIELD = 0x02
Set yield as default scheduling
CU_CTX_SCHED_BLOCKING_SYNC = 0x04
Set blocking synchronization as default scheduling
CU_CTX_BLOCKING_SYNC = 0x04
Set blocking synchronization as default scheduling Deprecated
This flag was deprecated as of CUDA 4.0 and was replaced with
CU_CTX_SCHED_BLOCKING_SYNC.
CU_CTX_SCHED_MASK = 0x07
CU_CTX_MAP_HOST = 0x08
Support mapped pinned allocations
CU_CTX_LMEM_RESIZE_TO_MAX = 0x10
Keep local memory allocation after launch
CU_CTX_FLAGS_MASK = 0x1f

enum CUdevice_attribute

Device properties

Values

CU_DEVICE_ATTRIBUTE_MAX_THREADS_PER_BLOCK = 1
Maximum number of threads per block
CU_DEVICE_ATTRIBUTE_MAX_BLOCK_DIM_X = 2
Maximum block dimension X
CU_DEVICE_ATTRIBUTE_MAX_BLOCK_DIM_Y = 3
Maximum block dimension Y
CU_DEVICE_ATTRIBUTE_MAX_BLOCK_DIM_Z = 4
Maximum block dimension Z
CU_DEVICE_ATTRIBUTE_MAX_GRID_DIM_X = 5
Maximum grid dimension X
CU_DEVICE_ATTRIBUTE_MAX_GRID_DIM_Y = 6
Maximum grid dimension Y
CU_DEVICE_ATTRIBUTE_MAX_GRID_DIM_Z = 7
Maximum grid dimension Z
CU_DEVICE_ATTRIBUTE_MAX_SHARED_MEMORY_PER_BLOCK = 8
Maximum shared memory available per block in bytes
CU_DEVICE_ATTRIBUTE_SHARED_MEMORY_PER_BLOCK = 8
Deprecated, use
CU_DEVICE_ATTRIBUTE_MAX_SHARED_MEMORY_PER_BLOCK
CU_DEVICE_ATTRIBUTE_TOTAL_CONSTANT_MEMORY = 9
Memory available on device for __constant__ variables in a CUDA C kernel in bytes
CU_DEVICE_ATTRIBUTE_WARP_SIZE = 10
Warp size in threads
CU_DEVICE_ATTRIBUTE_MAX_PITCH = 11
Maximum pitch in bytes allowed by memory copies
CU_DEVICE_ATTRIBUTE_MAX_REGISTERS_PER_BLOCK = 12
Maximum number of 32-bit registers available per block
CU_DEVICE_ATTRIBUTE_REGISTERS_PER_BLOCK = 12
Deprecated, use CU_DEVICE_ATTRIBUTE_MAX_REGISTERS_PER_BLOCK
CU_DEVICE_ATTRIBUTE_CLOCK_RATE = 13
Typical clock frequency in kilohertz
CU_DEVICE_ATTRIBUTE_TEXTURE_ALIGNMENT = 14
Alignment requirement for textures
CU_DEVICE_ATTRIBUTE_GPU_OVERLAP = 15
Device can possibly copy memory and execute a kernel concurrently. Deprecated. Use instead CU_DEVICE_ATTRIBUTE_ASYNC_ENGINE_COUNT.
CU_DEVICE_ATTRIBUTE_MULTIPROCESSOR_COUNT = 16

Number of multiprocessors on device

CU_DEVICE_ATTRIBUTE_KERNEL_EXEC_TIMEOUT = 17

Specifies whether there is a run time limit on kernels

CU_DEVICE_ATTRIBUTE_INTEGRATED = 18

Device is integrated with host memory

CU_DEVICE_ATTRIBUTE_CAN_MAP_HOST_MEMORY = 19

Device can map host memory into CUDA address space

CU_DEVICE_ATTRIBUTE_COMPUTE_MODE = 20

Compute mode (See CUcomputemode for details)

CU_DEVICE_ATTRIBUTE_MAXIMUM_TEXTURE1D_WIDTH = 21

Maximum 1D texture width

CU_DEVICE_ATTRIBUTE_MAXIMUM_TEXTURE2D_WIDTH = 22

Maximum 2D texture width

CU_DEVICE_ATTRIBUTE_MAXIMUM_TEXTURE2D_HEIGHT = 23

Maximum 2D texture height

CU_DEVICE_ATTRIBUTE_MAXIMUM_TEXTURE3D_WIDTH = 24

Maximum 3D texture width

CU_DEVICE_ATTRIBUTE_MAXIMUM_TEXTURE3D_HEIGHT = 25

Maximum 3D texture height

CU_DEVICE_ATTRIBUTE_MAXIMUM_TEXTURE3D_DEPTH = 26

Maximum 3D texture depth

CU_DEVICE_ATTRIBUTE_MAXIMUM_TEXTURE2D_LAYERED_WIDTH = 27

Maximum 2D layered texture width

CU_DEVICE_ATTRIBUTE_MAXIMUM_TEXTURE2D_LAYERED_HEIGHT = 28

Maximum 2D layered texture height

CU_DEVICE_ATTRIBUTE_MAXIMUM_TEXTURE2D_LAYERED_LAYERS = 29

Maximum layers in a 2D layered texture

CU_DEVICE_ATTRIBUTE_MAXIMUM_TEXTURE2D_ARRAY_WIDTH = 27

Deprecated, use
CU_DEVICE_ATTRIBUTE_MAXIMUM_TEXTURE2D_LAYERED_WIDTH

CU_DEVICE_ATTRIBUTE_MAXIMUM_TEXTURE2D_ARRAY_HEIGHT = 28

Deprecated, use
CU_DEVICE_ATTRIBUTE_MAXIMUM_TEXTURE2D_LAYERED_HEIGHT

CU_DEVICE_ATTRIBUTE_MAXIMUM_TEXTURE2D_ARRAY_NUMSLICES = 29

Deprecated, use
CU_DEVICE_ATTRIBUTE_MAXIMUM_TEXTURE2D_LAYERED_LAYERS

CU_DEVICE_ATTRIBUTE_SURFACE_ALIGNMENT = 30

Alignment requirement for surfaces

CU_DEVICE_ATTRIBUTE_CONCURRENT_KERNELS = 31

Device can possibly execute multiple kernels concurrently

CU_DEVICE_ATTRIBUTE_ECC_ENABLED = 32

Device has ECC support enabled

CU_DEVICE_ATTRIBUTE_PCI_BUS_ID = 33

PCI bus ID of the device

CU_DEVICE_ATTRIBUTE_PCI_DEVICE_ID = 34

PCI device ID of the device

CU_DEVICE_ATTRIBUTE_TCC_DRIVER = 35

Device is using TCC driver model

CU_DEVICE_ATTRIBUTE_MEMORY_CLOCK_RATE = 36

Peak memory clock frequency in kilohertz

CU_DEVICE_ATTRIBUTE_GLOBAL_MEMORY_BUS_WIDTH = 37

Global memory bus width in bits

CU_DEVICE_ATTRIBUTE_L2_CACHE_SIZE = 38

Size of L2 cache in bytes

CU_DEVICE_ATTRIBUTE_MAX_THREADS_PER_MULTIPROCESSOR = 39

Maximum resident threads per multiprocessor

CU_DEVICE_ATTRIBUTE_ASYNC_ENGINE_COUNT = 40

Number of asynchronous engines

CU_DEVICE_ATTRIBUTE_UNIFIED_ADDRESSING = 41

Device shares a unified address space with the host

CU_DEVICE_ATTRIBUTE_MAXIMUM_TEXTURE1D_LAYERED_WIDTH = 42

Maximum 1D layered texture width

CU_DEVICE_ATTRIBUTE_MAXIMUM_TEXTURE1D_LAYERED_LAYERS = 43

Maximum layers in a 1D layered texture

CU_DEVICE_ATTRIBUTE_CAN_TEX2D_GATHER = 44

Deprecated, do not use.

CU_DEVICE_ATTRIBUTE_MAXIMUM_TEXTURE2D_GATHER_WIDTH = 45

Maximum 2D texture width if CUDA_ARRAY3D_TEXTURE_GATHER is set

CU_DEVICE_ATTRIBUTE_MAXIMUM_TEXTURE2D_GATHER_HEIGHT = 46

Maximum 2D texture height if CUDA_ARRAY3D_TEXTURE_GATHER is set

CU_DEVICE_ATTRIBUTE_MAXIMUM_TEXTURE3D_WIDTH_ALTERNATE = 47

Alternate maximum 3D texture width

CU_DEVICE_ATTRIBUTE_MAXIMUM_TEXTURE3D_HEIGHT_ALTERNATE = 48

Alternate maximum 3D texture height

CU_DEVICE_ATTRIBUTE_MAXIMUM_TEXTURE3D_DEPTH_ALTERNATE = 49

Alternate maximum 3D texture depth

CU_DEVICE_ATTRIBUTE_PCI_DOMAIN_ID = 50

PCI domain ID of the device

CU_DEVICE_ATTRIBUTE_TEXTURE_PITCH_ALIGNMENT = 51

Pitch alignment requirement for textures

CU_DEVICE_ATTRIBUTE_MAXIMUM_TEXTURECUBEMAP_WIDTH = 52

Maximum cubemap texture width/height

CU_DEVICE_ATTRIBUTE_MAXIMUM_TEXTURECUBEMAP_LAYERED_WIDTH = 53

Maximum cubemap layered texture width/height

CU_DEVICE_ATTRIBUTE_MAXIMUM_TEXTURECUBEMAP_LAYERED_LAYERS = 54
     Maximum layers in a cubemap layered texture
CU_DEVICE_ATTRIBUTE_MAXIMUM_SURFACE1D_WIDTH = 55
     Maximum 1D surface width
CU_DEVICE_ATTRIBUTE_MAXIMUM_SURFACE2D_WIDTH = 56
     Maximum 2D surface width
CU_DEVICE_ATTRIBUTE_MAXIMUM_SURFACE2D_HEIGHT = 57
     Maximum 2D surface height
CU_DEVICE_ATTRIBUTE_MAXIMUM_SURFACE3D_WIDTH = 58
     Maximum 3D surface width
CU_DEVICE_ATTRIBUTE_MAXIMUM_SURFACE3D_HEIGHT = 59
     Maximum 3D surface height
CU_DEVICE_ATTRIBUTE_MAXIMUM_SURFACE3D_DEPTH = 60
     Maximum 3D surface depth
CU_DEVICE_ATTRIBUTE_MAXIMUM_SURFACE1D_LAYERED_WIDTH = 61
     Maximum 1D layered surface width
CU_DEVICE_ATTRIBUTE_MAXIMUM_SURFACE1D_LAYERED_LAYERS = 62
     Maximum layers in a 1D layered surface
CU_DEVICE_ATTRIBUTE_MAXIMUM_SURFACE2D_LAYERED_WIDTH = 63
     Maximum 2D layered surface width
CU_DEVICE_ATTRIBUTE_MAXIMUM_SURFACE2D_LAYERED_HEIGHT = 64
     Maximum 2D layered surface height
CU_DEVICE_ATTRIBUTE_MAXIMUM_SURFACE2D_LAYERED_LAYERS = 65
     Maximum layers in a 2D layered surface
CU_DEVICE_ATTRIBUTE_MAXIMUM_SURFACECUBEMAP_WIDTH = 66
     Maximum cubemap surface width
CU_DEVICE_ATTRIBUTE_MAXIMUM_SURFACECUBEMAP_LAYERED_WIDTH = 67
     Maximum cubemap layered surface width
CU_DEVICE_ATTRIBUTE_MAXIMUM_SURFACECUBEMAP_LAYERED_LAYERS = 68
     Maximum layers in a cubemap layered surface
CU_DEVICE_ATTRIBUTE_MAXIMUM_TEXTURE1D_LINEAR_WIDTH = 69
     Maximum 1D linear texture width
CU_DEVICE_ATTRIBUTE_MAXIMUM_TEXTURE2D_LINEAR_WIDTH = 70
     Maximum 2D linear texture width
CU_DEVICE_ATTRIBUTE_MAXIMUM_TEXTURE2D_LINEAR_HEIGHT = 71
     Maximum 2D linear texture height
CU_DEVICE_ATTRIBUTE_MAXIMUM_TEXTURE2D_LINEAR_PITCH = 72
     Maximum 2D linear texture pitch in bytes
CU_DEVICE_ATTRIBUTE_MAXIMUM_TEXTURE2D_MIPMAPPED_WIDTH = 73
     Maximum mipmapped 2D texture width

CU_DEVICE_ATTRIBUTE_MAXIMUM_TEXTURE2D_MIPMAPPED_HEIGHT = 74
Maximum mipmapped 2D texture height
CU_DEVICE_ATTRIBUTE_COMPUTE_CAPABILITY_MAJOR = 75
Major compute capability version number
CU_DEVICE_ATTRIBUTE_COMPUTE_CAPABILITY_MINOR = 76
Minor compute capability version number
CU_DEVICE_ATTRIBUTE_MAXIMUM_TEXTURE1D_MIPMAPPED_WIDTH = 77
Maximum mipmapped 1D texture width
CU_DEVICE_ATTRIBUTE_STREAM_PRIORITIES_SUPPORTED = 78
Device supports stream priorities
CU_DEVICE_ATTRIBUTE_GLOBAL_L1_CACHE_SUPPORTED = 79
Device supports caching globals in L1
CU_DEVICE_ATTRIBUTE_LOCAL_L1_CACHE_SUPPORTED = 80
Device supports caching locals in L1
CU_DEVICE_ATTRIBUTE_MAX_SHARED_MEMORY_PER_MULTIPROCESSOR = 81
Maximum shared memory available per multiprocessor in bytes
CU_DEVICE_ATTRIBUTE_MAX_REGISTERS_PER_MULTIPROCESSOR = 82
Maximum number of 32-bit registers available per multiprocessor
CU_DEVICE_ATTRIBUTE_MANAGED_MEMORY = 83
Device can allocate managed memory on this system
CU_DEVICE_ATTRIBUTE_MULTI_GPU_BOARD = 84
Device is on a multi-GPU board
CU_DEVICE_ATTRIBUTE_MULTI_GPU_BOARD_GROUP_ID = 85
Unique id for a group of devices on the same multi-GPU board
CU_DEVICE_ATTRIBUTE_MAX

enum CUevent_flags

Event creation flags

Values

CU_EVENT_DEFAULT = 0x0
Default event flag
CU_EVENT_BLOCKING_SYNC = 0x1
Event uses blocking synchronization
CU_EVENT_DISABLE_TIMING = 0x2
Event will not record timing data
CU_EVENT_INTERPROCESS = 0x4
Event is suitable for interprocess use. CU_EVENT_DISABLE_TIMING must be set

enum CUfilter_mode

Texture reference filtering modes

Values

CU_TR_FILTER_MODE_POINT = 0
    Point filter mode
CU_TR_FILTER_MODE_LINEAR = 1
    Linear filter mode

enum CUfunc_cache

Function cache configurations

Values

CU_FUNC_CACHE_PREFER_NONE = 0x00
    no preference for shared memory or L1 (default)
CU_FUNC_CACHE_PREFER_SHARED = 0x01
    prefer larger shared memory and smaller L1 cache
CU_FUNC_CACHE_PREFER_L1 = 0x02
    prefer larger L1 cache and smaller shared memory
CU_FUNC_CACHE_PREFER_EQUAL = 0x03
    prefer equal sized L1 cache and shared memory

enum CUfunction_attribute

Function properties

Values

CU_FUNC_ATTRIBUTE_MAX_THREADS_PER_BLOCK = 0
    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.
CU_FUNC_ATTRIBUTE_SHARED_SIZE_BYTES = 1
    The size in bytes of statically-allocated shared memory required by this function.
    This does not include dynamically-allocated shared memory requested by the user at
    runtime.
CU_FUNC_ATTRIBUTE_CONST_SIZE_BYTES = 2
    The size in bytes of user-allocated constant memory required by this function.
CU_FUNC_ATTRIBUTE_LOCAL_SIZE_BYTES = 3
    The size in bytes of local memory used by each thread of this function.
CU_FUNC_ATTRIBUTE_NUM_REGS = 4
    The number of registers used by each thread of this function.
CU_FUNC_ATTRIBUTE_PTX_VERSION = 5
    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. Note that this may return the undefined value of 0 for cubins compiled prior to CUDA 3.0.

CU_FUNC_ATTRIBUTE_BINARY_VERSION = 6

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. Note that this will return a value of 10 for legacy cubins that do not have a properly-encoded binary architecture version.

CU_FUNC_ATTRIBUTE_CACHE_MODE_CA = 7

The attribute to indicate whether the function has been compiled with user specified option "-Xptxas --dlcm=ca" set .

CU_FUNC_ATTRIBUTE_MAX

enum CUgraphicsMapResourceFlags

Flags for mapping and unmapping interop resources

Values

CU_GRAPHICS_MAP_RESOURCE_FLAGS_NONE = 0x00
CU_GRAPHICS_MAP_RESOURCE_FLAGS_READ_ONLY = 0x01
CU_GRAPHICS_MAP_RESOURCE_FLAGS_WRITE_DISCARD = 0x02

enum CUgraphicsRegisterFlags

Flags to register a graphics resource

Values

CU_GRAPHICS_REGISTER_FLAGS_NONE = 0x00
CU_GRAPHICS_REGISTER_FLAGS_READ_ONLY = 0x01
CU_GRAPHICS_REGISTER_FLAGS_WRITE_DISCARD = 0x02
CU_GRAPHICS_REGISTER_FLAGS_SURFACE_LDST = 0x04
CU_GRAPHICS_REGISTER_FLAGS_TEXTURE_GATHER = 0x08

enum CUipcMem_flags

CUDA Ipc Mem Flags

Values

CU_IPC_MEM_LAZY_ENABLE_PEER_ACCESS = 0x1

Automatically enable peer access between remote devices as needed

enum CUjit_cacheMode

Caching modes for dlcm

Values

CU_JIT_CACHE_OPTION_NONE = 0
 Compile with no -dlcm flag specified
CU_JIT_CACHE_OPTION_CG
 Compile with L1 cache disabled
CU_JIT_CACHE_OPTION_CA
 Compile with L1 cache enabled

enum CUjit_fallback

Cubin matching fallback strategies

Values

CU_PREFER_PTX = 0
 Prefer to compile ptx if exact binary match not found
CU_PREFER_BINARY
 Prefer to fall back to compatible binary code if exact match not found

enum CUjit_option

Online compiler and linker options

Values

CU_JIT_MAX_REGISTERS = 0
 Max number of registers that a thread may use. Option type: unsigned int Applies to: compiler only
CU_JIT_THREADS_PER_BLOCK
 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 fo 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. Cannot be combined with CU_JIT_TARGET. Option type: unsigned int Applies to: compiler only
CU_JIT_WALL_TIME
 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
CU_JIT_INFO_LOG_BUFFER
 Pointer to a buffer in which to print any log messages that are informational in nature (the buffer size is specified via option CU_JIT_INFO_LOG_BUFFER_SIZE_BYTES) Option type: char * Applies to: compiler and linker
CU_JIT_INFO_LOG_BUFFER_SIZE_BYTES

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

CU_JIT_ERROR_LOG_BUFFER

Pointer to a buffer in which to print any log messages that reflect errors (the buffer size is specified via option CU_JIT_ERROR_LOG_BUFFER_SIZE_BYTES) Option type: char * Applies to: compiler and linker

CU_JIT_ERROR_LOG_BUFFER_SIZE_BYTES

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

CU_JIT_OPTIMIZATION_LEVEL

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

CU_JIT_TARGET_FROM_CUCONTEXT

No option value required. Determines the target based on the current attached context (default) Option type: No option value needed Applies to: compiler and linker

CU_JIT_TARGET

Target is chosen based on supplied CUjit_target. Cannot be combined with CU_JIT_THREADS_PER_BLOCK. Option type: unsigned int for enumerated type CUjit_target Applies to: compiler and linker

CU_JIT_FALLBACK_STRATEGY

Specifies choice of fallback strategy if matching cubin is not found. Choice is based on supplied CUjit_fallback. This option cannot be used with cuLink* APIs as the linker requires exact matches. Option type: unsigned int for enumerated type CUjit_fallback Applies to: compiler only

CU_JIT_GENERATE_DEBUG_INFO

Specifies whether to create debug information in output (-g) (0: false, default) Option type: int Applies to: compiler and linker

CU_JIT_LOG_VERBOSE

Generate verbose log messages (0: false, default) Option type: int Applies to: compiler and linker

CU_JIT_GENERATE_LINE_INFO

Generate line number information (-lineinfo) (0: false, default) Option type: int Applies to: compiler only

CU_JIT_CACHE_MODE

Specifies whether to enable caching explicitly (-dlcm) Choice is based on supplied CUjit_cacheMode_enum. Option type: unsigned int for enumerated type CUjit_cacheMode_enum Applies to: compiler only

CU_JIT_NUM_OPTIONS

enum CUjit_target

Online compilation targets

Values

CU_TARGET_COMPUTE_10 = 10
 Compute device class 1.0
CU_TARGET_COMPUTE_11 = 11
 Compute device class 1.1
CU_TARGET_COMPUTE_12 = 12
 Compute device class 1.2
CU_TARGET_COMPUTE_13 = 13
 Compute device class 1.3
CU_TARGET_COMPUTE_20 = 20
 Compute device class 2.0
CU_TARGET_COMPUTE_21 = 21
 Compute device class 2.1
CU_TARGET_COMPUTE_30 = 30
 Compute device class 3.0
CU_TARGET_COMPUTE_32 = 32
 Compute device class 3.2
CU_TARGET_COMPUTE_35 = 35
 Compute device class 3.5
CU_TARGET_COMPUTE_37 = 37
 Compute device class 3.7
CU_TARGET_COMPUTE_50 = 50
 Compute device class 5.0

enum CUjitInputType

Device code formats

Values

CU_JIT_INPUT_CUBIN = 0
 Compiled device-class-specific device code Applicable options: none
CU_JIT_INPUT_PTX
 PTX source code Applicable options: PTX compiler options
CU_JIT_INPUT_FATBINARY
 Bundle of multiple cubins and/or PTX of some device code Applicable options: PTX compiler options, CU_JIT_FALLBACK_STRATEGY
CU_JIT_INPUT_OBJECT

Host object with embedded device code Applicable options: PTX compiler options, CU_JIT_FALLBACK_STRATEGY

CU_JIT_INPUT_LIBRARY

Archive of host objects with embedded device code Applicable options: PTX compiler options, CU_JIT_FALLBACK_STRATEGY

CU_JIT_NUM_INPUT_TYPES

enum CUlimit

Limits

Values

CU_LIMIT_STACK_SIZE = 0x00
GPU thread stack size
CU_LIMIT_PRINTF_FIFO_SIZE = 0x01
GPU printf FIFO size
CU_LIMIT_MALLOC_HEAP_SIZE = 0x02
GPU malloc heap size
CU_LIMIT_DEV_RUNTIME_SYNC_DEPTH = 0x03
GPU device runtime launch synchronize depth
CU_LIMIT_DEV_RUNTIME_PENDING_LAUNCH_COUNT = 0x04
GPU device runtime pending launch count
CU_LIMIT_MAX

enum CUmemAttach_flags

CUDA Mem Attach Flags

Values

CU_MEM_ATTACH_GLOBAL = 0x1
Memory can be accessed by any stream on any device
CU_MEM_ATTACH_HOST = 0x2
Memory cannot be accessed by any stream on any device
CU_MEM_ATTACH_SINGLE = 0x4
Memory can only be accessed by a single stream on the associated device

enum CUmemorytype

Memory types

Values

CU_MEMORYTYPE_HOST = 0x01
Host memory

CU_MEMORYTYPE_DEVICE = 0x02
   Device memory
CU_MEMORYTYPE_ARRAY = 0x03
   Array memory
CU_MEMORYTYPE_UNIFIED = 0x04
   Unified device or host memory

enum CUpointer_attribute

Pointer information

Values

CU_POINTER_ATTRIBUTE_CONTEXT = 1
   The CUcontext on which a pointer was allocated or registered
CU_POINTER_ATTRIBUTE_MEMORY_TYPE = 2
   The CUmemorytype describing the physical location of a pointer
CU_POINTER_ATTRIBUTE_DEVICE_POINTER = 3
   The address at which a pointer's memory may be accessed on the device
CU_POINTER_ATTRIBUTE_HOST_POINTER = 4
   The address at which a pointer's memory may be accessed on the host
CU_POINTER_ATTRIBUTE_P2P_TOKENS = 5
   A pair of tokens for use with the nv-p2p.h Linux kernel interface
CU_POINTER_ATTRIBUTE_SYNC_MEMOPS = 6
   Synchronize every synchronous memory operation initiated on this region
CU_POINTER_ATTRIBUTE_BUFFER_ID = 7
   A process-wide unique ID for an allocated memory region
CU_POINTER_ATTRIBUTE_IS_MANAGED = 8
   Indicates if the pointer points to managed memory

enum CUresourcetype

Resource types

Values

CU_RESOURCE_TYPE_ARRAY = 0x00
   Array resoure
CU_RESOURCE_TYPE_MIPMAPPED_ARRAY = 0x01
   Mipmapped array resource
CU_RESOURCE_TYPE_LINEAR = 0x02
   Linear resource
CU_RESOURCE_TYPE_PITCH2D = 0x03
   Pitch 2D resource

enum CUresourceViewFormat

Resource view format

Values

CU_RES_VIEW_FORMAT_NONE = 0x00
No resource view format (use underlying resource format)
CU_RES_VIEW_FORMAT_UINT_1X8 = 0x01
1 channel unsigned 8-bit integers
CU_RES_VIEW_FORMAT_UINT_2X8 = 0x02
2 channel unsigned 8-bit integers
CU_RES_VIEW_FORMAT_UINT_4X8 = 0x03
4 channel unsigned 8-bit integers
CU_RES_VIEW_FORMAT_SINT_1X8 = 0x04
1 channel signed 8-bit integers
CU_RES_VIEW_FORMAT_SINT_2X8 = 0x05
2 channel signed 8-bit integers
CU_RES_VIEW_FORMAT_SINT_4X8 = 0x06
4 channel signed 8-bit integers
CU_RES_VIEW_FORMAT_UINT_1X16 = 0x07
1 channel unsigned 16-bit integers
CU_RES_VIEW_FORMAT_UINT_2X16 = 0x08
2 channel unsigned 16-bit integers
CU_RES_VIEW_FORMAT_UINT_4X16 = 0x09
4 channel unsigned 16-bit integers
CU_RES_VIEW_FORMAT_SINT_1X16 = 0x0a
1 channel signed 16-bit integers
CU_RES_VIEW_FORMAT_SINT_2X16 = 0x0b
2 channel signed 16-bit integers
CU_RES_VIEW_FORMAT_SINT_4X16 = 0x0c
4 channel signed 16-bit integers
CU_RES_VIEW_FORMAT_UINT_1X32 = 0x0d
1 channel unsigned 32-bit integers
CU_RES_VIEW_FORMAT_UINT_2X32 = 0x0e
2 channel unsigned 32-bit integers
CU_RES_VIEW_FORMAT_UINT_4X32 = 0x0f
4 channel unsigned 32-bit integers
CU_RES_VIEW_FORMAT_SINT_1X32 = 0x10
1 channel signed 32-bit integers
CU_RES_VIEW_FORMAT_SINT_2X32 = 0x11
2 channel signed 32-bit integers
CU_RES_VIEW_FORMAT_SINT_4X32 = 0x12

4 channel signed 32-bit integers

CU_RES_VIEW_FORMAT_FLOAT_1X16 = 0x13

    1 channel 16-bit floating point

CU_RES_VIEW_FORMAT_FLOAT_2X16 = 0x14

    2 channel 16-bit floating point

CU_RES_VIEW_FORMAT_FLOAT_4X16 = 0x15

    4 channel 16-bit floating point

CU_RES_VIEW_FORMAT_FLOAT_1X32 = 0x16

    1 channel 32-bit floating point

CU_RES_VIEW_FORMAT_FLOAT_2X32 = 0x17

    2 channel 32-bit floating point

CU_RES_VIEW_FORMAT_FLOAT_4X32 = 0x18

    4 channel 32-bit floating point

CU_RES_VIEW_FORMAT_UNSIGNED_BC1 = 0x19

    Block compressed 1

CU_RES_VIEW_FORMAT_UNSIGNED_BC2 = 0x1a

    Block compressed 2

CU_RES_VIEW_FORMAT_UNSIGNED_BC3 = 0x1b

    Block compressed 3

CU_RES_VIEW_FORMAT_UNSIGNED_BC4 = 0x1c

    Block compressed 4 unsigned

CU_RES_VIEW_FORMAT_SIGNED_BC4 = 0x1d

    Block compressed 4 signed

CU_RES_VIEW_FORMAT_UNSIGNED_BC5 = 0x1e

    Block compressed 5 unsigned

CU_RES_VIEW_FORMAT_SIGNED_BC5 = 0x1f

    Block compressed 5 signed

CU_RES_VIEW_FORMAT_UNSIGNED_BC6H = 0x20

    Block compressed 6 unsigned half-float

CU_RES_VIEW_FORMAT_SIGNED_BC6H = 0x21

    Block compressed 6 signed half-float

CU_RES_VIEW_FORMAT_UNSIGNED_BC7 = 0x22

    Block compressed 7

enum CUresult

Error codes

Values

CUDA_SUCCESS = 0

    The API call returned with no errors. In the case of query calls, this can also mean that the operation being queried is complete (see cuEventQuery() and cuStreamQuery()).

CUDA_ERROR_INVALID_VALUE = 1

This indicates that one or more of the parameters passed to the API call is not within an acceptable range of values.

CUDA_ERROR_OUT_OF_MEMORY = 2

The API call failed because it was unable to allocate enough memory to perform the requested operation.

CUDA_ERROR_NOT_INITIALIZED = 3

This indicates that the CUDA driver has not been initialized with cuInit() or that initialization has failed.

CUDA_ERROR_DEINITIALIZED = 4

This indicates that the CUDA driver is in the process of shutting down.

CUDA_ERROR_PROFILER_DISABLED = 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.

CUDA_ERROR_PROFILER_NOT_INITIALIZED = 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 cuProfilerStart or cuProfilerStop without initialization.

CUDA_ERROR_PROFILER_ALREADY_STARTED = 7

Deprecated This error return is deprecated as of CUDA 5.0. It is no longer an error to call cuProfilerStart() when profiling is already enabled.

CUDA_ERROR_PROFILER_ALREADY_STOPPED = 8

Deprecated This error return is deprecated as of CUDA 5.0. It is no longer an error to call cuProfilerStop() when profiling is already disabled.

CUDA_ERROR_NO_DEVICE = 100

This indicates that no CUDA-capable devices were detected by the installed CUDA driver.

CUDA_ERROR_INVALID_DEVICE = 101

This indicates that the device ordinal supplied by the user does not correspond to a valid CUDA device.

CUDA_ERROR_INVALID_IMAGE = 200

This indicates that the device kernel image is invalid. This can also indicate an invalid CUDA module.

CUDA_ERROR_INVALID_CONTEXT = 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.

CUDA_ERROR_CONTEXT_ALREADY_CURRENT = 202

This indicated that the context being supplied as a parameter to the API call was already the active context. Deprecated This error return is deprecated as of CUDA 3.2. It is no longer an error to attempt to push the active context via cuCtxPushCurrent().

CUDA_ERROR_MAP_FAILED = 205

This indicates that a map or register operation has failed.

CUDA_ERROR_UNMAP_FAILED = 206

This indicates that an unmap or unregister operation has failed.

CUDA_ERROR_ARRAY_IS_MAPPED = 207

This indicates that the specified array is currently mapped and thus cannot be destroyed.

CUDA_ERROR_ALREADY_MAPPED = 208

This indicates that the resource is already mapped.

CUDA_ERROR_NO_BINARY_FOR_GPU = 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.

CUDA_ERROR_ALREADY_ACQUIRED = 210

This indicates that a resource has already been acquired.

CUDA_ERROR_NOT_MAPPED = 211

This indicates that a resource is not mapped.

CUDA_ERROR_NOT_MAPPED_AS_ARRAY = 212

This indicates that a mapped resource is not available for access as an array.

CUDA_ERROR_NOT_MAPPED_AS_POINTER = 213

This indicates that a mapped resource is not available for access as a pointer.

CUDA_ERROR_ECC_UNCORRECTABLE = 214

This indicates that an uncorrectable ECC error was detected during execution.

CUDA_ERROR_UNSUPPORTED_LIMIT = 215

This indicates that the CUlimit passed to the API call is not supported by the active device.

CUDA_ERROR_CONTEXT_ALREADY_IN_USE = 216

This indicates that the CUcontext passed to the API call can only be bound to a single CPU thread at a time but is already bound to a CPU thread.

CUDA_ERROR_PEER_ACCESS_UNSUPPORTED = 217

This indicates that peer access is not supported across the given devices.

CUDA_ERROR_INVALID_PTX = 218

This indicates that a PTX JIT compilation failed.

CUDA_ERROR_INVALID_GRAPHICS_CONTEXT = 219

This indicates an error with OpenGL or DirectX context.

CUDA_ERROR_INVALID_SOURCE = 300

This indicates that the device kernel source is invalid.

CUDA_ERROR_FILE_NOT_FOUND = 301

This indicates that the file specified was not found.

CUDA_ERROR_SHARED_OBJECT_SYMBOL_NOT_FOUND = 302

This indicates that a link to a shared object failed to resolve.

CUDA_ERROR_SHARED_OBJECT_INIT_FAILED = 303

This indicates that initialization of a shared object failed.

CUDA_ERROR_OPERATING_SYSTEM = 304

This indicates that an OS call failed.

CUDA_ERROR_INVALID_HANDLE = 400

This indicates that a resource handle passed to the API call was not valid. Resource handles are opaque types like CUstream and CUevent.

CUDA_ERROR_NOT_FOUND = 500

This indicates that a named symbol was not found. Examples of symbols are global/constant variable names, texture names, and surface names.

CUDA_ERROR_NOT_READY = 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 CUDA_SUCCESS (which indicates completion). Calls that may return this value include cuEventQuery() and cuStreamQuery().

CUDA_ERROR_ILLEGAL_ADDRESS = 700

While executing a kernel, the device encountered a load or store instruction on an invalid memory address. The context cannot be used, so it must be destroyed (and a new one should be created). All existing device memory allocations from this context are invalid and must be reconstructed if the program is to continue using CUDA.

CUDA_ERROR_LAUNCH_OUT_OF_RESOURCES = 701

This indicates that a launch did not occur because it did not have appropriate resources. 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. Passing arguments of the wrong size (i.e. a 64-bit pointer when a 32-bit int is expected) is equivalent to passing too many arguments and can also result in this error.

CUDA_ERROR_LAUNCH_TIMEOUT = 702

This indicates that the device kernel took too long to execute. This can only occur if timeouts are enabled - see the device attribute CU_DEVICE_ATTRIBUTE_KERNEL_EXEC_TIMEOUT for more information. The context cannot be used (and must be destroyed similar to CUDA_ERROR_LAUNCH_FAILED). All existing device memory allocations from this context are invalid and must be reconstructed if the program is to continue using CUDA.

CUDA_ERROR_LAUNCH_INCOMPATIBLE_TEXTURING = 703

This error indicates a kernel launch that uses an incompatible texturing mode.

CUDA_ERROR_PEER_ACCESS_ALREADY_ENABLED = 704

This error indicates that a call to cuCtxEnablePeerAccess() is trying to re-enable peer access to a context which has already had peer access to it enabled.

CUDA_ERROR_PEER_ACCESS_NOT_ENABLED = 705

This error indicates that cuCtxDisablePeerAccess() is trying to disable peer access which has not been enabled yet via cuCtxEnablePeerAccess().

CUDA_ERROR_PRIMARY_CONTEXT_ACTIVE = 708

This error indicates that the primary context for the specified device has already been initialized.

CUDA_ERROR_CONTEXT_IS_DESTROYED = 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.

CUDA_ERROR_ASSERT = 710

A device-side assert triggered during kernel execution. The context cannot be used anymore, and must be destroyed. All existing device memory allocations from this context are invalid and must be reconstructed if the program is to continue using CUDA.

CUDA_ERROR_TOO_MANY_PEERS = 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 cuCtxEnablePeerAccess().

CUDA_ERROR_HOST_MEMORY_ALREADY_REGISTERED = 712

This error indicates that the memory range passed to cuMemHostRegister() has already been registered.

CUDA_ERROR_HOST_MEMORY_NOT_REGISTERED = 713

This error indicates that the pointer passed to cuMemHostUnregister() does not correspond to any currently registered memory region.

CUDA_ERROR_HARDWARE_STACK_ERROR = 714

While executing a kernel, the device encountered a stack error. This can be due to stack corruption or exceeding the stack size limit. The context cannot be used, so it must be destroyed (and a new one should be created). All existing device memory allocations from this context are invalid and must be reconstructed if the program is to continue using CUDA.

CUDA_ERROR_ILLEGAL_INSTRUCTION = 715

While executing a kernel, the device encountered an illegal instruction. The context cannot be used, so it must be destroyed (and a new one should be created). All existing device memory allocations from this context are invalid and must be reconstructed if the program is to continue using CUDA.

CUDA_ERROR_MISALIGNED_ADDRESS = 716

While executing a kernel, the device encountered a load or store instruction on a memory address which is not aligned. The context cannot be used, so it must be destroyed (and a new one should be created). All existing device memory allocations from this context are invalid and must be reconstructed if the program is to continue using CUDA.

CUDA_ERROR_INVALID_ADDRESS_SPACE = 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. The context cannot be used, so it must be destroyed (and a new one should be created). All existing device memory allocations from this context are invalid and must be reconstructed if the program is to continue using CUDA.

CUDA_ERROR_INVALID_PC = 718

While executing a kernel, the device program counter wrapped its address space. The context cannot be used, so it must be destroyed (and a new one should be created). All existing device memory allocations from this context are invalid and must be reconstructed if the program is to continue using CUDA.

CUDA_ERROR_LAUNCH_FAILED = 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. The context cannot be used, so it must be destroyed (and a new one should be created). All existing device memory allocations from this context are invalid and must be reconstructed if the program is to continue using CUDA.

CUDA_ERROR_NOT_PERMITTED = 800

This error indicates that the attempted operation is not permitted.

CUDA_ERROR_NOT_SUPPORTED = 801

This error indicates that the attempted operation is not supported on the current system or device.

CUDA_ERROR_UNKNOWN = 999

This indicates that an unknown internal error has occurred.

enum CUsharedconfig

Shared memory configurations

Values

CU_SHARED_MEM_CONFIG_DEFAULT_BANK_SIZE = 0x00

set default shared memory bank size

CU_SHARED_MEM_CONFIG_FOUR_BYTE_BANK_SIZE = 0x01

set shared memory bank width to four bytes

CU_SHARED_MEM_CONFIG_EIGHT_BYTE_BANK_SIZE = 0x02

set shared memory bank width to eight bytes

enum CUstream_flags

Stream creation flags

Values

CU_STREAM_DEFAULT = 0x0

Default stream flag

CU_STREAM_NON_BLOCKING = 0x1

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

typedef struct CUarray_st *CUarray

CUDA array

typedef struct CUctx_st *CUcontext

CUDA context

typedef int CUdevice

CUDA device

typedef unsigned int CUdeviceptr

CUDA device pointer

typedef struct CUevent_st *CUevent

CUDA event

typedef struct CUfunc_st *CUfunction

CUDA function

typedef struct CUgraphicsResource_st *CUgraphicsResource

CUDA graphics interop resource

typedef struct CUmipmappedArray_st *CUmipmappedArray

CUDA mipmapped array

typedef struct CUmod_st *CUmodule

CUDA module

typedef size_t (CUDA_CB *CUoccupancyB2DSize) (int blockSize)

Block size to per-block dynamic shared memory mapping for a certain kernel

typedef struct CUstream_st *CUstream

CUDA stream

typedef void (CUDA_CB *CUstreamCallback) (CUstream hStream, CUresult status, void* userData)

CUDA stream callback

typedef unsigned long long CUsurfObject

CUDA surface object

typedef struct CUsurfref_st *CUsurfref

CUDA surface reference

typedef unsigned long long CUtexObject

CUDA texture object

typedef struct CUtexref_st *CUtexref

CUDA texture reference

#define CU_IPC_HANDLE_SIZE 64

CUDA IPC handle size

#define CU_LAUNCH_PARAM_BUFFER_POINTER ((void*)0x01)

Indicator that the next value in the `extra` parameter to cuLaunchKernel will be a pointer to a buffer containing all kernel parameters used for launching kernel `f`. This buffer needs to honor all alignment/padding requirements of the individual parameters. If CU_LAUNCH_PARAM_BUFFER_SIZE is not also specified in the `extra` array, then CU_LAUNCH_PARAM_BUFFER_POINTER will have no effect.

#define CU_LAUNCH_PARAM_BUFFER_SIZE ((void*)0x02)

Indicator that the next value in the `extra` parameter to cuLaunchKernel will be a pointer to a size_t which contains the size of the buffer specified with CU_LAUNCH_PARAM_BUFFER_POINTER. It is required that

CU_LAUNCH_PARAM_BUFFER_POINTER also be specified in the `extra` array if the value associated with CU_LAUNCH_PARAM_BUFFER_SIZE is not zero.

#define CU_LAUNCH_PARAM_END ((void*)0x00)

End of array terminator for the `extra` parameter to cuLaunchKernel

#define CU_MEMHOSTALLOC_DEVICEMAP 0x02

If set, host memory is mapped into CUDA address space and cuMemHostGetDevicePointer() may be called on the host pointer. Flag for cuMemHostAlloc()

#define CU_MEMHOSTALLOC_PORTABLE 0x01

If set, host memory is portable between CUDA contexts. Flag for cuMemHostAlloc()

#define CU_MEMHOSTALLOC_WRITECOMBINED 0x04

If set, host memory is allocated as write-combined - fast to write, faster to DMA, slow to read except via SSE4 streaming load instruction (MOVNTDQA). Flag for cuMemHostAlloc()

#define CU_MEMHOSTREGISTER_DEVICEMAP 0x02

If set, host memory is mapped into CUDA address space and cuMemHostGetDevicePointer() may be called on the host pointer. Flag for cuMemHostRegister()

#define CU_MEMHOSTREGISTER_PORTABLE 0x01

If set, host memory is portable between CUDA contexts. Flag for cuMemHostRegister()

#define CU_PARAM_TR_DEFAULT -1

For texture references loaded into the module, use default texunit from texture reference.

#define CU_TRSA_OVERRIDE_FORMAT 0x01

Override the texref format with a format inferred from the array. Flag for cuTexRefSetArray()

#define CU_TRSF_NORMALIZED_COORDINATES 0x02

Use normalized texture coordinates in the range [0,1) instead of [0,dim). Flag for cuTexRefSetFlags()

#define CU_TRSF_READ_AS_INTEGER 0x01

Read the texture as integers rather than promoting the values to floats in the range [0,1]. Flag for cuTexRefSetFlags()

#define CU_TRSF_SRGB 0x10

Perform sRGB->linear conversion during texture read. Flag for cuTexRefSetFlags()

#define CUDA_ARRAY3D_2DARRAY 0x01

Deprecated, use CUDA_ARRAY3D_LAYERED

#define CUDA_ARRAY3D_CUBEMAP 0x04

If set, the CUDA array is a collection of six 2D arrays, representing faces of a cube. The width of such a CUDA array must be equal to its height, and Depth must be six. If CUDA_ARRAY3D_LAYERED flag is also set, then the CUDA array is a collection of cubemaps and Depth must be a multiple of six.

#define CUDA_ARRAY3D_DEPTH_TEXTURE 0x10

This flag if set indicates that the CUDA array is a DEPTH_TEXTURE.

#define CUDA_ARRAY3D_LAYERED 0x01

If set, the CUDA array is a collection of layers, where each layer is either a 1D or a 2D array and the Depth member of CUDA_ARRAY3D_DESCRIPTOR specifies the number of layers, not the depth of a 3D array.

#define CUDA_ARRAY3D_SURFACE_LDST 0x02

This flag must be set in order to bind a surface reference to the CUDA array

#define CUDA_ARRAY3D_TEXTURE_GATHER 0x08

This flag must be set in order to perform texture gather operations on a CUDA array.

#define CUDA_VERSION 6050

CUDA API version number

3.2. Error Handling

This section describes the error handling functions of the low-level CUDA driver application programming interface.

CUresult cuGetErrorName (CUresult error, const char **pStr)

Gets the string representation of an error code enum name.

Parameters

error
- Error code to convert to string
pStr
- Address of the string pointer.

Returns

CUDA_SUCCESS, CUDA_ERROR_INVALID_VALUE

Description

Sets *pStr to the address of a NULL-terminated string representation of the name of the enum error code error. If the error code is not recognized, CUDA_ERROR_INVALID_VALUE will be returned and *pStr will be set to the NULL address.

See also:

CUresult

CUresult cuGetErrorString (CUresult error, const char **pStr)

Gets the string description of an error code.

Parameters

error
- Error code to convert to string

pStr
   - Address of the string pointer.

Returns

CUDA_SUCCESS, CUDA_ERROR_INVALID_VALUE

Description

Sets `*pStr` to the address of a NULL-terminated string description of the error code `error`. If the error code is not recognized, CUDA_ERROR_INVALID_VALUE will be returned and `*pStr` will be set to the NULL address.

See also:

CUresult

3.3. Initialization

This section describes the initialization functions of the low-level CUDA driver application programming interface.

CUresult cuInit (unsigned int Flags)

Initialize the CUDA driver API.

Parameters

Flags
   - Initialization flag for CUDA.

Returns

CUDA_SUCCESS, CUDA_ERROR_INVALID_VALUE,
CUDA_ERROR_INVALID_DEVICE

Description

Initializes the driver API and must be called before any other function from the driver API. Currently, the `Flags` parameter must be 0. If cuInit() has not been called, any function from the driver API will return CUDA_ERROR_NOT_INITIALIZED.

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

3.4. Version Management

This section describes the version management functions of the low-level CUDA driver application programming interface.

CUresult cuDriverGetVersion (int *driverVersion)

Returns the CUDA driver version.

Parameters

driverVersion
   - Returns the CUDA driver version

Returns

CUDA_SUCCESS, CUDA_ERROR_INVALID_VALUE

Description

Returns in `*driverVersion` the version number of the installed CUDA driver. This function automatically returns CUDA_ERROR_INVALID_VALUE if the `driverVersion` argument is NULL.

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

3.5. Device Management

This section describes the device management functions of the low-level CUDA driver application programming interface.

CUresult cuDeviceGet (CUdevice *device, int ordinal)

Returns a handle to a compute device.

Parameters

device
   - Returned device handle
ordinal
   - Device number to get handle for

Returns

CUDA_SUCCESS, CUDA_ERROR_DEINITIALIZED,
CUDA_ERROR_NOT_INITIALIZED, CUDA_ERROR_INVALID_CONTEXT,
CUDA_ERROR_INVALID_VALUE, CUDA_ERROR_INVALID_DEVICE

Description

Returns in `*device` a device handle given an ordinal in the range **[0, cuDeviceGetCount()-1]**.

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

See also:

cuDeviceGetAttribute, cuDeviceGetCount, cuDeviceGetName, cuDeviceTotalMem

CUresult cuDeviceGetAttribute (int *pi, CUdevice_attribute attrib, CUdevice dev)

Returns information about the device.

Parameters

pi
   - Returned device attribute value
attrib
   - Device attribute to query
dev
   - Device handle

Returns

CUDA_SUCCESS, CUDA_ERROR_DEINITIALIZED,
CUDA_ERROR_NOT_INITIALIZED, CUDA_ERROR_INVALID_CONTEXT,
CUDA_ERROR_INVALID_VALUE, CUDA_ERROR_INVALID_DEVICE

Description

Returns in `*pi` the integer value of the attribute `attrib` on device `dev`. The supported attributes are:

▸   CU_DEVICE_ATTRIBUTE_MAX_THREADS_PER_BLOCK: Maximum number of threads per block;

▸ CU_DEVICE_ATTRIBUTE_MAX_BLOCK_DIM_X: Maximum x-dimension of a block;

▸ CU_DEVICE_ATTRIBUTE_MAX_BLOCK_DIM_Y: Maximum y-dimension of a block;

▸ CU_DEVICE_ATTRIBUTE_MAX_BLOCK_DIM_Z: Maximum z-dimension of a block;

▸ CU_DEVICE_ATTRIBUTE_MAX_GRID_DIM_X: Maximum x-dimension of a grid;

▸ CU_DEVICE_ATTRIBUTE_MAX_GRID_DIM_Y: Maximum y-dimension of a grid;

▸ CU_DEVICE_ATTRIBUTE_MAX_GRID_DIM_Z: Maximum z-dimension of a grid;

▸ CU_DEVICE_ATTRIBUTE_MAX_SHARED_MEMORY_PER_BLOCK: Maximum amount of shared memory available to a thread block in bytes;

▸ CU_DEVICE_ATTRIBUTE_TOTAL_CONSTANT_MEMORY: Memory available on device for __constant__ variables in a CUDA C kernel in bytes;

▸ CU_DEVICE_ATTRIBUTE_WARP_SIZE: Warp size in threads;

▸ CU_DEVICE_ATTRIBUTE_MAX_PITCH: Maximum pitch in bytes allowed by the memory copy functions that involve memory regions allocated through cuMemAllocPitch();

▸ CU_DEVICE_ATTRIBUTE_MAXIMUM_TEXTURE1D_WIDTH: Maximum 1D texture width;

▸ CU_DEVICE_ATTRIBUTE_MAXIMUM_TEXTURE1D_LINEAR_WIDTH: Maximum width for a 1D texture bound to linear memory;

▸ CU_DEVICE_ATTRIBUTE_MAXIMUM_TEXTURE1D_MIPMAPPED_WIDTH: Maximum mipmapped 1D texture width;

▸ CU_DEVICE_ATTRIBUTE_MAXIMUM_TEXTURE2D_WIDTH: Maximum 2D texture width;

▸ CU_DEVICE_ATTRIBUTE_MAXIMUM_TEXTURE2D_HEIGHT: Maximum 2D texture height;

▸ CU_DEVICE_ATTRIBUTE_MAXIMUM_TEXTURE2D_LINEAR_WIDTH: Maximum width for a 2D texture bound to linear memory;

▸ CU_DEVICE_ATTRIBUTE_MAXIMUM_TEXTURE2D_LINEAR_HEIGHT: Maximum height for a 2D texture bound to linear memory;

▸ CU_DEVICE_ATTRIBUTE_MAXIMUM_TEXTURE2D_LINEAR_PITCH: Maximum pitch in bytes for a 2D texture bound to linear memory;

▸ CU_DEVICE_ATTRIBUTE_MAXIMUM_TEXTURE2D_MIPMAPPED_WIDTH: Maximum mipmapped 2D texture width;

▸ CU_DEVICE_ATTRIBUTE_MAXIMUM_TEXTURE2D_MIPMAPPED_HEIGHT: Maximum mipmapped 2D texture height;

▸ CU_DEVICE_ATTRIBUTE_MAXIMUM_TEXTURE3D_WIDTH: Maximum 3D texture width;

▸ CU_DEVICE_ATTRIBUTE_MAXIMUM_TEXTURE3D_HEIGHT: Maximum 3D texture height;

▸ CU_DEVICE_ATTRIBUTE_MAXIMUM_TEXTURE3D_DEPTH: Maximum 3D texture depth;

▸ CU_DEVICE_ATTRIBUTE_MAXIMUM_TEXTURE3D_WIDTH_ALTERNATE: Alternate maximum 3D texture width, 0 if no alternate maximum 3D texture size is supported;

▸ CU_DEVICE_ATTRIBUTE_MAXIMUM_TEXTURE3D_HEIGHT_ALTERNATE: Alternate maximum 3D texture height, 0 if no alternate maximum 3D texture size is supported;

▸ CU_DEVICE_ATTRIBUTE_MAXIMUM_TEXTURE3D_DEPTH_ALTERNATE: Alternate maximum 3D texture depth, 0 if no alternate maximum 3D texture size is supported;

▸ CU_DEVICE_ATTRIBUTE_MAXIMUM_TEXTURECUBEMAP_WIDTH: Maximum cubemap texture width or height;

▸ CU_DEVICE_ATTRIBUTE_MAXIMUM_TEXTURE1D_LAYERED_WIDTH: Maximum 1D layered texture width;

▸ CU_DEVICE_ATTRIBUTE_MAXIMUM_TEXTURE1D_LAYERED_LAYERS: Maximum layers in a 1D layered texture;

▸ CU_DEVICE_ATTRIBUTE_MAXIMUM_TEXTURE2D_LAYERED_WIDTH: Maximum 2D layered texture width;

▸ CU_DEVICE_ATTRIBUTE_MAXIMUM_TEXTURE2D_LAYERED_HEIGHT: Maximum 2D layered texture height;

▸ CU_DEVICE_ATTRIBUTE_MAXIMUM_TEXTURE2D_LAYERED_LAYERS: Maximum layers in a 2D layered texture;

▸ CU_DEVICE_ATTRIBUTE_MAXIMUM_TEXTURECUBEMAP_LAYERED_WIDTH: Maximum cubemap layered texture width or height;

▸ CU_DEVICE_ATTRIBUTE_MAXIMUM_TEXTURECUBEMAP_LAYERED_LAYERS: Maximum layers in a cubemap layered texture;

▸ CU_DEVICE_ATTRIBUTE_MAXIMUM_SURFACE1D_WIDTH: Maximum 1D surface width;

▸ CU_DEVICE_ATTRIBUTE_MAXIMUM_SURFACE2D_WIDTH: Maximum 2D surface width;

▸ CU_DEVICE_ATTRIBUTE_MAXIMUM_SURFACE2D_HEIGHT: Maximum 2D surface height;

▸ CU_DEVICE_ATTRIBUTE_MAXIMUM_SURFACE3D_WIDTH: Maximum 3D surface width;

▸ CU_DEVICE_ATTRIBUTE_MAXIMUM_SURFACE3D_HEIGHT: Maximum 3D surface height;

▸ CU_DEVICE_ATTRIBUTE_MAXIMUM_SURFACE3D_DEPTH: Maximum 3D surface depth;

▸ CU_DEVICE_ATTRIBUTE_MAXIMUM_SURFACE1D_LAYERED_WIDTH: Maximum 1D layered surface width;

▸ CU_DEVICE_ATTRIBUTE_MAXIMUM_SURFACE1D_LAYERED_LAYERS:
Maximum layers in a 1D layered surface;

▸ CU_DEVICE_ATTRIBUTE_MAXIMUM_SURFACE2D_LAYERED_WIDTH:
Maximum 2D layered surface width;

▸ CU_DEVICE_ATTRIBUTE_MAXIMUM_SURFACE2D_LAYERED_HEIGHT:
Maximum 2D layered surface height;

▸ CU_DEVICE_ATTRIBUTE_MAXIMUM_SURFACE2D_LAYERED_LAYERS:
Maximum layers in a 2D layered surface;

▸ CU_DEVICE_ATTRIBUTE_MAXIMUM_SURFACECUBEMAP_WIDTH: Maximum
cubemap surface width;

▸ CU_DEVICE_ATTRIBUTE_MAXIMUM_SURFACECUBEMAP_LAYERED_WIDTH:
Maximum cubemap layered surface width;

▸ CU_DEVICE_ATTRIBUTE_MAXIMUM_SURFACECUBEMAP_LAYERED_LAYERS:
Maximum layers in a cubemap layered surface;

▸ CU_DEVICE_ATTRIBUTE_MAX_REGISTERS_PER_BLOCK: Maximum number of
32-bit registers available to a thread block;

▸ CU_DEVICE_ATTRIBUTE_CLOCK_RATE: The typical clock frequency in kilohertz;

▸ CU_DEVICE_ATTRIBUTE_TEXTURE_ALIGNMENT: Alignment requirement;
texture base addresses aligned to textureAlign bytes do not need an offset applied to
texture fetches;

▸ CU_DEVICE_ATTRIBUTE_TEXTURE_PITCH_ALIGNMENT: Pitch alignment
requirement for 2D texture references bound to pitched memory;

▸ CU_DEVICE_ATTRIBUTE_GPU_OVERLAP: 1 if the device can concurrently copy
memory between host and device while executing a kernel, or 0 if not;

▸ CU_DEVICE_ATTRIBUTE_MULTIPROCESSOR_COUNT: Number of
multiprocessors on the device;

▸ CU_DEVICE_ATTRIBUTE_KERNEL_EXEC_TIMEOUT: 1 if there is a run time limit
for kernels executed on the device, or 0 if not;

▸ CU_DEVICE_ATTRIBUTE_INTEGRATED: 1 if the device is integrated with the
memory subsystem, or 0 if not;

▸ CU_DEVICE_ATTRIBUTE_CAN_MAP_HOST_MEMORY: 1 if the device can map
host memory into the CUDA address space, or 0 if not;

▸ CU_DEVICE_ATTRIBUTE_COMPUTE_MODE: Compute mode that device is
currently in. Available modes are as follows:

  ▸ CU_COMPUTEMODE_DEFAULT: Default mode - Device is not restricted and
  can have multiple CUDA contexts present at a single time.

  ▸ CU_COMPUTEMODE_EXCLUSIVE: Compute-exclusive mode - Device can
  have only one CUDA context present on it at a time.

  ▸ CU_COMPUTEMODE_PROHIBITED: Compute-prohibited mode - Device is
  prohibited from creating new CUDA contexts.

▸ **CU_COMPUTEMODE_EXCLUSIVE_PROCESS**: Compute-exclusive-process mode - Device can have only one context used by a single process at a time.

▸ **CU_DEVICE_ATTRIBUTE_CONCURRENT_KERNELS**: 1 if the device supports executing multiple kernels within the same context simultaneously, or 0 if not. It is not guaranteed that multiple kernels will be resident on the device concurrently so this feature should not be relied upon for correctness;

▸ **CU_DEVICE_ATTRIBUTE_ECC_ENABLED**: 1 if error correction is enabled on the device, 0 if error correction is disabled or not supported by the device;

▸ **CU_DEVICE_ATTRIBUTE_PCI_BUS_ID**: PCI bus identifier of the device;

▸ **CU_DEVICE_ATTRIBUTE_PCI_DEVICE_ID**: PCI device (also known as slot) identifier of the device;

▸ **CU_DEVICE_ATTRIBUTE_TCC_DRIVER**: 1 if the device is using a TCC driver. TCC is only available on Tesla hardware running Windows Vista or later;

▸ **CU_DEVICE_ATTRIBUTE_MEMORY_CLOCK_RATE**: Peak memory clock frequency in kilohertz;

▸ **CU_DEVICE_ATTRIBUTE_GLOBAL_MEMORY_BUS_WIDTH**: Global memory bus width in bits;

▸ **CU_DEVICE_ATTRIBUTE_L2_CACHE_SIZE**: Size of L2 cache in bytes. 0 if the device doesn't have L2 cache;

▸ **CU_DEVICE_ATTRIBUTE_MAX_THREADS_PER_MULTIPROCESSOR**: Maximum resident threads per multiprocessor;

▸ **CU_DEVICE_ATTRIBUTE_UNIFIED_ADDRESSING**: 1 if the device shares a unified address space with the host, or 0 if not;

▸ **CU_DEVICE_ATTRIBUTE_COMPUTE_CAPABILITY_MAJOR**: Major compute capability version number;

▸ **CU_DEVICE_ATTRIBUTE_COMPUTE_CAPABILITY_MINOR**: Minor compute capability version number;

▸ **CU_DEVICE_ATTRIBUTE_GLOBAL_L1_CACHE_SUPPORTED**: 1 if device supports caching globals in L1 cache, 0 if caching globals in L1 cache is not supported by the device;

▸ **CU_DEVICE_ATTRIBUTE_LOCAL_L1_CACHE_SUPPORTED**: 1 if device supports caching locals in L1 cache, 0 if caching locals in L1 cache is not supported by the device;

▸ **CU_DEVICE_ATTRIBUTE_MAX_SHARED_MEMORY_PER_MULTIPROCESSOR**: Maximum amount of shared memory available to a multiprocessor in bytes; this amount is shared by all thread blocks simultaneously resident on a multiprocessor;

▸ **CU_DEVICE_ATTRIBUTE_MAX_REGISTERS_PER_MULTIPROCESSOR**: Maximum number of 32-bit registers available to a multiprocessor; this number is shared by all thread blocks simultaneously resident on a multiprocessor;

▸ **CU_DEVICE_ATTRIBUTE_MANAGED_MEMORY**: 1 if device supports allocating managed memory on this system, 0 if allocating managed memory is not supported by the device on this system.

- ▸ CU_DEVICE_ATTRIBUTE_MULTI_GPU_BOARD: 1 if device is on a multi-GPU board, 0 if not.
- ▸ CU_DEVICE_ATTRIBUTE_MULTI_GPU_BOARD_GROUP_ID: Unique identifier for a group of devices associated with the same board. Devices on the same multi-GPU board will share the same identifier.

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

See also:

cuDeviceGetCount, cuDeviceGetName, cuDeviceGet, cuDeviceTotalMem

CUresult cuDeviceGetCount (int *count)

Returns the number of compute-capable devices.

Parameters

count
  - Returned number of compute-capable devices

Returns

CUDA_SUCCESS, CUDA_ERROR_DEINITIALIZED, CUDA_ERROR_NOT_INITIALIZED, CUDA_ERROR_INVALID_CONTEXT, CUDA_ERROR_INVALID_VALUE

Description

Returns in `*count` the number of devices with compute capability greater than or equal to 1.0 that are available for execution. If there is no such device, cuDeviceGetCount() returns 0.

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

See also:

cuDeviceGetAttribute, cuDeviceGetName, cuDeviceGet, cuDeviceTotalMem

CUresult cuDeviceGetName (char *name, int len, CUdevice dev)

Returns an identifer string for the device.

Parameters

name
   - Returned identifier string for the device
len
   - Maximum length of string to store in `name`
dev
   - Device to get identifier string for

Returns

CUDA_SUCCESS, CUDA_ERROR_DEINITIALIZED,
CUDA_ERROR_NOT_INITIALIZED, CUDA_ERROR_INVALID_CONTEXT,
CUDA_ERROR_INVALID_VALUE, CUDA_ERROR_INVALID_DEVICE

Description

Returns an ASCII string identifying the device `dev` in the NULL-terminated string pointed to by `name`. `len` specifies the maximum length of the string that may be returned.

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

See also:

cuDeviceGetAttribute, cuDeviceGetCount, cuDeviceGet, cuDeviceTotalMem

CUresult cuDeviceTotalMem (size_t *bytes, CUdevice dev)

Returns the total amount of memory on the device.

Parameters

bytes
   - Returned memory available on device in bytes
dev
   - Device handle

Returns

CUDA_SUCCESS, CUDA_ERROR_DEINITIALIZED,
CUDA_ERROR_NOT_INITIALIZED, CUDA_ERROR_INVALID_CONTEXT,
CUDA_ERROR_INVALID_VALUE, CUDA_ERROR_INVALID_DEVICE

Description

Returns in `*bytes` the total amount of memory available on the device `dev` in bytes.

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

See also:

cuDeviceGetAttribute, cuDeviceGetCount, cuDeviceGetName, cuDeviceGet,

3.6. Device Management [DEPRECATED]

This section describes the device management functions of the low-level CUDA driver application programming interface.

CUresult cuDeviceComputeCapability (int *major, int *minor, CUdevice dev)

Returns the compute capability of the device.

Parameters

major
   - Major revision number
minor
   - Minor revision number
dev
   - Device handle

Returns

CUDA_SUCCESS, CUDA_ERROR_DEINITIALIZED,
CUDA_ERROR_NOT_INITIALIZED, CUDA_ERROR_INVALID_CONTEXT,
CUDA_ERROR_INVALID_VALUE, CUDA_ERROR_INVALID_DEVICE

Description

Deprecated

This function was deprecated as of CUDA 5.0 and its functionality superceded by cuDeviceGetAttribute().

Returns in `*major` and `*minor` the major and minor revision numbers that define the compute capability of the device `dev`.

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

See also:

cuDeviceGetAttribute, cuDeviceGetCount, cuDeviceGetName, cuDeviceGet, cuDeviceTotalMem

CUresult cuDeviceGetProperties (CUdevprop *prop, CUdevice dev)

Returns properties for a selected device.

Parameters

prop
   - Returned properties of device
dev
   - Device to get properties for

Returns

CUDA_SUCCESS, CUDA_ERROR_DEINITIALIZED, CUDA_ERROR_NOT_INITIALIZED, CUDA_ERROR_INVALID_CONTEXT, CUDA_ERROR_INVALID_VALUE, CUDA_ERROR_INVALID_DEVICE

Description

Deprecated

This function was deprecated as of CUDA 5.0 and replaced by cuDeviceGetAttribute().

Returns in `*prop` the properties of device `dev`. The CUdevprop structure is defined as:

```
typedef struct CUdevprop_st {
        int maxThreadsPerBlock;
        int maxThreadsDim[3];
        int maxGridSize[3];
        int sharedMemPerBlock;
        int totalConstantMemory;
        int SIMDWidth;
        int memPitch;
        int regsPerBlock;
        int clockRate;
        int textureAlign
    } CUdevprop;
```

where:

▸   maxThreadsPerBlock is the maximum number of threads per block;

▸   maxThreadsDim[3] is the maximum sizes of each dimension of a block;

▸   maxGridSize[3] is the maximum sizes of each dimension of a grid;

▸   sharedMemPerBlock is the total amount of shared memory available per block in bytes;

▸   totalConstantMemory is the total amount of constant memory available on the device in bytes;

▸   SIMDWidth is the warp size;

▸   memPitch is the maximum pitch allowed by the memory copy functions that involve memory regions allocated through cuMemAllocPitch();

▸   regsPerBlock is the total number of registers available per block;

▸   clockRate is the clock frequency in kilohertz;

▸   textureAlign is the alignment requirement; texture base addresses that are aligned to textureAlign bytes do not need an offset applied to texture fetches.

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

See also:

cuDeviceGetAttribute, cuDeviceGetCount, cuDeviceGetName, cuDeviceGet, cuDeviceTotalMem

3.7. Context Management

This section describes the context management functions of the low-level CUDA driver application programming interface.

CUresult cuCtxCreate (CUcontext *pctx, unsigned int flags, CUdevice dev)

Create a CUDA context.

Parameters

pctx
  - Returned context handle of the new context
flags
  - Context creation flags
dev
  - Device to create context on

Returns

CUDA_SUCCESS, CUDA_ERROR_DEINITIALIZED,
CUDA_ERROR_NOT_INITIALIZED, CUDA_ERROR_INVALID_CONTEXT,
CUDA_ERROR_INVALID_DEVICE, CUDA_ERROR_INVALID_VALUE,
CUDA_ERROR_OUT_OF_MEMORY, CUDA_ERROR_UNKNOWN

Description

Creates a new CUDA context and associates it with the calling thread. The `flags` parameter is described below. The context is created with a usage count of 1 and the caller of cuCtxCreate() must call cuCtxDestroy() or when done using the context. If a context is already current to the thread, it is supplanted by the newly created context and may be restored by a subsequent call to cuCtxPopCurrent().

The three LSBs of the `flags` parameter can be used to control how the OS thread, which owns the CUDA context at the time of an API call, interacts with the OS scheduler when waiting for results from the GPU. Only one of the scheduling flags can be set when creating a context.

▸ CU_CTX_SCHED_AUTO: 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 GPU, otherwise CUDA will not yield while waiting for results and actively spin on the processor.

▸ CU_CTX_SCHED_SPIN: Instruct CUDA to actively spin when waiting for results from the GPU. This can decrease latency when waiting for the GPU, but may lower the performance of CPU threads if they are performing work in parallel with the CUDA thread.

▸ CU_CTX_SCHED_YIELD: Instruct CUDA to yield its thread when waiting for results from the GPU. This can increase latency when waiting for the GPU, but can increase the performance of CPU threads performing work in parallel with the GPU.

▸ CU_CTX_SCHED_BLOCKING_SYNC: Instruct CUDA to block the CPU thread on a synchronization primitive when waiting for the GPU to finish work.

▸ CU_CTX_BLOCKING_SYNC: Instruct CUDA to block the CPU thread on a synchronization primitive when waiting for the GPU to finish work.

   Deprecated: This flag was deprecated as of CUDA 4.0 and was replaced with CU_CTX_SCHED_BLOCKING_SYNC.

▸ CU_CTX_MAP_HOST: Instruct CUDA to support mapped pinned allocations. This flag must be set in order to allocate pinned host memory that is accessible to the GPU.

▸ CU_CTX_LMEM_RESIZE_TO_MAX: 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.

Context creation will fail with CUDA_ERROR_UNKNOWN if the compute mode of the device is CU_COMPUTEMODE_PROHIBITED. Similarly, context creation will also fail with CUDA_ERROR_UNKNOWN if the compute mode for the device is set to CU_COMPUTEMODE_EXCLUSIVE and there is already an active context on the device. The function cuDeviceGetAttribute() can be used with CU_DEVICE_ATTRIBUTE_COMPUTE_MODE to determine the compute mode of the device. The nvidia-smi tool can be used to set the compute mode for devices. Documentation for nvidia-smi can be obtained by passing a -h option to it.

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

See also:

cuCtxDestroy, cuCtxGetApiVersion, cuCtxGetCacheConfig, cuCtxGetDevice, cuCtxGetLimit, cuCtxPopCurrent, cuCtxPushCurrent, cuCtxSetCacheConfig, cuCtxSetLimit, cuCtxSynchronize

CUresult cuCtxDestroy (CUcontext ctx)

Destroy a CUDA context.

Parameters

ctx
- Context to destroy

Returns

CUDA_SUCCESS, CUDA_ERROR_DEINITIALIZED, CUDA_ERROR_NOT_INITIALIZED, CUDA_ERROR_INVALID_CONTEXT, CUDA_ERROR_INVALID_VALUE

Description

Destroys the CUDA context specified by `ctx`. The context `ctx` will be destroyed regardless of how many threads it is current to. It is the responsibility of the calling function to ensure that no API call issues using `ctx` while cuCtxDestroy() is executing.

If `ctx` is current to the calling thread then `ctx` will also be popped from the current thread's context stack (as though cuCtxPopCurrent() were called). If `ctx` is current to other threads, then `ctx` will remain current to those threads, and attempting to access `ctx` from those threads will result in the error CUDA_ERROR_CONTEXT_IS_DESTROYED.

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

See also:

cuCtxCreate, cuCtxGetApiVersion, cuCtxGetCacheConfig, cuCtxGetDevice, cuCtxGetLimit, cuCtxPopCurrent, cuCtxPushCurrent, cuCtxSetCacheConfig, cuCtxSetLimit, cuCtxSynchronize

CUresult cuCtxGetApiVersion (CUcontext ctx, unsigned int *version)

Gets the context's API version.

Parameters

ctx
- Context to check

version
   - Pointer to version

Returns

CUDA_SUCCESS, CUDA_ERROR_DEINITIALIZED,
CUDA_ERROR_NOT_INITIALIZED, CUDA_ERROR_INVALID_CONTEXT,
CUDA_ERROR_UNKNOWN

Description

Returns a version number in `version` corresponding to the capabilities of the context (e.g. 3010 or 3020), which library developers can use to direct callers to a specific API version. If `ctx` is NULL, returns the API version used to create the currently bound context.

Note that new API versions are only introduced when context capabilities are changed that break binary compatibility, so the API version and driver version may be different. For example, it is valid for the API version to be 3020 while the driver version is 4020.

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

See also:

cuCtxCreate, cuCtxDestroy, cuCtxGetDevice, cuCtxGetLimit, cuCtxPopCurrent, cuCtxPushCurrent, cuCtxSetCacheConfig, cuCtxSetLimit, cuCtxSynchronize

CUresult cuCtxGetCacheConfig (CUfunc_cache *pconfig)

Returns the preferred cache configuration for the current context.

Parameters

pconfig
   - Returned cache configuration

Returns

CUDA_SUCCESS, CUDA_ERROR_DEINITIALIZED,
CUDA_ERROR_NOT_INITIALIZED, CUDA_ERROR_INVALID_CONTEXT,
CUDA_ERROR_INVALID_VALUE

Description

On devices where the L1 cache and shared memory use the same hardware resources, this function returns through `pconfig` the preferred cache configuration for the current

context. This is only a preference. The driver 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 `pconfig` of CU_FUNC_CACHE_PREFER_NONE on devices where the size of the L1 cache and shared memory are fixed.

The supported cache configurations are:

▸ CU_FUNC_CACHE_PREFER_NONE: no preference for shared memory or L1 (default)
▸ CU_FUNC_CACHE_PREFER_SHARED: prefer larger shared memory and smaller L1 cache
▸ CU_FUNC_CACHE_PREFER_L1: prefer larger L1 cache and smaller shared memory
▸ CU_FUNC_CACHE_PREFER_EQUAL: prefer equal sized L1 cache and shared memory

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

See also:

cuCtxCreate, cuCtxDestroy, cuCtxGetApiVersion, cuCtxGetDevice, cuCtxGetLimit, cuCtxPopCurrent, cuCtxPushCurrent, cuCtxSetCacheConfig, cuCtxSetLimit, cuCtxSynchronize, cuFuncSetCacheConfig

CUresult cuCtxGetCurrent (CUcontext *pctx)

Returns the CUDA context bound to the calling CPU thread.

Parameters

pctx
   - Returned context handle

Returns

CUDA_SUCCESS, CUDA_ERROR_DEINITIALIZED, CUDA_ERROR_NOT_INITIALIZED,

Description

Returns in `*pctx` the CUDA context bound to the calling CPU thread. If no context is bound to the calling CPU thread then `*pctx` is set to NULL and CUDA_SUCCESS is returned.

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

See also:

cuCtxSetCurrent, cuCtxCreate, cuCtxDestroy

CUresult cuCtxGetDevice (CUdevice *device)

Returns the device ID for the current context.

Parameters

device
   - Returned device ID for the current context

Returns

CUDA_SUCCESS, CUDA_ERROR_DEINITIALIZED,
CUDA_ERROR_NOT_INITIALIZED, CUDA_ERROR_INVALID_CONTEXT,
CUDA_ERROR_INVALID_VALUE,

Description

Returns in `*device` the ordinal of the current context's device.

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

See also:

cuCtxCreate, cuCtxDestroy, cuCtxGetApiVersion, cuCtxGetCacheConfig,
cuCtxGetLimit, cuCtxPopCurrent, cuCtxPushCurrent, cuCtxSetCacheConfig,
cuCtxSetLimit, cuCtxSynchronize

CUresult cuCtxGetLimit (size_t *pvalue, CUlimit limit)

Returns resource limits.

Parameters

pvalue
   - Returned size of limit
limit
   - Limit to query

Returns

CUDA_SUCCESS, CUDA_ERROR_INVALID_VALUE,
CUDA_ERROR_UNSUPPORTED_LIMIT

Description

Returns in `*pvalue` the current size of `limit`. The supported CUlimit values are:

▸ CU_LIMIT_STACK_SIZE: stack size in bytes of each GPU thread.
▸ CU_LIMIT_PRINTF_FIFO_SIZE: size in bytes of the FIFO used by the printf() device
  system call.
▸ CU_LIMIT_MALLOC_HEAP_SIZE: size in bytes of the heap used by the malloc()
  and free() device system calls.
▸ CU_LIMIT_DEV_RUNTIME_SYNC_DEPTH: maximum grid depth at which a
  thread can issue the device runtime call cudaDeviceSynchronize() to wait on child
  grid launches to complete.
▸ CU_LIMIT_DEV_RUNTIME_PENDING_LAUNCH_COUNT: maximum number of
  outstanding device runtime launches that can be made from this context.

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

See also:

cuCtxCreate, cuCtxDestroy, cuCtxGetApiVersion, cuCtxGetCacheConfig,
cuCtxGetDevice, cuCtxPopCurrent, cuCtxPushCurrent, cuCtxSetCacheConfig,
cuCtxSetLimit, cuCtxSynchronize

CUresult cuCtxGetSharedMemConfig (CUsharedconfig *pConfig)

Returns the current shared memory configuration for the current context.

Parameters

pConfig
   - returned shared memory configuration

Returns

CUDA_SUCCESS, CUDA_ERROR_DEINITIALIZED,
CUDA_ERROR_NOT_INITIALIZED, CUDA_ERROR_INVALID_CONTEXT,
CUDA_ERROR_INVALID_VALUE

Description

This function will return in `pConfig` the current size of shared memory banks in the current context. On devices with configurable shared memory banks, cuCtxSetSharedMemConfig can be used to change this setting, so that all subsequent kernel launches will by default use the new bank size. When cuCtxGetSharedMemConfig 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:

▸ CU_SHARED_MEM_CONFIG_FOUR_BYTE_BANK_SIZE: shared memory bank width is four bytes.
▸ CU_SHARED_MEM_CONFIG_EIGHT_BYTE_BANK_SIZE: shared memory bank width will eight bytes.

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

See also:

cuCtxCreate, cuCtxDestroy, cuCtxGetApiVersion, cuCtxGetCacheConfig, cuCtxGetDevice, cuCtxGetLimit, cuCtxPopCurrent, cuCtxPushCurrent, cuCtxSetLimit, cuCtxSynchronize, cuCtxGetSharedMemConfig, cuFuncSetCacheConfig,

CUresult cuCtxGetStreamPriorityRange (int *leastPriority, int *greatestPriority)

Returns numerical values that correspond to the least and greatest stream priorities.

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

CUDA_SUCCESS, CUDA_ERROR_INVALID_VALUE,

Description

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 meaningful range as specified by this API, the priority is automatically clamped down or up to either *leastPriority or *greatestPriority respectively. See cuStreamCreateWithPriority 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 cuDeviceGetAttribute).

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

See also:

cuStreamCreateWithPriority, cuStreamGetPriority, cuCtxGetDevice, cuCtxSetLimit, cuCtxSynchronize

CUresult cuCtxPopCurrent (CUcontext *pctx)

Pops the current CUDA context from the current CPU thread.

Parameters

pctx
    - Returned new context handle

Returns

CUDA_SUCCESS, CUDA_ERROR_DEINITIALIZED, CUDA_ERROR_NOT_INITIALIZED, CUDA_ERROR_INVALID_CONTEXT

Description

Pops the current CUDA context from the CPU thread and passes back the old context handle in *pctx. That context may then be made current to a different CPU thread by calling cuCtxPushCurrent().

If a context was current to the CPU thread before cuCtxCreate() or cuCtxPushCurrent() was called, this function makes that context current to the CPU thread again.

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

See also:

cuCtxCreate, cuCtxDestroy, cuCtxGetApiVersion, cuCtxGetCacheConfig, cuCtxGetDevice, cuCtxGetLimit, cuCtxPushCurrent, cuCtxSetCacheConfig, cuCtxSetLimit, cuCtxSynchronize

CUresult cuCtxPushCurrent (CUcontext ctx)

Pushes a context on the current CPU thread.

Parameters

ctx
    - Context to push

Returns

CUDA_SUCCESS, CUDA_ERROR_DEINITIALIZED, CUDA_ERROR_NOT_INITIALIZED, CUDA_ERROR_INVALID_CONTEXT, CUDA_ERROR_INVALID_VALUE

Description

Pushes the given context `ctx` onto the CPU thread's stack of current contexts. The specified context becomes the CPU thread's current context, so all CUDA functions that operate on the current context are affected.

The previous current context may be made current again by calling cuCtxDestroy() or cuCtxPopCurrent().

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

See also:

cuCtxCreate, cuCtxDestroy, cuCtxGetApiVersion, cuCtxGetCacheConfig, cuCtxGetDevice, cuCtxGetLimit, cuCtxPopCurrent, cuCtxSetCacheConfig, cuCtxSetLimit, cuCtxSynchronize

CUresult cuCtxSetCacheConfig (CUfunc_cache config)

Sets the preferred cache configuration for the current context.

Parameters

config
    - Requested cache configuration

Returns

CUDA_SUCCESS, CUDA_ERROR_DEINITIALIZED,
CUDA_ERROR_NOT_INITIALIZED, CUDA_ERROR_INVALID_CONTEXT,
CUDA_ERROR_INVALID_VALUE

Description

On devices where the L1 cache and shared memory use the same hardware resources, this sets through `config` the preferred cache configuration for the current context. This is only a preference. The driver 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 cuFuncSetCacheConfig() will be preferred over this context-wide setting. Setting the context-wide cache configuration to CU_FUNC_CACHE_PREFER_NONE 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:

▸ CU_FUNC_CACHE_PREFER_NONE: no preference for shared memory or L1 (default)
▸ CU_FUNC_CACHE_PREFER_SHARED: prefer larger shared memory and smaller L1 cache
▸ CU_FUNC_CACHE_PREFER_L1: prefer larger L1 cache and smaller shared memory
▸ CU_FUNC_CACHE_PREFER_EQUAL: prefer equal sized L1 cache and shared memory

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

See also:

cuCtxCreate, cuCtxDestroy, cuCtxGetApiVersion, cuCtxGetCacheConfig, cuCtxGetDevice, cuCtxGetLimit, cuCtxPopCurrent, cuCtxPushCurrent, cuCtxSetLimit, cuCtxSynchronize, cuFuncSetCacheConfig

CUresult cuCtxSetCurrent (CUcontext ctx)

Binds the specified CUDA context to the calling CPU thread.

Parameters

ctx
  - Context to bind to the calling CPU thread

Returns

CUDA_SUCCESS, CUDA_ERROR_DEINITIALIZED,
CUDA_ERROR_NOT_INITIALIZED, CUDA_ERROR_INVALID_CONTEXT

Description

Binds the specified CUDA context to the calling CPU thread. If `ctx` is NULL then the CUDA context previously bound to the calling CPU thread is unbound and CUDA_SUCCESS is returned.

If there exists a CUDA context stack on the calling CPU thread, this will replace the top of that stack with `ctx`. If `ctx` is NULL then this will be equivalent to popping the top of the calling CPU thread's CUDA context stack (or a no-op if the calling CPU thread's CUDA context stack is empty).

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

See also:

cuCtxGetCurrent, cuCtxCreate, cuCtxDestroy

CUresult cuCtxSetLimit (CUlimit limit, size_t value)

Set resource limits.

Parameters

limit
  - Limit to set
value
  - Size of limit

Returns

CUDA_SUCCESS, CUDA_ERROR_INVALID_VALUE,
CUDA_ERROR_UNSUPPORTED_LIMIT, CUDA_ERROR_OUT_OF_MEMORY

Description

Setting limit to value is a request by the application to update the current limit maintained by the context. 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 cuCtxGetLimit() to find out exactly what the limit has been set to.

Setting each CUlimit has its own specific restrictions, so each is discussed here.

▸ CU_LIMIT_STACK_SIZE controls the stack size in bytes of each GPU thread. This limit is only applicable to devices of compute capability 2.0 and higher. Attempting to set this limit on devices of compute capability less than 2.0 will result in the error CUDA_ERROR_UNSUPPORTED_LIMIT being returned.

▸ CU_LIMIT_PRINTF_FIFO_SIZE controls the size in bytes of the FIFO used by the printf() device system call. Setting CU_LIMIT_PRINTF_FIFO_SIZE must be performed before launching any kernel that uses the printf() device system call, otherwise CUDA_ERROR_INVALID_VALUE will be returned. This limit is only applicable to devices of compute capability 2.0 and higher. Attempting to set this limit on devices of compute capability less than 2.0 will result in the error CUDA_ERROR_UNSUPPORTED_LIMIT being returned.

▸ CU_LIMIT_MALLOC_HEAP_SIZE controls the size in bytes of the heap used by the malloc() and free() device system calls. Setting CU_LIMIT_MALLOC_HEAP_SIZE must be performed before launching any kernel that uses the malloc() or free() device system calls, otherwise CUDA_ERROR_INVALID_VALUE will be returned. This limit is only applicable to devices of compute capability 2.0 and higher. Attempting to set this limit on devices of compute capability less than 2.0 will result in the error CUDA_ERROR_UNSUPPORTED_LIMIT being returned.

▸ CU_LIMIT_DEV_RUNTIME_SYNC_DEPTH 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 driver to reserve large amounts of device memory which can no longer be used for user allocations. If these reservations of device memory fail, cuCtxSetLimit will return CUDA_ERROR_OUT_OF_MEMORY, 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 CUDA_ERROR_UNSUPPORTED_LIMIT being returned.

▶ CU_LIMIT_DEV_RUNTIME_PENDING_LAUNCH_COUNT controls the maximum number of outstanding device runtime launches that can be made from the current context. 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 driver to reserve larger amounts of device memory upfront which can no longer be used for allocations. If these reservations fail, cuCtxSetLimit will return CUDA_ERROR_OUT_OF_MEMORY, 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 CUDA_ERROR_UNSUPPORTED_LIMIT being returned.

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

See also:

cuCtxCreate, cuCtxDestroy, cuCtxGetApiVersion, cuCtxGetCacheConfig, cuCtxGetDevice, cuCtxGetLimit, cuCtxPopCurrent, cuCtxPushCurrent, cuCtxSetCacheConfig, cuCtxSynchronize

CUresult cuCtxSetSharedMemConfig (CUsharedconfig config)

Sets the shared memory configuration for the current context.

Parameters

config
 - requested shared memory configuration

Returns

CUDA_SUCCESS, CUDA_ERROR_DEINITIALIZED, CUDA_ERROR_NOT_INITIALIZED, CUDA_ERROR_INVALID_CONTEXT, CUDA_ERROR_INVALID_VALUE

Description

On devices with configurable shared memory banks, this function will set the context's shared memory bank size which is used for subsequent kernel launches.

Changed the shared memory configuration between launches may insert a device side synchronization point between those launches.

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:

▸ CU_SHARED_MEM_CONFIG_DEFAULT_BANK_SIZE: set bank width to the default initial setting (currently, four bytes).

▸ CU_SHARED_MEM_CONFIG_FOUR_BYTE_BANK_SIZE: set shared memory bank width to be natively four bytes.

▸ CU_SHARED_MEM_CONFIG_EIGHT_BYTE_BANK_SIZE: set shared memory bank width to be natively eight bytes.

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

See also:

cuCtxCreate, cuCtxDestroy, cuCtxGetApiVersion, cuCtxGetCacheConfig, cuCtxGetDevice, cuCtxGetLimit, cuCtxPopCurrent, cuCtxPushCurrent, cuCtxSetLimit, cuCtxSynchronize, cuCtxGetSharedMemConfig, cuFuncSetCacheConfig,

CUresult cuCtxSynchronize (void)

Block for a context's tasks to complete.

Returns

CUDA_SUCCESS, CUDA_ERROR_DEINITIALIZED, CUDA_ERROR_NOT_INITIALIZED, CUDA_ERROR_INVALID_CONTEXT

Description

Blocks until the device has completed all preceding requested tasks. cuCtxSynchronize() returns an error if one of the preceding tasks failed. If the context was created with the

CU_CTX_SCHED_BLOCKING_SYNC flag, the CPU thread will block until the GPU context has finished its work.

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

See also:

cuCtxCreate, cuCtxDestroy, cuCtxGetApiVersion, cuCtxGetCacheConfig, cuCtxGetDevice, cuCtxGetLimit, cuCtxPopCurrent, cuCtxPushCurrent, cuCtxSetCacheConfig, cuCtxSetLimit

3.8. Context Management [DEPRECATED]

This section describes the deprecated context management functions of the low-level CUDA driver application programming interface.

CUresult cuCtxAttach (CUcontext *pctx, unsigned int flags)

Increment a context's usage-count.

Parameters

pctx
   - Returned context handle of the current context
flags
   - Context attach flags (must be 0)

Returns

CUDA_SUCCESS, CUDA_ERROR_DEINITIALIZED, CUDA_ERROR_NOT_INITIALIZED, CUDA_ERROR_INVALID_CONTEXT, CUDA_ERROR_INVALID_VALUE

Description

Deprecated

Note that this function is deprecated and should not be used.

Increments the usage count of the context and passes back a context handle in `*pctx` that must be passed to cuCtxDetach() when the application is done with the context. cuCtxAttach() fails if there is no context current to the thread.

Currently, the `flags` parameter must be 0.

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

See also:

cuCtxCreate, cuCtxDestroy, cuCtxDetach, cuCtxGetApiVersion, cuCtxGetCacheConfig, cuCtxGetDevice, cuCtxGetLimit, cuCtxPopCurrent, cuCtxPushCurrent, cuCtxSetCacheConfig, cuCtxSetLimit, cuCtxSynchronize

CUresult cuCtxDetach (CUcontext ctx)

Decrement a context's usage-count.

Parameters

ctx
  - Context to destroy

Returns

CUDA_SUCCESS, CUDA_ERROR_DEINITIALIZED, CUDA_ERROR_NOT_INITIALIZED, CUDA_ERROR_INVALID_CONTEXT

Description

Deprecated

Note that this function is deprecated and should not be used.

Decrements the usage count of the context `ctx`, and destroys the context if the usage count goes to 0. The context must be a handle that was passed back by cuCtxCreate() or cuCtxAttach(), and must be current to the calling thread.

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

See also:

cuCtxCreate, cuCtxDestroy, cuCtxGetApiVersion, cuCtxGetCacheConfig, cuCtxGetDevice, cuCtxGetLimit, cuCtxPopCurrent, cuCtxPushCurrent, cuCtxSetCacheConfig, cuCtxSetLimit, cuCtxSynchronize

3.9. Module Management

This section describes the module management functions of the low-level CUDA driver application programming interface.

CUresult cuLinkAddData (CUlinkState state, CUjitInputType type, void *data, size_t size, const char *name, unsigned int numOptions, CUjit_option *options, void **optionValues)

Add an input to a pending linker invocation.

Parameters

state
   A pending linker action.
type
   The type of the input data.
data
   The input data. PTX must be NULL-terminated.
size
   The length of the input data.
name
   An optional name for this input in log messages.
numOptions
   Size of options.
options
   Options to be applied only for this input (overrides options from cuLinkCreate).
optionValues
   Array of option values, each cast to void *.

Returns

CUDA_SUCCESS, CUDA_ERROR_INVALID_HANDLE, CUDA_ERROR_INVALID_VALUE, CUDA_ERROR_INVALID_IMAGE, CUDA_ERROR_INVALID_PTX, CUDA_ERROR_OUT_OF_MEMORY, CUDA_ERROR_NO_BINARY_FOR_GPU

Description

Ownership of `data` is retained by the caller. No reference is retained to any inputs after this call returns.

This method accepts only compiler options, which are used if the data must be compiled from PTX, and does not accept any of CU_JIT_WALL_TIME, CU_JIT_INFO_LOG_BUFFER, CU_JIT_ERROR_LOG_BUFFER, CU_JIT_TARGET_FROM_CUCONTEXT, or CU_JIT_TARGET.

See also:

cuLinkCreate, cuLinkAddFile, cuLinkComplete, cuLinkDestroy

CUresult cuLinkAddFile (CUlinkState state, CUjitInputType type, const char *path, unsigned int numOptions, CUjit_option *options, void **optionValues)

Add a file input to a pending linker invocation.

Parameters

state
    A pending linker action
type
    The type of the input data
path
    Path to the input file
numOptions
    Size of options
options
    Options to be applied only for this input (overrides options from cuLinkCreate)
optionValues
    Array of option values, each cast to void *

Returns

CUDA_SUCCESS, CUDA_ERROR_FILE_NOT_FOUND CUDA_ERROR_INVALID_HANDLE, CUDA_ERROR_INVALID_VALUE, CUDA_ERROR_INVALID_IMAGE, CUDA_ERROR_INVALID_PTX, CUDA_ERROR_OUT_OF_MEMORY, CUDA_ERROR_NO_BINARY_FOR_GPU

Description

No reference is retained to any inputs after this call returns.

This method accepts only compiler options, which are used if the input must be compiled from PTX, and does not accept any of CU_JIT_WALL_TIME, CU_JIT_INFO_LOG_BUFFER, CU_JIT_ERROR_LOG_BUFFER, CU_JIT_TARGET_FROM_CUCONTEXT, or CU_JIT_TARGET.

This method is equivalent to invoking cuLinkAddData on the contents of the file.

See also:

cuLinkCreate, cuLinkAddData, cuLinkComplete, cuLinkDestroy

CUresult cuLinkComplete (CUlinkState state, void **cubinOut, size_t *sizeOut)

Complete a pending linker invocation.

Parameters

state
  A pending linker invocation
cubinOut
  On success, this will point to the output image
sizeOut
  Optional parameter to receive the size of the generated image

Returns

CUDA_SUCCESS, CUDA_ERROR_INVALID_HANDLE,
CUDA_ERROR_OUT_OF_MEMORY

Description

Completes the pending linker action and returns the cubin image for the linked device code, which can be used with cuModuleLoadData. The cubin is owned by state, so it should be loaded before state is destroyed via cuLinkDestroy. This call does not destroy state.

See also:

cuLinkCreate, cuLinkAddData, cuLinkAddFile, cuLinkDestroy, cuModuleLoadData

CUresult cuLinkCreate (unsigned int numOptions, CUjit_option *options, void **optionValues, CUlinkState *stateOut)

Creates a pending JIT linker invocation.

Parameters

numOptions
  Size of options arrays

options

Array of linker and compiler options

optionValues

Array of option values, each cast to void *

stateOut

On success, this will contain a CUlinkState to specify and complete this action

Returns

CUDA_SUCCESS, CUDA_ERROR_DEINITIALIZED,
CUDA_ERROR_NOT_INITIALIZED, CUDA_ERROR_INVALID_CONTEXT,
CUDA_ERROR_INVALID_VALUE, CUDA_ERROR_OUT_OF_MEMORY

Description

If the call is successful, the caller owns the returned CUlinkState, which should eventually be destroyed with cuLinkDestroy. The device code machine size (32 or 64 bit) will match the calling application.

Both linker and compiler options may be specified. Compiler options will be applied to inputs to this linker action which must be compiled from PTX. The options CU_JIT_WALL_TIME, CU_JIT_INFO_LOG_BUFFER_SIZE_BYTES, and CU_JIT_ERROR_LOG_BUFFER_SIZE_BYTES will accumulate data until the CUlinkState is destroyed.

optionValues must remain valid for the life of the CUlinkState if output options are used. No other references to inputs are maintained after this call returns.

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

See also:

cuLinkAddData, cuLinkAddFile, cuLinkComplete, cuLinkDestroy

CUresult cuLinkDestroy (CUlinkState state)

Destroys state for a JIT linker invocation.

Parameters

state

State object for the linker invocation

Returns

CUDA_SUCCESS, CUDA_ERROR_INVALID_HANDLE

Description

See also:

cuLinkCreate

CUresult cuModuleGetFunction (CUfunction *hfunc, CUmodule hmod, const char *name)

Returns a function handle.

Parameters

hfunc
   - Returned function handle
hmod
   - Module to retrieve function from
name
   - Name of function to retrieve

Returns

CUDA_SUCCESS, CUDA_ERROR_DEINITIALIZED,
CUDA_ERROR_NOT_INITIALIZED, CUDA_ERROR_INVALID_CONTEXT,
CUDA_ERROR_INVALID_VALUE, CUDA_ERROR_NOT_FOUND

Description

Returns in `*hfunc` the handle of the function of name `name` located in module `hmod`. If no function of that name exists, cuModuleGetFunction() returns CUDA_ERROR_NOT_FOUND.

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

See also:

cuModuleGetGlobal, cuModuleGetTexRef, cuModuleLoad, cuModuleLoadData, cuModuleLoadDataEx, cuModuleLoadFatBinary, cuModuleUnload

CUresult cuModuleGetGlobal (CUdeviceptr *dptr, size_t *bytes, CUmodule hmod, const char *name)

Returns a global pointer from a module.

Parameters

dptr
   - Returned global device pointer
bytes
   - Returned global size in bytes
hmod
   - Module to retrieve global from
name
   - Name of global to retrieve

Returns

CUDA_SUCCESS, CUDA_ERROR_DEINITIALIZED,
CUDA_ERROR_NOT_INITIALIZED, CUDA_ERROR_INVALID_CONTEXT,
CUDA_ERROR_INVALID_VALUE, CUDA_ERROR_NOT_FOUND

Description

Returns in `*dptr` and `*bytes` the base pointer and size of the global of name `name` located in module `hmod`. If no variable of that name exists, cuModuleGetGlobal() returns CUDA_ERROR_NOT_FOUND. Both parameters `dptr` and `bytes` are optional. If one of them is NULL, it is ignored.

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

See also:

cuModuleGetFunction, cuModuleGetTexRef, cuModuleLoad, cuModuleLoadData, cuModuleLoadDataEx, cuModuleLoadFatBinary, cuModuleUnload

CUresult cuModuleGetSurfRef (CUsurfref *pSurfRef, CUmodule hmod, const char *name)

Returns a handle to a surface reference.

Parameters

pSurfRef
- Returned surface reference

hmod
- Module to retrieve surface reference from

name
- Name of surface reference to retrieve

Returns

CUDA_SUCCESS, CUDA_ERROR_DEINITIALIZED,
CUDA_ERROR_NOT_INITIALIZED, CUDA_ERROR_INVALID_CONTEXT,
CUDA_ERROR_INVALID_VALUE, CUDA_ERROR_NOT_FOUND

Description

Returns in `*pSurfRef` the handle of the surface reference of name `name` in the module `hmod`. If no surface reference of that name exists, cuModuleGetSurfRef() returns CUDA_ERROR_NOT_FOUND.

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

See also:

cuModuleGetFunction, cuModuleGetGlobal, cuModuleGetTexRef, cuModuleLoad, cuModuleLoadData, cuModuleLoadDataEx, cuModuleLoadFatBinary, cuModuleUnload

CUresult cuModuleGetTexRef (CUtexref *pTexRef, CUmodule hmod, const char *name)

Returns a handle to a texture reference.

Parameters

pTexRef
- Returned texture reference

hmod
- Module to retrieve texture reference from

name
- Name of texture reference to retrieve

Returns

CUDA_SUCCESS, CUDA_ERROR_DEINITIALIZED,
CUDA_ERROR_NOT_INITIALIZED, CUDA_ERROR_INVALID_CONTEXT,
CUDA_ERROR_INVALID_VALUE, CUDA_ERROR_NOT_FOUND

Description

Returns in `*pTexRef` the handle of the texture reference of name `name` in the module `hmod`. If no texture reference of that name exists, cuModuleGetTexRef() returns CUDA_ERROR_NOT_FOUND. This texture reference handle should not be destroyed, since it will be destroyed when the module is unloaded.

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

See also:

cuModuleGetFunction, cuModuleGetGlobal, cuModuleGetSurfRef, cuModuleLoad, cuModuleLoadData, cuModuleLoadDataEx, cuModuleLoadFatBinary, cuModuleUnload

CUresult cuModuleLoad (CUmodule *module, const char *fname)

Loads a compute module.

Parameters

module
- Returned module

fname
- Filename of module to load

Returns

CUDA_SUCCESS, CUDA_ERROR_DEINITIALIZED,
CUDA_ERROR_NOT_INITIALIZED, CUDA_ERROR_INVALID_CONTEXT,
CUDA_ERROR_INVALID_VALUE, CUDA_ERROR_INVALID_PTX,
CUDA_ERROR_NOT_FOUND, CUDA_ERROR_OUT_OF_MEMORY,

CUDA_ERROR_FILE_NOT_FOUND, CUDA_ERROR_NO_BINARY_FOR_GPU,
CUDA_ERROR_SHARED_OBJECT_SYMBOL_NOT_FOUND,
CUDA_ERROR_SHARED_OBJECT_INIT_FAILED

Description

Takes a filename `fname` and loads the corresponding module `module` into the current context. The CUDA driver API does not attempt to lazily allocate the resources needed by a module; if the memory for functions and data (constant and global) needed by the module cannot be allocated, cuModuleLoad() fails. The file should be a cubin file as output by **nvcc**, or a PTX file either as output by **nvcc** or handwritten, or a fatbin file as output by **nvcc** from toolchain 4.0 or later.

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

See also:

cuModuleGetFunction, cuModuleGetGlobal, cuModuleGetTexRef, cuModuleLoadData, cuModuleLoadDataEx, cuModuleLoadFatBinary, cuModuleUnload

CUresult cuModuleLoadData (CUmodule *module, const void *image)

Load a module's data.

Parameters

module
    - Returned module
image
    - Module data to load

Returns

CUDA_SUCCESS, CUDA_ERROR_DEINITIALIZED,
CUDA_ERROR_NOT_INITIALIZED, CUDA_ERROR_INVALID_CONTEXT,
CUDA_ERROR_INVALID_VALUE, CUDA_ERROR_INVALID_PTX,
CUDA_ERROR_OUT_OF_MEMORY, CUDA_ERROR_NO_BINARY_FOR_GPU,
CUDA_ERROR_SHARED_OBJECT_SYMBOL_NOT_FOUND,
CUDA_ERROR_SHARED_OBJECT_INIT_FAILED

Description

Takes a pointer `image` and loads the corresponding module `module` into the current context. The pointer may be obtained by mapping a cubin or PTX or fatbin file, passing a cubin or PTX or fatbin file as a NULL-terminated text string, or incorporating a cubin or fatbin object into the executable resources and using operating system calls such as Windows `FindResource()` to obtain the pointer.

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

See also:

cuModuleGetFunction, cuModuleGetGlobal, cuModuleGetTexRef, cuModuleLoad, cuModuleLoadDataEx, cuModuleLoadFatBinary, cuModuleUnload

CUresult cuModuleLoadDataEx (CUmodule *module, const void *image, unsigned int numOptions, CUjit_option *options, void **optionValues)

Load a module's data with options.

Parameters

module
  - Returned module
image
  - Module data to load
numOptions
  - Number of options
options
  - Options for JIT
optionValues
  - Option values for JIT

Returns

CUDA_SUCCESS, CUDA_ERROR_DEINITIALIZED, CUDA_ERROR_NOT_INITIALIZED, CUDA_ERROR_INVALID_CONTEXT, CUDA_ERROR_INVALID_VALUE, CUDA_ERROR_INVALID_PTX, CUDA_ERROR_OUT_OF_MEMORY, CUDA_ERROR_NO_BINARY_FOR_GPU, CUDA_ERROR_SHARED_OBJECT_SYMBOL_NOT_FOUND, CUDA_ERROR_SHARED_OBJECT_INIT_FAILED

Description

Takes a pointer `image` and loads the corresponding module `module` into the current context. The pointer may be obtained by mapping a cubin or PTX or fatbin file, passing a cubin or PTX or fatbin file as a NULL-terminated text string, or incorporating a cubin or fatbin object into the executable resources and using operating system calls such as Windows `FindResource()` to obtain the pointer. Options are passed as an array via `options` and any corresponding parameters are passed in `optionValues`. The number of total options is supplied via `numOptions`. Any outputs will be returned via `optionValues`.

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

See also:

cuModuleGetFunction, cuModuleGetGlobal, cuModuleGetTexRef, cuModuleLoad, cuModuleLoadData, cuModuleLoadFatBinary, cuModuleUnload

CUresult cuModuleLoadFatBinary (CUmodule *module, const void *fatCubin)

Load a module's data.

Parameters

module
   - Returned module
fatCubin
   - Fat binary to load

Returns

CUDA_SUCCESS, CUDA_ERROR_DEINITIALIZED, CUDA_ERROR_NOT_INITIALIZED, CUDA_ERROR_INVALID_CONTEXT, CUDA_ERROR_INVALID_VALUE, CUDA_ERROR_INVALID_PTX, CUDA_ERROR_NOT_FOUND, CUDA_ERROR_OUT_OF_MEMORY, CUDA_ERROR_NO_BINARY_FOR_GPU, CUDA_ERROR_SHARED_OBJECT_SYMBOL_NOT_FOUND, CUDA_ERROR_SHARED_OBJECT_INIT_FAILED

Description

Takes a pointer `fatCubin` and loads the corresponding module `module` into the current context. The pointer represents a fat binary object, which is a collection of different cubin

and/or PTX files, all representing the same device code, but compiled and optimized for different architectures.

Prior to CUDA 4.0, there was no documented API for constructing and using fat binary objects by programmers. Starting with CUDA 4.0, fat binary objects can be constructed by providing the -fatbin option to **nvcc**. More information can be found in the **nvcc** document.

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

See also:

cuModuleGetFunction, cuModuleGetGlobal, cuModuleGetTexRef, cuModuleLoad, cuModuleLoadData, cuModuleLoadDataEx, cuModuleUnload

CUresult cuModuleUnload (CUmodule hmod)

Unloads a module.

Parameters

hmod
   - Module to unload

Returns

CUDA_SUCCESS, CUDA_ERROR_DEINITIALIZED, CUDA_ERROR_NOT_INITIALIZED, CUDA_ERROR_INVALID_CONTEXT, CUDA_ERROR_INVALID_VALUE

Description

Unloads a module `hmod` from the current context.

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

See also:

cuModuleGetFunction, cuModuleGetGlobal, cuModuleGetTexRef, cuModuleLoad, cuModuleLoadData, cuModuleLoadDataEx, cuModuleLoadFatBinary

3.10. Memory Management

This section describes the memory management functions of the low-level CUDA driver application programming interface.

CUresult cuArray3DCreate (CUarray *pHandle, const CUDA_ARRAY3D_DESCRIPTOR *pAllocateArray)

Creates a 3D CUDA array.

Parameters

pHandle
   - Returned array
pAllocateArray
   - 3D array descriptor

Returns

CUDA_SUCCESS, CUDA_ERROR_DEINITIALIZED, CUDA_ERROR_NOT_INITIALIZED, CUDA_ERROR_INVALID_CONTEXT, CUDA_ERROR_INVALID_VALUE, CUDA_ERROR_OUT_OF_MEMORY, CUDA_ERROR_UNKNOWN

Description

Creates a CUDA array according to the CUDA_ARRAY3D_DESCRIPTOR structure pAllocateArray and returns a handle to the new CUDA array in *pHandle. The CUDA_ARRAY3D_DESCRIPTOR is defined as:

```
typedef struct {
        unsigned int Width;
        unsigned int Height;
        unsigned int Depth;
        CUarray_format Format;
        unsigned int NumChannels;
        unsigned int Flags;
    } CUDA_ARRAY3D_DESCRIPTOR;
```

where:

▸ Width, Height, and Depth are the width, height, and depth of the CUDA array (in elements); the following types of CUDA arrays can be allocated:

   ▸ A 1D array is allocated if Height and Depth extents are both zero.

   ▸ A 2D array is allocated if only 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 `Height` is zero and the CUDA_ARRAY3D_LAYERED 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 CUDA_ARRAY3D_LAYERED 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 CUDA_ARRAY3D_CUBEMAP 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 CUarray_cubemap_face.

▸ A cubemap layered CUDA array is allocated if all three extents are non-zero, and both, CUDA_ARRAY3D_CUBEMAP and CUDA_ARRAY3D_LAYERED 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.

▸ Format specifies the format of the elements; CUarray_format is defined as:

```
typedef enum CUarray_format_enum {
        CU_AD_FORMAT_UNSIGNED_INT8 = 0x01,
        CU_AD_FORMAT_UNSIGNED_INT16 = 0x02,
        CU_AD_FORMAT_UNSIGNED_INT32 = 0x03,
        CU_AD_FORMAT_SIGNED_INT8 = 0x08,
        CU_AD_FORMAT_SIGNED_INT16 = 0x09,
        CU_AD_FORMAT_SIGNED_INT32 = 0x0a,
        CU_AD_FORMAT_HALF = 0x10,
        CU_AD_FORMAT_FLOAT = 0x20
    } CUarray_format;
```

▸ `NumChannels` specifies the number of packed components per CUDA array element; it may be 1, 2, or 4;

▸ Flags may be set to

   ▸ CUDA_ARRAY3D_LAYERED to enable creation of layered CUDA arrays. If this flag is set, `Depth` specifies the number of layers, not the depth of a 3D array.

   ▸ CUDA_ARRAY3D_SURFACE_LDST to enable surface references to be bound to the CUDA array. If this flag is not set, cuSurfRefSetArray will fail when attempting to bind the CUDA array to a surface reference.

   ▸ CUDA_ARRAY3D_CUBEMAP to enable creation of cubemaps. If this flag is set, `Width` must be equal to `Height`, and `Depth` must be six. If the CUDA_ARRAY3D_LAYERED flag is also set, then `Depth` must be a multiple of six.

   ▸ CUDA_ARRAY3D_TEXTURE_GATHER to indicate that the CUDA array will be used for texture gather. Texture gather can only be performed on 2D CUDA arrays.

`Width`, `Height` and `Depth` must meet certain size requirements as listed in the following table. All values are specified in elements. Note that for brevity's sake, the full name of the device attribute is not specified. For ex., TEXTURE1D_WIDTH refers to the device attribute CU_DEVICE_ATTRIBUTE_MAXIMUM_TEXTURE1D_WIDTH.

Note that 2D CUDA arrays have different size requirements if the CUDA_ARRAY3D_TEXTURE_GATHER flag is set. `Width` and `Height` must not be greater than CU_DEVICE_ATTRIBUTE_MAXIMUM_TEXTURE2D_GATHER_WIDTH and CU_DEVICE_ATTRIBUTE_MAXIMUM_TEXTURE2D_GATHER_HEIGHT respectively, in that case.

CUDA array type	Valid extents that must always be met {(width range in elements), (height range), (depth range)}	Valid extents with CUDA_ARRAY3D_SURFACE_LDST set {(width range in elements), (height range), (depth range)}
1D	{ (1,TEXTURE1D_WIDTH), 0, 0 }	{ (1,SURFACE1D_WIDTH), 0, 0 }
2D	{ (1,TEXTURE2D_WIDTH), (1,TEXTURE2D_HEIGHT), 0 }	{ (1,SURFACE2D_WIDTH), (1,SURFACE2D_HEIGHT), 0 }
3D	{ (1,TEXTURE3D_WIDTH), (1,TEXTURE3D_HEIGHT), (1,TEXTURE3D_DEPTH) } OR { (1,TEXTURE3D_WIDTH_ALTERNATE), (1,TEXTURE3D_HEIGHT_ALTERNATE), (1,TEXTURE3D_DEPTH_ALTERNATE) }	{ (1,SURFACE3D_WIDTH), (1,SURFACE3D_HEIGHT), (1,SURFACE3D_DEPTH) }
1D Layered	{ (1,TEXTURE1D_LAYERED_WIDTH), 0, (1,TEXTURE1D_LAYERED_LAYERS) }	{ (1,SURFACE1D_LAYERED_WIDTH), 0, (1,SURFACE1D_LAYERED_LAYERS) }
2D Layered	{ (1,TEXTURE2D_LAYERED_WIDTH), (1,TEXTURE2D_LAYERED_HEIGHT), (1,TEXTURE2D_LAYERED_LAYERS) }	{ (1,SURFACE2D_LAYERED_WIDTH), (1,SURFACE2D_LAYERED_HEIGHT), (1,SURFACE2D_LAYERED_LAYERS) }
Cubemap	{ (1,TEXTURECUBEMAP_WIDTH), (1,TEXTURECUBEMAP_WIDTH), 6 }	{ (1,SURFACECUBEMAP_WIDTH), (1,SURFACECUBEMAP_WIDTH), 6 }
Cubemap Layered	{ (1,TEXTURECUBEMAP_LAYERED_WIDTH), (1,TEXTURECUBEMAP_LAYERED_WIDTH), (1,TEXTURECUBEMAP_LAYERED_LAYERS) }	{ (1,SURFACECUBEMAP_LAYERED_WIDTH), (1,SURFACECUBEMAP_LAYERED_WIDTH), (1,SURFACECUBEMAP_LAYERED_LAYERS) }

Here are examples of CUDA array descriptions:

Description for a CUDA array of 2048 floats:

```
CUDA_ARRAY3D_DESCRIPTOR desc;
    desc.Format = CU_AD_FORMAT_FLOAT;
    desc.NumChannels = 1;
    desc.Width = 2048;
    desc.Height = 0;
    desc.Depth = 0;
```

Description for a 64 x 64 CUDA array of floats:

```
CUDA_ARRAY3D_DESCRIPTOR desc;
    desc.Format = CU_AD_FORMAT_FLOAT;
    desc.NumChannels = 1;
    desc.Width = 64;
    desc.Height = 64;
    desc.Depth = 0;
```

Description for a `width x height x depth` CUDA array of 64-bit, 4x16-bit float16's:

```
CUDA_ARRAY3D_DESCRIPTOR desc;
    desc.FormatFlags = CU_AD_FORMAT_HALF;
    desc.NumChannels = 4;
    desc.Width = width;
    desc.Height = height;
    desc.Depth = depth;
```

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

See also:

cuArray3DGetDescriptor, cuArrayCreate, cuArrayDestroy, cuArrayGetDescriptor, cuMemAlloc, cuMemAllocHost, cuMemAllocPitch, cuMemcpy2D, cuMemcpy2DAsync, cuMemcpy2DUnaligned, cuMemcpy3D, cuMemcpy3DAsync, cuMemcpyAtoA, cuMemcpyAtoD, cuMemcpyAtoH, cuMemcpyAtoHAsync, cuMemcpyDtoA, cuMemcpyDtoD, cuMemcpyDtoDAsync, cuMemcpyDtoH, cuMemcpyDtoHAsync, cuMemcpyHtoA, cuMemcpyHtoAAsync, cuMemcpyHtoD, cuMemcpyHtoDAsync, cuMemFree, cuMemFreeHost, cuMemGetAddressRange, cuMemGetInfo, cuMemHostAlloc, cuMemHostGetDevicePointer, cuMemsetD2D8, cuMemsetD2D16, cuMemsetD2D32, cuMemsetD8, cuMemsetD16, cuMemsetD32

CUresult cuArray3DGetDescriptor (CUDA_ARRAY3D_DESCRIPTOR *pArrayDescriptor, CUarray hArray)

Get a 3D CUDA array descriptor.

Parameters

pArrayDescriptor
   - Returned 3D array descriptor

hArray
- 3D array to get descriptor of

Returns

CUDA_SUCCESS, CUDA_ERROR_DEINITIALIZED,
CUDA_ERROR_NOT_INITIALIZED, CUDA_ERROR_INVALID_CONTEXT,
CUDA_ERROR_INVALID_VALUE, CUDA_ERROR_INVALID_HANDLE

Description

Returns in `*pArrayDescriptor` a descriptor containing information on the format and dimensions of the CUDA array `hArray`. It is useful for subroutines that have been passed a CUDA array, but need to know the CUDA array parameters for validation or other purposes.

This function may be called on 1D and 2D arrays, in which case the `Height` and/or `Depth` members of the descriptor struct will be set to 0.

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

See also:

cuArray3DCreate, cuArrayCreate, cuArrayDestroy, cuArrayGetDescriptor, cuMemAlloc, cuMemAllocHost, cuMemAllocPitch, cuMemcpy2D, cuMemcpy2DAsync, cuMemcpy2DUnaligned, cuMemcpy3D, cuMemcpy3DAsync, cuMemcpyAtoA, cuMemcpyAtoD, cuMemcpyAtoH, cuMemcpyAtoHAsync, cuMemcpyDtoA, cuMemcpyDtoD, cuMemcpyDtoDAsync, cuMemcpyDtoH, cuMemcpyDtoHAsync, cuMemcpyHtoA, cuMemcpyHtoAAsync, cuMemcpyHtoD, cuMemcpyHtoDAsync, cuMemFree, cuMemFreeHost, cuMemGetAddressRange, cuMemGetInfo, cuMemHostAlloc, cuMemHostGetDevicePointer, cuMemsetD2D8, cuMemsetD2D16, cuMemsetD2D32, cuMemsetD8, cuMemsetD16, cuMemsetD32

CUresult cuArrayCreate (CUarray *pHandle, const CUDA_ARRAY_DESCRIPTOR *pAllocateArray)

Creates a 1D or 2D CUDA array.

Parameters

pHandle
- Returned array
pAllocateArray
- Array descriptor

Returns

CUDA_SUCCESS, CUDA_ERROR_DEINITIALIZED,
CUDA_ERROR_NOT_INITIALIZED, CUDA_ERROR_INVALID_CONTEXT,
CUDA_ERROR_INVALID_VALUE, CUDA_ERROR_OUT_OF_MEMORY,
CUDA_ERROR_UNKNOWN

Description

Creates a CUDA array according to the CUDA_ARRAY_DESCRIPTOR structure
pAllocateArray and returns a handle to the new CUDA array in *pHandle. The
CUDA_ARRAY_DESCRIPTOR is defined as:

```
typedef struct {
        unsigned int Width;
        unsigned int Height;
        CUarray_format Format;
        unsigned int NumChannels;
    } CUDA_ARRAY_DESCRIPTOR;
```

where:

▸ Width, and Height are the width, and height of the CUDA array (in elements); the
  CUDA array is one-dimensional if height is 0, two-dimensional otherwise;

▸ Format specifies the format of the elements; CUarray_format is defined as:

```
typedef enum CUarray_format_enum {
        CU_AD_FORMAT_UNSIGNED_INT8 = 0x01,
        CU_AD_FORMAT_UNSIGNED_INT16 = 0x02,
        CU_AD_FORMAT_UNSIGNED_INT32 = 0x03,
        CU_AD_FORMAT_SIGNED_INT8 = 0x08,
        CU_AD_FORMAT_SIGNED_INT16 = 0x09,
        CU_AD_FORMAT_SIGNED_INT32 = 0x0a,
        CU_AD_FORMAT_HALF = 0x10,
        CU_AD_FORMAT_FLOAT = 0x20
    } CUarray_format;
```

▸ NumChannels specifies the number of packed components per CUDA array
  element; it may be 1, 2, or 4;

Here are examples of CUDA array descriptions:

Description for a CUDA array of 2048 floats:

```
CUDA_ARRAY_DESCRIPTOR desc;
    desc.Format = CU_AD_FORMAT_FLOAT;
    desc.NumChannels = 1;
    desc.Width = 2048;
    desc.Height = 1;
```

Description for a 64 x 64 CUDA array of floats:

```
CUDA_ARRAY_DESCRIPTOR desc;
    desc.Format = CU_AD_FORMAT_FLOAT;
    desc.NumChannels = 1;
    desc.Width = 64;
    desc.Height = 64;
```

Description for a `width x height` CUDA array of 64-bit, 4x16-bit float16's:

```
    CUDA_ARRAY_DESCRIPTOR desc;
        desc.FormatFlags = CU_AD_FORMAT_HALF;
        desc.NumChannels = 4;
        desc.Width = width;
        desc.Height = height;
```

Description for a `width x height` CUDA array of 16-bit elements, each of which is two 8-bit unsigned chars:

```
    CUDA_ARRAY_DESCRIPTOR arrayDesc;
        desc.FormatFlags = CU_AD_FORMAT_UNSIGNED_INT8;
        desc.NumChannels = 2;
        desc.Width = width;
        desc.Height = height;
```

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

See also:

cuArray3DCreate, cuArray3DGetDescriptor, cuArrayDestroy, cuArrayGetDescriptor, cuMemAlloc, cuMemAllocHost, cuMemAllocPitch, cuMemcpy2D, cuMemcpy2DAsync, cuMemcpy2DUnaligned, cuMemcpy3D, cuMemcpy3DAsync, cuMemcpyAtoA, cuMemcpyAtoD, cuMemcpyAtoH, cuMemcpyAtoHAsync, cuMemcpyDtoA, cuMemcpyDtoD, cuMemcpyDtoDAsync, cuMemcpyDtoH, cuMemcpyDtoHAsync, cuMemcpyHtoA, cuMemcpyHtoAAsync, cuMemcpyHtoD, cuMemcpyHtoDAsync, cuMemFree, cuMemFreeHost, cuMemGetAddressRange, cuMemGetInfo, cuMemHostAlloc, cuMemHostGetDevicePointer, cuMemsetD2D8, cuMemsetD2D16, cuMemsetD2D32, cuMemsetD8, cuMemsetD16, cuMemsetD32

CUresult cuArrayDestroy (CUarray hArray)

Destroys a CUDA array.

Parameters

hArray
   - Array to destroy

Returns

CUDA_SUCCESS, CUDA_ERROR_DEINITIALIZED, CUDA_ERROR_NOT_INITIALIZED, CUDA_ERROR_INVALID_CONTEXT, CUDA_ERROR_INVALID_HANDLE, CUDA_ERROR_ARRAY_IS_MAPPED

Description

Destroys the CUDA array `hArray`.

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

See also:

cuArray3DCreate, cuArray3DGetDescriptor, cuArrayCreate, cuArrayGetDescriptor, cuMemAlloc, cuMemAllocHost, cuMemAllocPitch, cuMemcpy2D, cuMemcpy2DAsync, cuMemcpy2DUnaligned, cuMemcpy3D, cuMemcpy3DAsync, cuMemcpyAtoA, cuMemcpyAtoD, cuMemcpyAtoH, cuMemcpyAtoHAsync, cuMemcpyDtoA, cuMemcpyDtoD, cuMemcpyDtoDAsync, cuMemcpyDtoH, cuMemcpyDtoHAsync, cuMemcpyHtoA, cuMemcpyHtoAAsync, cuMemcpyHtoD, cuMemcpyHtoDAsync, cuMemFree, cuMemFreeHost, cuMemGetAddressRange, cuMemGetInfo, cuMemHostAlloc, cuMemHostGetDevicePointer, cuMemsetD2D8, cuMemsetD2D16, cuMemsetD2D32, cuMemsetD8, cuMemsetD16, cuMemsetD32

CUresult cuArrayGetDescriptor (CUDA_ARRAY_DESCRIPTOR *pArrayDescriptor, CUarray hArray)

Get a 1D or 2D CUDA array descriptor.

Parameters

pArrayDescriptor
   - Returned array descriptor
hArray
   - Array to get descriptor of

Returns

CUDA_SUCCESS, CUDA_ERROR_DEINITIALIZED, CUDA_ERROR_NOT_INITIALIZED, CUDA_ERROR_INVALID_CONTEXT, CUDA_ERROR_INVALID_VALUE, CUDA_ERROR_INVALID_HANDLE

Description

Returns in `*pArrayDescriptor` a descriptor containing information on the format and dimensions of the CUDA array `hArray`. It is useful for subroutines that have been passed a CUDA array, but need to know the CUDA array parameters for validation or other purposes.

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

See also:

cuArray3DCreate, cuArray3DGetDescriptor, cuArrayCreate, cuArrayDestroy, cuMemAlloc, cuMemAllocHost, cuMemAllocPitch, cuMemcpy2D, cuMemcpy2DAsync, cuMemcpy2DUnaligned, cuMemcpy3D, cuMemcpy3DAsync, cuMemcpyAtoA, cuMemcpyAtoD, cuMemcpyAtoH, cuMemcpyAtoHAsync, cuMemcpyDtoA, cuMemcpyDtoD, cuMemcpyDtoDAsync, cuMemcpyDtoH, cuMemcpyDtoHAsync, cuMemcpyHtoA, cuMemcpyHtoAAsync, cuMemcpyHtoD, cuMemcpyHtoDAsync, cuMemFree, cuMemFreeHost, cuMemGetAddressRange, cuMemGetInfo, cuMemHostAlloc, cuMemHostGetDevicePointer, cuMemsetD2D8, cuMemsetD2D16, cuMemsetD2D32, cuMemsetD8, cuMemsetD16, cuMemsetD32

CUresult cuDeviceGetByPCIBusId (CUdevice *dev, const char *pciBusId)

Returns a handle to a compute device.

Parameters

dev
  - Returned device handle
pciBusId
  - String in one of the following forms: [domain]:[bus]:[device].[function] [domain]:[bus]:[device] [bus]:[device].[function] where `domain`, `bus`, `device`, and `function` are all hexadecimal values

Returns

CUDA_SUCCESS, CUDA_ERROR_DEINITIALIZED, CUDA_ERROR_NOT_INITIALIZED, CUDA_ERROR_INVALID_VALUE, CUDA_ERROR_INVALID_DEVICE

Description

Returns in `*device` a device handle given a PCI bus ID string.

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

See also:

cuDeviceGet, cuDeviceGetAttribute, cuDeviceGetPCIBusId

CUresult cuDeviceGetPCIBusId (char *pciBusId, int len, CUdevice dev)

Returns a PCI Bus Id string for the device.

Parameters

pciBusId
  - Returned identifier string for the device in the following format [domain]:[bus]:
  [device].[function] where `domain`, `bus`, `device`, and `function` are all hexadecimal
  values. pciBusId should be large enough to store 13 characters including the NULL-
  terminator.
len
  - Maximum length of string to store in `name`
dev
  - Device to get identifier string for

Returns

CUDA_SUCCESS, CUDA_ERROR_DEINITIALIZED,
CUDA_ERROR_NOT_INITIALIZED, CUDA_ERROR_INVALID_VALUE,
CUDA_ERROR_INVALID_DEVICE

Description

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.

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

See also:

cuDeviceGet, cuDeviceGetAttribute, cuDeviceGetByPCIBusId

CUresult cuIpcCloseMemHandle (CUdeviceptr dptr)

Close memory mapped with cuIpcOpenMemHandle.

Parameters

dptr
   - Device pointer returned by cuIpcOpenMemHandle

Returns

CUDA_SUCCESS, CUDA_ERROR_INVALID_CONTEXT,
CUDA_ERROR_MAP_FAILED, CUDA_ERROR_INVALID_HANDLE,

Description

Unmaps memory returnd by cuIpcOpenMemHandle. 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 operating systems.

See also:

cuMemAlloc, cuMemFree, cuIpcGetEventHandle, cuIpcOpenEventHandle, cuIpcGetMemHandle, cuIpcOpenMemHandle,

CUresult cuIpcGetEventHandle (CUipcEventHandle *pHandle, CUevent event)

Gets an interprocess handle for a previously allocated event.

Parameters

pHandle
   - Pointer to a user allocated CUipcEventHandle in which to return the opaque event handle
event
   - Event allocated with CU_EVENT_INTERPROCESS and CU_EVENT_DISABLE_TIMING flags.

Returns

CUDA_SUCCESS, CUDA_ERROR_INVALID_HANDLE,
CUDA_ERROR_OUT_OF_MEMORY, CUDA_ERROR_MAP_FAILED

Description

Takes as input a previously allocated event. This event must have been created with the CU_EVENT_INTERPROCESS and CU_EVENT_DISABLE_TIMING flags set. This opaque handle may be copied into other processes and opened with cuIpcOpenEventHandle to allow efficient hardware synchronization between GPU work in different processes.

After the event has been opened in the importing process, cuEventRecord, cuEventSynchronize, cuStreamWaitEvent and cuEventQuery may be used in either process. Performing operations on the imported event after the exported event has been freed with cuEventDestroy will result in undefined behavior.

IPC functionality is restricted to devices with support for unified addressing on Linux operating systems.

See also:

cuEventCreate, cuEventDestroy, cuEventSynchronize, cuEventQuery, cuStreamWaitEvent, cuIpcOpenEventHandle, cuIpcGetMemHandle, cuIpcOpenMemHandle, cuIpcCloseMemHandle

CUresult cuIpcGetMemHandle (CUipcMemHandle *pHandle, CUdeviceptr dptr)

Gets an interprocess memory handle for an existing device memory allocation.

Parameters

pHandle
   - Pointer to user allocated CUipcMemHandle to return the handle in.
dptr
   - Base pointer to previously allocated device memory

Returns

CUDA_SUCCESS, CUDA_ERROR_INVALID_HANDLE, CUDA_ERROR_OUT_OF_MEMORY, CUDA_ERROR_MAP_FAILED,

Description

Takes a pointer to the base of an existing device memory allocation created with cuMemAlloc 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 cuMemFree and a subsequent call to cuMemAlloc returns memory with the same device address, cuIpcGetMemHandle will return a unique handle for the new memory.

IPC functionality is restricted to devices with support for unified addressing on Linux operating systems.

See also:

cuMemAlloc, cuMemFree, cuIpcGetEventHandle, cuIpcOpenEventHandle, cuIpcOpenMemHandle, cuIpcCloseMemHandle

CUresult cuIpcOpenEventHandle (CUevent *phEvent, CUipcEventHandle handle)

Opens an interprocess event handle for use in the current process.

Parameters

phEvent
   - Returns the imported event
handle
   - Interprocess handle to open

Returns

CUDA_SUCCESS, CUDA_ERROR_INVALID_CONTEXT, CUDA_ERROR_MAP_FAILED, CUDA_ERROR_PEER_ACCESS_UNSUPPORTED, CUDA_ERROR_INVALID_HANDLE

Description

Opens an interprocess event handle exported from another process with cuIpcGetEventHandle. This function returns a CUevent that behaves like a locally created event with the CU_EVENT_DISABLE_TIMING flag specified. This event must be freed with cuEventDestroy.

Performing operations on the imported event after the exported event has been freed with cuEventDestroy will result in undefined behavior.

IPC functionality is restricted to devices with support for unified addressing on Linux operating systems.

See also:

cuEventCreate, cuEventDestroy, cuEventSynchronize, cuEventQuery, cuStreamWaitEvent, cuIpcGetEventHandle, cuIpcGetMemHandle, cuIpcOpenMemHandle, cuIpcCloseMemHandle

CUresult cuIpcOpenMemHandle (CUdeviceptr *pdptr, CUipcMemHandle handle, unsigned int Flags)

Opens an interprocess memory handle exported from another process and returns a device pointer usable in the local process.

Parameters

pdptr
  - Returned device pointer
handle
  - CUipcMemHandle to open
Flags
  - Flags for this operation. Must be specified as CU_IPC_MEM_LAZY_ENABLE_PEER_ACCESS

Returns

CUDA_SUCCESS, CUDA_ERROR_INVALID_CONTEXT, CUDA_ERROR_MAP_FAILED, CUDA_ERROR_INVALID_HANDLE, CUDA_ERROR_TOO_MANY_PEERS

Description

Maps memory exported from another process with cuIpcGetMemHandle into the current device address space. For contexts on different devices cuIpcOpenMemHandle can attempt to enable peer access between the devices as if the user called cuCtxEnablePeerAccess. This behavior is controlled by the CU_IPC_MEM_LAZY_ENABLE_PEER_ACCESS flag. cuDeviceCanAccessPeer can determine if a mapping is possible.

Contexts that may open CUipcMemHandles are restricted in the following way. CUipcMemHandles from each CUdevice in a given process may only be opened by one CUcontext per CUdevice per other process.

Memory returned from cuIpcOpenMemHandle must be freed with cuIpcCloseMemHandle.

Calling cuMemFree on an exported memory region before calling cuIpcCloseMemHandle in the importing context will result in undefined behavior.

IPC functionality is restricted to devices with support for unified addressing on Linux operating systems.

> No guarantees are made about the address returned in `*pdptr`. In particular, multiple processes may not receive the same address for the same `handle`.

See also:

cuMemAlloc, cuMemFree, cuIpcGetEventHandle, cuIpcOpenEventHandle, cuIpcGetMemHandle, cuIpcCloseMemHandle, cuCtxEnablePeerAccess, cuDeviceCanAccessPeer,

CUresult cuMemAlloc (CUdeviceptr *dptr, size_t bytesize)

Allocates device memory.

Parameters

dptr
- Returned device pointer
bytesize
- Requested allocation size in bytes

Returns

CUDA_SUCCESS, CUDA_ERROR_DEINITIALIZED, CUDA_ERROR_NOT_INITIALIZED, CUDA_ERROR_INVALID_CONTEXT, CUDA_ERROR_INVALID_VALUE, CUDA_ERROR_OUT_OF_MEMORY

Description

Allocates `bytesize` bytes of linear memory on the device and returns in `*dptr` a pointer to the allocated memory. The allocated memory is suitably aligned for any kind of variable. The memory is not cleared. If `bytesize` is 0, cuMemAlloc() returns CUDA_ERROR_INVALID_VALUE.

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

See also:

cuArray3DCreate, cuArray3DGetDescriptor, cuArrayCreate, cuArrayDestroy, cuArrayGetDescriptor, cuMemAllocHost, cuMemAllocPitch, cuMemcpy2D, cuMemcpy2DAsync, cuMemcpy2DUnaligned, cuMemcpy3D, cuMemcpy3DAsync, cuMemcpyAtoA, cuMemcpyAtoD, cuMemcpyAtoH, cuMemcpyAtoHAsync,

cuMemcpyDtoA, cuMemcpyDtoD, cuMemcpyDtoDAsync, cuMemcpyDtoH, cuMemcpyDtoHAsync, cuMemcpyHtoA, cuMemcpyHtoAAsync, cuMemcpyHtoD, cuMemcpyHtoDAsync, cuMemFree, cuMemFreeHost, cuMemGetAddressRange, cuMemGetInfo, cuMemHostAlloc, cuMemHostGetDevicePointer, cuMemsetD2D8, cuMemsetD2D16, cuMemsetD2D32, cuMemsetD8, cuMemsetD16, cuMemsetD32

CUresult cuMemAllocHost (void **pp, size_t bytesize)

Allocates page-locked host memory.

Parameters

pp
 - Returned host pointer to page-locked memory
bytesize
 - Requested allocation size in bytes

Returns

CUDA_SUCCESS, CUDA_ERROR_DEINITIALIZED, CUDA_ERROR_NOT_INITIALIZED, CUDA_ERROR_INVALID_CONTEXT, CUDA_ERROR_INVALID_VALUE, CUDA_ERROR_OUT_OF_MEMORY

Description

Allocates `bytesize` 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 cuMemcpy(). 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 memory with cuMemAllocHost() 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.

Note all host memory allocated using cuMemHostAlloc() will automatically be immediately accessible to all contexts on all devices which support unified addressing (as may be queried using CU_DEVICE_ATTRIBUTE_UNIFIED_ADDRESSING). The device pointer that may be used to access this host memory from those contexts is always equal to the returned host pointer *pp. See Unified Addressing for additional details.

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

See also:

cuArray3DCreate, cuArray3DGetDescriptor, cuArrayCreate, cuArrayDestroy, cuArrayGetDescriptor, cuMemAlloc, cuMemAllocPitch, cuMemcpy2D, cuMemcpy2DAsync, cuMemcpy2DUnaligned, cuMemcpy3D, cuMemcpy3DAsync, cuMemcpyAtoA, cuMemcpyAtoD, cuMemcpyAtoH, cuMemcpyAtoHAsync, cuMemcpyDtoA, cuMemcpyDtoD, cuMemcpyDtoDAsync, cuMemcpyDtoH, cuMemcpyDtoHAsync, cuMemcpyHtoA, cuMemcpyHtoAAsync, cuMemcpyHtoD, cuMemcpyHtoDAsync, cuMemFree, cuMemFreeHost, cuMemGetAddressRange, cuMemGetInfo, cuMemHostAlloc, cuMemHostGetDevicePointer, cuMemsetD2D8, cuMemsetD2D16, cuMemsetD2D32, cuMemsetD8, cuMemsetD16, cuMemsetD32

CUresult cuMemAllocManaged (CUdeviceptr *dptr, size_t bytesize, unsigned int flags)

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

Parameters

dptr
   - Returned device pointer
bytesize
   - Requested allocation size in bytes
flags
   - Must be one of CU_MEM_ATTACH_GLOBAL or CU_MEM_ATTACH_HOST

Returns

CUDA_SUCCESS, CUDA_ERROR_DEINITIALIZED, CUDA_ERROR_NOT_INITIALIZED, CUDA_ERROR_INVALID_CONTEXT, CUDA_ERROR_NOT_SUPPORTED, CUDA_ERROR_INVALID_VALUE, CUDA_ERROR_OUT_OF_MEMORY

Description

Allocates `bytesize` bytes of managed memory on the device and returns in `*dptr` a pointer to the allocated memory. If the device doesn't support allocating managed memory, CUDA_ERROR_NOT_SUPPORTED is returned. Support for managed memory can be queried using the device attribute CU_DEVICE_ATTRIBUTE_MANAGED_MEMORY. The allocated memory is suitably aligned for any kind of variable. The memory is not cleared. If `bytesize` is 0, cuMemAllocManaged returns CUDA_ERROR_INVALID_VALUE. 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 CU_MEM_ATTACH_GLOBAL or CU_MEM_ATTACH_HOST. If CU_MEM_ATTACH_GLOBAL is specified, then this memory is accessible from any stream on any device. If CU_MEM_ATTACH_HOST is specified, then the allocation is created with initial visibility restricted to host access only; an explicit call to cuStreamAttachMemAsync will be required to enable access on the device.

If the association is later changed via cuStreamAttachMemAsync to a single stream, the default association as specifed during cuMemAllocManaged is restored when that stream is destroyed. For __managed__ variables, the default association is always CU_MEM_ATTACH_GLOBAL. 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 cuMemAllocManaged should be released with cuMemFree.

On a multi-GPU system with peer-to-peer support, where multiple GPUs support managed memory, the physical storage is created on the GPU which is active at the time cuMemAllocManaged is called. All other GPUs will reference the data at reduced bandwidth via peer mappings over the PCIe bus. The Unified Memory management system does not migrate memory between GPUs.

On a multi-GPU system where multiple GPUs support managed memory, but not all pairs of such GPUs have peer-to-peer support between them, the physical storage is created in 'zero-copy' or system 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 contexts created in that process on devices that support managed memory have to be peer-to-peer compatible with each other. Context creation will fail if a context is created on a device that supports managed memory and is not peer-to-peer compatible with any of the other managed memory supporting devices on which contexts were previously created, even if those contexts have been destroyed. These environment variables are described in the CUDA programming guide under the "CUDA environment variables" section.

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

See also:

cuArray3DCreate, cuArray3DGetDescriptor, cuArrayCreate, cuArrayDestroy, cuArrayGetDescriptor, cuMemAllocHost, cuMemAllocPitch, cuMemcpy2D,

cuMemcpy2DAsync, cuMemcpy2DUnaligned, cuMemcpy3D, cuMemcpy3DAsync,
cuMemcpyAtoA, cuMemcpyAtoD, cuMemcpyAtoH, cuMemcpyAtoHAsync,
cuMemcpyDtoA, cuMemcpyDtoD, cuMemcpyDtoDAsync, cuMemcpyDtoH,
cuMemcpyDtoHAsync, cuMemcpyHtoA, cuMemcpyHtoAAsync, cuMemcpyHtoD,
cuMemcpyHtoDAsync, cuMemFree, cuMemFreeHost, cuMemGetAddressRange,
cuMemGetInfo, cuMemHostAlloc, cuMemHostGetDevicePointer, cuMemsetD2D8,
cuMemsetD2D16, cuMemsetD2D32, cuMemsetD8, cuMemsetD16, cuMemsetD32,
cuDeviceGetAttribute, cuStreamAttachMemAsync

CUresult cuMemAllocPitch (CUdeviceptr *dptr, size_t *pPitch, size_t WidthInBytes, size_t Height, unsigned int ElementSizeBytes)

Allocates pitched device memory.

Parameters

dptr
  - Returned device pointer
pPitch
  - Returned pitch of allocation in bytes
WidthInBytes
  - Requested allocation width in bytes
Height
  - Requested allocation height in rows
ElementSizeBytes
  - Size of largest reads/writes for range

Returns

CUDA_SUCCESS, CUDA_ERROR_DEINITIALIZED,
CUDA_ERROR_NOT_INITIALIZED, CUDA_ERROR_INVALID_CONTEXT,
CUDA_ERROR_INVALID_VALUE, CUDA_ERROR_OUT_OF_MEMORY

Description

Allocates at least `WidthInBytes` * `Height` bytes of linear memory on the device
and returns in `*dptr` 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. `ElementSizeBytes` specifies the size of the largest reads and writes
that will be performed on the memory range. `ElementSizeBytes` may be 4, 8 or
16 (since coalesced memory transactions are not possible on other data sizes). If
`ElementSizeBytes` is smaller than the actual read/write size of a kernel, the kernel
will run correctly, but possibly at reduced speed. The pitch returned in `*pPitch` by

cuMemAllocPitch() 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;
```

The pitch returned by cuMemAllocPitch() is guaranteed to work with cuMemcpy2D() under all circumstances. For allocations of 2D arrays, it is recommended that programmers consider performing pitch allocations using cuMemAllocPitch(). Due to 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).

The byte alignment of the pitch returned by cuMemAllocPitch() is guaranteed to match or exceed the alignment requirement for texture binding with cuTexRefSetAddress2D().

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

See also:

cuArray3DCreate, cuArray3DGetDescriptor, cuArrayCreate, cuArrayDestroy, cuArrayGetDescriptor, cuMemAlloc, cuMemAllocHost, cuMemcpy2D, cuMemcpy2DAsync, cuMemcpy2DUnaligned, cuMemcpy3D, cuMemcpy3DAsync, cuMemcpyAtoA, cuMemcpyAtoD, cuMemcpyAtoH, cuMemcpyAtoHAsync, cuMemcpyDtoA, cuMemcpyDtoD, cuMemcpyDtoDAsync, cuMemcpyDtoH, cuMemcpyDtoHAsync, cuMemcpyHtoA, cuMemcpyHtoAAsync, cuMemcpyHtoD, cuMemcpyHtoDAsync, cuMemFree, cuMemFreeHost, cuMemGetAddressRange, cuMemGetInfo, cuMemHostAlloc, cuMemHostGetDevicePointer, cuMemsetD2D8, cuMemsetD2D16, cuMemsetD2D32, cuMemsetD8, cuMemsetD16, cuMemsetD32

CUresult cuMemcpy (CUdeviceptr dst, CUdeviceptr src, size_t ByteCount)

Copies memory.

Parameters

dst
   - Destination unified virtual address space pointer
src
   - Source unified virtual address space pointer
ByteCount
   - Size of memory copy in bytes

Returns

CUDA_SUCCESS, CUDA_ERROR_DEINITIALIZED,
CUDA_ERROR_NOT_INITIALIZED, CUDA_ERROR_INVALID_CONTEXT,
CUDA_ERROR_INVALID_VALUE

Description

Copies data between two pointers. `dst` and `src` are base pointers of the destination and source, respectively. `ByteCount` specifies the number of bytes to copy. Note that this function infers the type of the transfer (host to host, host to device, device to device, or device to host) from the pointer values. This function is only allowed in contexts which support unified addressing.

> ▸ Note that this function may also return error codes from previous, asynchronous launches.
> ▸ This function exhibits synchronous behavior for most use cases.

See also:

cuArray3DCreate, cuArray3DGetDescriptor, cuArrayCreate, cuArrayDestroy, cuArrayGetDescriptor, cuMemAlloc, cuMemAllocHost, cuMemAllocPitch, cuMemcpy2D, cuMemcpy2DAsync, cuMemcpy2DUnaligned, cuMemcpy3D, cuMemcpy3DAsync, cuMemcpyAtoA, cuMemcpyAtoD, cuMemcpyAtoH, cuMemcpyAtoHAsync, cuMemcpyDtoA, cuMemcpyDtoH, cuMemcpyDtoHAsync, cuMemcpyHtoA, cuMemcpyHtoAAsync, cuMemcpyHtoD, cuMemcpyHtoDAsync, cuMemFree, cuMemFreeHost, cuMemGetAddressRange, cuMemGetInfo, cuMemHostAlloc, cuMemHostGetDevicePointer, cuMemsetD2D8, cuMemsetD2D16, cuMemsetD2D32, cuMemsetD8, cuMemsetD16, cuMemsetD32

CUresult cuMemcpy2D (const CUDA_MEMCPY2D *pCopy)

Copies memory for 2D arrays.

Parameters

pCopy
  - Parameters for the memory copy

Returns

CUDA_SUCCESS, CUDA_ERROR_DEINITIALIZED,
CUDA_ERROR_NOT_INITIALIZED, CUDA_ERROR_INVALID_CONTEXT,
CUDA_ERROR_INVALID_VALUE

Description

Perform a 2D memory copy according to the parameters specified in `pCopy`. The CUDA_MEMCPY2D structure is defined as:

```
typedef struct CUDA_MEMCPY2D_st {
        unsigned int srcXInBytes, srcY;
        CUmemorytype srcMemoryType;
            const void *srcHost;
            CUdeviceptr srcDevice;
            CUarray srcArray;
            unsigned int srcPitch;

        unsigned int dstXInBytes, dstY;
        CUmemorytype dstMemoryType;
            void *dstHost;
            CUdeviceptr dstDevice;
            CUarray dstArray;
            unsigned int dstPitch;

        unsigned int WidthInBytes;
        unsigned int Height;
    } CUDA_MEMCPY2D;
```

where:

▶ srcMemoryType and dstMemoryType specify the type of memory of the source and destination, respectively; CUmemorytype_enum is defined as:

```
typedef enum CUmemorytype_enum {
        CU_MEMORYTYPE_HOST = 0x01,
        CU_MEMORYTYPE_DEVICE = 0x02,
        CU_MEMORYTYPE_ARRAY = 0x03,
        CU_MEMORYTYPE_UNIFIED = 0x04
    } CUmemorytype;
```

If srcMemoryType is CU_MEMORYTYPE_UNIFIED, srcDevice and srcPitch specify the (unified virtual address space) base address of the source data and the bytes per row to apply. srcArray is ignored. This value may be used only if unified addressing is supported in the calling context.

If srcMemoryType is CU_MEMORYTYPE_HOST, srcHost and srcPitch specify the (host) base address of the source data and the bytes per row to apply. srcArray is ignored.

If srcMemoryType is CU_MEMORYTYPE_DEVICE, srcDevice and srcPitch specify the (device) base address of the source data and the bytes per row to apply. srcArray is ignored.

If srcMemoryType is CU_MEMORYTYPE_ARRAY, srcArray specifies the handle of the source data. srcHost, srcDevice and srcPitch are ignored.

If dstMemoryType is CU_MEMORYTYPE_HOST, dstHost and dstPitch specify the (host) base address of the destination data and the bytes per row to apply. dstArray is ignored.

If dstMemoryType is CU_MEMORYTYPE_UNIFIED, dstDevice and dstPitch specify the (unified virtual address space) base address of the source data and the bytes per row to apply. dstArray is ignored. This value may be used only if unified addressing is supported in the calling context.

If dstMemoryType is CU_MEMORYTYPE_DEVICE, dstDevice and dstPitch specify the (device) base address of the destination data and the bytes per row to apply. dstArray is ignored.

If dstMemoryType is CU_MEMORYTYPE_ARRAY, dstArray specifies the handle of the destination data. dstHost, dstDevice and dstPitch are ignored.

▸ srcXInBytes and srcY specify the base address of the source data for the copy.

For host pointers, the starting address is

```
void* Start = (void*)((char*)srcHost+srcY*srcPitch + srcXInBytes);
```

For device pointers, the starting address is

```
CUdeviceptr Start = srcDevice+srcY*srcPitch+srcXInBytes;
```

For CUDA arrays, srcXInBytes must be evenly divisible by the array element size.

▸ dstXInBytes and dstY specify the base address of the destination data for the copy.

For host pointers, the base address is

```
void* dstStart = (void*)((char*)dstHost+dstY*dstPitch + dstXInBytes);
```

For device pointers, the starting address is

```
CUdeviceptr dstStart = dstDevice+dstY*dstPitch+dstXInBytes;
```

For CUDA arrays, dstXInBytes must be evenly divisible by the array element size.

▸ WidthInBytes and Height specify the width (in bytes) and height of the 2D copy being performed.
▸ If specified, srcPitch must be greater than or equal to WidthInBytes + srcXInBytes, and dstPitch must be greater than or equal to WidthInBytes + dstXInBytes.

cuMemcpy2D() returns an error if any pitch is greater than the maximum allowed (CU_DEVICE_ATTRIBUTE_MAX_PITCH). cuMemAllocPitch() passes back pitches that always work with cuMemcpy2D(). On intra-device memory copies (device to device, CUDA array to device, CUDA array to CUDA array), cuMemcpy2D() may fail for pitches not computed by cuMemAllocPitch(). cuMemcpy2DUnaligned() does not have this restriction, but may run significantly slower in the cases where cuMemcpy2D() would have returned an error code.

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

See also:

cuArray3DCreate, cuArray3DGetDescriptor, cuArrayCreate, cuArrayDestroy, cuArrayGetDescriptor, cuMemAlloc, cuMemAllocHost, cuMemAllocPitch, cuMemcpy2DAsync, cuMemcpy2DUnaligned, cuMemcpy3D, cuMemcpy3DAsync, cuMemcpyAtoA, cuMemcpyAtoD, cuMemcpyAtoH, cuMemcpyAtoHAsync, cuMemcpyDtoA, cuMemcpyDtoD, cuMemcpyDtoDAsync, cuMemcpyDtoH, cuMemcpyDtoHAsync, cuMemcpyHtoA, cuMemcpyHtoAAsync, cuMemcpyHtoD, cuMemcpyHtoDAsync, cuMemFree, cuMemFreeHost, cuMemGetAddressRange, cuMemGetInfo, cuMemHostAlloc, cuMemHostGetDevicePointer, cuMemsetD2D8, cuMemsetD2D16, cuMemsetD2D32, cuMemsetD8, cuMemsetD16, cuMemsetD32

CUresult cuMemcpy2DAsync (const CUDA_MEMCPY2D *pCopy, CUstream hStream)

Copies memory for 2D arrays.

Parameters

pCopy
  - Parameters for the memory copy
hStream
  - Stream identifier

Returns

CUDA_SUCCESS, CUDA_ERROR_DEINITIALIZED, CUDA_ERROR_NOT_INITIALIZED, CUDA_ERROR_INVALID_CONTEXT, CUDA_ERROR_INVALID_VALUE

Description

Perform a 2D memory copy according to the parameters specified in `pCopy`. The CUDA_MEMCPY2D structure is defined as:

```
typedef struct CUDA_MEMCPY2D_st {
        unsigned int srcXInBytes, srcY;
        CUmemorytype srcMemoryType;
        const void *srcHost;
        CUdeviceptr srcDevice;
        CUarray srcArray;
        unsigned int srcPitch;
        unsigned int dstXInBytes, dstY;
        CUmemorytype dstMemoryType;
        void *dstHost;
        CUdeviceptr dstDevice;
        CUarray dstArray;
        unsigned int dstPitch;
        unsigned int WidthInBytes;
        unsigned int Height;
    } CUDA_MEMCPY2D;
```

where:

▸ srcMemoryType and dstMemoryType specify the type of memory of the source and destination, respectively; CUmemorytype_enum is defined as:

```
typedef enum CUmemorytype_enum {
        CU_MEMORYTYPE_HOST = 0x01,
        CU_MEMORYTYPE_DEVICE = 0x02,
        CU_MEMORYTYPE_ARRAY = 0x03,
        CU_MEMORYTYPE_UNIFIED = 0x04
    } CUmemorytype;
```

If srcMemoryType is CU_MEMORYTYPE_HOST, srcHost and srcPitch specify the (host) base address of the source data and the bytes per row to apply. srcArray is ignored.

If srcMemoryType is CU_MEMORYTYPE_UNIFIED, srcDevice and srcPitch specify the (unified virtual address space) base address of the source data and the bytes per row to apply. srcArray is ignored. This value may be used only if unified addressing is supported in the calling context.

If srcMemoryType is CU_MEMORYTYPE_DEVICE, srcDevice and srcPitch specify the (device) base address of the source data and the bytes per row to apply. srcArray is ignored.

If srcMemoryType is CU_MEMORYTYPE_ARRAY, srcArray specifies the handle of the source data. srcHost, srcDevice and srcPitch are ignored.

If dstMemoryType is CU_MEMORYTYPE_UNIFIED, dstDevice and dstPitch specify the (unified virtual address space) base address of the source data and the bytes per row to apply. dstArray is ignored. This value may be used only if unified addressing is supported in the calling context.

If dstMemoryType is CU_MEMORYTYPE_HOST, dstHost and dstPitch specify the (host) base address of the destination data and the bytes per row to apply. dstArray is ignored.

If dstMemoryType is CU_MEMORYTYPE_DEVICE, dstDevice and dstPitch specify the (device) base address of the destination data and the bytes per row to apply. dstArray is ignored.

If dstMemoryType is CU_MEMORYTYPE_ARRAY, dstArray specifies the handle of the destination data. dstHost, dstDevice and dstPitch are ignored.

▶ srcXInBytes and srcY specify the base address of the source data for the copy.

For host pointers, the starting address is

```
void* Start = (void*)((char*)srcHost+srcY*srcPitch + srcXInBytes);
```

For device pointers, the starting address is

```
CUdeviceptr Start = srcDevice+srcY*srcPitch+srcXInBytes;
```

For CUDA arrays, srcXInBytes must be evenly divisible by the array element size.

▶ dstXInBytes and dstY specify the base address of the destination data for the copy.

For host pointers, the base address is

```
void* dstStart = (void*)((char*)dstHost+dstY*dstPitch + dstXInBytes);
```

For device pointers, the starting address is

```
CUdeviceptr dstStart = dstDevice+dstY*dstPitch+dstXInBytes;
```

For CUDA arrays, dstXInBytes must be evenly divisible by the array element size.

▶ WidthInBytes and Height specify the width (in bytes) and height of the 2D copy being performed.
▶ If specified, srcPitch must be greater than or equal to WidthInBytes + srcXInBytes, and dstPitch must be greater than or equal to WidthInBytes + dstXInBytes.
▶ If specified, srcPitch must be greater than or equal to WidthInBytes + srcXInBytes, and dstPitch must be greater than or equal to WidthInBytes + dstXInBytes.
▶ If specified, srcHeight must be greater than or equal to Height + srcY, and dstHeight must be greater than or equal to Height + dstY.

cuMemcpy2DAsync() returns an error if any pitch is greater than the maximum allowed (CU_DEVICE_ATTRIBUTE_MAX_PITCH). cuMemAllocPitch() passes back pitches that always work with cuMemcpy2D(). On intra-device memory copies (device to device, CUDA array to device, CUDA array to CUDA array), cuMemcpy2DAsync() may fail for pitches not computed by cuMemAllocPitch().

> ▸ Note that this function may also return error codes from previous, asynchronous launches.
>
> ▸ This function exhibits asynchronous behavior for most use cases.
>
> ▸ This function uses standard NULL stream semantics.

See also:

cuArray3DCreate, cuArray3DGetDescriptor, cuArrayCreate, cuArrayDestroy, cuArrayGetDescriptor, cuMemAlloc, cuMemAllocHost, cuMemAllocPitch, cuMemcpy2D, cuMemcpy2DUnaligned, cuMemcpy3D, cuMemcpy3DAsync, cuMemcpyAtoA, cuMemcpyAtoD, cuMemcpyAtoH, cuMemcpyAtoHAsync, cuMemcpyDtoA, cuMemcpyDtoD, cuMemcpyDtoDAsync, cuMemcpyDtoH, cuMemcpyDtoHAsync, cuMemcpyHtoA, cuMemcpyHtoAAsync, cuMemcpyHtoD, cuMemcpyHtoDAsync, cuMemFree, cuMemFreeHost, cuMemGetAddressRange, cuMemGetInfo, cuMemHostAlloc, cuMemHostGetDevicePointer, cuMemsetD2D8, cuMemsetD2D8Async, cuMemsetD2D16, cuMemsetD2D16Async, cuMemsetD2D32, cuMemsetD2D32Async, cuMemsetD8, cuMemsetD8Async, cuMemsetD16, cuMemsetD16Async, cuMemsetD32, cuMemsetD32Async

CUresult cuMemcpy2DUnaligned (const CUDA_MEMCPY2D *pCopy)

Copies memory for 2D arrays.

Parameters

pCopy
  - Parameters for the memory copy

Returns

CUDA_SUCCESS, CUDA_ERROR_DEINITIALIZED, CUDA_ERROR_NOT_INITIALIZED, CUDA_ERROR_INVALID_CONTEXT, CUDA_ERROR_INVALID_VALUE

Description

Perform a 2D memory copy according to the parameters specified in `pCopy`. The CUDA_MEMCPY2D structure is defined as:

```
typedef struct CUDA_MEMCPY2D_st {
        unsigned int srcXInBytes, srcY;
        CUmemorytype srcMemoryType;
        const void *srcHost;
        CUdeviceptr srcDevice;
        CUarray srcArray;
        unsigned int srcPitch;
        unsigned int dstXInBytes, dstY;
        CUmemorytype dstMemoryType;
        void *dstHost;
        CUdeviceptr dstDevice;
        CUarray dstArray;
        unsigned int dstPitch;
        unsigned int WidthInBytes;
        unsigned int Height;
    } CUDA_MEMCPY2D;
```

where:

▶ srcMemoryType and dstMemoryType specify the type of memory of the source and destination, respectively; CUmemorytype_enum is defined as:

```
typedef enum CUmemorytype_enum {
        CU_MEMORYTYPE_HOST = 0x01,
        CU_MEMORYTYPE_DEVICE = 0x02,
        CU_MEMORYTYPE_ARRAY = 0x03,
        CU_MEMORYTYPE_UNIFIED = 0x04
    } CUmemorytype;
```

If srcMemoryType is CU_MEMORYTYPE_UNIFIED, srcDevice and srcPitch specify the (unified virtual address space) base address of the source data and the bytes per row to apply. srcArray is ignored. This value may be used only if unified addressing is supported in the calling context.

If srcMemoryType is CU_MEMORYTYPE_HOST, srcHost and srcPitch specify the (host) base address of the source data and the bytes per row to apply. srcArray is ignored.

If srcMemoryType is CU_MEMORYTYPE_DEVICE, srcDevice and srcPitch specify the (device) base address of the source data and the bytes per row to apply. srcArray is ignored.

If srcMemoryType is CU_MEMORYTYPE_ARRAY, srcArray specifies the handle of the source data. srcHost, srcDevice and srcPitch are ignored.

If dstMemoryType is CU_MEMORYTYPE_UNIFIED, dstDevice and dstPitch specify the (unified virtual address space) base address of the source data and the bytes per row to apply. dstArray is ignored. This value may be used only if unified addressing is supported in the calling context.

If dstMemoryType is CU_MEMORYTYPE_HOST, dstHost and dstPitch specify the (host) base address of the destination data and the bytes per row to apply. dstArray is ignored.

If dstMemoryType is CU_MEMORYTYPE_DEVICE, dstDevice and dstPitch specify the (device) base address of the destination data and the bytes per row to apply. dstArray is ignored.

If dstMemoryType is CU_MEMORYTYPE_ARRAY, dstArray specifies the handle of the destination data. dstHost, dstDevice and dstPitch are ignored.

▶ srcXInBytes and srcY specify the base address of the source data for the copy.

For host pointers, the starting address is

```
void* Start = (void*)((char*)srcHost+srcY*srcPitch + srcXInBytes);
```

For device pointers, the starting address is

```
CUdeviceptr Start = srcDevice+srcY*srcPitch+srcXInBytes;
```

For CUDA arrays, srcXInBytes must be evenly divisible by the array element size.

▶ dstXInBytes and dstY specify the base address of the destination data for the copy.

For host pointers, the base address is

```
void* dstStart = (void*)((char*)dstHost+dstY*dstPitch + dstXInBytes);
```

For device pointers, the starting address is

```
CUdeviceptr dstStart = dstDevice+dstY*dstPitch+dstXInBytes;
```

For CUDA arrays, dstXInBytes must be evenly divisible by the array element size.

▶ WidthInBytes and Height specify the width (in bytes) and height of the 2D copy being performed.
▶ If specified, srcPitch must be greater than or equal to WidthInBytes + srcXInBytes, and dstPitch must be greater than or equal to WidthInBytes + dstXInBytes.

cuMemcpy2D() returns an error if any pitch is greater than the maximum allowed (CU_DEVICE_ATTRIBUTE_MAX_PITCH). cuMemAllocPitch() passes back pitches that always work with cuMemcpy2D(). On intra-device memory copies (device to device, CUDA array to device, CUDA array to CUDA array), cuMemcpy2D() may fail for pitches not computed by cuMemAllocPitch(). cuMemcpy2DUnaligned() does not have this restriction, but may run significantly slower in the cases where cuMemcpy2D() would have returned an error code.

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

▶ This function exhibits synchronous behavior for most use cases.

See also:

cuArray3DCreate, cuArray3DGetDescriptor, cuArrayCreate, cuArrayDestroy, cuArrayGetDescriptor, cuMemAlloc, cuMemAllocHost, cuMemAllocPitch, cuMemcpy2D, cuMemcpy2DAsync, cuMemcpy3D, cuMemcpy3DAsync, cuMemcpyAtoA, cuMemcpyAtoD, cuMemcpyAtoH, cuMemcpyAtoHAsync, cuMemcpyDtoA, cuMemcpyDtoD, cuMemcpyDtoDAsync, cuMemcpyDtoH, cuMemcpyDtoHAsync, cuMemcpyHtoA, cuMemcpyHtoAAsync, cuMemcpyHtoD, cuMemcpyHtoDAsync, cuMemFree, cuMemFreeHost, cuMemGetAddressRange, cuMemGetInfo, cuMemHostAlloc, cuMemHostGetDevicePointer, cuMemsetD2D8, cuMemsetD2D16, cuMemsetD2D32, cuMemsetD8, cuMemsetD16, cuMemsetD32

CUresult cuMemcpy3D (const CUDA_MEMCPY3D *pCopy)

Copies memory for 3D arrays.

Parameters

pCopy
  - Parameters for the memory copy

Returns

CUDA_SUCCESS, CUDA_ERROR_DEINITIALIZED, CUDA_ERROR_NOT_INITIALIZED, CUDA_ERROR_INVALID_CONTEXT, CUDA_ERROR_INVALID_VALUE

Description

Perform a 3D memory copy according to the parameters specified in `pCopy`. The CUDA_MEMCPY3D structure is defined as:

```
typedef struct CUDA_MEMCPY3D_st {

        unsigned int srcXInBytes, srcY, srcZ;
        unsigned int srcLOD;
        CUmemorytype srcMemoryType;
            const void *srcHost;
            CUdeviceptr srcDevice;
            CUarray srcArray;
            unsigned int srcPitch;  // ignored when src is array
            unsigned int srcHeight; // ignored when src is array; may
 be 0 if Depth==1

        unsigned int dstXInBytes, dstY, dstZ;
        unsigned int dstLOD;
        CUmemorytype dstMemoryType;
            void *dstHost;
            CUdeviceptr dstDevice;
            CUarray dstArray;
            unsigned int dstPitch;  // ignored when dst is array
            unsigned int dstHeight; // ignored when dst is array; may
 be 0 if Depth==1

        unsigned int WidthInBytes;
        unsigned int Height;
        unsigned int Depth;
    } CUDA_MEMCPY3D;
```

where:

▶ srcMemoryType and dstMemoryType specify the type of memory of the source and destination, respectively; CUmemorytype_enum is defined as:

```
typedef enum CUmemorytype_enum {
        CU_MEMORYTYPE_HOST = 0x01,
        CU_MEMORYTYPE_DEVICE = 0x02,
        CU_MEMORYTYPE_ARRAY = 0x03,
        CU_MEMORYTYPE_UNIFIED = 0x04
    } CUmemorytype;
```

If srcMemoryType is CU_MEMORYTYPE_UNIFIED, srcDevice and srcPitch specify the (unified virtual address space) base address of the source data and the bytes per row to apply. srcArray is ignored. This value may be used only if unified addressing is supported in the calling context.

If srcMemoryType is CU_MEMORYTYPE_HOST, srcHost, srcPitch and srcHeight specify the (host) base address of the source data, the bytes per row, and the height of each 2D slice of the 3D array. srcArray is ignored.

If srcMemoryType is CU_MEMORYTYPE_DEVICE, srcDevice, srcPitch and srcHeight specify the (device) base address of the source data, the bytes per row, and the height of each 2D slice of the 3D array. srcArray is ignored.

If srcMemoryType is CU_MEMORYTYPE_ARRAY, srcArray specifies the handle of the source data. srcHost, srcDevice, srcPitch and srcHeight are ignored.

If dstMemoryType is CU_MEMORYTYPE_UNIFIED, dstDevice and dstPitch specify the (unified virtual address space) base address of the source data and the bytes per row to apply. dstArray is ignored. This value may be used only if unified addressing is supported in the calling context.

If dstMemoryType is CU_MEMORYTYPE_HOST, dstHost and dstPitch specify the (host) base address of the destination data, the bytes per row, and the height of each 2D slice of the 3D array. dstArray is ignored.

If dstMemoryType is CU_MEMORYTYPE_DEVICE, dstDevice and dstPitch specify the (device) base address of the destination data, the bytes per row, and the height of each 2D slice of the 3D array. dstArray is ignored.

If dstMemoryType is CU_MEMORYTYPE_ARRAY, dstArray specifies the handle of the destination data. dstHost, dstDevice, dstPitch and dstHeight are ignored.

▸ srcXInBytes, srcY and srcZ specify the base address of the source data for the copy.

For host pointers, the starting address is

```
  void* Start = (void*)((char*)srcHost+(srcZ*srcHeight+srcY)*srcPitch +
 srcXInBytes);
```

For device pointers, the starting address is

```
  CUdeviceptr Start = srcDevice+(srcZ*srcHeight+srcY)*srcPitch+srcXInBytes;
```

For CUDA arrays, srcXInBytes must be evenly divisible by the array element size.

▸ dstXInBytes, dstY and dstZ specify the base address of the destination data for the copy.

For host pointers, the base address is

```
  void* dstStart = (void*)((char*)dstHost+(dstZ*dstHeight+dstY)*dstPitch +
 dstXInBytes);
```

For device pointers, the starting address is

```
  CUdeviceptr dstStart = dstDevice+(dstZ*dstHeight+dstY)*dstPitch+dstXInBytes;
```

For CUDA arrays, dstXInBytes must be evenly divisible by the array element size.

▸ WidthInBytes, Height and Depth specify the width (in bytes), height and depth of the 3D copy being performed.
▸ If specified, srcPitch must be greater than or equal to WidthInBytes + srcXInBytes, and dstPitch must be greater than or equal to WidthInBytes + dstXInBytes.
▸ If specified, srcHeight must be greater than or equal to Height + srcY, and dstHeight must be greater than or equal to Height + dstY.

cuMemcpy3D() returns an error if any pitch is greater than the maximum allowed (CU_DEVICE_ATTRIBUTE_MAX_PITCH).

The srcLOD and dstLOD members of the CUDA_MEMCPY3D structure must be set to 0.

> ▸ Note that this function may also return error codes from previous, asynchronous launches.
> ▸ This function exhibits synchronous behavior for most use cases.

See also:

cuArray3DCreate, cuArray3DGetDescriptor, cuArrayCreate, cuArrayDestroy, cuArrayGetDescriptor, cuMemAlloc, cuMemAllocHost, cuMemAllocPitch, cuMemcpy2D, cuMemcpy2DAsync, cuMemcpy2DUnaligned, cuMemcpy3DAsync, cuMemcpyAtoA, cuMemcpyAtoD, cuMemcpyAtoH, cuMemcpyAtoHAsync, cuMemcpyDtoA, cuMemcpyDtoD, cuMemcpyDtoDAsync, cuMemcpyDtoH, cuMemcpyDtoHAsync, cuMemcpyHtoA, cuMemcpyHtoAAsync, cuMemcpyHtoD, cuMemcpyHtoDAsync, cuMemFree, cuMemFreeHost, cuMemGetAddressRange, cuMemGetInfo, cuMemHostAlloc, cuMemHostGetDevicePointer, cuMemsetD2D8, cuMemsetD2D16, cuMemsetD2D32, cuMemsetD8, cuMemsetD16, cuMemsetD32

CUresult cuMemcpy3DAsync (const CUDA_MEMCPY3D *pCopy, CUstream hStream)

Copies memory for 3D arrays.

Parameters

pCopy
   - Parameters for the memory copy
hStream
   - Stream identifier

Returns

CUDA_SUCCESS, CUDA_ERROR_DEINITIALIZED, CUDA_ERROR_NOT_INITIALIZED, CUDA_ERROR_INVALID_CONTEXT, CUDA_ERROR_INVALID_VALUE

Description

Perform a 3D memory copy according to the parameters specified in `pCopy`. The CUDA_MEMCPY3D structure is defined as:

```
        typedef struct CUDA_MEMCPY3D_st {

                unsigned int srcXInBytes, srcY, srcZ;
                unsigned int srcLOD;
                CUmemorytype srcMemoryType;
                    const void *srcHost;
                    CUdeviceptr srcDevice;
                    CUarray srcArray;
                    unsigned int srcPitch;  // ignored when src is array
                    unsigned int srcHeight; // ignored when src is array; may
 be 0 if Depth==1

                unsigned int dstXInBytes, dstY, dstZ;
                unsigned int dstLOD;
                CUmemorytype dstMemoryType;
                    void *dstHost;
                    CUdeviceptr dstDevice;
                    CUarray dstArray;
                    unsigned int dstPitch;  // ignored when dst is array
                    unsigned int dstHeight; // ignored when dst is array; may
 be 0 if Depth==1

                unsigned int WidthInBytes;
                unsigned int Height;
                unsigned int Depth;
            } CUDA_MEMCPY3D;
```

where:

▸ srcMemoryType and dstMemoryType specify the type of memory of the source and destination, respectively; CUmemorytype_enum is defined as:

```
    typedef enum CUmemorytype_enum {
            CU_MEMORYTYPE_HOST = 0x01,
            CU_MEMORYTYPE_DEVICE = 0x02,
            CU_MEMORYTYPE_ARRAY = 0x03,
            CU_MEMORYTYPE_UNIFIED = 0x04
        } CUmemorytype;
```

If srcMemoryType is CU_MEMORYTYPE_UNIFIED, srcDevice and srcPitch specify the (unified virtual address space) base address of the source data and the bytes per row to apply. srcArray is ignored. This value may be used only if unified addressing is supported in the calling context.

If srcMemoryType is CU_MEMORYTYPE_HOST, srcHost, srcPitch and srcHeight specify the (host) base address of the source data, the bytes per row, and the height of each 2D slice of the 3D array. srcArray is ignored.

If srcMemoryType is CU_MEMORYTYPE_DEVICE, srcDevice, srcPitch and srcHeight specify the (device) base address of the source data, the bytes per row, and the height of each 2D slice of the 3D array. srcArray is ignored.

If srcMemoryType is CU_MEMORYTYPE_ARRAY, srcArray specifies the handle of the source data. srcHost, srcDevice, srcPitch and srcHeight are ignored.

If dstMemoryType is CU_MEMORYTYPE_UNIFIED, dstDevice and dstPitch specify the (unified virtual address space) base address of the source data and the bytes per row to apply. dstArray is ignored. This value may be used only if unified addressing is supported in the calling context.

If dstMemoryType is CU_MEMORYTYPE_HOST, dstHost and dstPitch specify the (host) base address of the destination data, the bytes per row, and the height of each 2D slice of the 3D array. dstArray is ignored.

If dstMemoryType is CU_MEMORYTYPE_DEVICE, dstDevice and dstPitch specify the (device) base address of the destination data, the bytes per row, and the height of each 2D slice of the 3D array. dstArray is ignored.

If dstMemoryType is CU_MEMORYTYPE_ARRAY, dstArray specifies the handle of the destination data. dstHost, dstDevice, dstPitch and dstHeight are ignored.

▸   srcXInBytes, srcY and srcZ specify the base address of the source data for the copy.

For host pointers, the starting address is

```
  void* Start = (void*)((char*)srcHost+(srcZ*srcHeight+srcY)*srcPitch +
 srcXInBytes);
```

For device pointers, the starting address is

```
  CUdeviceptr Start = srcDevice+(srcZ*srcHeight+srcY)*srcPitch+srcXInBytes;
```

For CUDA arrays, srcXInBytes must be evenly divisible by the array element size.

▸   dstXInBytes, dstY and dstZ specify the base address of the destination data for the copy.

For host pointers, the base address is

```
  void* dstStart = (void*)((char*)dstHost+(dstZ*dstHeight+dstY)*dstPitch +
 dstXInBytes);
```

For device pointers, the starting address is

```
  CUdeviceptr dstStart = dstDevice+(dstZ*dstHeight+dstY)*dstPitch+dstXInBytes;
```

For CUDA arrays, dstXInBytes must be evenly divisible by the array element size.

▸   WidthInBytes, Height and Depth specify the width (in bytes), height and depth of the 3D copy being performed.
▸   If specified, srcPitch must be greater than or equal to WidthInBytes + srcXInBytes, and dstPitch must be greater than or equal to WidthInBytes + dstXInBytes.
▸   If specified, srcHeight must be greater than or equal to Height + srcY, and dstHeight must be greater than or equal to Height + dstY.

cuMemcpy3DAsync() returns an error if any pitch is greater than the maximum allowed (CU_DEVICE_ATTRIBUTE_MAX_PITCH).

The srcLOD and dstLOD members of the CUDA_MEMCPY3D structure must be set to 0.

> ‣ Note that this function may also return error codes from previous, asynchronous launches.
> ‣ This function exhibits asynchronous behavior for most use cases.
> ‣ This function uses standard NULL stream semantics.

See also:

cuArray3DCreate, cuArray3DGetDescriptor, cuArrayCreate, cuArrayDestroy, cuArrayGetDescriptor, cuMemAlloc, cuMemAllocHost, cuMemAllocPitch, cuMemcpy2D, cuMemcpy2DAsync, cuMemcpy2DUnaligned, cuMemcpy3D, cuMemcpyAtoA, cuMemcpyAtoD, cuMemcpyAtoH, cuMemcpyAtoHAsync, cuMemcpyDtoA, cuMemcpyDtoD, cuMemcpyDtoDAsync, cuMemcpyDtoH, cuMemcpyDtoHAsync, cuMemcpyHtoA, cuMemcpyHtoAAsync, cuMemcpyHtoD, cuMemcpyHtoDAsync, cuMemFree, cuMemFreeHost, cuMemGetAddressRange, cuMemGetInfo, cuMemHostAlloc, cuMemHostGetDevicePointer, cuMemsetD2D8, cuMemsetD2D8Async, cuMemsetD2D16, cuMemsetD2D16Async, cuMemsetD2D32, cuMemsetD2D32Async, cuMemsetD8, cuMemsetD8Async, cuMemsetD16, cuMemsetD16Async, cuMemsetD32, cuMemsetD32Async

CUresult cuMemcpy3DPeer (const CUDA_MEMCPY3D_PEER *pCopy)

Copies memory between contexts.

Parameters

pCopy
   - Parameters for the memory copy

Returns

CUDA_SUCCESS, CUDA_ERROR_DEINITIALIZED, CUDA_ERROR_NOT_INITIALIZED, CUDA_ERROR_INVALID_CONTEXT, CUDA_ERROR_INVALID_VALUE

Description

Perform a 3D memory copy according to the parameters specified in `pCopy`. See the definition of the CUDA_MEMCPY3D_PEER structure for documentation of its parameters.

> ▸ Note that this function may also return error codes from previous, asynchronous launches.
> ▸ This function exhibits synchronous behavior for most use cases.

See also:

cuMemcpyDtoD, cuMemcpyPeer, cuMemcpyDtoDAsync, cuMemcpyPeerAsync, cuMemcpy3DPeerAsync

CUresult cuMemcpy3DPeerAsync (const CUDA_MEMCPY3D_PEER *pCopy, CUstream hStream)

Copies memory between contexts asynchronously.

Parameters

pCopy
   - Parameters for the memory copy
hStream
   - Stream identifier

Returns

CUDA_SUCCESS, CUDA_ERROR_DEINITIALIZED, CUDA_ERROR_NOT_INITIALIZED, CUDA_ERROR_INVALID_CONTEXT, CUDA_ERROR_INVALID_VALUE

Description

Perform a 3D memory copy according to the parameters specified in `pCopy`. See the definition of the CUDA_MEMCPY3D_PEER structure for documentation of its parameters.

> ▸ Note that this function may also return error codes from previous, asynchronous launches.
> ▸ This function exhibits asynchronous behavior for most use cases.
> ▸ This function uses standard NULL stream semantics.

See also:

cuMemcpyDtoD, cuMemcpyPeer, cuMemcpyDtoDAsync, cuMemcpyPeerAsync, cuMemcpy3DPeerAsync

CUresult cuMemcpyAsync (CUdeviceptr dst, CUdeviceptr src, size_t ByteCount, CUstream hStream)

Copies memory asynchronously.

Parameters

dst
   - Destination unified virtual address space pointer
src
   - Source unified virtual address space pointer
ByteCount
   - Size of memory copy in bytes
hStream
   - Stream identifier

Returns

CUDA_SUCCESS, CUDA_ERROR_DEINITIALIZED, CUDA_ERROR_NOT_INITIALIZED, CUDA_ERROR_INVALID_CONTEXT, CUDA_ERROR_INVALID_VALUE

Description

Copies data between two pointers. `dst` and `src` are base pointers of the destination and source, respectively. `ByteCount` specifies the number of bytes to copy. Note that this function infers the type of the transfer (host to host, host to device, device to device, or device to host) from the pointer values. This function is only allowed in contexts which support unified addressing.

> ▸ Note that this function may also return error codes from previous, asynchronous launches.
> ▸ This function exhibits asynchronous behavior for most use cases.
> ▸ This function uses standard NULL stream semantics.

See also:

cuArray3DCreate, cuArray3DGetDescriptor, cuArrayCreate, cuArrayDestroy, cuArrayGetDescriptor, cuMemAlloc, cuMemAllocHost, cuMemAllocPitch, cuMemcpy2D, cuMemcpy2DAsync, cuMemcpy2DUnaligned, cuMemcpy3D,

cuMemcpy3DAsync, cuMemcpyAtoA, cuMemcpyAtoD, cuMemcpyAtoH,
cuMemcpyAtoHAsync, cuMemcpyDtoA, cuMemcpyDtoD, cuMemcpyDtoH,
cuMemcpyDtoHAsync, cuMemcpyHtoA, cuMemcpyHtoAAsync, cuMemcpyHtoD,
cuMemcpyHtoDAsync, cuMemFree, cuMemFreeHost, cuMemGetAddressRange,
cuMemGetInfo, cuMemHostAlloc, cuMemHostGetDevicePointer, cuMemsetD2D8,
cuMemsetD2D8Async, cuMemsetD2D16, cuMemsetD2D16Async, cuMemsetD2D32,
cuMemsetD2D32Async, cuMemsetD8, cuMemsetD8Async, cuMemsetD16,
cuMemsetD16Async, cuMemsetD32, cuMemsetD32Async

CUresult cuMemcpyAtoA (CUarray dstArray, size_t dstOffset, CUarray srcArray, size_t srcOffset, size_t ByteCount)

Copies memory from Array to Array.

Parameters

dstArray
   - Destination array
dstOffset
   - Offset in bytes of destination array
srcArray
   - Source array
srcOffset
   - Offset in bytes of source array
ByteCount
   - Size of memory copy in bytes

Returns

CUDA_SUCCESS, CUDA_ERROR_DEINITIALIZED,
CUDA_ERROR_NOT_INITIALIZED, CUDA_ERROR_INVALID_CONTEXT,
CUDA_ERROR_INVALID_VALUE

Description

Copies from one 1D CUDA array to another. `dstArray` and `srcArray` specify
the handles of the destination and source CUDA arrays for the copy, respectively.
`dstOffset` and `srcOffset` specify the destination and source offsets in bytes into the
CUDA arrays. `ByteCount` is the number of bytes to be copied. The size of the elements
in the CUDA arrays need not be the same format, but the elements must be the same
size; and count must be evenly divisible by that size.

> ▸ Note that this function may also return error codes from previous, asynchronous launches.
>
> ▸ This function exhibits synchronous behavior for most use cases.

See also:

cuArray3DCreate, cuArray3DGetDescriptor, cuArrayCreate, cuArrayDestroy, cuArrayGetDescriptor, cuMemAlloc, cuMemAllocHost, cuMemAllocPitch, cuMemcpy2D, cuMemcpy2DAsync, cuMemcpy2DUnaligned, cuMemcpy3D, cuMemcpy3DAsync, cuMemcpyAtoD, cuMemcpyAtoH, cuMemcpyAtoHAsync, cuMemcpyDtoA, cuMemcpyDtoD, cuMemcpyDtoDAsync, cuMemcpyDtoH, cuMemcpyDtoHAsync, cuMemcpyHtoA, cuMemcpyHtoAAsync, cuMemcpyHtoD, cuMemcpyHtoDAsync, cuMemFree, cuMemFreeHost, cuMemGetAddressRange, cuMemGetInfo, cuMemHostAlloc, cuMemHostGetDevicePointer, cuMemsetD2D8, cuMemsetD2D16, cuMemsetD2D32, cuMemsetD8, cuMemsetD16, cuMemsetD32

CUresult cuMemcpyAtoD (CUdeviceptr dstDevice, CUarray srcArray, size_t srcOffset, size_t ByteCount)

Copies memory from Array to Device.

Parameters

dstDevice
   - Destination device pointer
srcArray
   - Source array
srcOffset
   - Offset in bytes of source array
ByteCount
   - Size of memory copy in bytes

Returns

CUDA_SUCCESS, CUDA_ERROR_DEINITIALIZED, CUDA_ERROR_NOT_INITIALIZED, CUDA_ERROR_INVALID_CONTEXT, CUDA_ERROR_INVALID_VALUE

Description

Copies from one 1D CUDA array to device memory. `dstDevice` specifies the base pointer of the destination and must be naturally aligned with the CUDA array elements. `srcArray` and `srcOffset` specify the CUDA array handle and the offset in bytes into

the array where the copy is to begin. `ByteCount` specifies the number of bytes to copy and must be evenly divisible by the array element size.

> ▸ Note that this function may also return error codes from previous, asynchronous launches.
> ▸ This function exhibits synchronous behavior for most use cases.

See also:

cuArray3DCreate, cuArray3DGetDescriptor, cuArrayCreate, cuArrayDestroy, cuArrayGetDescriptor, cuMemAlloc, cuMemAllocHost, cuMemAllocPitch, cuMemcpy2D, cuMemcpy2DAsync, cuMemcpy2DUnaligned, cuMemcpy3D, cuMemcpy3DAsync, cuMemcpyAtoA, cuMemcpyAtoH, cuMemcpyAtoHAsync, cuMemcpyDtoA, cuMemcpyDtoD, cuMemcpyDtoDAsync, cuMemcpyDtoH, cuMemcpyDtoHAsync, cuMemcpyHtoA, cuMemcpyHtoAAsync, cuMemcpyHtoD, cuMemcpyHtoDAsync, cuMemFree, cuMemFreeHost, cuMemGetAddressRange, cuMemGetInfo, cuMemHostAlloc, cuMemHostGetDevicePointer, cuMemsetD2D8, cuMemsetD2D16, cuMemsetD2D32, cuMemsetD8, cuMemsetD16, cuMemsetD32

CUresult cuMemcpyAtoH (void *dstHost, CUarray srcArray, size_t srcOffset, size_t ByteCount)

Copies memory from Array to Host.

Parameters

dstHost
 - Destination device pointer
srcArray
 - Source array
srcOffset
 - Offset in bytes of source array
ByteCount
 - Size of memory copy in bytes

Returns

CUDA_SUCCESS, CUDA_ERROR_DEINITIALIZED, CUDA_ERROR_NOT_INITIALIZED, CUDA_ERROR_INVALID_CONTEXT, CUDA_ERROR_INVALID_VALUE

Description

Copies from one 1D CUDA array to host memory. `dstHost` specifies the base pointer of the destination. `srcArray` and `srcOffset` specify the CUDA array handle and starting offset in bytes of the source data. `ByteCount` specifies the number of bytes to copy.

> ▶ Note that this function may also return error codes from previous, asynchronous launches.
> ▶ This function exhibits synchronous behavior for most use cases.

See also:

cuArray3DCreate, cuArray3DGetDescriptor, cuArrayCreate, cuArrayDestroy, cuArrayGetDescriptor, cuMemAlloc, cuMemAllocHost, cuMemAllocPitch, cuMemcpy2D, cuMemcpy2DAsync, cuMemcpy2DUnaligned, cuMemcpy3D, cuMemcpy3DAsync, cuMemcpyAtoA, cuMemcpyAtoD, cuMemcpyAtoHAsync, cuMemcpyDtoA, cuMemcpyDtoD, cuMemcpyDtoDAsync, cuMemcpyDtoH, cuMemcpyDtoHAsync, cuMemcpyHtoA, cuMemcpyHtoAAsync, cuMemcpyHtoD, cuMemcpyHtoDAsync, cuMemFree, cuMemFreeHost, cuMemGetAddressRange, cuMemGetInfo, cuMemHostAlloc, cuMemHostGetDevicePointer, cuMemsetD2D8, cuMemsetD2D16, cuMemsetD2D32, cuMemsetD8, cuMemsetD16, cuMemsetD32

CUresult cuMemcpyAtoHAsync (void *dstHost, CUarray srcArray, size_t srcOffset, size_t ByteCount, CUstream hStream)

Copies memory from Array to Host.

Parameters

dstHost
    - Destination pointer
srcArray
    - Source array
srcOffset
    - Offset in bytes of source array
ByteCount
    - Size of memory copy in bytes
hStream
    - Stream identifier

Returns

CUDA_SUCCESS, CUDA_ERROR_DEINITIALIZED,
CUDA_ERROR_NOT_INITIALIZED, CUDA_ERROR_INVALID_CONTEXT,
CUDA_ERROR_INVALID_VALUE

Description

Copies from one 1D CUDA array to host memory. `dstHost` specifies the base pointer of the destination. `srcArray` and `srcOffset` specify the CUDA array handle and starting offset in bytes of the source data. `ByteCount` specifies the number of bytes to copy.

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

▸ This function exhibits asynchronous behavior for most use cases.

▸ This function uses standard NULL stream semantics.

See also:

cuArray3DCreate, cuArray3DGetDescriptor, cuArrayCreate, cuArrayDestroy, cuArrayGetDescriptor, cuMemAlloc, cuMemAllocHost, cuMemAllocPitch, cuMemcpy2D, cuMemcpy2DAsync, cuMemcpy2DUnaligned, cuMemcpy3D, cuMemcpy3DAsync, cuMemcpyAtoA, cuMemcpyAtoD, cuMemcpyAtoH, cuMemcpyDtoA, cuMemcpyDtoD, cuMemcpyDtoDAsync, cuMemcpyDtoH, cuMemcpyDtoHAsync, cuMemcpyHtoA, cuMemcpyHtoAAsync, cuMemcpyHtoD, cuMemcpyHtoDAsync, cuMemFree, cuMemFreeHost, cuMemGetAddressRange, cuMemGetInfo, cuMemHostAlloc, cuMemHostGetDevicePointer, cuMemsetD2D8, cuMemsetD2D8Async, cuMemsetD2D16, cuMemsetD2D16Async, cuMemsetD2D32, cuMemsetD2D32Async, cuMemsetD8, cuMemsetD8Async, cuMemsetD16, cuMemsetD16Async, cuMemsetD32, cuMemsetD32Async

CUresult cuMemcpyDtoA (CUarray dstArray, size_t dstOffset, CUdeviceptr srcDevice, size_t ByteCount)

Copies memory from Device to Array.

Parameters

dstArray
  - Destination array
dstOffset
  - Offset in bytes of destination array
srcDevice
  - Source device pointer

ByteCount
   - Size of memory copy in bytes

Returns

CUDA_SUCCESS, CUDA_ERROR_DEINITIALIZED,
CUDA_ERROR_NOT_INITIALIZED, CUDA_ERROR_INVALID_CONTEXT,
CUDA_ERROR_INVALID_VALUE

Description

Copies from device memory to a 1D CUDA array. `dstArray` and `dstOffset` specify
the CUDA array handle and starting index of the destination data. `srcDevice` specifies
the base pointer of the source. `ByteCount` specifies the number of bytes to copy.

> ▸ Note that this function may also return error codes from previous, asynchronous
>   launches.
> ▸ This function exhibits synchronous behavior for most use cases.

See also:

cuArray3DCreate, cuArray3DGetDescriptor, cuArrayCreate, cuArrayDestroy,
cuArrayGetDescriptor, cuMemAlloc, cuMemAllocHost, cuMemAllocPitch,
cuMemcpy2D, cuMemcpy2DAsync, cuMemcpy2DUnaligned, cuMemcpy3D,
cuMemcpy3DAsync, cuMemcpyAtoA, cuMemcpyAtoD, cuMemcpyAtoH,
cuMemcpyAtoHAsync, cuMemcpyDtoD, cuMemcpyDtoDAsync, cuMemcpyDtoH,
cuMemcpyDtoHAsync, cuMemcpyHtoA, cuMemcpyHtoAAsync, cuMemcpyHtoD,
cuMemcpyHtoDAsync, cuMemFree, cuMemFreeHost, cuMemGetAddressRange,
cuMemGetInfo, cuMemHostAlloc, cuMemHostGetDevicePointer, cuMemsetD2D8,
cuMemsetD2D16, cuMemsetD2D32, cuMemsetD8, cuMemsetD16, cuMemsetD32

CUresult cuMemcpyDtoD (CUdeviceptr dstDevice, CUdeviceptr srcDevice, size_t ByteCount)

Copies memory from Device to Device.

Parameters

dstDevice
   - Destination device pointer
srcDevice
   - Source device pointer
ByteCount
   - Size of memory copy in bytes

Returns

CUDA_SUCCESS, CUDA_ERROR_DEINITIALIZED, CUDA_ERROR_NOT_INITIALIZED, CUDA_ERROR_INVALID_CONTEXT, CUDA_ERROR_INVALID_VALUE

Description

Copies from device memory to device memory. `dstDevice` and `srcDevice` are the base pointers of the destination and source, respectively. `ByteCount` specifies the number of bytes to copy.

> ▸ Note that this function may also return error codes from previous, asynchronous launches.
> ▸ This function exhibits synchronous behavior for most use cases.

See also:

cuArray3DCreate, cuArray3DGetDescriptor, cuArrayCreate, cuArrayDestroy, cuArrayGetDescriptor, cuMemAlloc, cuMemAllocHost, cuMemAllocPitch, cuMemcpy2D, cuMemcpy2DAsync, cuMemcpy2DUnaligned, cuMemcpy3D, cuMemcpy3DAsync, cuMemcpyAtoA, cuMemcpyAtoD, cuMemcpyAtoH, cuMemcpyAtoHAsync, cuMemcpyDtoA, cuMemcpyDtoH, cuMemcpyDtoHAsync, cuMemcpyHtoA, cuMemcpyHtoAAsync, cuMemcpyHtoD, cuMemcpyHtoDAsync, cuMemFree, cuMemFreeHost, cuMemGetAddressRange, cuMemGetInfo, cuMemHostAlloc, cuMemHostGetDevicePointer, cuMemsetD2D8, cuMemsetD2D16, cuMemsetD2D32, cuMemsetD8, cuMemsetD16, cuMemsetD32

CUresult cuMemcpyDtoDAsync (CUdeviceptr dstDevice, CUdeviceptr srcDevice, size_t ByteCount, CUstream hStream)

Copies memory from Device to Device.

Parameters

dstDevice
    - Destination device pointer
srcDevice
    - Source device pointer
ByteCount
    - Size of memory copy in bytes
hStream
    - Stream identifier

Returns

CUDA_SUCCESS, CUDA_ERROR_DEINITIALIZED,
CUDA_ERROR_NOT_INITIALIZED, CUDA_ERROR_INVALID_CONTEXT,
CUDA_ERROR_INVALID_VALUE

Description

Copies from device memory to device memory. `dstDevice` and `srcDevice` are the base pointers of the destination and source, respectively. `ByteCount` specifies the number of bytes to copy.

> ▸ Note that this function may also return error codes from previous, asynchronous launches.
> ▸ This function exhibits asynchronous behavior for most use cases.
> ▸ This function uses standard NULL stream semantics.

See also:

cuArray3DCreate, cuArray3DGetDescriptor, cuArrayCreate, cuArrayDestroy, cuArrayGetDescriptor, cuMemAlloc, cuMemAllocHost, cuMemAllocPitch, cuMemcpy2D, cuMemcpy2DAsync, cuMemcpy2DUnaligned, cuMemcpy3D, cuMemcpy3DAsync, cuMemcpyAtoA, cuMemcpyAtoD, cuMemcpyAtoH, cuMemcpyAtoHAsync, cuMemcpyDtoA, cuMemcpyDtoD, cuMemcpyDtoH, cuMemcpyDtoHAsync, cuMemcpyHtoA, cuMemcpyHtoAAsync, cuMemcpyHtoD, cuMemcpyHtoDAsync, cuMemFree, cuMemFreeHost, cuMemGetAddressRange, cuMemGetInfo, cuMemHostAlloc, cuMemHostGetDevicePointer, cuMemsetD2D8, cuMemsetD2D8Async, cuMemsetD2D16, cuMemsetD2D16Async, cuMemsetD2D32, cuMemsetD2D32Async, cuMemsetD8, cuMemsetD8Async, cuMemsetD16, cuMemsetD16Async, cuMemsetD32, cuMemsetD32Async

CUresult cuMemcpyDtoH (void *dstHost, CUdeviceptr srcDevice, size_t ByteCount)

Copies memory from Device to Host.

Parameters

dstHost
    - Destination host pointer
srcDevice
    - Source device pointer
ByteCount
    - Size of memory copy in bytes

Returns

CUDA_SUCCESS, CUDA_ERROR_DEINITIALIZED,
CUDA_ERROR_NOT_INITIALIZED, CUDA_ERROR_INVALID_CONTEXT,
CUDA_ERROR_INVALID_VALUE

Description

Copies from device to host memory. `dstHost` and `srcDevice` specify the base pointers of the destination and source, respectively. `ByteCount` specifies the number of bytes to copy.

> ▸ Note that this function may also return error codes from previous, asynchronous launches.
>
> ▸ This function exhibits synchronous behavior for most use cases.

See also:

cuArray3DCreate, cuArray3DGetDescriptor, cuArrayCreate, cuArrayDestroy, cuArrayGetDescriptor, cuMemAlloc, cuMemAllocHost, cuMemAllocPitch, cuMemcpy2D, cuMemcpy2DAsync, cuMemcpy2DUnaligned, cuMemcpy3D, cuMemcpy3DAsync, cuMemcpyAtoA, cuMemcpyAtoD, cuMemcpyAtoH, cuMemcpyAtoHAsync, cuMemcpyDtoA, cuMemcpyDtoD, cuMemcpyDtoDAsync, cuMemcpyDtoHAsync, cuMemcpyHtoA, cuMemcpyHtoAAsync, cuMemcpyHtoD, cuMemcpyHtoDAsync, cuMemFree, cuMemFreeHost, cuMemGetAddressRange, cuMemGetInfo, cuMemHostAlloc, cuMemHostGetDevicePointer, cuMemsetD2D8, cuMemsetD2D16, cuMemsetD2D32, cuMemsetD8, cuMemsetD16, cuMemsetD32

CUresult cuMemcpyDtoHAsync (void *dstHost, CUdeviceptr srcDevice, size_t ByteCount, CUstream hStream)

Copies memory from Device to Host.

Parameters

dstHost
  - Destination host pointer
srcDevice
  - Source device pointer
ByteCount
  - Size of memory copy in bytes
hStream
  - Stream identifier

Returns

CUDA_SUCCESS, CUDA_ERROR_DEINITIALIZED, CUDA_ERROR_NOT_INITIALIZED, CUDA_ERROR_INVALID_CONTEXT, CUDA_ERROR_INVALID_VALUE

Description

Copies from device to host memory. `dstHost` and `srcDevice` specify the base pointers of the destination and source, respectively. `ByteCount` specifies the number of bytes to copy.

> ▸ Note that this function may also return error codes from previous, asynchronous launches.
> ▸ This function exhibits asynchronous behavior for most use cases.
> ▸ This function uses standard NULL stream semantics.

See also:

cuArray3DCreate, cuArray3DGetDescriptor, cuArrayCreate, cuArrayDestroy, cuArrayGetDescriptor, cuMemAlloc, cuMemAllocHost, cuMemAllocPitch, cuMemcpy2D, cuMemcpy2DAsync, cuMemcpy2DUnaligned, cuMemcpy3D, cuMemcpy3DAsync, cuMemcpyAtoA, cuMemcpyAtoD, cuMemcpyAtoH, cuMemcpyAtoHAsync, cuMemcpyDtoA, cuMemcpyDtoD, cuMemcpyDtoDAsync, cuMemcpyDtoH, cuMemcpyHtoA, cuMemcpyHtoAAsync, cuMemcpyHtoD, cuMemcpyHtoDAsync, cuMemFree, cuMemFreeHost, cuMemGetAddressRange, cuMemGetInfo, cuMemHostAlloc, cuMemHostGetDevicePointer, cuMemsetD2D8, cuMemsetD2D8Async, cuMemsetD2D16, cuMemsetD2D16Async, cuMemsetD2D32, cuMemsetD2D32Async, cuMemsetD8, cuMemsetD8Async, cuMemsetD16, cuMemsetD16Async, cuMemsetD32, cuMemsetD32Async

CUresult cuMemcpyHtoA (CUarray dstArray, size_t dstOffset, const void *srcHost, size_t ByteCount)

Copies memory from Host to Array.

Parameters

dstArray
   - Destination array
dstOffset
   - Offset in bytes of destination array
srcHost
   - Source host pointer

ByteCount
  - Size of memory copy in bytes

Returns

CUDA_SUCCESS, CUDA_ERROR_DEINITIALIZED,
CUDA_ERROR_NOT_INITIALIZED, CUDA_ERROR_INVALID_CONTEXT,
CUDA_ERROR_INVALID_VALUE

Description

Copies from host memory to a 1D CUDA array. `dstArray` and `dstOffset` specify the CUDA array handle and starting offset in bytes of the destination data. `pSrc` specifies the base address of the source. `ByteCount` specifies the number of bytes to copy.

> ▸ Note that this function may also return error codes from previous, asynchronous launches.
>
> ▸ This function exhibits synchronous behavior for most use cases.

See also:

cuArray3DCreate, cuArray3DGetDescriptor, cuArrayCreate, cuArrayDestroy, cuArrayGetDescriptor, cuMemAlloc, cuMemAllocHost, cuMemAllocPitch, cuMemcpy2D, cuMemcpy2DAsync, cuMemcpy2DUnaligned, cuMemcpy3D, cuMemcpy3DAsync, cuMemcpyAtoA, cuMemcpyAtoD, cuMemcpyAtoH, cuMemcpyAtoHAsync, cuMemcpyDtoA, cuMemcpyDtoD, cuMemcpyDtoDAsync, cuMemcpyDtoH, cuMemcpyDtoHAsync, cuMemcpyHtoAAsync, cuMemcpyHtoD, cuMemcpyHtoDAsync, cuMemFree, cuMemFreeHost, cuMemGetAddressRange, cuMemGetInfo, cuMemHostAlloc, cuMemHostGetDevicePointer, cuMemsetD2D8, cuMemsetD2D16, cuMemsetD2D32, cuMemsetD8, cuMemsetD16, cuMemsetD32

CUresult cuMemcpyHtoAAsync (CUarray dstArray, size_t dstOffset, const void *srcHost, size_t ByteCount, CUstream hStream)

Copies memory from Host to Array.

Parameters

dstArray
  - Destination array
dstOffset
  - Offset in bytes of destination array

srcHost
    - Source host pointer
ByteCount
    - Size of memory copy in bytes
hStream
    - Stream identifier

Returns

CUDA_SUCCESS, CUDA_ERROR_DEINITIALIZED,
CUDA_ERROR_NOT_INITIALIZED, CUDA_ERROR_INVALID_CONTEXT,
CUDA_ERROR_INVALID_VALUE

Description

Copies from host memory to a 1D CUDA array. `dstArray` and `dstOffset` specify
the CUDA array handle and starting offset in bytes of the destination data. `srcHost`
specifies the base address of the source. `ByteCount` specifies the number of bytes to
copy.

> ▸ Note that this function may also return error codes from previous, asynchronous
>   launches.
> ▸ This function exhibits asynchronous behavior for most use cases.
> ▸ This function uses standard NULL stream semantics.

See also:

cuArray3DCreate, cuArray3DGetDescriptor, cuArrayCreate, cuArrayDestroy,
cuArrayGetDescriptor, cuMemAlloc, cuMemAllocHost, cuMemAllocPitch,
cuMemcpy2D, cuMemcpy2DAsync, cuMemcpy2DUnaligned, cuMemcpy3D,
cuMemcpy3DAsync, cuMemcpyAtoA, cuMemcpyAtoD, cuMemcpyAtoH,
cuMemcpyAtoHAsync, cuMemcpyDtoA, cuMemcpyDtoD, cuMemcpyDtoDAsync,
cuMemcpyDtoH, cuMemcpyDtoHAsync, cuMemcpyHtoA, cuMemcpyHtoD,
cuMemcpyHtoDAsync, cuMemFree, cuMemFreeHost, cuMemGetAddressRange,
cuMemGetInfo, cuMemHostAlloc, cuMemHostGetDevicePointer, cuMemsetD2D8,
cuMemsetD2D8Async, cuMemsetD2D16, cuMemsetD2D16Async, cuMemsetD2D32,
cuMemsetD2D32Async, cuMemsetD8, cuMemsetD8Async, cuMemsetD16,
cuMemsetD16Async, cuMemsetD32, cuMemsetD32Async

CUresult cuMemcpyHtoD (CUdeviceptr dstDevice, const void *srcHost, size_t ByteCount)

Copies memory from Host to Device.

Parameters

dstDevice
    - Destination device pointer
srcHost
    - Source host pointer
ByteCount
    - Size of memory copy in bytes

Returns

CUDA_SUCCESS, CUDA_ERROR_DEINITIALIZED, CUDA_ERROR_NOT_INITIALIZED, CUDA_ERROR_INVALID_CONTEXT, CUDA_ERROR_INVALID_VALUE

Description

Copies from host memory to device memory. `dstDevice` and `srcHost` are the base addresses of the destination and source, respectively. `ByteCount` specifies the number of bytes to copy.

> ▸ Note that this function may also return error codes from previous, asynchronous launches.
> ▸ This function exhibits synchronous behavior for most use cases.

See also:

cuArray3DCreate, cuArray3DGetDescriptor, cuArrayCreate, cuArrayDestroy, cuArrayGetDescriptor, cuMemAlloc, cuMemAllocHost, cuMemAllocPitch, cuMemcpy2D, cuMemcpy2DAsync, cuMemcpy2DUnaligned, cuMemcpy3D, cuMemcpy3DAsync, cuMemcpyAtoA, cuMemcpyAtoD, cuMemcpyAtoH, cuMemcpyAtoHAsync, cuMemcpyDtoA, cuMemcpyDtoD, cuMemcpyDtoDAsync, cuMemcpyDtoH, cuMemcpyDtoHAsync, cuMemcpyHtoA, cuMemcpyHtoAAsync, cuMemcpyHtoDAsync, cuMemFree, cuMemFreeHost, cuMemGetAddressRange, cuMemGetInfo, cuMemHostAlloc, cuMemHostGetDevicePointer, cuMemsetD2D8, cuMemsetD2D16, cuMemsetD2D32, cuMemsetD8, cuMemsetD16, cuMemsetD32

CUresult cuMemcpyHtoDAsync (CUdeviceptr dstDevice, const void *srcHost, size_t ByteCount, CUstream hStream)

Copies memory from Host to Device.

Parameters

dstDevice
- Destination device pointer
srcHost
- Source host pointer
ByteCount
- Size of memory copy in bytes
hStream
- Stream identifier

Returns

CUDA_SUCCESS, CUDA_ERROR_DEINITIALIZED, CUDA_ERROR_NOT_INITIALIZED, CUDA_ERROR_INVALID_CONTEXT, CUDA_ERROR_INVALID_VALUE

Description

Copies from host memory to device memory. `dstDevice` and `srcHost` are the base addresses of the destination and source, respectively. `ByteCount` specifies the number of bytes to copy.

> ▶ Note that this function may also return error codes from previous, asynchronous launches.
> ▶ This function exhibits asynchronous behavior for most use cases.
> ▶ This function uses standard NULL stream semantics.

See also:

cuArray3DCreate, cuArray3DGetDescriptor, cuArrayCreate, cuArrayDestroy, cuArrayGetDescriptor, cuMemAlloc, cuMemAllocHost, cuMemAllocPitch, cuMemcpy2D, cuMemcpy2DAsync, cuMemcpy2DUnaligned, cuMemcpy3D, cuMemcpy3DAsync, cuMemcpyAtoA, cuMemcpyAtoD, cuMemcpyAtoH, cuMemcpyAtoHAsync, cuMemcpyDtoA, cuMemcpyDtoD, cuMemcpyDtoDAsync, cuMemcpyDtoH, cuMemcpyDtoHAsync, cuMemcpyHtoA, cuMemcpyHtoAAsync, cuMemcpyHtoD, cuMemFree, cuMemFreeHost, cuMemGetAddressRange,

cuMemGetInfo, cuMemHostAlloc, cuMemHostGetDevicePointer, cuMemsetD2D8, cuMemsetD2D8Async, cuMemsetD2D16, cuMemsetD2D16Async, cuMemsetD2D32, cuMemsetD2D32Async, cuMemsetD8, cuMemsetD8Async, cuMemsetD16, cuMemsetD16Async, cuMemsetD32, cuMemsetD32Async

CUresult cuMemcpyPeer (CUdeviceptr dstDevice, CUcontext dstContext, CUdeviceptr srcDevice, CUcontext srcContext, size_t ByteCount)

Copies device memory between two contexts.

Parameters

dstDevice
   - Destination device pointer
dstContext
   - Destination context
srcDevice
   - Source device pointer
srcContext
   - Source context
ByteCount
   - Size of memory copy in bytes

Returns

CUDA_SUCCESS, CUDA_ERROR_DEINITIALIZED, CUDA_ERROR_NOT_INITIALIZED, CUDA_ERROR_INVALID_CONTEXT, CUDA_ERROR_INVALID_VALUE

Description

Copies from device memory in one context to device memory in another context. `dstDevice` is the base device pointer of the destination memory and `dstContext` is the destination context. `srcDevice` is the base device pointer of the source memory and `srcContext` is the source pointer. `ByteCount` specifies the number of bytes to copy.

> ‣ Note that this function may also return error codes from previous, asynchronous launches.
> ‣ This function exhibits synchronous behavior for most use cases.

See also:

cuMemcpyDtoD, cuMemcpy3DPeer, cuMemcpyDtoDAsync, cuMemcpyPeerAsync, cuMemcpy3DPeerAsync

CUresult cuMemcpyPeerAsync (CUdeviceptr dstDevice, CUcontext dstContext, CUdeviceptr srcDevice, CUcontext srcContext, size_t ByteCount, CUstream hStream)

Copies device memory between two contexts asynchronously.

Parameters

dstDevice
    - Destination device pointer
dstContext
    - Destination context
srcDevice
    - Source device pointer
srcContext
    - Source context
ByteCount
    - Size of memory copy in bytes
hStream
    - Stream identifier

Returns

CUDA_SUCCESS, CUDA_ERROR_DEINITIALIZED, CUDA_ERROR_NOT_INITIALIZED, CUDA_ERROR_INVALID_CONTEXT, CUDA_ERROR_INVALID_VALUE

Description

Copies from device memory in one context to device memory in another context. `dstDevice` is the base device pointer of the destination memory and `dstContext` is the destination context. `srcDevice` is the base device pointer of the source memory and `srcContext` is the source pointer. `ByteCount` specifies the number of bytes to copy.

> ▸ Note that this function may also return error codes from previous, asynchronous launches.
> ▸ This function exhibits asynchronous behavior for most use cases.
> ▸ This function uses standard NULL stream semantics.

See also:

cuMemcpyDtoD, cuMemcpyPeer, cuMemcpy3DPeer, cuMemcpyDtoDAsync, cuMemcpy3DPeerAsync

CUresult cuMemFree (CUdeviceptr dptr)

Frees device memory.

Parameters

dptr

   - Pointer to memory to free

Returns

CUDA_SUCCESS, CUDA_ERROR_DEINITIALIZED, CUDA_ERROR_NOT_INITIALIZED, CUDA_ERROR_INVALID_CONTEXT, CUDA_ERROR_INVALID_VALUE

Description

Frees the memory space pointed to by `dptr`, which must have been returned by a previous call to cuMemAlloc() or cuMemAllocPitch().

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

See also:

cuArray3DCreate, cuArray3DGetDescriptor, cuArrayCreate, cuArrayDestroy, cuArrayGetDescriptor, cuMemAlloc, cuMemAllocHost, cuMemAllocPitch, cuMemcpy2D, cuMemcpy2DAsync, cuMemcpy2DUnaligned, cuMemcpy3D, cuMemcpy3DAsync, cuMemcpyAtoA, cuMemcpyAtoD, cuMemcpyAtoH, cuMemcpyAtoHAsync, cuMemcpyDtoA, cuMemcpyDtoD, cuMemcpyDtoDAsync, cuMemcpyDtoH, cuMemcpyDtoHAsync, cuMemcpyHtoA, cuMemcpyHtoAAsync, cuMemcpyHtoD, cuMemcpyHtoDAsync, cuMemFreeHost, cuMemGetAddressRange, cuMemGetInfo, cuMemHostAlloc, cuMemHostGetDevicePointer, cuMemsetD2D8, cuMemsetD2D16, cuMemsetD2D32, cuMemsetD8, cuMemsetD16, cuMemsetD32

CUresult cuMemFreeHost (void *p)

Frees page-locked host memory.

Parameters

p
   - Pointer to memory to free

Returns

CUDA_SUCCESS, CUDA_ERROR_DEINITIALIZED,
CUDA_ERROR_NOT_INITIALIZED, CUDA_ERROR_INVALID_CONTEXT,
CUDA_ERROR_INVALID_VALUE

Description

Frees the memory space pointed to by p, which must have been returned by a previous call to cuMemAllocHost().

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

See also:

cuArray3DCreate, cuArray3DGetDescriptor, cuArrayCreate, cuArrayDestroy, cuArrayGetDescriptor, cuMemAlloc, cuMemAllocHost, cuMemAllocPitch, cuMemcpy2D, cuMemcpy2DAsync, cuMemcpy2DUnaligned, cuMemcpy3D, cuMemcpy3DAsync, cuMemcpyAtoA, cuMemcpyAtoD, cuMemcpyAtoH, cuMemcpyAtoHAsync, cuMemcpyDtoA, cuMemcpyDtoD, cuMemcpyDtoDAsync, cuMemcpyDtoH, cuMemcpyDtoHAsync, cuMemcpyHtoA, cuMemcpyHtoAAsync, cuMemcpyHtoD, cuMemcpyHtoDAsync, cuMemFree, cuMemGetAddressRange, cuMemGetInfo, cuMemHostAlloc, cuMemHostGetDevicePointer, cuMemsetD2D8, cuMemsetD2D16, cuMemsetD2D32, cuMemsetD8, cuMemsetD16, cuMemsetD32

CUresult cuMemGetAddressRange (CUdeviceptr *pbase, size_t *psize, CUdeviceptr dptr)

Get information on memory allocations.

Parameters

pbase
   - Returned base address

psize
- Returned size of device memory allocation

dptr
- Device pointer to query

Returns

CUDA_SUCCESS, CUDA_ERROR_DEINITIALIZED,
CUDA_ERROR_NOT_INITIALIZED, CUDA_ERROR_INVALID_CONTEXT,
CUDA_ERROR_INVALID_VALUE

Description

Returns the base address in `*pbase` and size in `*psize` of the allocation by
cuMemAlloc() or cuMemAllocPitch() that contains the input pointer `dptr`. Both
parameters `pbase` and `psize` are optional. If one of them is NULL, it is ignored.

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

See also:

cuArray3DCreate, cuArray3DGetDescriptor, cuArrayCreate, cuArrayDestroy,
cuArrayGetDescriptor, cuMemAlloc, cuMemAllocHost, cuMemAllocPitch,
cuMemcpy2D, cuMemcpy2DAsync, cuMemcpy2DUnaligned, cuMemcpy3D,
cuMemcpy3DAsync, cuMemcpyAtoA, cuMemcpyAtoD, cuMemcpyAtoH,
cuMemcpyAtoHAsync, cuMemcpyDtoA, cuMemcpyDtoD, cuMemcpyDtoDAsync,
cuMemcpyDtoH, cuMemcpyDtoHAsync, cuMemcpyHtoA, cuMemcpyHtoAAsync,
cuMemcpyHtoD, cuMemcpyHtoDAsync, cuMemFree, cuMemFreeHost, cuMemGetInfo,
cuMemHostAlloc, cuMemHostGetDevicePointer, cuMemsetD2D8, cuMemsetD2D16,
cuMemsetD2D32, cuMemsetD8, cuMemsetD16, cuMemsetD32

CUresult cuMemGetInfo (size_t *free, size_t *total)

Gets free and total memory.

Parameters

free
- Returned free memory in bytes

total
- Returned total memory in bytes

Returns

CUDA_SUCCESS, CUDA_ERROR_DEINITIALIZED,
CUDA_ERROR_NOT_INITIALIZED, CUDA_ERROR_INVALID_CONTEXT,
CUDA_ERROR_INVALID_VALUE

Description

Returns in `*free` and `*total` respectively, the free and total amount of memory
available for allocation by the CUDA context, in bytes.

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

See also:

cuArray3DCreate, cuArray3DGetDescriptor, cuArrayCreate, cuArrayDestroy,
cuArrayGetDescriptor, cuMemAlloc, cuMemAllocHost, cuMemAllocPitch,
cuMemcpy2D, cuMemcpy2DAsync, cuMemcpy2DUnaligned, cuMemcpy3D,
cuMemcpy3DAsync, cuMemcpyAtoA, cuMemcpyAtoD, cuMemcpyAtoH,
cuMemcpyAtoHAsync, cuMemcpyDtoA, cuMemcpyDtoD, cuMemcpyDtoDAsync,
cuMemcpyDtoH, cuMemcpyDtoHAsync, cuMemcpyHtoA, cuMemcpyHtoAAsync,
cuMemcpyHtoD, cuMemcpyHtoDAsync, cuMemFree, cuMemFreeHost,
cuMemGetAddressRange, cuMemHostAlloc, cuMemHostGetDevicePointer,
cuMemsetD2D8, cuMemsetD2D16, cuMemsetD2D32, cuMemsetD8, cuMemsetD16,
cuMemsetD32

CUresult cuMemHostAlloc (void **pp, size_t bytesize, unsigned int Flags)

Allocates page-locked host memory.

Parameters

pp
   - Returned host pointer to page-locked memory
bytesize
   - Requested allocation size in bytes
Flags
   - Flags for allocation request

Returns

CUDA_SUCCESS, CUDA_ERROR_DEINITIALIZED,
CUDA_ERROR_NOT_INITIALIZED, CUDA_ERROR_INVALID_CONTEXT,
CUDA_ERROR_INVALID_VALUE, CUDA_ERROR_OUT_OF_MEMORY

Description

Allocates `bytesize` 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 cuMemcpyHtoD(). 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.

▶ CU_MEMHOSTALLOC_PORTABLE: The memory returned by this call will be considered as pinned memory by all CUDA contexts, not just the one that performed the allocation.

▶ CU_MEMHOSTALLOC_DEVICEMAP: Maps the allocation into the CUDA address space. The device pointer to the memory may be obtained by calling cuMemHostGetDevicePointer(). This feature is available only on GPUs with compute capability greater than or equal to 1.1.

▶ CU_MEMHOSTALLOC_WRITECOMBINED: 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 GPU 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.

The CUDA context must have been created with the CU_CTX_MAP_HOST flag in order for the CU_MEMHOSTALLOC_DEVICEMAP flag to have any effect.

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

The memory allocated by this function must be freed with cuMemFreeHost().

Note all host memory allocated using cuMemHostAlloc() will automatically be immediately accessible to all contexts on all devices which support unified addressing

(as may be queried using CU_DEVICE_ATTRIBUTE_UNIFIED_ADDRESSING). Unless the flag CU_MEMHOSTALLOC_WRITECOMBINED is specified, the device pointer that may be used to access this host memory from those contexts is always equal to the returned host pointer *pp. If the flag CU_MEMHOSTALLOC_WRITECOMBINED is specified, then the function cuMemHostGetDevicePointer() must be used to query the device pointer, even if the context supports unified addressing. See Unified Addressing for additional details.

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

See also:

cuArray3DCreate, cuArray3DGetDescriptor, cuArrayCreate, cuArrayDestroy, cuArrayGetDescriptor, cuMemAlloc, cuMemAllocHost, cuMemAllocPitch, cuMemcpy2D, cuMemcpy2DAsync, cuMemcpy2DUnaligned, cuMemcpy3D, cuMemcpy3DAsync, cuMemcpyAtoA, cuMemcpyAtoD, cuMemcpyAtoH, cuMemcpyAtoHAsync, cuMemcpyDtoA, cuMemcpyDtoD, cuMemcpyDtoDAsync, cuMemcpyDtoH, cuMemcpyDtoHAsync, cuMemcpyHtoA, cuMemcpyHtoAAsync, cuMemcpyHtoD, cuMemcpyHtoDAsync, cuMemFree, cuMemFreeHost, cuMemGetAddressRange, cuMemGetInfo, cuMemHostGetDevicePointer, cuMemsetD2D8, cuMemsetD2D16, cuMemsetD2D32, cuMemsetD8, cuMemsetD16, cuMemsetD32

CUresult cuMemHostGetDevicePointer (CUdeviceptr *pdptr, void *p, unsigned int Flags)

Passes back device pointer of mapped pinned memory.

Parameters

pdptr
   - Returned device pointer
p
   - Host pointer
Flags
   - Options (must be 0)

Returns

CUDA_SUCCESS, CUDA_ERROR_DEINITIALIZED, CUDA_ERROR_NOT_INITIALIZED, CUDA_ERROR_INVALID_CONTEXT, CUDA_ERROR_INVALID_VALUE

Description

Passes back the device pointer `pdptr` corresponding to the mapped, pinned host buffer `p` allocated by cuMemHostAlloc.

cuMemHostGetDevicePointer() will fail if the CU_MEMHOSTALLOC_DEVICEMAP flag was not specified at the time the memory was allocated, or if the function is called on a GPU that does not support mapped pinned memory.

`Flags` provides for future releases. For now, it must be set to 0.

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

See also:

cuArray3DCreate, cuArray3DGetDescriptor, cuArrayCreate, cuArrayDestroy, cuArrayGetDescriptor, cuMemAlloc, cuMemAllocHost, cuMemAllocPitch, cuMemcpy2D, cuMemcpy2DAsync, cuMemcpy2DUnaligned, cuMemcpy3D, cuMemcpy3DAsync, cuMemcpyAtoA, cuMemcpyAtoD, cuMemcpyAtoH, cuMemcpyAtoHAsync, cuMemcpyDtoA, cuMemcpyDtoD, cuMemcpyDtoDAsync, cuMemcpyDtoH, cuMemcpyDtoHAsync, cuMemcpyHtoA, cuMemcpyHtoAAsync, cuMemcpyHtoD, cuMemcpyHtoDAsync, cuMemFree, cuMemFreeHost, cuMemGetAddressRange, cuMemGetInfo, cuMemHostAlloc, cuMemsetD2D8, cuMemsetD2D16, cuMemsetD2D32, cuMemsetD8, cuMemsetD16, cuMemsetD32

CUresult cuMemHostGetFlags (unsigned int *pFlags, void *p)

Passes back flags that were used for a pinned allocation.

Parameters

pFlags
   - Returned flags word
p
   - Host pointer

Returns

CUDA_SUCCESS, CUDA_ERROR_DEINITIALIZED, CUDA_ERROR_NOT_INITIALIZED, CUDA_ERROR_INVALID_CONTEXT, CUDA_ERROR_INVALID_VALUE

Description

Passes back the flags `pFlags` that were specified when allocating the pinned host buffer `p` allocated by cuMemHostAlloc.

cuMemHostGetFlags() will fail if the pointer does not reside in an allocation performed by cuMemAllocHost() or cuMemHostAlloc().

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

See also:

cuMemAllocHost, cuMemHostAlloc

CUresult cuMemHostRegister (void *p, size_t bytesize, unsigned int Flags)

Registers an existing host memory range for use by CUDA.

Parameters

p
  - Host pointer to memory to page-lock
bytesize
  - Size in bytes of the address range to page-lock
Flags
  - Flags for allocation request

Returns

CUDA_SUCCESS, CUDA_ERROR_DEINITIALIZED,
CUDA_ERROR_NOT_INITIALIZED, CUDA_ERROR_INVALID_CONTEXT,
CUDA_ERROR_INVALID_VALUE, CUDA_ERROR_OUT_OF_MEMORY,
CUDA_ERROR_HOST_MEMORY_ALREADY_REGISTERED

Description

Page-locks the memory range specified by `p` and `bytesize` and maps it for the device(s) as specified by `Flags`. This memory range also is added to the same tracking mechanism as cuMemHostAlloc to automatically accelerate calls to functions such as cuMemcpyHtoD(). 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.

This function has limited support on Mac OS X. OS 10.7 or higher is required.

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

▸ CU_MEMHOSTREGISTER_PORTABLE: The memory returned by this call will be considered as pinned memory by all CUDA contexts, not just the one that performed the allocation.

▸ CU_MEMHOSTREGISTER_DEVICEMAP: Maps the allocation into the CUDA address space. The device pointer to the memory may be obtained by calling cuMemHostGetDevicePointer(). This feature is available only on GPUs with compute capability greater than or equal to 1.1.

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 CU_CTX_MAP_HOST flag in order for the CU_MEMHOSTREGISTER_DEVICEMAP flag to have any effect.

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

The memory page-locked by this function must be unregistered with cuMemHostUnregister().

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

See also:

cuMemHostUnregister, cuMemHostGetFlags, cuMemHostGetDevicePointer

CUresult cuMemHostUnregister (void *p)

Unregisters a memory range that was registered with cuMemHostRegister.

Parameters

p
    - Host pointer to memory to unregister

Returns

CUDA_SUCCESS, CUDA_ERROR_DEINITIALIZED,
CUDA_ERROR_NOT_INITIALIZED, CUDA_ERROR_INVALID_CONTEXT,
CUDA_ERROR_INVALID_VALUE, CUDA_ERROR_OUT_OF_MEMORY,
CUDA_ERROR_HOST_MEMORY_NOT_REGISTERED,

Description

Unmaps the memory range whose base address is specified by p, and makes it pageable again.

The base address must be the same one specified to cuMemHostRegister().

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

See also:

cuMemHostRegister

CUresult cuMemsetD16 (CUdeviceptr dstDevice, unsigned short us, size_t N)

Initializes device memory.

Parameters

dstDevice
  - Destination device pointer
us
  - Value to set
N
  - Number of elements

Returns

CUDA_SUCCESS, CUDA_ERROR_DEINITIALIZED,
CUDA_ERROR_NOT_INITIALIZED, CUDA_ERROR_INVALID_CONTEXT,
CUDA_ERROR_INVALID_VALUE

Description

Sets the memory range of N 16-bit values to the specified value us. The dstDevice pointer must be two byte aligned.

> ▸ Note that this function may also return error codes from previous, asynchronous launches.
> ▸ See also memset synchronization details.

See also:

cuArray3DCreate, cuArray3DGetDescriptor, cuArrayCreate, cuArrayDestroy, cuArrayGetDescriptor, cuMemAlloc, cuMemAllocHost, cuMemAllocPitch, cuMemcpy2D, cuMemcpy2DAsync, cuMemcpy2DUnaligned, cuMemcpy3D, cuMemcpy3DAsync, cuMemcpyAtoA, cuMemcpyAtoD, cuMemcpyAtoH, cuMemcpyAtoHAsync, cuMemcpyDtoA, cuMemcpyDtoD, cuMemcpyDtoDAsync, cuMemcpyDtoH, cuMemcpyDtoHAsync, cuMemcpyHtoA, cuMemcpyHtoAAsync, cuMemcpyHtoD, cuMemcpyHtoDAsync, cuMemFree, cuMemFreeHost, cuMemGetAddressRange, cuMemGetInfo, cuMemHostAlloc, cuMemHostGetDevicePointer, cuMemsetD2D8, cuMemsetD2D8Async, cuMemsetD2D16, cuMemsetD2D16Async, cuMemsetD2D32, cuMemsetD2D32Async, cuMemsetD8, cuMemsetD8Async, cuMemsetD16Async, cuMemsetD32, cuMemsetD32Async

CUresult cuMemsetD16Async (CUdeviceptr dstDevice, unsigned short us, size_t N, CUstream hStream)

Sets device memory.

Parameters

dstDevice
    - Destination device pointer
us
    - Value to set
N
    - Number of elements
hStream
    - Stream identifier

Returns

CUDA_SUCCESS, CUDA_ERROR_DEINITIALIZED, CUDA_ERROR_NOT_INITIALIZED, CUDA_ERROR_INVALID_CONTEXT, CUDA_ERROR_INVALID_VALUE

Description

Sets the memory range of `N` 16-bit values to the specified value `us`. The `dstDevice` pointer must be two byte aligned.

> ▸ Note that this function may also return error codes from previous, asynchronous launches.
> ▸ See also memset synchronization details.
> ▸ This function uses standard NULL stream semantics.

See also:

cuArray3DCreate, cuArray3DGetDescriptor, cuArrayCreate, cuArrayDestroy, cuArrayGetDescriptor, cuMemAlloc, cuMemAllocHost, cuMemAllocPitch, cuMemcpy2D, cuMemcpy2DAsync, cuMemcpy2DUnaligned, cuMemcpy3D, cuMemcpy3DAsync, cuMemcpyAtoA, cuMemcpyAtoD, cuMemcpyAtoH, cuMemcpyAtoHAsync, cuMemcpyDtoA, cuMemcpyDtoD, cuMemcpyDtoDAsync, cuMemcpyDtoH, cuMemcpyDtoHAsync, cuMemcpyHtoA, cuMemcpyHtoAAsync, cuMemcpyHtoD, cuMemcpyHtoDAsync, cuMemFree, cuMemFreeHost, cuMemGetAddressRange, cuMemGetInfo, cuMemHostAlloc, cuMemHostGetDevicePointer, cuMemsetD2D8, cuMemsetD2D8Async, cuMemsetD2D16, cuMemsetD2D16Async, cuMemsetD2D32, cuMemsetD2D32Async, cuMemsetD8, cuMemsetD8Async, cuMemsetD16, cuMemsetD32, cuMemsetD32Async

CUresult cuMemsetD2D16 (CUdeviceptr dstDevice, size_t dstPitch, unsigned short us, size_t Width, size_t Height)

Initializes device memory.

Parameters

dstDevice
   - Destination device pointer
dstPitch
   - Pitch of destination device pointer
us
   - Value to set
Width
   - Width of row
Height
   - Number of rows

Returns

CUDA_SUCCESS, CUDA_ERROR_DEINITIALIZED,
CUDA_ERROR_NOT_INITIALIZED, CUDA_ERROR_INVALID_CONTEXT,
CUDA_ERROR_INVALID_VALUE

Description

Sets the 2D memory range of `Width` 16-bit values to the specified value `us`. `Height` specifies the number of rows to set, and `dstPitch` specifies the number of bytes between each row. The `dstDevice` pointer and `dstPitch` offset must be two byte aligned. This function performs fastest when the pitch is one that has been passed back by cuMemAllocPitch().

> ▸ Note that this function may also return error codes from previous, asynchronous launches.
> ▸ See also memset synchronization details.

See also:

cuArray3DCreate, cuArray3DGetDescriptor, cuArrayCreate, cuArrayDestroy, cuArrayGetDescriptor, cuMemAlloc, cuMemAllocHost, cuMemAllocPitch, cuMemcpy2D, cuMemcpy2DAsync, cuMemcpy2DUnaligned, cuMemcpy3D, cuMemcpy3DAsync, cuMemcpyAtoA, cuMemcpyAtoD, cuMemcpyAtoH, cuMemcpyAtoHAsync, cuMemcpyDtoA, cuMemcpyDtoD, cuMemcpyDtoDAsync, cuMemcpyDtoH, cuMemcpyDtoHAsync, cuMemcpyHtoA, cuMemcpyHtoAAsync, cuMemcpyHtoD, cuMemcpyHtoDAsync, cuMemFree, cuMemFreeHost, cuMemGetAddressRange, cuMemGetInfo, cuMemHostAlloc, cuMemHostGetDevicePointer, cuMemsetD2D8, cuMemsetD2D8Async, cuMemsetD2D16Async, cuMemsetD2D32, cuMemsetD2D32Async, cuMemsetD8, cuMemsetD8Async, cuMemsetD16, cuMemsetD16Async, cuMemsetD32, cuMemsetD32Async

CUresult cuMemsetD2D16Async (CUdeviceptr dstDevice, size_t dstPitch, unsigned short us, size_t Width, size_t Height, CUstream hStream)

Sets device memory.

Parameters

dstDevice
   - Destination device pointer

dstPitch
- Pitch of destination device pointer
us
- Value to set
Width
- Width of row
Height
- Number of rows
hStream
- Stream identifier

Returns

CUDA_SUCCESS, CUDA_ERROR_DEINITIALIZED,
CUDA_ERROR_NOT_INITIALIZED, CUDA_ERROR_INVALID_CONTEXT,
CUDA_ERROR_INVALID_VALUE

Description

Sets the 2D memory range of `Width` 16-bit values to the specified value `us`. `Height` specifies the number of rows to set, and `dstPitch` specifies the number of bytes between each row. The `dstDevice` pointer and `dstPitch` offset must be two byte aligned. This function performs fastest when the pitch is one that has been passed back by cuMemAllocPitch().

> ▸ Note that this function may also return error codes from previous, asynchronous launches.
> ▸ See also memset synchronization details.
> ▸ This function uses standard NULL stream semantics.

See also:

cuArray3DCreate, cuArray3DGetDescriptor, cuArrayCreate, cuArrayDestroy, cuArrayGetDescriptor, cuMemAlloc, cuMemAllocHost, cuMemAllocPitch, cuMemcpy2D, cuMemcpy2DAsync, cuMemcpy2DUnaligned, cuMemcpy3D, cuMemcpy3DAsync, cuMemcpyAtoA, cuMemcpyAtoD, cuMemcpyAtoH, cuMemcpyAtoHAsync, cuMemcpyDtoA, cuMemcpyDtoD, cuMemcpyDtoDAsync, cuMemcpyDtoH, cuMemcpyDtoHAsync, cuMemcpyHtoA, cuMemcpyHtoAAsync, cuMemcpyHtoD, cuMemcpyHtoDAsync, cuMemFree, cuMemFreeHost, cuMemGetAddressRange, cuMemGetInfo, cuMemHostAlloc, cuMemHostGetDevicePointer, cuMemsetD2D8, cuMemsetD2D8Async, cuMemsetD2D16, cuMemsetD2D32, cuMemsetD2D32Async, cuMemsetD8, cuMemsetD8Async, cuMemsetD16, cuMemsetD16Async, cuMemsetD32, cuMemsetD32Async

CUresult cuMemsetD2D32 (CUdeviceptr dstDevice, size_t dstPitch, unsigned int ui, size_t Width, size_t Height)

Initializes device memory.

Parameters

dstDevice
   - Destination device pointer
dstPitch
   - Pitch of destination device pointer
ui
   - Value to set
Width
   - Width of row
Height
   - Number of rows

Returns

CUDA_SUCCESS, CUDA_ERROR_DEINITIALIZED,
CUDA_ERROR_NOT_INITIALIZED, CUDA_ERROR_INVALID_CONTEXT,
CUDA_ERROR_INVALID_VALUE

Description

Sets the 2D memory range of `Width` 32-bit values to the specified value `ui`. `Height` specifies the number of rows to set, and `dstPitch` specifies the number of bytes between each row. The `dstDevice` pointer and `dstPitch` offset must be four byte aligned. This function performs fastest when the pitch is one that has been passed back by cuMemAllocPitch().

> ▸ Note that this function may also return error codes from previous, asynchronous launches.
> ▸ See also memset synchronization details.

See also:

cuArray3DCreate, cuArray3DGetDescriptor, cuArrayCreate, cuArrayDestroy, cuArrayGetDescriptor, cuMemAlloc, cuMemAllocHost, cuMemAllocPitch, cuMemcpy2D, cuMemcpy2DAsync, cuMemcpy2DUnaligned, cuMemcpy3D, cuMemcpy3DAsync, cuMemcpyAtoA, cuMemcpyAtoD,

cuMemcpyAtoH, cuMemcpyAtoHAsync, cuMemcpyDtoA, cuMemcpyDtoD, cuMemcpyDtoDAsync, cuMemcpyDtoH, cuMemcpyDtoHAsync, cuMemcpyHtoA, cuMemcpyHtoAAsync, cuMemcpyHtoD, cuMemcpyHtoDAsync, cuMemFree, cuMemFreeHost, cuMemGetAddressRange, cuMemGetInfo, cuMemHostAlloc, cuMemHostGetDevicePointer, cuMemsetD2D8, cuMemsetD2D8Async, cuMemsetD2D16, cuMemsetD2D16Async, cuMemsetD2D32Async, cuMemsetD8, cuMemsetD8Async, cuMemsetD16, cuMemsetD16Async, cuMemsetD32, cuMemsetD32Async

CUresult cuMemsetD2D32Async (CUdeviceptr dstDevice, size_t dstPitch, unsigned int ui, size_t Width, size_t Height, CUstream hStream)

Sets device memory.

Parameters

dstDevice
  - Destination device pointer
dstPitch
  - Pitch of destination device pointer
ui
  - Value to set
Width
  - Width of row
Height
  - Number of rows
hStream
  - Stream identifier

Returns

CUDA_SUCCESS, CUDA_ERROR_DEINITIALIZED, CUDA_ERROR_NOT_INITIALIZED, CUDA_ERROR_INVALID_CONTEXT, CUDA_ERROR_INVALID_VALUE

Description

Sets the 2D memory range of `Width` 32-bit values to the specified value `ui`. `Height` specifies the number of rows to set, and `dstPitch` specifies the number of bytes between each row. The `dstDevice` pointer and `dstPitch` offset must be four byte aligned. This function performs fastest when the pitch is one that has been passed back by cuMemAllocPitch().

> ▸ Note that this function may also return error codes from previous, asynchronous launches.
> ▸ See also memset synchronization details.
> ▸ This function uses standard NULL stream semantics.

See also:

cuArray3DCreate, cuArray3DGetDescriptor, cuArrayCreate, cuArrayDestroy, cuArrayGetDescriptor, cuMemAlloc, cuMemAllocHost, cuMemAllocPitch, cuMemcpy2D, cuMemcpy2DAsync, cuMemcpy2DUnaligned, cuMemcpy3D, cuMemcpy3DAsync, cuMemcpyAtoA, cuMemcpyAtoD, cuMemcpyAtoH, cuMemcpyAtoHAsync, cuMemcpyDtoA, cuMemcpyDtoD, cuMemcpyDtoDAsync, cuMemcpyDtoH, cuMemcpyDtoHAsync, cuMemcpyHtoA, cuMemcpyHtoAAsync, cuMemcpyHtoD, cuMemcpyHtoDAsync, cuMemFree, cuMemFreeHost, cuMemGetAddressRange, cuMemGetInfo, cuMemHostAlloc, cuMemHostGetDevicePointer, cuMemsetD2D8, cuMemsetD2D8Async, cuMemsetD2D16, cuMemsetD2D16Async, cuMemsetD2D32, cuMemsetD8, cuMemsetD8Async, cuMemsetD16, cuMemsetD16Async, cuMemsetD32, cuMemsetD32Async

CUresult cuMemsetD2D8 (CUdeviceptr dstDevice, size_t dstPitch, unsigned char uc, size_t Width, size_t Height)

Initializes device memory.

Parameters

dstDevice
   - Destination device pointer
dstPitch
   - Pitch of destination device pointer
uc
   - Value to set
Width
   - Width of row
Height
   - Number of rows

Returns

CUDA_SUCCESS, CUDA_ERROR_DEINITIALIZED, CUDA_ERROR_NOT_INITIALIZED, CUDA_ERROR_INVALID_CONTEXT, CUDA_ERROR_INVALID_VALUE

Description

Sets the 2D memory range of `Width` 8-bit values to the specified value `uc`. `Height` specifies the number of rows to set, and `dstPitch` specifies the number of bytes between each row. This function performs fastest when the pitch is one that has been passed back by cuMemAllocPitch().

> ▸ Note that this function may also return error codes from previous, asynchronous launches.
>
> ▸ See also memset synchronization details.

See also:

cuArray3DCreate, cuArray3DGetDescriptor, cuArrayCreate, cuArrayDestroy, cuArrayGetDescriptor, cuMemAlloc, cuMemAllocHost, cuMemAllocPitch, cuMemcpy2D, cuMemcpy2DAsync, cuMemcpy2DUnaligned, cuMemcpy3D, cuMemcpy3DAsync, cuMemcpyAtoA, cuMemcpyAtoD, cuMemcpyAtoH, cuMemcpyAtoHAsync, cuMemcpyDtoA, cuMemcpyDtoD, cuMemcpyDtoDAsync, cuMemcpyDtoH, cuMemcpyDtoHAsync, cuMemcpyHtoA, cuMemcpyHtoAAsync, cuMemcpyHtoD, cuMemcpyHtoDAsync, cuMemFree, cuMemFreeHost, cuMemGetAddressRange, cuMemGetInfo, cuMemHostAlloc, cuMemHostGetDevicePointer, cuMemsetD2D8Async, cuMemsetD2D16, cuMemsetD2D16Async, cuMemsetD2D32, cuMemsetD2D32Async, cuMemsetD8, cuMemsetD8Async, cuMemsetD16, cuMemsetD16Async, cuMemsetD32, cuMemsetD32Async

CUresult cuMemsetD2D8Async (CUdeviceptr dstDevice, size_t dstPitch, unsigned char uc, size_t Width, size_t Height, CUstream hStream)

Sets device memory.

Parameters

dstDevice
  - Destination device pointer
dstPitch
  - Pitch of destination device pointer
uc
  - Value to set
Width
  - Width of row

Height
- Number of rows

hStream
- Stream identifier

Returns

CUDA_SUCCESS, CUDA_ERROR_DEINITIALIZED,
CUDA_ERROR_NOT_INITIALIZED, CUDA_ERROR_INVALID_CONTEXT,
CUDA_ERROR_INVALID_VALUE

Description

Sets the 2D memory range of `Width` 8-bit values to the specified value `uc`. `Height`
specifies the number of rows to set, and `dstPitch` specifies the number of bytes
between each row. This function performs fastest when the pitch is one that has been
passed back by cuMemAllocPitch().

> ▸ Note that this function may also return error codes from previous, asynchronous
>   launches.
> ▸ See also memset synchronization details.
> ▸ This function uses standard NULL stream semantics.

See also:

cuArray3DCreate, cuArray3DGetDescriptor, cuArrayCreate, cuArrayDestroy,
cuArrayGetDescriptor, cuMemAlloc, cuMemAllocHost, cuMemAllocPitch,
cuMemcpy2D, cuMemcpy2DAsync, cuMemcpy2DUnaligned,
cuMemcpy3D, cuMemcpy3DAsync, cuMemcpyAtoA, cuMemcpyAtoD,
cuMemcpyAtoH, cuMemcpyAtoHAsync, cuMemcpyDtoA, cuMemcpyDtoD,
cuMemcpyDtoDAsync, cuMemcpyDtoH, cuMemcpyDtoHAsync, cuMemcpyHtoA,
cuMemcpyHtoAAsync, cuMemcpyHtoD, cuMemcpyHtoDAsync, cuMemFree,
cuMemFreeHost, cuMemGetAddressRange, cuMemGetInfo, cuMemHostAlloc,
cuMemHostGetDevicePointer, cuMemsetD2D8, cuMemsetD2D16,
cuMemsetD2D16Async, cuMemsetD2D32, cuMemsetD2D32Async, cuMemsetD8,
cuMemsetD8Async, cuMemsetD16, cuMemsetD16Async, cuMemsetD32,
cuMemsetD32Async

CUresult cuMemsetD32 (CUdeviceptr dstDevice, unsigned int ui, size_t N)

Initializes device memory.

Parameters

dstDevice
   - Destination device pointer
ui
   - Value to set
N
   - Number of elements

Returns

CUDA_SUCCESS, CUDA_ERROR_DEINITIALIZED, CUDA_ERROR_NOT_INITIALIZED, CUDA_ERROR_INVALID_CONTEXT, CUDA_ERROR_INVALID_VALUE

Description

Sets the memory range of N 32-bit values to the specified value ui. The dstDevice pointer must be four byte aligned.

> ‣ Note that this function may also return error codes from previous, asynchronous launches.
> ‣ See also memset synchronization details.

See also:

cuArray3DCreate, cuArray3DGetDescriptor, cuArrayCreate, cuArrayDestroy, cuArrayGetDescriptor, cuMemAlloc, cuMemAllocHost, cuMemAllocPitch, cuMemcpy2D, cuMemcpy2DAsync, cuMemcpy2DUnaligned, cuMemcpy3D, cuMemcpy3DAsync, cuMemcpyAtoA, cuMemcpyAtoD, cuMemcpyAtoH, cuMemcpyAtoHAsync, cuMemcpyDtoA, cuMemcpyDtoD, cuMemcpyDtoDAsync, cuMemcpyDtoH, cuMemcpyDtoHAsync, cuMemcpyHtoA, cuMemcpyHtoAAsync, cuMemcpyHtoD, cuMemcpyHtoDAsync, cuMemFree, cuMemFreeHost, cuMemGetAddressRange, cuMemGetInfo, cuMemHostAlloc, cuMemHostGetDevicePointer, cuMemsetD2D8, cuMemsetD2D8Async, cuMemsetD2D16, cuMemsetD2D16Async, cuMemsetD2D32, cuMemsetD2D32Async, cuMemsetD8, cuMemsetD8Async, cuMemsetD16, cuMemsetD16Async, cuMemsetD32Async

CUresult cuMemsetD32Async (CUdeviceptr dstDevice, unsigned int ui, size_t N, CUstream hStream)

Sets device memory.

Parameters

dstDevice
   - Destination device pointer
ui
   - Value to set
N
   - Number of elements
hStream
   - Stream identifier

Returns

CUDA_SUCCESS, CUDA_ERROR_DEINITIALIZED,
CUDA_ERROR_NOT_INITIALIZED, CUDA_ERROR_INVALID_CONTEXT,
CUDA_ERROR_INVALID_VALUE

Description

Sets the memory range of N 32-bit values to the specified value ui. The dstDevice pointer must be four byte aligned.

> ‣ Note that this function may also return error codes from previous, asynchronous launches.
> ‣ See also memset synchronization details.
> ‣ This function uses standard NULL stream semantics.

See also:

cuArray3DCreate, cuArray3DGetDescriptor, cuArrayCreate, cuArrayDestroy, cuArrayGetDescriptor, cuMemAlloc, cuMemAllocHost, cuMemAllocPitch, cuMemcpy2D, cuMemcpy2DAsync, cuMemcpy2DUnaligned, cuMemcpy3D, cuMemcpy3DAsync, cuMemcpyAtoA, cuMemcpyAtoD, cuMemcpyAtoH, cuMemcpyAtoHAsync, cuMemcpyDtoA, cuMemcpyDtoD, cuMemcpyDtoDAsync, cuMemcpyDtoH, cuMemcpyDtoHAsync, cuMemcpyHtoA, cuMemcpyHtoAAsync, cuMemcpyHtoD, cuMemcpyHtoDAsync, cuMemFree, cuMemFreeHost, cuMemGetAddressRange, cuMemGetInfo, cuMemHostAlloc, cuMemHostGetDevicePointer, cuMemsetD2D8, cuMemsetD2D8Async,

—

cuMemsetD2D16, cuMemsetD2D16Async, cuMemsetD2D32, cuMemsetD2D32Async, cuMemsetD8, cuMemsetD8Async, cuMemsetD16, cuMemsetD16Async, cuMemsetD32

CUresult cuMemsetD8 (CUdeviceptr dstDevice, unsigned char uc, size_t N)

Initializes device memory.

Parameters

dstDevice
   - Destination device pointer
uc
   - Value to set
N
   - Number of elements

Returns

CUDA_SUCCESS, CUDA_ERROR_DEINITIALIZED, CUDA_ERROR_NOT_INITIALIZED, CUDA_ERROR_INVALID_CONTEXT, CUDA_ERROR_INVALID_VALUE

Description

Sets the memory range of N 8-bit values to the specified value uc.

> ▸ Note that this function may also return error codes from previous, asynchronous launches.
> ▸ See also memset synchronization details.

See also:

cuArray3DCreate, cuArray3DGetDescriptor, cuArrayCreate, cuArrayDestroy, cuArrayGetDescriptor, cuMemAlloc, cuMemAllocHost, cuMemAllocPitch, cuMemcpy2D, cuMemcpy2DAsync, cuMemcpy2DUnaligned, cuMemcpy3D, cuMemcpy3DAsync, cuMemcpyAtoA, cuMemcpyAtoD, cuMemcpyAtoH, cuMemcpyAtoHAsync, cuMemcpyDtoA, cuMemcpyDtoD, cuMemcpyDtoDAsync, cuMemcpyDtoH, cuMemcpyDtoHAsync, cuMemcpyHtoA, cuMemcpyHtoAAsync, cuMemcpyHtoD, cuMemcpyHtoDAsync, cuMemFree, cuMemFreeHost, cuMemGetAddressRange, cuMemGetInfo, cuMemHostAlloc, cuMemHostGetDevicePointer, cuMemsetD2D8, cuMemsetD2D8Async, cuMemsetD2D16, cuMemsetD2D16Async, cuMemsetD2D32, cuMemsetD2D32Async,

cuMemsetD8Async, cuMemsetD16, cuMemsetD16Async, cuMemsetD32, cuMemsetD32Async

CUresult cuMemsetD8Async (CUdeviceptr dstDevice, unsigned char uc, size_t N, CUstream hStream)

Sets device memory.

Parameters

dstDevice
   - Destination device pointer
uc
   - Value to set
N
   - Number of elements
hStream
   - Stream identifier

Returns

CUDA_SUCCESS, CUDA_ERROR_DEINITIALIZED, CUDA_ERROR_NOT_INITIALIZED, CUDA_ERROR_INVALID_CONTEXT, CUDA_ERROR_INVALID_VALUE

Description

Sets the memory range of N 8-bit values to the specified value uc.

> ▸ Note that this function may also return error codes from previous, asynchronous launches.
> ▸ See also memset synchronization details.
> ▸ This function uses standard NULL stream semantics.

See also:

cuArray3DCreate, cuArray3DGetDescriptor, cuArrayCreate, cuArrayDestroy, cuArrayGetDescriptor, cuMemAlloc, cuMemAllocHost, cuMemAllocPitch, cuMemcpy2D, cuMemcpy2DAsync, cuMemcpy2DUnaligned, cuMemcpy3D, cuMemcpy3DAsync, cuMemcpyAtoA, cuMemcpyAtoD, cuMemcpyAtoH, cuMemcpyAtoHAsync, cuMemcpyDtoA, cuMemcpyDtoD, cuMemcpyDtoDAsync, cuMemcpyDtoH, cuMemcpyDtoHAsync, cuMemcpyHtoA, cuMemcpyHtoAAsync, cuMemcpyHtoD, cuMemcpyHtoDAsync, cuMemFree, cuMemFreeHost, cuMemGetAddressRange, cuMemGetInfo, cuMemHostAlloc,

cuMemHostGetDevicePointer, cuMemsetD2D8, cuMemsetD2D8Async, cuMemsetD2D16, cuMemsetD2D16Async, cuMemsetD2D32, cuMemsetD2D32Async, cuMemsetD8, cuMemsetD16, cuMemsetD16Async, cuMemsetD32, cuMemsetD32Async

CUresult cuMipmappedArrayCreate (CUmipmappedArray *pHandle, const CUDA_ARRAY3D_DESCRIPTOR *pMipmappedArrayDesc, unsigned int numMipmapLevels)

Creates a CUDA mipmapped array.

Parameters

pHandle
   - Returned mipmapped array
pMipmappedArrayDesc
   - mipmapped array descriptor
numMipmapLevels
   - Number of mipmap levels

Returns

CUDA_SUCCESS, CUDA_ERROR_DEINITIALIZED, CUDA_ERROR_NOT_INITIALIZED, CUDA_ERROR_INVALID_CONTEXT, CUDA_ERROR_INVALID_VALUE, CUDA_ERROR_OUT_OF_MEMORY, CUDA_ERROR_UNKNOWN

Description

Creates a CUDA mipmapped array according to the CUDA_ARRAY3D_DESCRIPTOR structure `pMipmappedArrayDesc` and returns a handle to the new CUDA mipmapped array in `*pHandle`. `numMipmapLevels` 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 CUDA_ARRAY3D_DESCRIPTOR is defined as:

```
    typedef struct {
            unsigned int Width;
            unsigned int Height;
            unsigned int Depth;
            CUarray_format Format;
            unsigned int NumChannels;
            unsigned int Flags;
        } CUDA_ARRAY3D_DESCRIPTOR;
```

where:

- ▶ `Width`, `Height`, and `Depth` are the width, height, and depth of the CUDA array (in elements); the following types of CUDA arrays can be allocated:

  - ▶ A 1D mipmapped array is allocated if `Height` and `Depth` extents are both zero.
  - ▶ A 2D mipmapped array is allocated if only `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 `Height` is zero and the CUDA_ARRAY3D_LAYERED flag is set. Each layer is a 1D 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 CUDA_ARRAY3D_LAYERED flag is set. Each layer is a 2D 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 CUDA_ARRAY3D_CUBEMAP 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 CUarray_cubemap_face.
  - ▶ A cubemap layered CUDA mipmapped array is allocated if all three extents are non-zero, and both, CUDA_ARRAY3D_CUBEMAP and CUDA_ARRAY3D_LAYERED 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.

- ▶ Format specifies the format of the elements; CUarray_format is defined as:

```
typedef enum CUarray_format_enum {
        CU_AD_FORMAT_UNSIGNED_INT8 = 0x01,
        CU_AD_FORMAT_UNSIGNED_INT16 = 0x02,
        CU_AD_FORMAT_UNSIGNED_INT32 = 0x03,
        CU_AD_FORMAT_SIGNED_INT8 = 0x08,
        CU_AD_FORMAT_SIGNED_INT16 = 0x09,
        CU_AD_FORMAT_SIGNED_INT32 = 0x0a,
        CU_AD_FORMAT_HALF = 0x10,
        CU_AD_FORMAT_FLOAT = 0x20
} CUarray_format;
```

- ▶ `NumChannels` specifies the number of packed components per CUDA array element; it may be 1, 2, or 4;

- ▶ Flags may be set to

  - ▶ CUDA_ARRAY3D_LAYERED to enable creation of layered CUDA mipmapped arrays. If this flag is set, `Depth` specifies the number of layers, not the depth of a 3D array.
  - ▶ CUDA_ARRAY3D_SURFACE_LDST to enable surface references to be bound to individual mipmap levels of the CUDA mipmapped array. If this flag is not set, cuSurfRefSetArray will fail when attempting to bind a mipmap level of the CUDA mipmapped array to a surface reference.

▶ CUDA_ARRAY3D_CUBEMAP to enable creation of mipmapped cubemaps. If this flag is set, `Width` must be equal to `Height`, and `Depth` must be six. If the CUDA_ARRAY3D_LAYERED flag is also set, then `Depth` must be a multiple of six.

▶ CUDA_ARRAY3D_TEXTURE_GATHER to indicate that the CUDA mipmapped array will be used for texture gather. Texture gather can only be performed on 2D CUDA mipmapped arrays.

`Width`, `Height` and `Depth` must meet certain size requirements as listed in the following table. All values are specified in elements. Note that for brevity's sake, the full name of the device attribute is not specified. For ex., TEXTURE1D_MIPMAPPED_WIDTH refers to the device attribute CU_DEVICE_ATTRIBUTE_MAXIMUM_TEXTURE1D_MIPMAPPED_WIDTH.

CUDA array type	Valid extents that must always be met {(width range in elements), (height range), (depth range)}
1D	{ (1,TEXTURE1D_MIPMAPPED_WIDTH), 0, 0 }
2D	{ (1,TEXTURE2D_MIPMAPPED_WIDTH), (1,TEXTURE2D_MIPMAPPED_HEIGHT), 0 }
3D	{ (1,TEXTURE3D_WIDTH), (1,TEXTURE3D_HEIGHT), (1,TEXTURE3D_DEPTH) } OR { (1,TEXTURE3D_WIDTH_ALTERNATE), (1,TEXTURE3D_HEIGHT_ALTERNATE), (1,TEXTURE3D_DEPTH_ALTERNATE) }
1D Layered	{ (1,TEXTURE1D_LAYERED_WIDTH), 0, (1,TEXTURE1D_LAYERED_LAYERS) }
2D Layered	{ (1,TEXTURE2D_LAYERED_WIDTH), (1,TEXTURE2D_LAYERED_HEIGHT), (1,TEXTURE2D_LAYERED_LAYERS) }
Cubemap	{ (1,TEXTURECUBEMAP_WIDTH), (1,TEXTURECUBEMAP_WIDTH), 6 }
Cubemap Layered	{ (1,TEXTURECUBEMAP_LAYERED_WIDTH), (1,TEXTURECUBEMAP_LAYERED_WIDTH), (1,TEXTURECUBEMAP_LAYERED_LAYERS) }

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

See also:

cuMipmappedArrayDestroy, cuMipmappedArrayGetLevel, cuArrayCreate,

CUresult cuMipmappedArrayDestroy (CUmipmappedArray hMipmappedArray)

Destroys a CUDA mipmapped array.

Parameters

hMipmappedArray
   - Mipmapped array to destroy

Returns

CUDA_SUCCESS, CUDA_ERROR_DEINITIALIZED,
CUDA_ERROR_NOT_INITIALIZED, CUDA_ERROR_INVALID_CONTEXT,
CUDA_ERROR_INVALID_HANDLE, CUDA_ERROR_ARRAY_IS_MAPPED

Description

Destroys the CUDA mipmapped array `hMipmappedArray`.

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

See also:

cuMipmappedArrayCreate, cuMipmappedArrayGetLevel, cuArrayCreate,

CUresult cuMipmappedArrayGetLevel (CUarray *pLevelArray, CUmipmappedArray hMipmappedArray, unsigned int level)

Gets a mipmap level of a CUDA mipmapped array.

Parameters

pLevelArray
   - Returned mipmap level CUDA array
hMipmappedArray
   - CUDA mipmapped array
level
   - Mipmap level

Returns

CUDA_SUCCESS, CUDA_ERROR_DEINITIALIZED,
CUDA_ERROR_NOT_INITIALIZED, CUDA_ERROR_INVALID_CONTEXT,
CUDA_ERROR_INVALID_VALUE, CUDA_ERROR_INVALID_HANDLE

Description

Returns in `*pLevelArray` a CUDA array that represents a single mipmap level of the
CUDA mipmapped array `hMipmappedArray`.

If `level` is greater than the maximum number of levels in this mipmapped array,
CUDA_ERROR_INVALID_VALUE is returned.

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

See also:

cuMipmappedArrayCreate, cuMipmappedArrayDestroy, cuArrayCreate,

3.11. Unified Addressing

This section describes the unified addressing functions of the low-level CUDA driver
application programming interface.

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).

Supported Platforms

Whether or not a device supports unified addressing may be
queried by calling cuDeviceGetAttribute() with the device attribute
CU_DEVICE_ATTRIBUTE_UNIFIED_ADDRESSING.

Unified addressing is automatically enabled in 64-bit processes on devices with compute
capability greater than or equal to 2.0.

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 cuPointerGetAttribute()

Since pointers are unique, it is not necessary to specify information about the pointers specified to the various copy functions in the CUDA API. The function cuMemcpy() may be used to perform a copy between two pointers, ignoring whether they point to host or device memory (making cuMemcpyHtoD(), cuMemcpyDtoD(), and cuMemcpyDtoH() unnecessary for devices supporting unified addressing). For multidimensional copies, the memory type CU_MEMORYTYPE_UNIFIED may be used to specify that the CUDA driver should infer the location of the pointer from its value.

Automatic Mapping of Host Allocated Host Memory

All host memory allocated in all contexts using cuMemAllocHost() and cuMemHostAlloc() is always directly accessible from all contexts on all devices that support unified addressing. This is the case regardless of whether or not the flags CU_MEMHOSTALLOC_PORTABLE and CU_MEMHOSTALLOC_DEVICEMAP 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, so it is not necessary to call cuMemHostGetDevicePointer() to get the device pointer for these allocations.

Note that this is not the case for memory allocated using the flag CU_MEMHOSTALLOC_WRITECOMBINED, as discussed below.

Automatic Registration of Peer Memory

Upon enabling direct access from a context that supports unified addressing to another peer context that supports unified addressing using cuCtxEnablePeerAccess() all memory allocated in the peer context using cuMemAlloc() and cuMemAllocPitch() will immediately be accessible by the current context. The device pointer value through which any peer memory may be accessed in the current context is the same pointer value through which that memory may be accessed in the peer context.

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 cuMemHostRegister() and host memory allocated using the flag CU_MEMHOSTALLOC_WRITECOMBINED. 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 contexts that support unified addressing.

This device address may be queried using cuMemHostGetDevicePointer() when a context using unified addressing is current. Either the host or the unified device pointer

value may be used to refer to this memory through cuMemcpy() and similar functions using the CU_MEMORYTYPE_UNIFIED memory type.

CUresult cuPointerGetAttribute (void *data, CUpointer_attribute attribute, CUdeviceptr ptr)

Returns information about a pointer.

Parameters

data
   - Returned pointer attribute value
attribute
   - Pointer attribute to query
ptr
   - Pointer

Returns

CUDA_SUCCESS, CUDA_ERROR_DEINITIALIZED, CUDA_ERROR_NOT_INITIALIZED, CUDA_ERROR_INVALID_CONTEXT, CUDA_ERROR_INVALID_VALUE, CUDA_ERROR_INVALID_DEVICE

Description

The supported attributes are:

▶   CU_POINTER_ATTRIBUTE_CONTEXT:

Returns in `*data` the CUcontext in which `ptr` was allocated or registered. The type of `data` must be CUcontext *.

If `ptr` was not allocated by, mapped by, or registered with a CUcontext which uses unified virtual addressing then CUDA_ERROR_INVALID_VALUE is returned.

▶   CU_POINTER_ATTRIBUTE_MEMORY_TYPE:

Returns in `*data` the physical memory type of the memory that `ptr` addresses as a CUmemorytype enumerated value. The type of `data` must be unsigned int.

If `ptr` addresses device memory then `*data` is set to CU_MEMORYTYPE_DEVICE. The particular CUdevice on which the memory resides is the CUdevice of the CUcontext returned by the CU_POINTER_ATTRIBUTE_CONTEXT attribute of `ptr`.

If `ptr` addresses host memory then `*data` is set to CU_MEMORYTYPE_HOST.

If `ptr` was not allocated by, mapped by, or registered with a CUcontext which uses unified virtual addressing then CUDA_ERROR_INVALID_VALUE is returned.

If the current CUcontext does not support unified virtual addressing then CUDA_ERROR_INVALID_CONTEXT is returned.

▸ CU_POINTER_ATTRIBUTE_DEVICE_POINTER:

Returns in `*data` the device pointer value through which `ptr` may be accessed by kernels running in the current CUcontext. The type of `data` must be CUdeviceptr *.

If there exists no device pointer value through which kernels running in the current CUcontext may access `ptr` then CUDA_ERROR_INVALID_VALUE is returned.

If there is no current CUcontext then CUDA_ERROR_INVALID_CONTEXT is returned.

Except in the exceptional disjoint addressing cases discussed below, the value returned in `*data` will equal the input value `ptr`.

▸ CU_POINTER_ATTRIBUTE_HOST_POINTER:

Returns in `*data` the host pointer value through which `ptr` may be accessed by by the host program. The type of `data` must be void **. If there exists no host pointer value through which the host program may directly access `ptr` then CUDA_ERROR_INVALID_VALUE is returned.

Except in the exceptional disjoint addressing cases discussed below, the value returned in `*data` will equal the input value `ptr`.

▸ CU_POINTER_ATTRIBUTE_P2P_TOKENS:

Returns in `*data` two tokens for use with the nv-p2p.h Linux kernel interface. `data` must be a struct of type CUDA_POINTER_ATTRIBUTE_P2P_TOKENS.

`ptr` must be a pointer to memory obtained from :cuMemAlloc(). Note that p2pToken and vaSpaceToken are only valid for the lifetime of the source allocation. A subsequent allocation at the same address may return completely different tokens. Querying this attribute has a side effect of setting the attribute CU_POINTER_ATTRIBUTE_SYNC_MEMOPS for the region of memory that `ptr` points to.

▸ CU_POINTER_ATTRIBUTE_SYNC_MEMOPS:

A boolean attribute which when set, ensures that synchronous memory operations initiated on the region of memory that `ptr` points to 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.

▸ CU_POINTER_ATTRIBUTE_BUFFER_ID:

Returns in `*data` a buffer ID which is guaranteed to be unique within the process. `data` must point to an unsigned long long.

`ptr` must be a pointer to memory obtained from a CUDA memory allocation API. Every memory allocation from any of the CUDA memory allocation APIs will have a unique ID over a process lifetime. Subsequent allocations do not reuse IDs from previous freed allocations. IDs are only unique within a single process.

▶ CU_POINTER_ATTRIBUTE_IS_MANAGED:

Returns in `*data` a boolean that indicates whether the pointer points to managed memory or not.

Note that for most allocations in the unified virtual address space the host and device pointer for accessing the allocation will be the same. The exceptions to this are

▶ user memory registered using cuMemHostRegister

▶ host memory allocated using cuMemHostAlloc with the CU_MEMHOSTALLOC_WRITECOMBINED flag For these types of allocation there will exist separate, disjoint host and device addresses for accessing the allocation. In particular

▶ The host address will correspond to an invalid unmapped device address (which will result in an exception if accessed from the device)

▶ The device address will correspond to an invalid unmapped host address (which will result in an exception if accessed from the host). For these types of allocations, querying CU_POINTER_ATTRIBUTE_HOST_POINTER and CU_POINTER_ATTRIBUTE_DEVICE_POINTER may be used to retrieve the host and device addresses from either address.

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

See also:

cuPointerSetAttribute, cuMemAlloc, cuMemFree, cuMemAllocHost, cuMemFreeHost, cuMemHostAlloc, cuMemHostRegister, cuMemHostUnregister

CUresult cuPointerSetAttribute (const void *value, CUpointer_attribute attribute, CUdeviceptr ptr)

Set attributes on a previously allocated memory region.

Parameters

value
   - Pointer to memory containing the value to be set
attribute
   - Pointer attribute to set
ptr
   - Pointer to a memory region allocated using CUDA memory allocation APIs

Returns

CUDA_SUCCESS, CUDA_ERROR_DEINITIALIZED,
CUDA_ERROR_NOT_INITIALIZED, CUDA_ERROR_INVALID_CONTEXT,
CUDA_ERROR_INVALID_VALUE, CUDA_ERROR_INVALID_DEVICE

Description

The supported attributes are:

▸ CU_POINTER_ATTRIBUTE_SYNC_MEMOPS:

A boolean attribute that can either be set (1) or unset (0). When set, the region of memory that `ptr` points to is guaranteed to always synchronize memory operations that are synchronous. If there are some previously initiated synchronous memory operations that are pending when this attribute is set, the function does not return until those memory operations are complete. See further documentation in the section titled "API synchronization behavior" to learn more about cases when synchronous memory operations can exhibit asynchronous behavior. `value` will be considered as a pointer to an unsigned integer to which this attribute is to be set.

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

See also:

cuPointerGetAttribute, cuMemAlloc, cuMemFree, cuMemAllocHost, cuMemFreeHost, cuMemHostAlloc, cuMemHostRegister, cuMemHostUnregister

3.12. Stream Management

This section describes the stream management functions of the low-level CUDA driver application programming interface.

CUresult cuStreamAddCallback (CUstream hStream, CUstreamCallback callback, void *userData, unsigned int flags)

Add a callback to a compute stream.

Parameters

hStream
   - 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

CUDA_SUCCESS, CUDA_ERROR_DEINITIALIZED,
CUDA_ERROR_NOT_INITIALIZED, CUDA_ERROR_INVALID_CONTEXT,
CUDA_ERROR_INVALID_HANDLE, CUDA_ERROR_NOT_SUPPORTED

Description

Adds a callback to be called on the host after all currently enqueued items in the stream have completed. For each cuStreamAddCallback call, the callback will be executed exactly once. The callback will block later work in the stream until it is finished.

The callback may be passed CUDA_SUCCESS or an error code. In the event of a device error, all subsequently executed callbacks will receive an appropriate CUresult.

Callbacks must not make any CUDA API calls. Attempting to use a CUDA API will result in CUDA_ERROR_NOT_PERMITTED. 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.

This API requires compute capability 1.1 or greater. See cuDeviceGetAttribute or cuDeviceGetProperties to query compute capability. Attempting to use this API with earlier compute versions will return CUDA_ERROR_NOT_SUPPORTED.

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.

> ▸ This function uses standard NULL stream semantics.
> ▸ Note that this function may also return error codes from previous, asynchronous launches.

See also:

cuStreamCreate, cuStreamQuery, cuStreamSynchronize, cuStreamWaitEvent, cuStreamDestroy, cuMemAllocManaged, cuStreamAttachMemAsync

CUresult cuStreamAttachMemAsync (CUstream hStream, CUdeviceptr dptr, size_t length, unsigned int flags)

Attach memory to a stream asynchronously.

Parameters

hStream
  - Stream in which to enqueue the attach operation
dptr
  - Pointer to memory (must be a pointer to managed memory)
length
  - Length of memory (must be zero)
flags
  - Must be one of CUmemAttach_flags

Returns

CUDA_SUCCESS, CUDA_ERROR_DEINITIALIZED, CUDA_ERROR_NOT_INITIALIZED, CUDA_ERROR_INVALID_CONTEXT, CUDA_ERROR_INVALID_HANDLE, CUDA_ERROR_NOT_SUPPORTED

Description

Enqueues an operation in `hStream` to specify stream association of `length` bytes of memory starting from `dptr`. 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.

`dptr` must point to an address within managed memory space declared using the __managed__ keyword or allocated with cuMemAllocManaged.

`length` must be zero, to indicate that the entire allocation's stream association is being changed. Currently, it's not possible to change stream association for a portion of an allocation.

The stream association is specified using `flags` which must be one of CUmemAttach_flags. If the CU_MEM_ATTACH_GLOBAL flag is specified, the memory can be accessed by any stream on any device. If the CU_MEM_ATTACH_HOST flag is specified, the program makes a guarantee that it won't access the memory on the device from any stream. If the CU_MEM_ATTACH_SINGLE flag is specified, the program makes a guarantee that it will only access the memory on the device from `hStream`. 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 `hStream` 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 cuStreamAttachMemAsync 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 `hStream` 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 cuMemAllocManaged. For __managed__ variables, the default association is always CU_MEM_ATTACH_GLOBAL. 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.

> ▸ This function uses standard NULL stream semantics.
> ▸ Note that this function may also return error codes from previous, asynchronous launches.

See also:

cuStreamCreate, cuStreamQuery, cuStreamSynchronize, cuStreamWaitEvent, cuStreamDestroy, cuMemAllocManaged

CUresult cuStreamCreate (CUstream *phStream, unsigned int Flags)

Create a stream.

Parameters

phStream
    - Returned newly created stream
Flags
    - Parameters for stream creation

Returns

CUDA_SUCCESS, CUDA_ERROR_DEINITIALIZED,
CUDA_ERROR_NOT_INITIALIZED, CUDA_ERROR_INVALID_CONTEXT,
CUDA_ERROR_INVALID_VALUE, CUDA_ERROR_OUT_OF_MEMORY

Description

Creates a stream and returns a handle in `phStream`. The `Flags` argument determines behaviors of the stream. Valid values for `Flags` are:

▸ CU_STREAM_DEFAULT: Default stream creation flag.
▸ CU_STREAM_NON_BLOCKING: 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.

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

See also:

cuStreamDestroy, cuStreamCreateWithPriority, cuStreamGetPriority, cuStreamGetFlags, cuStreamWaitEvent, cuStreamQuery, cuStreamSynchronize, cuStreamAddCallback

CUresult cuStreamCreateWithPriority (CUstream *phStream, unsigned int flags, int priority)

Create a stream with the given priority.

Parameters

phStream
    - Returned newly created stream

flags

    - Flags for stream creation. See cuStreamCreate for a list of valid flags

priority

    - Stream priority. Lower numbers represent higher priorities. See
    cuCtxGetStreamPriorityRange for more information about meaningful stream
    priorities that can be passed.

Returns

CUDA_SUCCESS, CUDA_ERROR_DEINITIALIZED,
CUDA_ERROR_NOT_INITIALIZED, CUDA_ERROR_INVALID_CONTEXT,
CUDA_ERROR_INVALID_VALUE, CUDA_ERROR_OUT_OF_MEMORY

Description

Creates a stream with the specified priority and returns a handle in `phStream`. This API
alters the scheduler priority of work in the stream. Work in a higher priority stream may
preempt work already executing in a low priority stream.

`priority` follows a convention where lower numbers represent higher priorities. '0'
represents default priority. The range of meaningful numerical priorities can be queried
using cuCtxGetStreamPriorityRange. If the specified priority is outside the numerical
range returned by cuCtxGetStreamPriorityRange, it will automatically be clamped to the
lowest or the highest number in the range.

> ▸ Note that this function may also return error codes from previous, asynchronous
>   launches.
>
> ▸ Stream priorities are supported only on Quadro and Tesla GPUs with compute
>   capability 3.5 or higher.
>
> ▸ 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.

See also:

cuStreamDestroy, cuStreamCreate, cuStreamGetPriority, cuCtxGetStreamPriorityRange,
cuStreamGetFlags, cuStreamWaitEvent, cuStreamQuery, cuStreamSynchronize,
cuStreamAddCallback

CUresult cuStreamDestroy (CUstream hStream)

Destroys a stream.

Parameters

hStream
   - Stream to destroy

Returns

CUDA_SUCCESS, CUDA_ERROR_DEINITIALIZED,
CUDA_ERROR_NOT_INITIALIZED, CUDA_ERROR_INVALID_CONTEXT,
CUDA_ERROR_INVALID_VALUE

Description

Destroys the stream specified by `hStream`.

In case the device is still doing work in the stream `hStream` when cuStreamDestroy() is called, the function will return immediately and the resources associated with `hStream` will be released automatically once the device has completed all work in `hStream`.

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

See also:

cuStreamCreate, cuStreamWaitEvent, cuStreamQuery, cuStreamSynchronize, cuStreamAddCallback

CUresult cuStreamGetFlags (CUstream hStream, unsigned int *flags)

Query the flags of a given stream.

Parameters

hStream
   - Handle to the stream to be queried
flags
   - Pointer to an unsigned integer in which the stream's flags are returned The value returned in `flags` is a logical 'OR' of all flags that were used while creating this stream. See cuStreamCreate for the list of valid flags

Returns

CUDA_SUCCESS, CUDA_ERROR_DEINITIALIZED,
CUDA_ERROR_NOT_INITIALIZED, CUDA_ERROR_INVALID_CONTEXT,
CUDA_ERROR_INVALID_VALUE, CUDA_ERROR_INVALID_HANDLE,
CUDA_ERROR_OUT_OF_MEMORY

Description

Query the flags of a stream created using cuStreamCreate or
cuStreamCreateWithPriority and return the flags in `flags`.

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

See also:

cuStreamDestroy, cuStreamCreate, cuStreamGetPriority

CUresult cuStreamGetPriority (CUstream hStream, int *priority)

Query the priority of a given stream.

Parameters

hStream
   - Handle to the stream to be queried
priority
   - Pointer to a signed integer in which the stream's priority is returned

Returns

CUDA_SUCCESS, CUDA_ERROR_DEINITIALIZED,
CUDA_ERROR_NOT_INITIALIZED, CUDA_ERROR_INVALID_CONTEXT,
CUDA_ERROR_INVALID_VALUE, CUDA_ERROR_INVALID_HANDLE,
CUDA_ERROR_OUT_OF_MEMORY

Description

Query the priority of a stream created using cuStreamCreate or
cuStreamCreateWithPriority and return the priority in `priority`. Note that if
the stream was created with a priority outside the numerical range returned by
cuCtxGetStreamPriorityRange, this function returns the clamped priority. See
cuStreamCreateWithPriority for details about priority clamping.

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

See also:

cuStreamDestroy, cuStreamCreate, cuStreamCreateWithPriority, cuCtxGetStreamPriorityRange, cuStreamGetFlags

CUresult cuStreamQuery (CUstream hStream)

Determine status of a compute stream.

Parameters

hStream
   - Stream to query status of

Returns

CUDA_SUCCESS, CUDA_ERROR_DEINITIALIZED, CUDA_ERROR_NOT_INITIALIZED, CUDA_ERROR_INVALID_CONTEXT, CUDA_ERROR_INVALID_HANDLE, CUDA_ERROR_NOT_READY

Description

Returns CUDA_SUCCESS if all operations in the stream specified by `hStream` have completed, or CUDA_ERROR_NOT_READY if not.

For the purposes of Unified Memory, a return value of CUDA_SUCCESS is equivalent to having called cuStreamSynchronize().

> ‣ This function uses standard NULL stream semantics.
> ‣ Note that this function may also return error codes from previous, asynchronous launches.

See also:

cuStreamCreate, cuStreamWaitEvent, cuStreamDestroy, cuStreamSynchronize, cuStreamAddCallback

CUresult cuStreamSynchronize (CUstream hStream)

Wait until a stream's tasks are completed.

Parameters

hStream
   - Stream to wait for

Returns

CUDA_SUCCESS, CUDA_ERROR_DEINITIALIZED,
CUDA_ERROR_NOT_INITIALIZED, CUDA_ERROR_INVALID_CONTEXT,
CUDA_ERROR_INVALID_HANDLE

Description

Waits until the device has completed all operations in the stream specified by `hStream`. If the context was created with the CU_CTX_SCHED_BLOCKING_SYNC flag, the CPU thread will block until the stream is finished with all of its tasks.

> ▸ This function uses standard NULL stream semantics.
> ▸ Note that this function may also return error codes from previous, asynchronous launches.

See also:

cuStreamCreate, cuStreamDestroy, cuStreamWaitEvent, cuStreamQuery, cuStreamAddCallback

CUresult cuStreamWaitEvent (CUstream hStream, CUevent hEvent, unsigned int Flags)

Make a compute stream wait on an event.

Parameters

hStream
   - Stream to wait
hEvent
   - Event to wait on (may not be NULL)
Flags
   - Parameters for the operation (must be 0)

Returns

CUDA_SUCCESS, CUDA_ERROR_DEINITIALIZED,
CUDA_ERROR_NOT_INITIALIZED, CUDA_ERROR_INVALID_CONTEXT,
CUDA_ERROR_INVALID_HANDLE,

Description

Makes all future work submitted to `hStream` wait until `hEvent` reports completion before beginning execution. This synchronization will be performed efficiently on the device. The event `hEvent` may be from a different context than `hStream`, in which case this function will perform cross-device synchronization.

The stream `hStream` will wait only for the completion of the most recent host call to cuEventRecord() on `hEvent`. Once this call has returned, any functions (including cuEventRecord() and cuEventDestroy()) may be called on `hEvent` again, and subsequent calls will not have any effect on `hStream`.

If cuEventRecord() has not been called on `hEvent`, this call acts as if the record has already completed, and so is a functional no-op.

> ▸ This function uses standard NULL stream semantics.
> ▸ Note that this function may also return error codes from previous, asynchronous launches.

See also:

cuStreamCreate, cuEventRecord, cuStreamQuery, cuStreamSynchronize, cuStreamAddCallback, cuStreamDestroy

3.13. Event Management

This section describes the event management functions of the low-level CUDA driver application programming interface.

CUresult cuEventCreate (CUevent *phEvent, unsigned int Flags)

Creates an event.

Parameters

phEvent
  - Returns newly created event

Flags
  - Event creation flags

Returns

CUDA_SUCCESS, CUDA_ERROR_DEINITIALIZED,
CUDA_ERROR_NOT_INITIALIZED, CUDA_ERROR_INVALID_CONTEXT,
CUDA_ERROR_INVALID_VALUE, CUDA_ERROR_OUT_OF_MEMORY

Description

Creates an event *phEvent with the flags specified via `Flags`. Valid flags include:

▶ CU_EVENT_DEFAULT: Default event creation flag.
▶ CU_EVENT_BLOCKING_SYNC: Specifies that the created event should use blocking synchronization. A CPU thread that uses cuEventSynchronize() to wait on an event created with this flag will block until the event has actually been recorded.
▶ CU_EVENT_DISABLE_TIMING: Specifies that the created event does not need to record timing data. Events created with this flag specified and the CU_EVENT_BLOCKING_SYNC flag not specified will provide the best performance when used with cuStreamWaitEvent() and cuEventQuery().
▶ CU_EVENT_INTERPROCESS: Specifies that the created event may be used as an interprocess event by cuIpcGetEventHandle(). CU_EVENT_INTERPROCESS must be specified along with CU_EVENT_DISABLE_TIMING.

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

See also:

cuEventRecord, cuEventQuery, cuEventSynchronize, cuEventDestroy,
cuEventElapsedTime

CUresult cuEventDestroy (CUevent hEvent)

Destroys an event.

Parameters

hEvent
  - Event to destroy

Returns

CUDA_SUCCESS, CUDA_ERROR_DEINITIALIZED, CUDA_ERROR_NOT_INITIALIZED, CUDA_ERROR_INVALID_CONTEXT, CUDA_ERROR_INVALID_HANDLE

Description

Destroys the event specified by `hEvent`.

In case `hEvent` has been recorded but has not yet been completed when cuEventDestroy() is called, the function will return immediately and the resources associated with `hEvent` will be released automatically once the device has completed `hEvent`.

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

See also:

cuEventCreate, cuEventRecord, cuEventQuery, cuEventSynchronize, cuEventElapsedTime

CUresult cuEventElapsedTime (float *pMilliseconds, CUevent hStart, CUevent hEnd)

Computes the elapsed time between two events.

Parameters

pMilliseconds
   - Time between `hStart` and `hEnd` in ms
hStart
   - Starting event
hEnd
   - Ending event

Returns

CUDA_SUCCESS, CUDA_ERROR_DEINITIALIZED, CUDA_ERROR_NOT_INITIALIZED, CUDA_ERROR_INVALID_CONTEXT, CUDA_ERROR_INVALID_HANDLE, CUDA_ERROR_NOT_READY

Description

Computes the elapsed time between two events (in milliseconds with a resolution of around 0.5 microseconds).

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 cuEventRecord() 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 cuEventRecord() has not been called on either event then CUDA_ERROR_INVALID_HANDLE is returned. If cuEventRecord() has been called on both events but one or both of them has not yet been completed (that is, cuEventQuery() would return CUDA_ERROR_NOT_READY on at least one of the events), CUDA_ERROR_NOT_READY is returned. If either event was created with the CU_EVENT_DISABLE_TIMING flag, then this function will return CUDA_ERROR_INVALID_HANDLE.

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

See also:

cuEventCreate, cuEventRecord, cuEventQuery, cuEventSynchronize, cuEventDestroy

CUresult cuEventQuery (CUevent hEvent)

Queries an event's status.

Parameters

hEvent
  - Event to query

Returns

CUDA_SUCCESS, CUDA_ERROR_DEINITIALIZED, CUDA_ERROR_NOT_INITIALIZED, CUDA_ERROR_INVALID_HANDLE, CUDA_ERROR_INVALID_VALUE, CUDA_ERROR_NOT_READY

Description

Query the status of all device work preceding the most recent call to cuEventRecord() (in the appropriate compute streams, as specified by the arguments to cuEventRecord()).

If this work has successfully been completed by the device, or if cuEventRecord() has not been called on hEvent, then CUDA_SUCCESS is returned. If this work has not yet been completed by the device then CUDA_ERROR_NOT_READY is returned.

For the purposes of Unified Memory, a return value of CUDA_SUCCESS is equivalent to having called cuEventSynchronize().

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

See also:

cuEventCreate, cuEventRecord, cuEventSynchronize, cuEventDestroy, cuEventElapsedTime

CUresult cuEventRecord (CUevent hEvent, CUstream hStream)

Records an event.

Parameters

hEvent
   - Event to record
hStream
   - Stream to record event for

Returns

CUDA_SUCCESS, CUDA_ERROR_DEINITIALIZED, CUDA_ERROR_NOT_INITIALIZED, CUDA_ERROR_INVALID_CONTEXT, CUDA_ERROR_INVALID_HANDLE, CUDA_ERROR_INVALID_VALUE

Description

Records an event. See note on NULL stream behavior. Since operation is asynchronous, cuEventQuery or cuEventSynchronize() must be used to determine when the event has actually been recorded.

If cuEventRecord() has previously been called on hEvent, then this call will overwrite any existing state in hEvent. Any subsequent calls which examine the status of hEvent will only examine the completion of this most recent call to cuEventRecord().

It is necessary that hEvent and hStream be created on the same context.

> ‣ This function uses standard NULL stream semantics.
> ‣ Note that this function may also return error codes from previous, asynchronous launches.

See also:

cuEventCreate, cuEventQuery, cuEventSynchronize, cuStreamWaitEvent, cuEventDestroy, cuEventElapsedTime

CUresult cuEventSynchronize (CUevent hEvent)

Waits for an event to complete.

Parameters

hEvent
   - Event to wait for

Returns

CUDA_SUCCESS, CUDA_ERROR_DEINITIALIZED, CUDA_ERROR_NOT_INITIALIZED, CUDA_ERROR_INVALID_CONTEXT, CUDA_ERROR_INVALID_HANDLE

Description

Wait until the completion of all device work preceding the most recent call to cuEventRecord() (in the appropriate compute streams, as specified by the arguments to cuEventRecord()).

If cuEventRecord() has not been called on hEvent, CUDA_SUCCESS is returned immediately.

Waiting for an event that was created with the CU_EVENT_BLOCKING_SYNC flag will cause the calling CPU thread to block until the event has been completed by the device. If the CU_EVENT_BLOCKING_SYNC flag has not been set, then the CPU thread will busy-wait until the event has been completed by the device.

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

See also:

cuEventCreate, cuEventRecord, cuEventQuery, cuEventDestroy, cuEventElapsedTime

3.14. Execution Control

This section describes the execution control functions of the low-level CUDA driver application programming interface.

CUresult cuFuncGetAttribute (int *pi, CUfunction_attribute attrib, CUfunction hfunc)

Returns information about a function.

Parameters

pi
    - Returned attribute value
attrib
    - Attribute requested
hfunc
    - Function to query attribute of

Returns

CUDA_SUCCESS, CUDA_ERROR_DEINITIALIZED, CUDA_ERROR_NOT_INITIALIZED, CUDA_ERROR_INVALID_CONTEXT, CUDA_ERROR_INVALID_HANDLE, CUDA_ERROR_INVALID_VALUE

Description

Returns in `*pi` the integer value of the attribute `attrib` on the kernel given by `hfunc`. The supported attributes are:

▸ CU_FUNC_ATTRIBUTE_MAX_THREADS_PER_BLOCK: 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.

▸ CU_FUNC_ATTRIBUTE_SHARED_SIZE_BYTES: 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 runtime.

▸ CU_FUNC_ATTRIBUTE_CONST_SIZE_BYTES: The size in bytes of user-allocated constant memory required by this function.

▸ CU_FUNC_ATTRIBUTE_LOCAL_SIZE_BYTES: The size in bytes of local memory used by each thread of this function.

▸ CU_FUNC_ATTRIBUTE_NUM_REGS: The number of registers used by each thread of this function.

▸ CU_FUNC_ATTRIBUTE_PTX_VERSION: 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. Note that this may return the undefined value of 0 for cubins compiled prior to CUDA 3.0.

▸ CU_FUNC_ATTRIBUTE_BINARY_VERSION: 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. Note that this will return a value of 10 for legacy cubins that do not have a properly-encoded binary architecture version.

▸ CU_FUNC_CACHE_MODE_CA: The attribute to indicate whether the function has been compiled with user specified option "-Xptxas --dlcm=ca" set .

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

See also:

cuCtxGetCacheConfig, cuCtxSetCacheConfig, cuFuncSetCacheConfig, cuLaunchKernel

CUresult cuFuncSetCacheConfig (CUfunction hfunc, CUfunc_cache config)

Sets the preferred cache configuration for a device function.

Parameters

hfunc
   - Kernel to configure cache for
config
   - Requested cache configuration

Returns

CUDA_SUCCESS, CUDA_ERROR_INVALID_VALUE, CUDA_ERROR_DEINITIALIZED, CUDA_ERROR_NOT_INITIALIZED, CUDA_ERROR_INVALID_CONTEXT

Description

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

to execute `hfunc`. Any context-wide preference set via cuCtxSetCacheConfig() will be overridden by this per-function setting unless the per-function setting is CU_FUNC_CACHE_PREFER_NONE. In that case, the current context-wide setting will be used.

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:

▸ CU_FUNC_CACHE_PREFER_NONE: no preference for shared memory or L1 (default)
▸ CU_FUNC_CACHE_PREFER_SHARED: prefer larger shared memory and smaller L1 cache
▸ CU_FUNC_CACHE_PREFER_L1: prefer larger L1 cache and smaller shared memory
▸ CU_FUNC_CACHE_PREFER_EQUAL: prefer equal sized L1 cache and shared memory

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

See also:

cuCtxGetCacheConfig, cuCtxSetCacheConfig, cuFuncGetAttribute, cuLaunchKernel

CUresult cuFuncSetSharedMemConfig (CUfunction hfunc, CUsharedconfig config)

Sets the shared memory configuration for a device function.

Parameters

hfunc
    - kernel to be given a shared memory config
config
    - requested shared memory configuration

Returns

CUDA_SUCCESS, CUDA_ERROR_INVALID_VALUE, CUDA_ERROR_DEINITIALIZED, CUDA_ERROR_NOT_INITIALIZED, CUDA_ERROR_INVALID_CONTEXT

Description

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 cuFuncSetSharedMemConfig will override the context wide setting set with cuCtxSetSharedMemConfig.

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:

▸ CU_SHARED_MEM_CONFIG_DEFAULT_BANK_SIZE: use the context's shared memory configuration when launching this function.
▸ CU_SHARED_MEM_CONFIG_FOUR_BYTE_BANK_SIZE: set shared memory bank width to be natively four bytes when launching this function.
▸ CU_SHARED_MEM_CONFIG_EIGHT_BYTE_BANK_SIZE: set shared memory bank width to be natively eight bytes when launching this function.

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

See also:

cuCtxGetCacheConfig, cuCtxSetCacheConfig, cuCtxGetSharedMemConfig, cuCtxSetSharedMemConfig, cuFuncGetAttribute, cuLaunchKernel

CUresult cuLaunchKernel (CUfunction f, unsigned int gridDimX, unsigned int gridDimY, unsigned int gridDimZ, unsigned int blockDimX, unsigned int blockDimY,

unsigned int blockDimZ, unsigned int sharedMemBytes, CUstream hStream, void **kernelParams, void **extra)

Launches a CUDA function.

Parameters

f
  - Kernel to launch
gridDimX
  - Width of grid in blocks
gridDimY
  - Height of grid in blocks
gridDimZ
  - Depth of grid in blocks
blockDimX
  - X dimension of each thread block
blockDimY
  - Y dimension of each thread block
blockDimZ
  - Z dimension of each thread block
sharedMemBytes
  - Dynamic shared-memory size per thread block in bytes
hStream
  - Stream identifier
kernelParams
  - Array of pointers to kernel parameters
extra
  - Extra options

Returns

CUDA_SUCCESS, CUDA_ERROR_DEINITIALIZED,
CUDA_ERROR_NOT_INITIALIZED, CUDA_ERROR_INVALID_CONTEXT,
CUDA_ERROR_INVALID_HANDLE, CUDA_ERROR_INVALID_IMAGE,
CUDA_ERROR_INVALID_VALUE, CUDA_ERROR_LAUNCH_FAILED,
CUDA_ERROR_LAUNCH_OUT_OF_RESOURCES,
CUDA_ERROR_LAUNCH_TIMEOUT,
CUDA_ERROR_LAUNCH_INCOMPATIBLE_TEXTURING,
CUDA_ERROR_SHARED_OBJECT_INIT_FAILED

Description

Invokes the kernel `f` on a `gridDimX` x `gridDimY` x `gridDimZ` grid of blocks. Each block contains `blockDimX` x `blockDimY` x `blockDimZ` threads.

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

Kernel parameters to `f` can be specified in one of two ways:

1) Kernel parameters can be specified via `kernelParams`. If `f` has N parameters, then `kernelParams` needs to be an array of N pointers. Each of `kernelParams`[0] through `kernelParams`[N-1] must point to a region of memory from which the actual kernel 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 the `extra` parameter. This places the burden on the application of knowing each kernel parameter's size and alignment/padding within the buffer. Here is an example of using the `extra` parameter in this manner:

```
    size_t argBufferSize;
        char argBuffer[256];

        // populate argBuffer and argBufferSize

        void *config[] = {
            CU_LAUNCH_PARAM_BUFFER_POINTER, argBuffer,
            CU_LAUNCH_PARAM_BUFFER_SIZE,    &argBufferSize,
            CU_LAUNCH_PARAM_END
        };
        status = cuLaunchKernel(f, gx, gy, gz, bx, by, bz, sh, s, NULL,
 config);
```

The `extra` parameter exists to allow cuLaunchKernel 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 `f`;
▸ 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 CUDA_ERROR_INVALID_VALUE will be returned if kernel parameters are specified with both `kernelParams` and `extra` (i.e. both `kernelParams` and `extra` are non-NULL).

Calling cuLaunchKernel() sets persistent function state that is the same as function state set through the following deprecated APIs: cuFuncSetBlockShape(), cuFuncSetSharedSize(), cuParamSetSize(), cuParamSeti(), cuParamSetf(), cuParamSetv().

When the kernel f is launched via cuLaunchKernel(), the previous block shape, shared size and parameter info associated with f is overwritten.

Note that to use cuLaunchKernel(), the kernel f must either have been compiled with toolchain version 3.2 or later so that it will contain kernel parameter information, or have no kernel parameters. If either of these conditions is not met, then cuLaunchKernel() will return CUDA_ERROR_INVALID_IMAGE.

> ▸ This function uses standard NULL stream semantics.
> ▸ Note that this function may also return error codes from previous, asynchronous launches.

See also:

cuCtxGetCacheConfig, cuCtxSetCacheConfig, cuFuncSetCacheConfig, cuFuncGetAttribute

3.15. Execution Control [DEPRECATED]

This section describes the deprecated execution control functions of the low-level CUDA driver application programming interface.

CUresult cuFuncSetBlockShape (CUfunction hfunc, int x, int y, int z)

Sets the block-dimensions for the function.

Parameters

hfunc
　- Kernel to specify dimensions of
x
　- X dimension
y
　- Y dimension
z
　- Z dimension

Returns

CUDA_SUCCESS, CUDA_ERROR_DEINITIALIZED, CUDA_ERROR_NOT_INITIALIZED, CUDA_ERROR_INVALID_CONTEXT, CUDA_ERROR_INVALID_HANDLE, CUDA_ERROR_INVALID_VALUE

Description

Deprecated

Specifies the x, y, and z dimensions of the thread blocks that are created when the kernel given by hfunc is launched.

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

See also:

cuFuncSetSharedSize, cuFuncSetCacheConfig, cuFuncGetAttribute, cuParamSetSize, cuParamSeti, cuParamSetf, cuParamSetv, cuLaunch, cuLaunchGrid, cuLaunchGridAsync, cuLaunchKernel

CUresult cuFuncSetSharedSize (CUfunction hfunc, unsigned int bytes)

Sets the dynamic shared-memory size for the function.

Parameters

hfunc
   - Kernel to specify dynamic shared-memory size for
bytes
   - Dynamic shared-memory size per thread in bytes

Returns

CUDA_SUCCESS, CUDA_ERROR_DEINITIALIZED, CUDA_ERROR_NOT_INITIALIZED, CUDA_ERROR_INVALID_CONTEXT, CUDA_ERROR_INVALID_HANDLE, CUDA_ERROR_INVALID_VALUE

Description

Deprecated

Sets through bytes the amount of dynamic shared memory that will be available to each thread block when the kernel given by hfunc is launched.

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

See also:

cuFuncSetBlockShape, cuFuncSetCacheConfig, cuFuncGetAttribute, cuParamSetSize, cuParamSeti, cuParamSetf, cuParamSetv, cuLaunch, cuLaunchGrid, cuLaunchGridAsync, cuLaunchKernel

CUresult cuLaunch (CUfunction f)

Launches a CUDA function.

Parameters

f
    - Kernel to launch

Returns

CUDA_SUCCESS, CUDA_ERROR_DEINITIALIZED, CUDA_ERROR_NOT_INITIALIZED, CUDA_ERROR_INVALID_CONTEXT, CUDA_ERROR_INVALID_VALUE, CUDA_ERROR_LAUNCH_FAILED, CUDA_ERROR_LAUNCH_OUT_OF_RESOURCES, CUDA_ERROR_LAUNCH_TIMEOUT, CUDA_ERROR_LAUNCH_INCOMPATIBLE_TEXTURING, CUDA_ERROR_SHARED_OBJECT_INIT_FAILED

Description

Deprecated

Invokes the kernel f on a 1 x 1 x 1 grid of blocks. The block contains the number of threads specified by a previous call to cuFuncSetBlockShape().

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

See also:

cuFuncSetBlockShape, cuFuncSetSharedSize, cuFuncGetAttribute, cuParamSetSize, cuParamSetf, cuParamSeti, cuParamSetv, cuLaunchGrid, cuLaunchGridAsync, cuLaunchKernel

CUresult cuLaunchGrid (CUfunction f, int grid_width, int grid_height)

Launches a CUDA function.

Parameters

f
   - Kernel to launch
grid_width
   - Width of grid in blocks
grid_height
   - Height of grid in blocks

Returns

CUDA_SUCCESS, CUDA_ERROR_DEINITIALIZED,
CUDA_ERROR_NOT_INITIALIZED, CUDA_ERROR_INVALID_CONTEXT,
CUDA_ERROR_INVALID_VALUE, CUDA_ERROR_LAUNCH_FAILED,
CUDA_ERROR_LAUNCH_OUT_OF_RESOURCES,
CUDA_ERROR_LAUNCH_TIMEOUT,
CUDA_ERROR_LAUNCH_INCOMPATIBLE_TEXTURING,
CUDA_ERROR_SHARED_OBJECT_INIT_FAILED

Description

Deprecated

Invokes the kernel `f` on a `grid_width` x `grid_height` grid of blocks. Each block contains the number of threads specified by a previous call to cuFuncSetBlockShape().

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

See also:

cuFuncSetBlockShape, cuFuncSetSharedSize, cuFuncGetAttribute, cuParamSetSize, cuParamSetf, cuParamSeti, cuParamSetv, cuLaunch, cuLaunchGridAsync, cuLaunchKernel

CUresult cuLaunchGridAsync (CUfunction f, int grid_width, int grid_height, CUstream hStream)

Launches a CUDA function.

Parameters

f
   - Kernel to launch
grid_width
   - Width of grid in blocks
grid_height
   - Height of grid in blocks
hStream
   - Stream identifier

Returns

CUDA_SUCCESS, CUDA_ERROR_DEINITIALIZED, CUDA_ERROR_NOT_INITIALIZED, CUDA_ERROR_INVALID_CONTEXT, CUDA_ERROR_INVALID_HANDLE, CUDA_ERROR_INVALID_VALUE, CUDA_ERROR_LAUNCH_FAILED, CUDA_ERROR_LAUNCH_OUT_OF_RESOURCES, CUDA_ERROR_LAUNCH_TIMEOUT, CUDA_ERROR_LAUNCH_INCOMPATIBLE_TEXTURING, CUDA_ERROR_SHARED_OBJECT_INIT_FAILED

Description

Deprecated

Invokes the kernel f on a grid_width x grid_height grid of blocks. Each block contains the number of threads specified by a previous call to cuFuncSetBlockShape().

> ▸ In certain cases where cubins are created with no ABI (i.e., using ptxas --abi-compile no), this function may serialize kernel launches. In order to force the CUDA driver to retain asynchronous behavior, set the CU_CTX_LMEM_RESIZE_TO_MAX flag during context creation (see cuCtxCreate).
> ▸ This function uses standard NULL stream semantics.
> ▸ Note that this function may also return error codes from previous, asynchronous launches.

See also:

cuFuncSetBlockShape, cuFuncSetSharedSize, cuFuncGetAttribute, cuParamSetSize, cuParamSetf, cuParamSeti, cuParamSetv, cuLaunch, cuLaunchGrid, cuLaunchKernel

CUresult cuParamSetf (CUfunction hfunc, int offset, float value)

Adds a floating-point parameter to the function's argument list.

Parameters

hfunc
- Kernel to add parameter to
offset
- Offset to add parameter to argument list
value
- Value of parameter

Returns

CUDA_SUCCESS, CUDA_ERROR_DEINITIALIZED, CUDA_ERROR_NOT_INITIALIZED, CUDA_ERROR_INVALID_CONTEXT, CUDA_ERROR_INVALID_VALUE

Description

Deprecated

Sets a floating-point parameter that will be specified the next time the kernel corresponding to `hfunc` will be invoked. `offset` is a byte offset.

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

See also:

cuFuncSetBlockShape, cuFuncSetSharedSize, cuFuncGetAttribute, cuParamSetSize, cuParamSeti, cuParamSetv, cuLaunch, cuLaunchGrid, cuLaunchGridAsync, cuLaunchKernel

CUresult cuParamSeti (CUfunction hfunc, int offset, unsigned int value)

Adds an integer parameter to the function's argument list.

Parameters

hfunc
   - Kernel to add parameter to
offset
   - Offset to add parameter to argument list
value
   - Value of parameter

Returns

CUDA_SUCCESS, CUDA_ERROR_DEINITIALIZED,
CUDA_ERROR_NOT_INITIALIZED, CUDA_ERROR_INVALID_CONTEXT,
CUDA_ERROR_INVALID_VALUE

Description

Deprecated

Sets an integer parameter that will be specified the next time the kernel corresponding to `hfunc` will be invoked. `offset` is a byte offset.

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

See also:

cuFuncSetBlockShape, cuFuncSetSharedSize, cuFuncGetAttribute, cuParamSetSize, cuParamSetf, cuParamSetv, cuLaunch, cuLaunchGrid, cuLaunchGridAsync, cuLaunchKernel

CUresult cuParamSetSize (CUfunction hfunc, unsigned int numbytes)

Sets the parameter size for the function.

Parameters

hfunc
   - Kernel to set parameter size for

numbytes
   - Size of parameter list in bytes

Returns

CUDA_SUCCESS, CUDA_ERROR_DEINITIALIZED,
CUDA_ERROR_NOT_INITIALIZED, CUDA_ERROR_INVALID_CONTEXT,
CUDA_ERROR_INVALID_VALUE

Description

Deprecated

Sets through `numbytes` the total size in bytes needed by the function parameters of the kernel corresponding to `hfunc`.

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

See also:

cuFuncSetBlockShape, cuFuncSetSharedSize, cuFuncGetAttribute, cuParamSetf,
cuParamSeti, cuParamSetv, cuLaunch, cuLaunchGrid, cuLaunchGridAsync,
cuLaunchKernel

CUresult cuParamSetTexRef (CUfunction hfunc, int texunit, CUtexref hTexRef)

Adds a texture-reference to the function's argument list.

Parameters

hfunc
   - Kernel to add texture-reference to
texunit
   - Texture unit (must be CU_PARAM_TR_DEFAULT)
hTexRef
   - Texture-reference to add to argument list

Returns

CUDA_SUCCESS, CUDA_ERROR_DEINITIALIZED,
CUDA_ERROR_NOT_INITIALIZED, CUDA_ERROR_INVALID_CONTEXT,
CUDA_ERROR_INVALID_VALUE

Description

Deprecated

Makes the CUDA array or linear memory bound to the texture reference `hTexRef` available to a device program as a texture. In this version of CUDA, the texture-reference must be obtained via cuModuleGetTexRef() and the `texunit` parameter must be set to CU_PARAM_TR_DEFAULT.

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

CUresult cuParamSetv (CUfunction hfunc, int offset, void *ptr, unsigned int numbytes)

Adds arbitrary data to the function's argument list.

Parameters

hfunc
    - Kernel to add data to
offset
    - Offset to add data to argument list
ptr
    - Pointer to arbitrary data
numbytes
    - Size of data to copy in bytes

Returns

CUDA_SUCCESS, CUDA_ERROR_DEINITIALIZED, CUDA_ERROR_NOT_INITIALIZED, CUDA_ERROR_INVALID_CONTEXT, CUDA_ERROR_INVALID_VALUE

Description

Deprecated

Copies an arbitrary amount of data (specified in `numbytes`) from `ptr` into the parameter space of the kernel corresponding to `hfunc`. `offset` is a byte offset.

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

See also:

cuFuncSetBlockShape, cuFuncSetSharedSize, cuFuncGetAttribute, cuParamSetSize, cuParamSetf, cuParamSeti, cuLaunch, cuLaunchGrid, cuLaunchGridAsync, cuLaunchKernel

3.16. Occupancy

This section describes the occupancy calculation functions of the low-level CUDA driver application programming interface.

CUresult cuOccupancyMaxActiveBlocksPerMultiprocessor (int *numBlocks, CUfunction func, int blockSize, size_t dynamicSMemSize)

Returns occupancy of a function.

Parameters

numBlocks
   - Returned occupancy
func
   - Kernel 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

CUDA_SUCCESS, CUDA_ERROR_DEINITIALIZED, CUDA_ERROR_NOT_INITIALIZED, CUDA_ERROR_INVALID_CONTEXT, CUDA_ERROR_INVALID_VALUE, CUDA_ERROR_UNKNOWN

Description

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

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

CUresult cuOccupancyMaxPotentialBlockSize (int *minGridSize, int *blockSize, CUfunction func, CUoccupancyB2DSize blockSizeToDynamicSMemSize, size_t dynamicSMemSize, int blockSizeLimit)

Suggest a launch configuration with reasonable occupancy.

Parameters

minGridSize
  - Returned minimum grid size needed to achieve the maximum occupancy
blockSize
  - Returned maximum block size that can achieve the maximum occupancy
func
  - Kernel for which launch configuration is calulated
blockSizeToDynamicSMemSize
  - A function that calculates how much per-block dynamic shared memory `func` uses based on the block size
dynamicSMemSize
  - Dynamic shared memory usage intended, in bytes
blockSizeLimit
  - The maximum block size `func` is designed to handle

Returns

CUDA_SUCCESS, CUDA_ERROR_DEINITIALIZED, CUDA_ERROR_NOT_INITIALIZED, CUDA_ERROR_INVALID_CONTEXT, CUDA_ERROR_INVALID_VALUE, CUDA_ERROR_UNKNOWN

Description

Returns in `*blockSize` a reasonable block size that can achieve the maximum occupancy (or, the maximum number of active warps with the fewest blocks per multiprocessor), and in `*minGridSize` the minimum grid size to achieve the maximum occupancy.

If `blockSizeLimit` is 0, the configurator will use the maximum block size permitted by the device / function instead.

If per-block dynamic shared memory allocation is not needed, the user should leave both `blockSizeToDynamicSMemSize` and `dynamicSMemSize` as 0.

If per-block dynamic shared memory allocation is needed, then if the dynamic shared memory size is constant regardless of block size, the size should be passed through `dynamicSMemSize`, and `blockSizeToDynamicSMemSize` should be NULL.

Otherwise, if the per-block dynamic shared memory size varies with different block sizes, the user needs to provide a unary function through `blockSizeToDynamicSMemSize` that computes the dynamic shared memory needed by `func` for any given block size. `dynamicSMemSize` is ignored. An example signature is:

```
    // Take block size, returns dynamic shared memory needed
        size_t blockToSmem(int blockSize);
```

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

3.17. Texture Reference Management

This section describes the texture reference management functions of the low-level CUDA driver application programming interface.

CUresult cuTexRefGetAddress (CUdeviceptr *pdptr, CUtexref hTexRef)

Gets the address associated with a texture reference.

Parameters

pdptr
    - Returned device address
hTexRef
    - Texture reference

Returns

CUDA_SUCCESS, CUDA_ERROR_DEINITIALIZED, CUDA_ERROR_NOT_INITIALIZED, CUDA_ERROR_INVALID_CONTEXT, CUDA_ERROR_INVALID_VALUE

Description

Returns in `*pdptr` the base address bound to the texture reference `hTexRef`, or returns CUDA_ERROR_INVALID_VALUE if the texture reference is not bound to any device memory range.

See also:

cuTexRefSetAddress, cuTexRefSetAddress2D, cuTexRefSetAddressMode,
cuTexRefSetArray, cuTexRefSetFilterMode, cuTexRefSetFlags, cuTexRefSetFormat,
cuTexRefGetAddressMode, cuTexRefGetArray, cuTexRefGetFilterMode,
cuTexRefGetFlags, cuTexRefGetFormat

CUresult cuTexRefGetAddressMode (CUaddress_mode *pam, CUtexref hTexRef, int dim)

Gets the addressing mode used by a texture reference.

Parameters

pam
 - Returned addressing mode
hTexRef
 - Texture reference
dim
 - Dimension

Returns

CUDA_SUCCESS, CUDA_ERROR_DEINITIALIZED,
CUDA_ERROR_NOT_INITIALIZED, CUDA_ERROR_INVALID_CONTEXT,
CUDA_ERROR_INVALID_VALUE

Description

Returns in `*pam` the addressing mode corresponding to the dimension `dim` of the
texture reference `hTexRef`. Currently, the only valid value for `dim` are 0 and 1.

See also:

cuTexRefSetAddress, cuTexRefSetAddress2D, cuTexRefSetAddressMode,
cuTexRefSetArray, cuTexRefSetFilterMode, cuTexRefSetFlags, cuTexRefSetFormat,
cuTexRefGetAddress, cuTexRefGetArray, cuTexRefGetFilterMode, cuTexRefGetFlags,
cuTexRefGetFormat

CUresult cuTexRefGetArray (CUarray *phArray, CUtexref hTexRef)

Gets the array bound to a texture reference.

Parameters

phArray
 - Returned array

hTexRef
- Texture reference

Returns

CUDA_SUCCESS, CUDA_ERROR_DEINITIALIZED,
CUDA_ERROR_NOT_INITIALIZED, CUDA_ERROR_INVALID_CONTEXT,
CUDA_ERROR_INVALID_VALUE

Description

Returns in `*phArray` the CUDA array bound to the texture reference `hTexRef`, or
returns CUDA_ERROR_INVALID_VALUE if the texture reference is not bound to any
CUDA array.

See also:

cuTexRefSetAddress, cuTexRefSetAddress2D, cuTexRefSetAddressMode,
cuTexRefSetArray, cuTexRefSetFilterMode, cuTexRefSetFlags, cuTexRefSetFormat,
cuTexRefGetAddress, cuTexRefGetAddressMode, cuTexRefGetFilterMode,
cuTexRefGetFlags, cuTexRefGetFormat

CUresult cuTexRefGetFilterMode (CUfilter_mode *pfm, CUtexref hTexRef)

Gets the filter-mode used by a texture reference.

Parameters

pfm
- Returned filtering mode
hTexRef
- Texture reference

Returns

CUDA_SUCCESS, CUDA_ERROR_DEINITIALIZED,
CUDA_ERROR_NOT_INITIALIZED, CUDA_ERROR_INVALID_CONTEXT,
CUDA_ERROR_INVALID_VALUE

Description

Returns in `*pfm` the filtering mode of the texture reference `hTexRef`.

See also:

cuTexRefSetAddress, cuTexRefSetAddress2D, cuTexRefSetAddressMode, cuTexRefSetArray, cuTexRefSetFilterMode, cuTexRefSetFlags, cuTexRefSetFormat, cuTexRefGetAddress, cuTexRefGetAddressMode, cuTexRefGetArray, cuTexRefGetFlags, cuTexRefGetFormat

CUresult cuTexRefGetFlags (unsigned int *pFlags, CUtexref hTexRef)

Gets the flags used by a texture reference.

Parameters

pFlags
- Returned flags
hTexRef
- Texture reference

Returns

CUDA_SUCCESS, CUDA_ERROR_DEINITIALIZED, CUDA_ERROR_NOT_INITIALIZED, CUDA_ERROR_INVALID_CONTEXT, CUDA_ERROR_INVALID_VALUE

Description

Returns in `*pFlags` the flags of the texture reference `hTexRef`.

See also:

cuTexRefSetAddress, cuTexRefSetAddress2D, cuTexRefSetAddressMode, cuTexRefSetArray, cuTexRefSetFilterMode, cuTexRefSetFlags, cuTexRefSetFormat, cuTexRefGetAddress, cuTexRefGetAddressMode, cuTexRefGetArray, cuTexRefGetFilterMode, cuTexRefGetFormat

CUresult cuTexRefGetFormat (CUarray_format *pFormat, int *pNumChannels, CUtexref hTexRef)

Gets the format used by a texture reference.

Parameters

pFormat
- Returned format
pNumChannels
- Returned number of components

hTexRef
  - Texture reference

Returns

CUDA_SUCCESS, CUDA_ERROR_DEINITIALIZED,
CUDA_ERROR_NOT_INITIALIZED, CUDA_ERROR_INVALID_CONTEXT,
CUDA_ERROR_INVALID_VALUE

Description

Returns in `*pFormat` and `*pNumChannels` the format and number of components of the CUDA array bound to the texture reference `hTexRef`. If `pFormat` or `pNumChannels` is NULL, it will be ignored.

See also:

cuTexRefSetAddress, cuTexRefSetAddress2D, cuTexRefSetAddressMode,
cuTexRefSetArray, cuTexRefSetFilterMode, cuTexRefSetFlags, cuTexRefSetFormat,
cuTexRefGetAddress, cuTexRefGetAddressMode, cuTexRefGetArray,
cuTexRefGetFilterMode, cuTexRefGetFlags

CUresult cuTexRefGetMaxAnisotropy (int *pmaxAniso, CUtexref hTexRef)

Gets the maximum anisotropy for a texture reference.

Parameters

pmaxAniso
  - Returned maximum anisotropy
hTexRef
  - Texture reference

Returns

CUDA_SUCCESS, CUDA_ERROR_DEINITIALIZED,
CUDA_ERROR_NOT_INITIALIZED, CUDA_ERROR_INVALID_CONTEXT,
CUDA_ERROR_INVALID_VALUE

Description

Returns the maximum anisotropy in `pmaxAniso` that's used when reading memory through the texture reference `hTexRef`.

See also:

cuTexRefSetAddress, cuTexRefSetAddress2D, cuTexRefSetAddressMode, cuTexRefSetArray, cuTexRefSetFlags, cuTexRefSetFormat, cuTexRefGetAddress, cuTexRefGetAddressMode, cuTexRefGetArray, cuTexRefGetFilterMode, cuTexRefGetFlags, cuTexRefGetFormat

CUresult cuTexRefGetMipmapFilterMode (CUfilter_mode *pfm, CUtexref hTexRef)

Gets the mipmap filtering mode for a texture reference.

Parameters

pfm
   - Returned mipmap filtering mode
hTexRef
   - Texture reference

Returns

CUDA_SUCCESS, CUDA_ERROR_DEINITIALIZED, CUDA_ERROR_NOT_INITIALIZED, CUDA_ERROR_INVALID_CONTEXT, CUDA_ERROR_INVALID_VALUE

Description

Returns the mipmap filtering mode in `pfm` that's used when reading memory through the texture reference `hTexRef`.

See also:

cuTexRefSetAddress, cuTexRefSetAddress2D, cuTexRefSetAddressMode, cuTexRefSetArray, cuTexRefSetFlags, cuTexRefSetFormat, cuTexRefGetAddress, cuTexRefGetAddressMode, cuTexRefGetArray, cuTexRefGetFilterMode, cuTexRefGetFlags, cuTexRefGetFormat

CUresult cuTexRefGetMipmapLevelBias (float *pbias, CUtexref hTexRef)

Gets the mipmap level bias for a texture reference.

Parameters

pbias
   - Returned mipmap level bias
hTexRef
   - Texture reference

Returns

CUDA_SUCCESS, CUDA_ERROR_DEINITIALIZED,
CUDA_ERROR_NOT_INITIALIZED, CUDA_ERROR_INVALID_CONTEXT,
CUDA_ERROR_INVALID_VALUE

Description

Returns the mipmap level bias in `pBias` that's added to the specified mipmap level
when reading memory through the texture reference `hTexRef`.

See also:

cuTexRefSetAddress, cuTexRefSetAddress2D, cuTexRefSetAddressMode,
cuTexRefSetArray, cuTexRefSetFlags, cuTexRefSetFormat, cuTexRefGetAddress,
cuTexRefGetAddressMode, cuTexRefGetArray, cuTexRefGetFilterMode,
cuTexRefGetFlags, cuTexRefGetFormat

CUresult cuTexRefGetMipmapLevelClamp (float *pminMipmapLevelClamp, float *pmaxMipmapLevelClamp, CUtexref hTexRef)

Gets the min/max mipmap level clamps for a texture reference.

Parameters

pminMipmapLevelClamp
   - Returned mipmap min level clamp
pmaxMipmapLevelClamp
   - Returned mipmap max level clamp
hTexRef
   - Texture reference

Returns

CUDA_SUCCESS, CUDA_ERROR_DEINITIALIZED,
CUDA_ERROR_NOT_INITIALIZED, CUDA_ERROR_INVALID_CONTEXT,
CUDA_ERROR_INVALID_VALUE

Description

Returns the min/max mipmap level clamps in `pminMipmapLevelClamp` and
`pmaxMipmapLevelClamp` that's used when reading memory through the texture
reference `hTexRef`.

See also:

cuTexRefSetAddress, cuTexRefSetAddress2D, cuTexRefSetAddressMode, cuTexRefSetArray, cuTexRefSetFlags, cuTexRefSetFormat, cuTexRefGetAddress, cuTexRefGetAddressMode, cuTexRefGetArray, cuTexRefGetFilterMode, cuTexRefGetFlags, cuTexRefGetFormat

CUresult cuTexRefGetMipmappedArray (CUmipmappedArray *phMipmappedArray, CUtexref hTexRef)

Gets the mipmapped array bound to a texture reference.

Parameters

phMipmappedArray
   - Returned mipmapped array
hTexRef
   - Texture reference

Returns

CUDA_SUCCESS, CUDA_ERROR_DEINITIALIZED, CUDA_ERROR_NOT_INITIALIZED, CUDA_ERROR_INVALID_CONTEXT, CUDA_ERROR_INVALID_VALUE

Description

Returns in `*phMipmappedArray` the CUDA mipmapped array bound to the texture reference `hTexRef`, or returns CUDA_ERROR_INVALID_VALUE if the texture reference is not bound to any CUDA mipmapped array.

See also:

cuTexRefSetAddress, cuTexRefSetAddress2D, cuTexRefSetAddressMode, cuTexRefSetArray, cuTexRefSetFilterMode, cuTexRefSetFlags, cuTexRefSetFormat, cuTexRefGetAddress, cuTexRefGetAddressMode, cuTexRefGetFilterMode, cuTexRefGetFlags, cuTexRefGetFormat

CUresult cuTexRefSetAddress (size_t *ByteOffset, CUtexref hTexRef, CUdeviceptr dptr, size_t bytes)

Binds an address as a texture reference.

Parameters

ByteOffset
   - Returned byte offset

hTexRef
   - Texture reference to bind
dptr
   - Device pointer to bind
bytes
   - Size of memory to bind in bytes

Returns

CUDA_SUCCESS, CUDA_ERROR_DEINITIALIZED,
CUDA_ERROR_NOT_INITIALIZED, CUDA_ERROR_INVALID_CONTEXT,
CUDA_ERROR_INVALID_VALUE

Description

Binds a linear address range to the texture reference `hTexRef`. Any previous address or CUDA array state associated with the texture reference is superseded by this function. Any memory previously bound to `hTexRef` is unbound.

Since the hardware enforces an alignment requirement on texture base addresses, cuTexRefSetAddress() passes back a byte offset in `*ByteOffset` that must be applied to texture fetches in order to read from the desired memory. This offset must be divided by the texel size and passed to kernels that read from the texture so they can be applied to the tex1Dfetch() function.

If the device memory pointer was returned from cuMemAlloc(), the offset is guaranteed to be 0 and NULL may be passed as the `ByteOffset` parameter.

The total number of elements (or texels) in the linear address range cannot exceed CU_DEVICE_ATTRIBUTE_MAXIMUM_TEXTURE1D_LINEAR_WIDTH. The number of elements is computed as (`bytes` / bytesPerElement), where bytesPerElement is determined from the data format and number of components set using cuTexRefSetFormat().

See also:

cuTexRefSetAddress2D, cuTexRefSetAddressMode, cuTexRefSetArray, cuTexRefSetFilterMode, cuTexRefSetFlags, cuTexRefSetFormat, cuTexRefGetAddress, cuTexRefGetAddressMode, cuTexRefGetArray, cuTexRefGetFilterMode, cuTexRefGetFlags, cuTexRefGetFormat

CUresult cuTexRefSetAddress2D (CUtexref hTexRef, const CUDA_ARRAY_DESCRIPTOR *desc, CUdeviceptr dptr, size_t Pitch)

Binds an address as a 2D texture reference.

Parameters

hTexRef
   - Texture reference to bind
desc
   - Descriptor of CUDA array
dptr
   - Device pointer to bind
Pitch
   - Line pitch in bytes

Returns

CUDA_SUCCESS, CUDA_ERROR_DEINITIALIZED, CUDA_ERROR_NOT_INITIALIZED, CUDA_ERROR_INVALID_CONTEXT, CUDA_ERROR_INVALID_VALUE

Description

Binds a linear address range to the texture reference `hTexRef`. Any previous address or CUDA array state associated with the texture reference is superseded by this function. Any memory previously bound to `hTexRef` is unbound.

Using a tex2D() function inside a kernel requires a call to either cuTexRefSetArray() to bind the corresponding texture reference to an array, or cuTexRefSetAddress2D() to bind the texture reference to linear memory.

Function calls to cuTexRefSetFormat() cannot follow calls to cuTexRefSetAddress2D() for the same texture reference.

It is required that `dptr` be aligned to the appropriate hardware-specific texture alignment. You can query this value using the device attribute CU_DEVICE_ATTRIBUTE_TEXTURE_ALIGNMENT. If an unaligned `dptr` is supplied, CUDA_ERROR_INVALID_VALUE is returned.

`Pitch` has to be aligned to the hardware-specific texture pitch alignment. This value can be queried using the device attribute CU_DEVICE_ATTRIBUTE_TEXTURE_PITCH_ALIGNMENT. If an unaligned `Pitch` is supplied, CUDA_ERROR_INVALID_VALUE is returned.

Width and Height, which are specified in elements (or texels), cannot exceed
CU_DEVICE_ATTRIBUTE_MAXIMUM_TEXTURE2D_LINEAR_WIDTH and
CU_DEVICE_ATTRIBUTE_MAXIMUM_TEXTURE2D_LINEAR_HEIGHT
respectively. Pitch, which is specified in bytes, cannot exceed
CU_DEVICE_ATTRIBUTE_MAXIMUM_TEXTURE2D_LINEAR_PITCH.

See also:

cuTexRefSetAddress, cuTexRefSetAddressMode, cuTexRefSetArray,
cuTexRefSetFilterMode, cuTexRefSetFlags, cuTexRefSetFormat, cuTexRefGetAddress,
cuTexRefGetAddressMode, cuTexRefGetArray, cuTexRefGetFilterMode,
cuTexRefGetFlags, cuTexRefGetFormat

CUresult cuTexRefSetAddressMode (CUtexref hTexRef, int dim, CUaddress_mode am)

Sets the addressing mode for a texture reference.

Parameters

hTexRef
   - Texture reference
dim
   - Dimension
am
   - Addressing mode to set

Returns

CUDA_SUCCESS, CUDA_ERROR_DEINITIALIZED,
CUDA_ERROR_NOT_INITIALIZED, CUDA_ERROR_INVALID_CONTEXT,
CUDA_ERROR_INVALID_VALUE

Description

Specifies the addressing mode am for the given dimension dim of the texture reference
hTexRef. If dim is zero, the addressing mode is applied to the first parameter
of the functions used to fetch from the texture; if dim is 1, the second, and so on.
CUaddress_mode is defined as:

```
    typedef enum CUaddress_mode_enum {
            CU_TR_ADDRESS_MODE_WRAP = 0,
            CU_TR_ADDRESS_MODE_CLAMP = 1,
            CU_TR_ADDRESS_MODE_MIRROR = 2,
            CU_TR_ADDRESS_MODE_BORDER = 3
        } CUaddress_mode;
```

Note that this call has no effect if `hTexRef` is bound to linear memory. Also, if the flag, CU_TRSF_NORMALIZED_COORDINATES, is not set, the only supported address mode is CU_TR_ADDRESS_MODE_CLAMP.

See also:

cuTexRefSetAddress, cuTexRefSetAddress2D, cuTexRefSetArray, cuTexRefSetFilterMode, cuTexRefSetFlags, cuTexRefSetFormat, cuTexRefGetAddress, cuTexRefGetAddressMode, cuTexRefGetArray, cuTexRefGetFilterMode, cuTexRefGetFlags, cuTexRefGetFormat

CUresult cuTexRefSetArray (CUtexref hTexRef, CUarray hArray, unsigned int Flags)

Binds an array as a texture reference.

Parameters

hTexRef
   - Texture reference to bind
hArray
   - Array to bind
Flags
   - Options (must be CU_TRSA_OVERRIDE_FORMAT)

Returns

CUDA_SUCCESS, CUDA_ERROR_DEINITIALIZED, CUDA_ERROR_NOT_INITIALIZED, CUDA_ERROR_INVALID_CONTEXT, CUDA_ERROR_INVALID_VALUE

Description

Binds the CUDA array `hArray` to the texture reference `hTexRef`. Any previous address or CUDA array state associated with the texture reference is superseded by this function. `Flags` must be set to CU_TRSA_OVERRIDE_FORMAT. Any CUDA array previously bound to `hTexRef` is unbound.

See also:

cuTexRefSetAddress, cuTexRefSetAddress2D, cuTexRefSetAddressMode, cuTexRefSetFilterMode, cuTexRefSetFlags, cuTexRefSetFormat, cuTexRefGetAddress, cuTexRefGetAddressMode, cuTexRefGetArray, cuTexRefGetFilterMode, cuTexRefGetFlags, cuTexRefGetFormat

CUresult cuTexRefSetFilterMode (CUtexref hTexRef, CUfilter_mode fm)

Sets the filtering mode for a texture reference.

Parameters

hTexRef
  - Texture reference
fm
  - Filtering mode to set

Returns

CUDA_SUCCESS, CUDA_ERROR_DEINITIALIZED,
CUDA_ERROR_NOT_INITIALIZED, CUDA_ERROR_INVALID_CONTEXT,
CUDA_ERROR_INVALID_VALUE

Description

Specifies the filtering mode `fm` to be used when reading memory through the texture reference `hTexRef`. CUfilter_mode_enum is defined as:

```
typedef enum CUfilter_mode_enum {
        CU_TR_FILTER_MODE_POINT = 0,
        CU_TR_FILTER_MODE_LINEAR = 1
    } CUfilter_mode;
```

Note that this call has no effect if `hTexRef` is bound to linear memory.

See also:

cuTexRefSetAddress, cuTexRefSetAddress2D, cuTexRefSetAddressMode,
cuTexRefSetArray, cuTexRefSetFlags, cuTexRefSetFormat, cuTexRefGetAddress,
cuTexRefGetAddressMode, cuTexRefGetArray, cuTexRefGetFilterMode,
cuTexRefGetFlags, cuTexRefGetFormat

CUresult cuTexRefSetFlags (CUtexref hTexRef, unsigned int Flags)

Sets the flags for a texture reference.

Parameters

hTexRef
  - Texture reference
Flags
  - Optional flags to set

Returns

CUDA_SUCCESS, CUDA_ERROR_DEINITIALIZED,
CUDA_ERROR_NOT_INITIALIZED, CUDA_ERROR_INVALID_CONTEXT,
CUDA_ERROR_INVALID_VALUE

Description

Specifies optional flags via `Flags` to specify the behavior of data returned through the texture reference `hTexRef`. The valid flags are:

▶ CU_TRSF_READ_AS_INTEGER, which suppresses the default behavior of having the texture promote integer data to floating point data in the range [0, 1]. Note that texture with 32-bit integer format would not be promoted, regardless of whether or not this flag is specified;

▶ CU_TRSF_NORMALIZED_COORDINATES, which suppresses the default behavior of having the texture coordinates range from [0, Dim) where Dim is the width or height of the CUDA array. Instead, the texture coordinates [0, 1.0) reference the entire breadth of the array dimension;

See also:

cuTexRefSetAddress, cuTexRefSetAddress2D, cuTexRefSetAddressMode, cuTexRefSetArray, cuTexRefSetFilterMode, cuTexRefSetFormat, cuTexRefGetAddress, cuTexRefGetAddressMode, cuTexRefGetArray, cuTexRefGetFilterMode, cuTexRefGetFlags, cuTexRefGetFormat

CUresult cuTexRefSetFormat (CUtexref hTexRef, CUarray_format fmt, int NumPackedComponents)

Sets the format for a texture reference.

Parameters

hTexRef
 - Texture reference
fmt
 - Format to set
NumPackedComponents
 - Number of components per array element

Returns

CUDA_SUCCESS, CUDA_ERROR_DEINITIALIZED,
CUDA_ERROR_NOT_INITIALIZED, CUDA_ERROR_INVALID_CONTEXT,
CUDA_ERROR_INVALID_VALUE

Description

Specifies the format of the data to be read by the texture reference `hTexRef`. `fmt` and `NumPackedComponents` are exactly analogous to the Format and NumChannels members of the CUDA_ARRAY_DESCRIPTOR structure: They specify the format of each component and the number of components per array element.

See also:

cuTexRefSetAddress, cuTexRefSetAddress2D, cuTexRefSetAddressMode, cuTexRefSetArray, cuTexRefSetFilterMode, cuTexRefSetFlags, cuTexRefGetAddress, cuTexRefGetAddressMode, cuTexRefGetArray, cuTexRefGetFilterMode, cuTexRefGetFlags, cuTexRefGetFormat

CUresult cuTexRefSetMaxAnisotropy (CUtexref hTexRef, unsigned int maxAniso)

Sets the maximum anisotropy for a texture reference.

Parameters

hTexRef
   - Texture reference
maxAniso
   - Maximum anisotropy

Returns

CUDA_SUCCESS, CUDA_ERROR_DEINITIALIZED, CUDA_ERROR_NOT_INITIALIZED, CUDA_ERROR_INVALID_CONTEXT, CUDA_ERROR_INVALID_VALUE

Description

Specifies the maximum anisotropy `maxAniso` to be used when reading memory through the texture reference `hTexRef`.

Note that this call has no effect if `hTexRef` is bound to linear memory.

See also:

cuTexRefSetAddress, cuTexRefSetAddress2D, cuTexRefSetAddressMode, cuTexRefSetArray, cuTexRefSetFlags, cuTexRefSetFormat, cuTexRefGetAddress, cuTexRefGetAddressMode, cuTexRefGetArray, cuTexRefGetFilterMode, cuTexRefGetFlags, cuTexRefGetFormat

CUresult cuTexRefSetMipmapFilterMode (CUtexref hTexRef, CUfilter_mode fm)

Sets the mipmap filtering mode for a texture reference.

Parameters

hTexRef
- Texture reference
fm
- Filtering mode to set

Returns

CUDA_SUCCESS, CUDA_ERROR_DEINITIALIZED,
CUDA_ERROR_NOT_INITIALIZED, CUDA_ERROR_INVALID_CONTEXT,
CUDA_ERROR_INVALID_VALUE

Description

Specifies the mipmap filtering mode `fm` to be used when reading memory through the texture reference `hTexRef`. CUfilter_mode_enum is defined as:

```
typedef enum CUfilter_mode_enum {
        CU_TR_FILTER_MODE_POINT = 0,
        CU_TR_FILTER_MODE_LINEAR = 1
    } CUfilter_mode;
```

Note that this call has no effect if `hTexRef` is not bound to a mipmapped array.

See also:

cuTexRefSetAddress, cuTexRefSetAddress2D, cuTexRefSetAddressMode,
cuTexRefSetArray, cuTexRefSetFlags, cuTexRefSetFormat, cuTexRefGetAddress,
cuTexRefGetAddressMode, cuTexRefGetArray, cuTexRefGetFilterMode,
cuTexRefGetFlags, cuTexRefGetFormat

CUresult cuTexRefSetMipmapLevelBias (CUtexref hTexRef, float bias)

Sets the mipmap level bias for a texture reference.

Parameters

hTexRef
- Texture reference
bias
- Mipmap level bias

Returns

CUDA_SUCCESS, CUDA_ERROR_DEINITIALIZED,
CUDA_ERROR_NOT_INITIALIZED, CUDA_ERROR_INVALID_CONTEXT,
CUDA_ERROR_INVALID_VALUE

Description

Specifies the mipmap level bias `bias` to be added to the specified mipmap level when reading memory through the texture reference `hTexRef`.

Note that this call has no effect if `hTexRef` is not bound to a mipmapped array.

See also:

cuTexRefSetAddress, cuTexRefSetAddress2D, cuTexRefSetAddressMode, cuTexRefSetArray, cuTexRefSetFlags, cuTexRefSetFormat, cuTexRefGetAddress, cuTexRefGetAddressMode, cuTexRefGetArray, cuTexRefGetFilterMode, cuTexRefGetFlags, cuTexRefGetFormat

CUresult cuTexRefSetMipmapLevelClamp (CUtexref hTexRef, float minMipmapLevelClamp, float maxMipmapLevelClamp)

Sets the mipmap min/max mipmap level clamps for a texture reference.

Parameters

hTexRef
   - Texture reference
minMipmapLevelClamp
   - Mipmap min level clamp
maxMipmapLevelClamp
   - Mipmap max level clamp

Returns

CUDA_SUCCESS, CUDA_ERROR_DEINITIALIZED,
CUDA_ERROR_NOT_INITIALIZED, CUDA_ERROR_INVALID_CONTEXT,
CUDA_ERROR_INVALID_VALUE

Description

Specifies the min/max mipmap level clamps, `minMipmapLevelClamp` and `maxMipmapLevelClamp` respectively, to be used when reading memory through the texture reference `hTexRef`.

Note that this call has no effect if `hTexRef` is not bound to a mipmapped array.

See also:

cuTexRefSetAddress, cuTexRefSetAddress2D, cuTexRefSetAddressMode, cuTexRefSetArray, cuTexRefSetFlags, cuTexRefSetFormat, cuTexRefGetAddress, cuTexRefGetAddressMode, cuTexRefGetArray, cuTexRefGetFilterMode, cuTexRefGetFlags, cuTexRefGetFormat

CUresult cuTexRefSetMipmappedArray (CUtexref hTexRef, CUmipmappedArray hMipmappedArray, unsigned int Flags)

Binds a mipmapped array to a texture reference.

Parameters

hTexRef
   - Texture reference to bind
hMipmappedArray
   - Mipmapped array to bind
Flags
   - Options (must be CU_TRSA_OVERRIDE_FORMAT)

Returns

CUDA_SUCCESS, CUDA_ERROR_DEINITIALIZED, CUDA_ERROR_NOT_INITIALIZED, CUDA_ERROR_INVALID_CONTEXT, CUDA_ERROR_INVALID_VALUE

Description

Binds the CUDA mipmapped array `hMipmappedArray` to the texture reference `hTexRef`. Any previous address or CUDA array state associated with the texture reference is superseded by this function. `Flags` must be set to CU_TRSA_OVERRIDE_FORMAT. Any CUDA array previously bound to `hTexRef` is unbound.

See also:

cuTexRefSetAddress, cuTexRefSetAddress2D, cuTexRefSetAddressMode, cuTexRefSetFilterMode, cuTexRefSetFlags, cuTexRefSetFormat, cuTexRefGetAddress, cuTexRefGetAddressMode, cuTexRefGetArray, cuTexRefGetFilterMode, cuTexRefGetFlags, cuTexRefGetFormat

3.18. Texture Reference Management [DEPRECATED]

This section describes the deprecated texture reference management functions of the low-level CUDA driver application programming interface.

CUresult cuTexRefCreate (CUtexref *pTexRef)

Creates a texture reference.

Parameters

pTexRef
- Returned texture reference

Returns

CUDA_SUCCESS, CUDA_ERROR_DEINITIALIZED, CUDA_ERROR_NOT_INITIALIZED, CUDA_ERROR_INVALID_CONTEXT, CUDA_ERROR_INVALID_VALUE

Description

Deprecated

Creates a texture reference and returns its handle in `*pTexRef`. Once created, the application must call cuTexRefSetArray() or cuTexRefSetAddress() to associate the reference with allocated memory. Other texture reference functions are used to specify the format and interpretation (addressing, filtering, etc.) to be used when the memory is read through this texture reference.

See also:

cuTexRefDestroy

CUresult cuTexRefDestroy (CUtexref hTexRef)

Destroys a texture reference.

Parameters

hTexRef
- Texture reference to destroy

Returns

CUDA_SUCCESS, CUDA_ERROR_DEINITIALIZED,
CUDA_ERROR_NOT_INITIALIZED, CUDA_ERROR_INVALID_CONTEXT,
CUDA_ERROR_INVALID_VALUE

Description

Deprecated

Destroys the texture reference specified by `hTexRef`.

See also:

cuTexRefCreate

3.19. Surface Reference Management

This section describes the surface reference management functions of the low-level
CUDA driver application programming interface.

CUresult cuSurfRefGetArray (CUarray *phArray, CUsurfref hSurfRef)

Passes back the CUDA array bound to a surface reference.

Parameters

phArray
  - Surface reference handle
hSurfRef
  - Surface reference handle

Returns

CUDA_SUCCESS, CUDA_ERROR_DEINITIALIZED,
CUDA_ERROR_NOT_INITIALIZED, CUDA_ERROR_INVALID_CONTEXT,
CUDA_ERROR_INVALID_VALUE

Description

Returns in `*phArray` the CUDA array bound to the surface reference `hSurfRef`, or
returns CUDA_ERROR_INVALID_VALUE if the surface reference is not bound to any
CUDA array.

See also:

cuModuleGetSurfRef, cuSurfRefSetArray

CUresult cuSurfRefSetArray (CUsurfref hSurfRef, CUarray hArray, unsigned int Flags)

Sets the CUDA array for a surface reference.

Parameters

hSurfRef
   - Surface reference handle
hArray
   - CUDA array handle
Flags
   - set to 0

Returns

CUDA_SUCCESS, CUDA_ERROR_DEINITIALIZED,
CUDA_ERROR_NOT_INITIALIZED, CUDA_ERROR_INVALID_CONTEXT,
CUDA_ERROR_INVALID_VALUE

Description

Sets the CUDA array `hArray` to be read and written by the surface reference `hSurfRef`. Any previous CUDA array state associated with the surface reference is superseded by this function. `Flags` must be set to 0. The CUDA_ARRAY3D_SURFACE_LDST flag must have been set for the CUDA array. Any CUDA array previously bound to `hSurfRef` is unbound.

See also:

cuModuleGetSurfRef, cuSurfRefGetArray

3.20. Texture Object Management

This section describes the texture object management functions of the low-level CUDA driver application programming interface. The texture object API is only supported on devices of compute capability 3.0 or higher.

CUresult cuTexObjectCreate (CUtexObject *pTexObject, const CUDA_RESOURCE_DESC *pResDesc,

const CUDA_TEXTURE_DESC *pTexDesc, const CUDA_RESOURCE_VIEW_DESC *pResViewDesc)

Creates a texture object.

Parameters

pTexObject
- Texture object to create

pResDesc
- Resource descriptor

pTexDesc
- Texture descriptor

pResViewDesc
- Resource view descriptor

Returns

CUDA_SUCCESS, CUDA_ERROR_DEINITIALIZED, CUDA_ERROR_NOT_INITIALIZED, CUDA_ERROR_INVALID_CONTEXT, CUDA_ERROR_INVALID_VALUE

Description

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.

Texture objects are only supported on devices of compute capability 3.0 or higher.

The CUDA_RESOURCE_DESC structure is defined as:

```
typedef struct CUDA_RESOURCE_DESC_st
{
    CUresourcetype resType;

    union {
        struct {
            CUarray hArray;
        } array;
        struct {
            CUmipmappedArray hMipmappedArray;
        } mipmap;
        struct {
            CUdeviceptr devPtr;
            CUarray_format format;
            unsigned int numChannels;
            size_t sizeInBytes;
        } linear;
        struct {
            CUdeviceptr devPtr;
            CUarray_format format;
            unsigned int numChannels;
            size_t width;
            size_t height;
            size_t pitchInBytes;
        } pitch2D;
    } res;

    unsigned int flags;
} CUDA_RESOURCE_DESC;
```

where:

▶ CUDA_RESOURCE_DESC::resType specifies the type of resource to texture from.
CUresourceType is defined as:

```
typedef enum CUresourcetype_enum {
    CU_RESOURCE_TYPE_ARRAY          = 0x00,
    CU_RESOURCE_TYPE_MIPMAPPED_ARRAY = 0x01,
    CU_RESOURCE_TYPE_LINEAR         = 0x02,
    CU_RESOURCE_TYPE_PITCH2D        = 0x03
} CUresourcetype;
```

If CUDA_RESOURCE_DESC::resType is set to CU_RESOURCE_TYPE_ARRAY,
CUDA_RESOURCE_DESC::res::array::hArray must be set to a valid CUDA array
handle.

If CUDA_RESOURCE_DESC::resType is set to
CU_RESOURCE_TYPE_MIPMAPPED_ARRAY,
CUDA_RESOURCE_DESC::res::mipmap::hMipmappedArray must be set to a valid
CUDA mipmapped array handle.

If CUDA_RESOURCE_DESC::resType is set to CU_RESOURCE_TYPE_LINEAR,
CUDA_RESOURCE_DESC::res::linear::devPtr must be set to a valid device
pointer, that is aligned to CU_DEVICE_ATTRIBUTE_TEXTURE_ALIGNMENT.
CUDA_RESOURCE_DESC::res::linear::format and
CUDA_RESOURCE_DESC::res::linear::numChannels describe the format
of each component and the number of components per array element.

CUDA_RESOURCE_DESC::res::linear::sizeInBytes specifies the size of the array in bytes. The total number of elements in the linear address range cannot exceed CU_DEVICE_ATTRIBUTE_MAXIMUM_TEXTURE1D_LINEAR_WIDTH. The number of elements is computed as (sizeInBytes / (sizeof(format) * numChannels)).

If CUDA_RESOURCE_DESC::resType is set to CU_RESOURCE_TYPE_PITCH2D, CUDA_RESOURCE_DESC::res::pitch2D::devPtr must be set to a valid device pointer, that is aligned to CU_DEVICE_ATTRIBUTE_TEXTURE_ALIGNMENT. CUDA_RESOURCE_DESC::res::pitch2D::format and CUDA_RESOURCE_DESC::res::pitch2D::numChannels describe the format of each component and the number of components per array element. CUDA_RESOURCE_DESC::res::pitch2D::width and CUDA_RESOURCE_DESC::res::pitch2D::height specify the width and height of the array in elements, and cannot exceed CU_DEVICE_ATTRIBUTE_MAXIMUM_TEXTURE2D_LINEAR_WIDTH and CU_DEVICE_ATTRIBUTE_MAXIMUM_TEXTURE2D_LINEAR_HEIGHT respectively. CUDA_RESOURCE_DESC::res::pitch2D::pitchInBytes specifies the pitch between two rows in bytes and has to be aligned to CU_DEVICE_ATTRIBUTE_TEXTURE_PITCH_ALIGNMENT. Pitch cannot exceed CU_DEVICE_ATTRIBUTE_MAXIMUM_TEXTURE2D_LINEAR_PITCH.

▶ flags must be set to zero.

The CUDA_TEXTURE_DESC struct is defined as

```
    typedef struct CUDA_TEXTURE_DESC_st {
            CUaddress_mode addressMode[3];
            CUfilter_mode filterMode;
            unsigned int flags;
            unsigned int maxAnisotropy;
            CUfilter_mode mipmapFilterMode;
            float mipmapLevelBias;
            float minMipmapLevelClamp;
            float maxMipmapLevelClamp;
        } CUDA_TEXTURE_DESC;
```

where

▶ CUDA_TEXTURE_DESC::addressMode specifies the addressing mode for each dimension of the texture data. CUaddress_mode is defined as:

```
    typedef enum CUaddress_mode_enum {
            CU_TR_ADDRESS_MODE_WRAP = 0,
            CU_TR_ADDRESS_MODE_CLAMP = 1,
            CU_TR_ADDRESS_MODE_MIRROR = 2,
            CU_TR_ADDRESS_MODE_BORDER = 3
        } CUaddress_mode;
```

This is ignored if CUDA_RESOURCE_DESC::resType is CU_RESOURCE_TYPE_LINEAR. Also, if the flag, CU_TRSF_NORMALIZED_COORDINATES is not set, the only supported address mode is CU_TR_ADDRESS_MODE_CLAMP.

▶ CUDA_TEXTURE_DESC::filterMode specifies the filtering mode to be used when fetching from the texture. CUfilter_mode is defined as:

```
typedef enum CUfilter_mode_enum {
        CU_TR_FILTER_MODE_POINT = 0,
        CU_TR_FILTER_MODE_LINEAR = 1
    } CUfilter_mode;
```

This is ignored if CUDA_RESOURCE_DESC::resType is CU_RESOURCE_TYPE_LINEAR.

▶ CUDA_TEXTURE_DESC::flags can be any combination of the following:

   ▶ CU_TRSF_READ_AS_INTEGER, which suppresses the default behavior of having the texture promote integer data to floating point data in the range [0, 1]. Note that texture with 32-bit integer format would not be promoted, regardless of whether or not this flag is specified.

   ▶ CU_TRSF_NORMALIZED_COORDINATES, which suppresses the default behavior of having the texture coordinates range from [0, Dim) where Dim is the width or height of the CUDA array. Instead, the texture coordinates [0, 1.0) reference the entire breadth of the array dimension; Note that for CUDA mipmapped arrays, this flag has to be set.

▶ CUDA_TEXTURE_DESC::maxAnisotropy specifies the maximum anisotropy ratio to be used when doing anisotropic filtering. This value will be clamped to the range [1,16].

▶ CUDA_TEXTURE_DESC::mipmapFilterMode specifies the filter mode when the calculated mipmap level lies between two defined mipmap levels.

▶ CUDA_TEXTURE_DESC::mipmapLevelBias specifies the offset to be applied to the calculated mipmap level.

▶ CUDA_TEXTURE_DESC::minMipmapLevelClamp specifies the lower end of the mipmap level range to clamp access to.

▶ CUDA_TEXTURE_DESC::maxMipmapLevelClamp specifies the upper end of the mipmap level range to clamp access to.

The CUDA_RESOURCE_VIEW_DESC struct is defined as

```
typedef struct CUDA_RESOURCE_VIEW_DESC_st
    {
        CUresourceViewFormat format;
        size_t width;
        size_t height;
        size_t depth;
        unsigned int firstMipmapLevel;
        unsigned int lastMipmapLevel;
        unsigned int firstLayer;
        unsigned int lastLayer;
    } CUDA_RESOURCE_VIEW_DESC;
```

where:

▶ CUDA_RESOURCE_VIEW_DESC::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 base of format CU_AD_FORMAT_UNSIGNED_INT32. with 2 or 4 channels, depending on the block compressed format. For ex., BC1 and BC4 require the underlying CUDA array to have a format of CU_AD_FORMAT_UNSIGNED_INT32 with 2 channels. The other BC formats require the underlying resource to have the same base format but with 4 channels.

▶ CUDA_RESOURCE_VIEW_DESC::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.

▶ CUDA_RESOURCE_VIEW_DESC::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.

▶ CUDA_RESOURCE_VIEW_DESC::depth specifies the new depth of the texture data. This value has to be equal to that of the original resource.

▶ CUDA_RESOURCE_VIEW_DESC::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.CUDA_TEXTURE_DESC::minMipmapLevelClamp and CUDA_TEXTURE_DESC::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.

▶ CUDA_RESOURCE_VIEW_DESC::lastMipmapLevel specifies the least detailed mipmap level. For non-mipmapped resources, this value has to be zero.

▶ CUDA_RESOURCE_VIEW_DESC::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.

▶ CUDA_RESOURCE_VIEW_DESC::lastLayer specifies the last layer index for layered textures. For non-layered resources, this value has to be zero.

See also:

cuTexObjectDestroy

CUresult cuTexObjectDestroy (CUtexObject texObject)

Destroys a texture object.

Parameters

texObject
   - Texture object to destroy

Returns

CUDA_SUCCESS, CUDA_ERROR_DEINITIALIZED,
CUDA_ERROR_NOT_INITIALIZED, CUDA_ERROR_INVALID_CONTEXT,
CUDA_ERROR_INVALID_VALUE

Description

Destroys the texture object specified by `texObject`.

See also:

cuTexObjectCreate

CUresult cuTexObjectGetResourceDesc (CUDA_RESOURCE_DESC *pResDesc, CUtexObject texObject)

Returns a texture object's resource descriptor.

Parameters

pResDesc
   - Resource descriptor
texObject
   - Texture object

Returns

CUDA_SUCCESS, CUDA_ERROR_DEINITIALIZED,
CUDA_ERROR_NOT_INITIALIZED, CUDA_ERROR_INVALID_CONTEXT,
CUDA_ERROR_INVALID_VALUE

Description

Returns the resource descriptor for the texture object specified by `texObject`.

See also:

cuTexObjectCreate

CUresult cuTexObjectGetResourceViewDesc (CUDA_RESOURCE_VIEW_DESC *pResViewDesc, CUtexObject texObject)

Returns a texture object's resource view descriptor.

Parameters

pResViewDesc
   - Resource view descriptor
texObject
   - Texture object

Returns

CUDA_SUCCESS, CUDA_ERROR_DEINITIALIZED, CUDA_ERROR_NOT_INITIALIZED, CUDA_ERROR_INVALID_CONTEXT, CUDA_ERROR_INVALID_VALUE

Description

Returns the resource view descriptor for the texture object specified by `texObject`. If no resource view was set for `texObject`, the CUDA_ERROR_INVALID_VALUE is returned.

See also:

cuTexObjectCreate

CUresult cuTexObjectGetTextureDesc (CUDA_TEXTURE_DESC *pTexDesc, CUtexObject texObject)

Returns a texture object's texture descriptor.

Parameters

pTexDesc
   - Texture descriptor
texObject
   - Texture object

Returns

CUDA_SUCCESS, CUDA_ERROR_DEINITIALIZED,
CUDA_ERROR_NOT_INITIALIZED, CUDA_ERROR_INVALID_CONTEXT,
CUDA_ERROR_INVALID_VALUE

Description

Returns the texture descriptor for the texture object specified by `texObject`.

See also:

cuTexObjectCreate

3.21. Surface Object Management

This section describes the surface object management functions of the low-level CUDA driver application programming interface. The surface object API is only supported on devices of compute capability 3.0 or higher.

CUresult cuSurfObjectCreate (CUsurfObject *pSurfObject, const CUDA_RESOURCE_DESC *pResDesc)

Creates a surface object.

Parameters

pSurfObject
    - Surface object to create
pResDesc
    - Resource descriptor

Returns

CUDA_SUCCESS, CUDA_ERROR_DEINITIALIZED,
CUDA_ERROR_NOT_INITIALIZED, CUDA_ERROR_INVALID_CONTEXT,
CUDA_ERROR_INVALID_VALUE

Description

Creates a surface object and returns it in `pSurfObject`. `pResDesc` describes the data to perform surface load/stores on. CUDA_RESOURCE_DESC::resType must be CU_RESOURCE_TYPE_ARRAY and CUDA_RESOURCE_DESC::res::array::hArray must be set to a valid CUDA array handle. CUDA_RESOURCE_DESC::flags must be set to zero.

Surface objects are only supported on devices of compute capability 3.0 or higher.

See also:

cuSurfObjectDestroy

CUresult cuSurfObjectDestroy (CUsurfObject surfObject)

Destroys a surface object.

Parameters

surfObject
   - Surface object to destroy

Returns

CUDA_SUCCESS, CUDA_ERROR_DEINITIALIZED,
CUDA_ERROR_NOT_INITIALIZED, CUDA_ERROR_INVALID_CONTEXT,
CUDA_ERROR_INVALID_VALUE

Description

Destroys the surface object specified by `surfObject`.

See also:

cuSurfObjectCreate

CUresult cuSurfObjectGetResourceDesc (CUDA_RESOURCE_DESC *pResDesc, CUsurfObject surfObject)

Returns a surface object's resource descriptor.

Parameters

pResDesc
   - Resource descriptor
surfObject
   - Surface object

Returns

CUDA_SUCCESS, CUDA_ERROR_DEINITIALIZED,
CUDA_ERROR_NOT_INITIALIZED, CUDA_ERROR_INVALID_CONTEXT,
CUDA_ERROR_INVALID_VALUE

Description

Returns the resource descriptor for the surface object specified by `surfObject`.

See also:

cuSurfObjectCreate

3.22. Peer Context Memory Access

This section describes the direct peer context memory access functions of the low-level CUDA driver application programming interface.

CUresult cuCtxDisablePeerAccess (CUcontext peerContext)

Disables direct access to memory allocations in a peer context and unregisters any registered allocations.

Parameters

peerContext
- Peer context to disable direct access to

Returns

CUDA_SUCCESS, CUDA_ERROR_DEINITIALIZED, CUDA_ERROR_NOT_INITIALIZED, CUDA_ERROR_PEER_ACCESS_NOT_ENABLED, CUDA_ERROR_INVALID_CONTEXT,

Description

Returns CUDA_ERROR_PEER_ACCESS_NOT_ENABLED if direct peer access has not yet been enabled from `peerContext` to the current context.

Returns CUDA_ERROR_INVALID_CONTEXT if there is no current context, or if `peerContext` is not a valid context.

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

See also:

cuDeviceCanAccessPeer, cuCtxEnablePeerAccess

CUresult cuCtxEnablePeerAccess (CUcontext peerContext, unsigned int Flags)

Enables direct access to memory allocations in a peer context.

Parameters

peerContext
   - Peer context to enable direct access to from the current context
Flags
   - Reserved for future use and must be set to 0

Returns

CUDA_SUCCESS, CUDA_ERROR_DEINITIALIZED,
CUDA_ERROR_NOT_INITIALIZED,
CUDA_ERROR_PEER_ACCESS_ALREADY_ENABLED,
CUDA_ERROR_TOO_MANY_PEERS, CUDA_ERROR_INVALID_CONTEXT,
CUDA_ERROR_PEER_ACCESS_UNSUPPORTED, CUDA_ERROR_INVALID_VALUE

Description

If both the current context and `peerContext` are on devices
which support unified addressing (as may be queried using
CU_DEVICE_ATTRIBUTE_UNIFIED_ADDRESSING) and same major compute
capability, then on success all allocations from `peerContext` will immediately be
accessible by the current context. See Unified Addressing for additional details.

Note that access granted by this call is unidirectional and that in order to access
memory from the current context in `peerContext`, a separate symmetric call to
cuCtxEnablePeerAccess() is required.

Returns CUDA_ERROR_PEER_ACCESS_UNSUPPORTED if cuDeviceCanAccessPeer()
indicates that the CUdevice of the current context cannot directly access memory from
the CUdevice of `peerContext`.

Returns CUDA_ERROR_PEER_ACCESS_ALREADY_ENABLED if direct access of
`peerContext` from the current context has already been enabled.

Returns CUDA_ERROR_TOO_MANY_PEERS if direct peer access is not possible
because hardware resources required for peer access have been exhausted.

Returns CUDA_ERROR_INVALID_CONTEXT if there is no current context,
`peerContext` is not a valid context, or if the current context is `peerContext`.

Returns CUDA_ERROR_INVALID_VALUE if `Flags` is not 0.

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

See also:

cuDeviceCanAccessPeer, cuCtxDisablePeerAccess

CUresult cuDeviceCanAccessPeer (int *canAccessPeer, CUdevice dev, CUdevice peerDev)

Queries if a device may directly access a peer device's memory.

Parameters

canAccessPeer
  - Returned access capability
dev
  - Device from which allocations on `peerDev` are to be directly accessed.
peerDev
  - Device on which the allocations to be directly accessed by `dev` reside.

Returns

CUDA_SUCCESS, CUDA_ERROR_DEINITIALIZED, CUDA_ERROR_NOT_INITIALIZED, CUDA_ERROR_INVALID_DEVICE

Description

Returns in `*canAccessPeer` a value of 1 if contexts on `dev` are capable of directly accessing memory from contexts on `peerDev` and 0 otherwise. If direct access of `peerDev` from `dev` is possible, then access may be enabled on two specific contexts by calling cuCtxEnablePeerAccess().

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

See also:

cuCtxEnablePeerAccess, cuCtxDisablePeerAccess

3.23. Graphics Interoperability

This section describes the graphics interoperability functions of the low-level CUDA driver application programming interface.

CUresult cuGraphicsMapResources (unsigned int count, CUgraphicsResource *resources, CUstream hStream)

Map graphics resources for access by CUDA.

Parameters

count
   - Number of resources to map
resources
   - Resources to map for CUDA usage
hStream
   - Stream with which to synchronize

Returns

CUDA_SUCCESS, CUDA_ERROR_DEINITIALIZED,
CUDA_ERROR_NOT_INITIALIZED, CUDA_ERROR_INVALID_CONTEXT,
CUDA_ERROR_INVALID_HANDLE, CUDA_ERROR_ALREADY_MAPPED,
CUDA_ERROR_UNKNOWN

Description

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 cuGraphicsMapResources() will complete before any subsequent CUDA work issued in `stream` begins.

If `resources` includes any duplicate entries then CUDA_ERROR_INVALID_HANDLE is returned. If any of `resources` are presently mapped for access by CUDA then CUDA_ERROR_ALREADY_MAPPED is returned.

> ▸ This function uses standard NULL stream semantics.

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

See also:

cuGraphicsResourceGetMappedPointer, cuGraphicsSubResourceGetMappedArray, cuGraphicsUnmapResources

CUresult cuGraphicsResourceGetMappedMipmappedArray (CUmipmappedArray *pMipmappedArray, CUgraphicsResource resource)

Get a mipmapped array through which to access a mapped graphics resource.

Parameters

pMipmappedArray
   - Returned mipmapped array through which `resource` may be accessed
resource
   - Mapped resource to access

Returns

CUDA_SUCCESS, CUDA_ERROR_DEINITIALIZED, CUDA_ERROR_NOT_INITIALIZED, CUDA_ERROR_INVALID_CONTEXT, CUDA_ERROR_INVALID_VALUE, CUDA_ERROR_INVALID_HANDLE, CUDA_ERROR_NOT_MAPPED, CUDA_ERROR_NOT_MAPPED_AS_ARRAY

Description

Returns in `*pMipmappedArray` a mipmapped array through which the mapped graphics resource `resource`. The value set in `*pMipmappedArray` may change every time that `resource` is mapped.

If `resource` is not a texture then it cannot be accessed via a mipmapped array and CUDA_ERROR_NOT_MAPPED_AS_ARRAY is returned. If `resource` is not mapped then CUDA_ERROR_NOT_MAPPED is returned.

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

See also:

cuGraphicsResourceGetMappedPointer

CUresult cuGraphicsResourceGetMappedPointer (CUdeviceptr *pDevPtr, size_t *pSize, CUgraphicsResource resource)

Get a device pointer through which to access a mapped graphics resource.

Parameters

pDevPtr
 - Returned pointer through which `resource` may be accessed
pSize
 - Returned size of the buffer accessible starting at `*pPointer`
resource
 - Mapped resource to access

Returns

CUDA_SUCCESS, CUDA_ERROR_DEINITIALIZED, CUDA_ERROR_NOT_INITIALIZED, CUDA_ERROR_INVALID_CONTEXT, CUDA_ERROR_INVALID_VALUE, CUDA_ERROR_INVALID_HANDLE, CUDA_ERROR_NOT_MAPPED, CUDA_ERROR_NOT_MAPPED_AS_POINTER

Description

Returns in `*pDevPtr` a pointer through which the mapped graphics resource `resource` may be accessed. Returns in `pSize` the size of the memory in bytes which may be accessed from that pointer. The value set in `pPointer` may change every time that `resource` is mapped.

If `resource` is not a buffer then it cannot be accessed via a pointer and CUDA_ERROR_NOT_MAPPED_AS_POINTER is returned. If `resource` is not mapped then CUDA_ERROR_NOT_MAPPED is returned. *

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

See also:

cuGraphicsMapResources, cuGraphicsSubResourceGetMappedArray

CUresult cuGraphicsResourceSetMapFlags (CUgraphicsResource resource, unsigned int flags)

Set usage flags for mapping a graphics resource.

Parameters

resource
- Registered resource to set flags for
flags
- Parameters for resource mapping

Returns

CUDA_SUCCESS, CUDA_ERROR_DEINITIALIZED,
CUDA_ERROR_NOT_INITIALIZED, CUDA_ERROR_INVALID_CONTEXT,
CUDA_ERROR_INVALID_VALUE, CUDA_ERROR_INVALID_HANDLE,
CUDA_ERROR_ALREADY_MAPPED

Description

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:

▶ CU_GRAPHICS_MAP_RESOURCE_FLAGS_NONE: 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.
▶ CU_GRAPHICS_MAP_RESOURCE_FLAGS_READONLY: Specifies that CUDA kernels which access this resource will not write to this resource.
▶ CU_GRAPHICS_MAP_RESOURCE_FLAGS_WRITEDISCARD: 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 `resource` is presently mapped for access by CUDA then CUDA_ERROR_ALREADY_MAPPED is returned. If `flags` is not one of the above values then CUDA_ERROR_INVALID_VALUE is returned.

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

See also:

cuGraphicsMapResources

CUresult cuGraphicsSubResourceGetMappedArray (CUarray *pArray, CUgraphicsResource resource, unsigned int arrayIndex, unsigned int mipLevel)

Get an array through which to access a subresource of a mapped graphics resource.

Parameters

pArray
   - 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
   CUarray_cubemap_face for cubemap textures for the subresource to access
mipLevel
   - Mipmap level for the subresource to access

Returns

CUDA_SUCCESS, CUDA_ERROR_DEINITIALIZED,
CUDA_ERROR_NOT_INITIALIZED, CUDA_ERROR_INVALID_CONTEXT,
CUDA_ERROR_INVALID_VALUE, CUDA_ERROR_INVALID_HANDLE,
CUDA_ERROR_NOT_MAPPED, CUDA_ERROR_NOT_MAPPED_AS_ARRAY

Description

Returns in `*pArray` 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 `*pArray` may change every time that `resource` is mapped.

If `resource` is not a texture then it cannot be accessed via an array and CUDA_ERROR_NOT_MAPPED_AS_ARRAY is returned. If `arrayIndex` is not a valid array index for `resource` then CUDA_ERROR_INVALID_VALUE is returned. If `mipLevel` is not a valid mipmap level for `resource` then CUDA_ERROR_INVALID_VALUE is returned. If `resource` is not mapped then CUDA_ERROR_NOT_MAPPED is returned.

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

See also:

cuGraphicsResourceGetMappedPointer

CUresult cuGraphicsUnmapResources (unsigned int count, CUgraphicsResource *resources, CUstream hStream)

Unmap graphics resources.

Parameters

count
- Number of resources to unmap
resources
- Resources to unmap
hStream
- Stream with which to synchronize

Returns

CUDA_SUCCESS, CUDA_ERROR_DEINITIALIZED,
CUDA_ERROR_NOT_INITIALIZED, CUDA_ERROR_INVALID_CONTEXT,
CUDA_ERROR_INVALID_HANDLE, CUDA_ERROR_NOT_MAPPED,
CUDA_ERROR_UNKNOWN

Description

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 cuGraphicsUnmapResources() will complete before any subsequently issued graphics work begins.

If `resources` includes any duplicate entries then CUDA_ERROR_INVALID_HANDLE is returned. If any of `resources` are not presently mapped for access by CUDA then CUDA_ERROR_NOT_MAPPED is returned.

> ▸ This function uses standard NULL stream semantics.
> ▸ Note that this function may also return error codes from previous, asynchronous launches.

See also:

cuGraphicsMapResources

CUresult cuGraphicsUnregisterResource (CUgraphicsResource resource)

Unregisters a graphics resource for access by CUDA.

Parameters

resource
   - Resource to unregister

Returns

CUDA_SUCCESS, CUDA_ERROR_DEINITIALIZED,
CUDA_ERROR_NOT_INITIALIZED, CUDA_ERROR_INVALID_CONTEXT,
CUDA_ERROR_INVALID_HANDLE, CUDA_ERROR_UNKNOWN

Description

Unregisters the graphics resource `resource` so it is not accessible by CUDA unless registered again.

If `resource` is invalid then CUDA_ERROR_INVALID_HANDLE is returned.

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

See also:

cuGraphicsD3D9RegisterResource, cuGraphicsD3D10RegisterResource, cuGraphicsD3D11RegisterResource, cuGraphicsGLRegisterBuffer, cuGraphicsGLRegisterImage

3.24. Profiler Control

This section describes the profiler control functions of the low-level CUDA driver application programming interface.

CUresult cuProfilerInitialize (const char *configFile, const char *outputFile, CUoutput_mode outputMode)

Initialize the profiling.

Parameters

configFile
   - Name of the config file that lists the counters/options for profiling.
outputFile
   - Name of the outputFile where the profiling results will be stored.
outputMode
   - outputMode, can be CU_OUT_KEY_VALUE_PAIR or CU_OUT_CSV.

Returns

CUDA_SUCCESS, CUDA_ERROR_INVALID_CONTEXT, CUDA_ERROR_INVALID_VALUE, CUDA_ERROR_PROFILER_DISABLED

Description

Using this API user can initialize the CUDA profiler by specifying the configuration file, output file and output file format. This API is generally used to profile different set of counters by looping the kernel launch. The `configFile` parameter can be used to select profiling options including profiler counters. Refer to the "Compute Command Line Profiler User Guide" for supported profiler options and counters.

Limitation: The CUDA profiler cannot be initialized with this API if another profiling tool is already active, as indicated by the CUDA_ERROR_PROFILER_DISABLED return code.

Typical usage of the profiling APIs is as follows:

for each set of counters/options { cuProfilerInitialize(); //Initialize profiling, set the counters or options in the config file ... cuProfilerStart(); // code to be profiled cuProfilerStop(); ... cuProfilerStart(); // code to be profiled cuProfilerStop(); ... }

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

See also:

cuProfilerStart, cuProfilerStop

CUresult cuProfilerStart (void)

Enable profiling.

Returns

CUDA_SUCCESS, CUDA_ERROR_INVALID_CONTEXT

Description

Enables profile collection by the active profiling tool for the current context. If profiling is already enabled, then cuProfilerStart() has no effect.

cuProfilerStart and cuProfilerStop APIs are used to programmatically control the profiling granularity by allowing profiling to be done only on selective pieces of code.

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

See also:

cuProfilerInitialize, cuProfilerStop

CUresult cuProfilerStop (void)

Disable profiling.

Returns

CUDA_SUCCESS, CUDA_ERROR_INVALID_CONTEXT

Description

Disables profile collection by the active profiling tool for the current context. If profiling is already disabled, then cuProfilerStop() has no effect.

cuProfilerStart and cuProfilerStop APIs are used to programmatically control the profiling granularity by allowing profiling to be done only on selective pieces of code.

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

See also:

cuProfilerInitialize, cuProfilerStart

3.25. OpenGL Interoperability

This section describes the OpenGL interoperability functions of the low-level CUDA driver application programming interface. Note that mapping of OpenGL resources is performed with the graphics API agnostic, resource mapping interface described in Graphics Interoperability.

OpenGL Interoperability [DEPRECATED]

enum CUGLDeviceList

CUDA devices corresponding to an OpenGL device

Values

CU_GL_DEVICE_LIST_ALL = 0x01
  The CUDA devices for all GPUs used by the current OpenGL context
CU_GL_DEVICE_LIST_CURRENT_FRAME = 0x02
  The CUDA devices for the GPUs used by the current OpenGL context in its currently rendering frame
CU_GL_DEVICE_LIST_NEXT_FRAME = 0x03
  The CUDA devices for the GPUs to be used by the current OpenGL context in the next frame

CUresult cuGLGetDevices (unsigned int *pCudaDeviceCount, CUdevice *pCudaDevices, unsigned int cudaDeviceCount, CUGLDeviceList deviceList)

Gets the CUDA devices associated with the current OpenGL context.

Parameters

pCudaDeviceCount
  - Returned number of CUDA devices.
pCudaDevices
  - Returned CUDA devices.
cudaDeviceCount
  - The size of the output device array pCudaDevices.
deviceList
  - The set of devices to return.

Returns

CUDA_SUCCESS, CUDA_ERROR_NO_DEVICE,
CUDA_ERROR_INVALID_VALUE, CUDA_ERROR_INVALID_CONTEXT,
CUDA_ERROR_INVALID_GRAPHICS_CONTEXT

Description

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 CUDA_ERROR_NO_DEVICE.

The `deviceList` argument may be any of the following:

▸ CU_GL_DEVICE_LIST_ALL: Query all devices used by the current OpenGL
context.

▸ CU_GL_DEVICE_LIST_CURRENT_FRAME: Query the devices used by the current
OpenGL context to render the current frame (in SLI).

▸ CU_GL_DEVICE_LIST_NEXT_FRAME: Query the devices used by the current
OpenGL context to render the next frame (in SLI). Note that this is a prediction, it
can't be guaranteed that this is correct in all cases.

> ▸ This function is not supported on Mac OS X.
> ▸ Note that this function may also return error codes from previous, asynchronous
>   launches.

See also:

cuWGLGetDevice

CUresult cuGraphicsGLRegisterBuffer (CUgraphicsResource *pCudaResource, GLuint buffer, unsigned int Flags)

Registers an OpenGL buffer object.

Parameters

pCudaResource
    - Pointer to the returned object handle
buffer
    - name of buffer object to be registered

Flags
- Register flags

Returns

CUDA_SUCCESS, CUDA_ERROR_INVALID_HANDLE,
CUDA_ERROR_ALREADY_MAPPED, CUDA_ERROR_INVALID_CONTEXT,

Description

Registers the buffer object specified by `buffer` for access by CUDA. A handle to the registered object is returned as `pCudaResource`. The register flags `Flags` specify the intended usage, as follows:

▶ CU_GRAPHICS_REGISTER_FLAGS_NONE: 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.

▶ CU_GRAPHICS_REGISTER_FLAGS_READ_ONLY: Specifies that CUDA will not write to this resource.

▶ CU_GRAPHICS_REGISTER_FLAGS_WRITE_DISCARD: 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.

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

See also:

cuGraphicsUnregisterResource, cuGraphicsMapResources,
cuGraphicsResourceGetMappedPointer

CUresult cuGraphicsGLRegisterImage (CUgraphicsResource *pCudaResource, GLuint image, GLenum target, unsigned int Flags)

Register an OpenGL texture or renderbuffer object.

Parameters

pCudaResource
- 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

CUDA_SUCCESS, CUDA_ERROR_INVALID_HANDLE,
CUDA_ERROR_ALREADY_MAPPED, CUDA_ERROR_INVALID_CONTEXT,

Description

Registers the texture or renderbuffer object specified by `image` for access by CUDA. A handle to the registered object is returned as `pCudaResource`.

`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:

▸ CU_GRAPHICS_REGISTER_FLAGS_NONE: 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.

▸ CU_GRAPHICS_REGISTER_FLAGS_READ_ONLY: Specifies that CUDA will not write to this resource.

▸ CU_GRAPHICS_REGISTER_FLAGS_WRITE_DISCARD: 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.

▸ CU_GRAPHICS_REGISTER_FLAGS_SURFACE_LDST: Specifies that CUDA will bind this resource to a surface reference.

▸ CU_GRAPHICS_REGISTER_FLAGS_TEXTURE_GATHER: 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

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

See also:

cuGraphicsUnregisterResource, cuGraphicsMapResources, cuGraphicsSubResourceGetMappedArray

CUresult cuWGLGetDevice (CUdevice *pDevice, HGPUNV hGpu)

Gets the CUDA device associated with hGpu.

Parameters

pDevice
   - Device associated with hGpu
hGpu
   - Handle to a GPU, as queried via WGL_NV_gpu_affinity()

Returns

CUDA_SUCCESS, CUDA_ERROR_DEINITIALIZED, CUDA_ERROR_NOT_INITIALIZED, CUDA_ERROR_INVALID_CONTEXT, CUDA_ERROR_INVALID_VALUE

Description

Returns in `*pDevice` the CUDA device associated with a `hGpu`, if applicable.

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

See also:

cuGLMapBufferObject, cuGLRegisterBufferObject, cuGLUnmapBufferObject, cuGLUnregisterBufferObject, cuGLUnmapBufferObjectAsync, cuGLSetBufferObjectMapFlags

3.25.1. OpenGL Interoperability [DEPRECATED]

OpenGL Interoperability

This section describes deprecated OpenGL interoperability functionality.

enum CUGLmap_flags

Flags to map or unmap a resource

Values

CU_GL_MAP_RESOURCE_FLAGS_NONE = 0x00
CU_GL_MAP_RESOURCE_FLAGS_READ_ONLY = 0x01
CU_GL_MAP_RESOURCE_FLAGS_WRITE_DISCARD = 0x02

CUresult cuGLCtxCreate (CUcontext *pCtx, unsigned int Flags, CUdevice device)

Create a CUDA context for interoperability with OpenGL.

Parameters

pCtx
   - Returned CUDA context
Flags
   - Options for CUDA context creation
device
   - Device on which to create the context

Returns

CUDA_SUCCESS, CUDA_ERROR_DEINITIALIZED,
CUDA_ERROR_NOT_INITIALIZED, CUDA_ERROR_INVALID_CONTEXT,
CUDA_ERROR_INVALID_VALUE, CUDA_ERROR_OUT_OF_MEMORY

Description

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 context with an OpenGL context in order to achieve maximum interoperability performance.

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

See also:

cuCtxCreate, cuGLInit, cuGLMapBufferObject, cuGLRegisterBufferObject, cuGLUnmapBufferObject, cuGLUnregisterBufferObject, cuGLMapBufferObjectAsync, cuGLUnmapBufferObjectAsync, cuGLSetBufferObjectMapFlags, cuWGLGetDevice

CUresult cuGLInit (void)

Initializes OpenGL interoperability.

Returns

CUDA_SUCCESS, CUDA_ERROR_DEINITIALIZED, CUDA_ERROR_NOT_INITIALIZED, CUDA_ERROR_INVALID_CONTEXT, CUDA_ERROR_UNKNOWN

Description

Deprecated This function is deprecated as of Cuda 3.0.

Initializes OpenGL interoperability. This function is deprecated and calling it is no longer required. It may fail if the needed OpenGL driver facilities are not available.

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

See also:

cuGLMapBufferObject, cuGLRegisterBufferObject, cuGLUnmapBufferObject, cuGLUnregisterBufferObject, cuGLMapBufferObjectAsync, cuGLUnmapBufferObjectAsync, cuGLSetBufferObjectMapFlags, cuWGLGetDevice

CUresult cuGLMapBufferObject (CUdeviceptr *dptr, size_t *size, GLuint buffer)

Maps an OpenGL buffer object.

Parameters

dptr
   - Returned mapped base pointer
size
   - Returned size of mapping
buffer
   - The name of the buffer object to map

Returns

CUDA_SUCCESS, CUDA_ERROR_DEINITIALIZED,
CUDA_ERROR_NOT_INITIALIZED, CUDA_ERROR_INVALID_CONTEXT,
CUDA_ERROR_INVALID_VALUE, CUDA_ERROR_MAP_FAILED

Description

Deprecated This function is deprecated as of Cuda 3.0.

Maps the buffer object specified by `buffer` into the address space of the current CUDA context and returns in `*dptr` and `*size` the base pointer and size of the resulting mapping.

There must be a valid OpenGL context bound to the current thread when this function is called. This must be the same context, or a member of the same shareGroup, as the context that was bound when the buffer was registered.

All streams in the current CUDA context are synchronized with the current GL context.

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

See also:

cuGraphicsMapResources

CUresult cuGLMapBufferObjectAsync (CUdeviceptr *dptr, size_t *size, GLuint buffer, CUstream hStream)

Maps an OpenGL buffer object.

Parameters

dptr
 - Returned mapped base pointer
size
 - Returned size of mapping
buffer
 - The name of the buffer object to map
hStream
 - Stream to synchronize

Returns

CUDA_SUCCESS, CUDA_ERROR_DEINITIALIZED,
CUDA_ERROR_NOT_INITIALIZED, CUDA_ERROR_INVALID_CONTEXT,
CUDA_ERROR_INVALID_VALUE, CUDA_ERROR_MAP_FAILED

Description

Deprecated This function is deprecated as of Cuda 3.0.

Maps the buffer object specified by `buffer` into the address space of the current CUDA context and returns in `*dptr` and `*size` the base pointer and size of the resulting mapping.

There must be a valid OpenGL context bound to the current thread when this function is called. This must be the same context, or a member of the same shareGroup, as the context that was bound when the buffer was registered.

Stream `hStream` in the current CUDA context is synchronized with the current GL context.

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

See also:

cuGraphicsMapResources

CUresult cuGLRegisterBufferObject (GLuint buffer)
Registers an OpenGL buffer object.

Parameters

buffer
   - The name of the buffer object to register.

Returns

CUDA_SUCCESS, CUDA_ERROR_DEINITIALIZED,
CUDA_ERROR_NOT_INITIALIZED, CUDA_ERROR_INVALID_CONTEXT,
CUDA_ERROR_ALREADY_MAPPED

Description

Deprecated This function is deprecated as of Cuda 3.0.

Registers the buffer object specified by `buffer` for access by CUDA. This function must be called before CUDA can map the buffer object. There must be a valid OpenGL context bound to the current thread when this function is called, and the buffer name is resolved by that context.

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

See also:

cuGraphicsGLRegisterBuffer

CUresult cuGLSetBufferObjectMapFlags (GLuint buffer, unsigned int Flags)

Set the map flags for an OpenGL buffer object.

Parameters

buffer
   - Buffer object to unmap
Flags
   - Map flags

Returns

CUDA_SUCCESS, CUDA_ERROR_NOT_INITIALIZED, CUDA_ERROR_INVALID_HANDLE, CUDA_ERROR_ALREADY_MAPPED, CUDA_ERROR_INVALID_CONTEXT,

Description

Deprecated This function is deprecated as of Cuda 3.0.

Sets the map flags for the buffer object specified by `buffer`.

Changes to `Flags` will take effect the next time `buffer` is mapped. The `Flags` argument may be any of the following:

▸ CU_GL_MAP_RESOURCE_FLAGS_NONE: 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.
▸ CU_GL_MAP_RESOURCE_FLAGS_READ_ONLY: Specifies that CUDA kernels which access this resource will not write to this resource.
▸ CU_GL_MAP_RESOURCE_FLAGS_WRITE_DISCARD: 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 `buffer` has not been registered for use with CUDA, then CUDA_ERROR_INVALID_HANDLE is returned. If `buffer` is presently mapped for access by CUDA, then CUDA_ERROR_ALREADY_MAPPED is returned.

There must be a valid OpenGL context bound to the current thread when this function is called. This must be the same context, or a member of the same shareGroup, as the context that was bound when the buffer was registered.

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

See also:

cuGraphicsResourceSetMapFlags

CUresult cuGLUnmapBufferObject (GLuint buffer)

Unmaps an OpenGL buffer object.

Parameters

buffer
   - Buffer object to unmap

Returns

CUDA_SUCCESS, CUDA_ERROR_DEINITIALIZED, CUDA_ERROR_NOT_INITIALIZED, CUDA_ERROR_INVALID_CONTEXT, CUDA_ERROR_INVALID_VALUE

Description

Deprecated This function is deprecated as of Cuda 3.0.

Unmaps the buffer object specified by `buffer` for access by CUDA.

There must be a valid OpenGL context bound to the current thread when this function is called. This must be the same context, or a member of the same shareGroup, as the context that was bound when the buffer was registered.

All streams in the current CUDA context are synchronized with the current GL context.

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

See also:

cuGraphicsUnmapResources

CUresult cuGLUnmapBufferObjectAsync (GLuint buffer, CUstream hStream)

Unmaps an OpenGL buffer object.

Parameters

buffer
- Name of the buffer object to unmap
hStream
- Stream to synchronize

Returns

CUDA_SUCCESS, CUDA_ERROR_DEINITIALIZED,
CUDA_ERROR_NOT_INITIALIZED, CUDA_ERROR_INVALID_CONTEXT,
CUDA_ERROR_INVALID_VALUE

Description

Deprecated This function is deprecated as of Cuda 3.0.

Unmaps the buffer object specified by `buffer` for access by CUDA.

There must be a valid OpenGL context bound to the current thread when this function is called. This must be the same context, or a member of the same shareGroup, as the context that was bound when the buffer was registered.

Stream `hStream` in the current CUDA context is synchronized with the current GL context.

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

See also:

cuGraphicsUnmapResources

CUresult cuGLUnregisterBufferObject (GLuint buffer)

Unregister an OpenGL buffer object.

Parameters

buffer
   - Name of the buffer object to unregister

Returns

CUDA_SUCCESS, CUDA_ERROR_DEINITIALIZED,
CUDA_ERROR_NOT_INITIALIZED, CUDA_ERROR_INVALID_CONTEXT,
CUDA_ERROR_INVALID_VALUE

Description

Deprecated This function is deprecated as of Cuda 3.0.

Unregisters the buffer object specified by `buffer`. This releases any resources associated with the registered buffer. After this call, the buffer may no longer be mapped for access by CUDA.

There must be a valid OpenGL context bound to the current thread when this function is called. This must be the same context, or a member of the same shareGroup, as the context that was bound when the buffer was registered.

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

See also:

cuGraphicsUnregisterResource

3.26. Direct3D 9 Interoperability

This section describes the Direct3D 9 interoperability functions of the low-level CUDA driver application programming interface. Note that mapping of Direct3D 9 resources is performed with the graphics API agnostic, resource mapping interface described in Graphics Interoperability.

Direct3D 9 Interoperability [DEPRECATED]

enum CUd3d9DeviceList

CUDA devices corresponding to a D3D9 device

Values

CU_D3D9_DEVICE_LIST_ALL = 0x01
   The CUDA devices for all GPUs used by a D3D9 device
CU_D3D9_DEVICE_LIST_CURRENT_FRAME = 0x02
   The CUDA devices for the GPUs used by a D3D9 device in its currently rendering
   frame
CU_D3D9_DEVICE_LIST_NEXT_FRAME = 0x03
   The CUDA devices for the GPUs to be used by a D3D9 device in the next frame

CUresult cuD3D9CtxCreate (CUcontext *pCtx, CUdevice *pCudaDevice, unsigned int Flags, IDirect3DDevice9 *pD3DDevice)

Create a CUDA context for interoperability with Direct3D 9.

Parameters

pCtx
   - Returned newly created CUDA context
pCudaDevice
   - Returned pointer to the device on which the context was created
Flags
   - Context creation flags (see cuCtxCreate() for details)
pD3DDevice
   - Direct3D device to create interoperability context with

Returns

CUDA_SUCCESS, CUDA_ERROR_DEINITIALIZED,
CUDA_ERROR_NOT_INITIALIZED, CUDA_ERROR_INVALID_VALUE,
CUDA_ERROR_OUT_OF_MEMORY, CUDA_ERROR_UNKNOWN

Description

Creates a new CUDA context, enables interoperability for that context with the Direct3D
device `pD3DDevice`, and associates the created CUDA context with the calling thread.
The created CUcontext will be returned in `*pCtx`. Direct3D resources from this
device may be registered and mapped through the lifetime of this CUDA context. If
`pCudaDevice` is non-NULL then the CUdevice on which this CUDA context was
created will be returned in `*pCudaDevice`.

On success, this call will increase the internal reference count on `pD3DDevice`. This reference count will be decremented upon destruction of this context through cuCtxDestroy(). This context will cease to function if `pD3DDevice` is destroyed or encounters an error.

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 circumstances, this function is not necessary.

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

See also:

cuD3D9GetDevice, cuGraphicsD3D9RegisterResource

CUresult cuD3D9CtxCreateOnDevice (CUcontext *pCtx, unsigned int flags, IDirect3DDevice9 *pD3DDevice, CUdevice cudaDevice)

Create a CUDA context for interoperability with Direct3D 9.

Parameters

pCtx
  - Returned newly created CUDA context
flags
  - Context creation flags (see cuCtxCreate() for details)
pD3DDevice
  - Direct3D device to create interoperability context with
cudaDevice
  - The CUDA device on which to create the context. This device must be among the devices returned when querying CU_D3D9_DEVICES_ALL from cuD3D9GetDevices.

Returns

CUDA_SUCCESS, CUDA_ERROR_DEINITIALIZED, CUDA_ERROR_NOT_INITIALIZED, CUDA_ERROR_INVALID_VALUE, CUDA_ERROR_OUT_OF_MEMORY, CUDA_ERROR_UNKNOWN

Description

Creates a new CUDA context, enables interoperability for that context with the Direct3D device `pD3DDevice`, and associates the created CUDA context with the calling thread.

The created CUcontext will be returned in *pCtx. Direct3D resources from this device may be registered and mapped through the lifetime of this CUDA context.

On success, this call will increase the internal reference count on pD3DDevice. This reference count will be decremented upon destruction of this context through cuCtxDestroy(). This context will cease to function if pD3DDevice is destroyed or encounters an error.

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 circumstances, this function is not necessary.

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

See also:

cuD3D9GetDevices, cuGraphicsD3D9RegisterResource

CUresult cuD3D9GetDevice (CUdevice *pCudaDevice, const char *pszAdapterName)

Gets the CUDA device corresponding to a display adapter.

Parameters

pCudaDevice
  - Returned CUDA device corresponding to pszAdapterName
pszAdapterName
  - Adapter name to query for device

Returns

CUDA_SUCCESS, CUDA_ERROR_DEINITIALIZED, CUDA_ERROR_NOT_INITIALIZED, CUDA_ERROR_INVALID_VALUE, CUDA_ERROR_NOT_FOUND, CUDA_ERROR_UNKNOWN

Description

Returns in *pCudaDevice 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.

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

See also:

cuD3D9CtxCreate

CUresult cuD3D9GetDevices (unsigned int *pCudaDeviceCount, CUdevice *pCudaDevices, unsigned int cudaDeviceCount, IDirect3DDevice9 *pD3D9Device, CUd3d9DeviceList deviceList)

Gets the CUDA devices corresponding to a Direct3D 9 device.

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 CU_D3D9_DEVICE_LIST_ALL for all devices, CU_D3D9_DEVICE_LIST_CURRENT_FRAME for the devices used to render the current frame (in SLI), or CU_D3D9_DEVICE_LIST_NEXT_FRAME for the devices used to render the next frame (in SLI).

Returns

CUDA_SUCCESS, CUDA_ERROR_DEINITIALIZED, CUDA_ERROR_NOT_INITIALIZED, CUDA_ERROR_NO_DEVICE, CUDA_ERROR_INVALID_VALUE, CUDA_ERROR_NOT_FOUND, CUDA_ERROR_UNKNOWN

Description

Returns in `*pCudaDeviceCount` the number of CUDA-compatible device corresponding to the Direct3D 9 device `pD3D9Device`. Also returns in `*pCudaDevices` at most `cudaDeviceCount` of 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 CUDA_ERROR_NO_DEVICE.

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

See also:

cuD3D9CtxCreate

CUresult cuD3D9GetDirect3DDevice (IDirect3DDevice9 **ppD3DDevice)

Get the Direct3D 9 device against which the current CUDA context was created.

Parameters

ppD3DDevice
  - Returned Direct3D device corresponding to CUDA context

Returns

CUDA_SUCCESS, CUDA_ERROR_DEINITIALIZED,
CUDA_ERROR_NOT_INITIALIZED, CUDA_ERROR_INVALID_CONTEXT

Description

Returns in `*ppD3DDevice` the Direct3D device against which this CUDA context was created in cuD3D9CtxCreate().

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

See also:

cuD3D9GetDevice

CUresult cuGraphicsD3D9RegisterResource (CUgraphicsResource *pCudaResource, IDirect3DResource9 *pD3DResource, unsigned int Flags)

Register a Direct3D 9 resource for access by CUDA.

Parameters

pCudaResource
- Returned graphics resource handle
pD3DResource
- Direct3D resource to register
Flags
- Parameters for resource registration

Returns

CUDA_SUCCESS, CUDA_ERROR_DEINITIALIZED, CUDA_ERROR_NOT_INITIALIZED, CUDA_ERROR_INVALID_CONTEXT, CUDA_ERROR_INVALID_VALUE, CUDA_ERROR_INVALID_HANDLE, CUDA_ERROR_OUT_OF_MEMORY, CUDA_ERROR_UNKNOWN

Description

Registers the Direct3D 9 resource `pD3DResource` for access by CUDA and returns a CUDA handle to `pD3Dresource` in `pCudaResource`. The handle returned in `pCudaResource` may be used to map and unmap this resource until it is unregistered. On success this call will increase the internal reference count on `pD3DResource`. This reference count will be decremented when this resource is unregistered through cuGraphicsUnregisterResource().

This call is potentially 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

- CU_GRAPHICS_REGISTER_FLAGS_NONE: Specifies no hints about how this resource will be used.
- CU_GRAPHICS_REGISTER_FLAGS_SURFACE_LDST: Specifies that CUDA will bind this resource to a surface reference.
- CU_GRAPHICS_REGISTER_FLAGS_TEXTURE_GATHER: 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 Direct3D interoperability is not initialized for this context using cuD3D9CtxCreate then CUDA_ERROR_INVALID_CONTEXT is returned. If `pD3DResource` is of incorrect type or is already registered then CUDA_ERROR_INVALID_HANDLE is returned. If `pD3DResource` cannot be registered then CUDA_ERROR_UNKNOWN is returned. If

`Flags` is not one of the above specified value then CUDA_ERROR_INVALID_VALUE is returned.

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

See also:

cuD3D9CtxCreate, cuGraphicsUnregisterResource, cuGraphicsMapResources, cuGraphicsSubResourceGetMappedArray, cuGraphicsResourceGetMappedPointer

3.26.1. Direct3D 9 Interoperability [DEPRECATED]

Direct3D 9 Interoperability

This section describes deprecated Direct3D 9 interoperability functionality.

enum CUd3d9map_flags

Flags to map or unmap a resource

Values

CU_D3D9_MAPRESOURCE_FLAGS_NONE = 0x00
CU_D3D9_MAPRESOURCE_FLAGS_READONLY = 0x01
CU_D3D9_MAPRESOURCE_FLAGS_WRITEDISCARD = 0x02

enum CUd3d9register_flags

Flags to register a resource

Values

CU_D3D9_REGISTER_FLAGS_NONE = 0x00
CU_D3D9_REGISTER_FLAGS_ARRAY = 0x01

CUresult cuD3D9MapResources (unsigned int count, IDirect3DResource9 **ppResource)

Map Direct3D resources for access by CUDA.

Parameters

count
 - Number of resources in ppResource
ppResource
 - Resources to map for CUDA usage

Returns

CUDA_SUCCESS, CUDA_ERROR_DEINITIALIZED,
CUDA_ERROR_NOT_INITIALIZED, CUDA_ERROR_INVALID_CONTEXT,
CUDA_ERROR_INVALID_HANDLE, CUDA_ERROR_ALREADY_MAPPED,
CUDA_ERROR_UNKNOWN

Description

Deprecated This function is deprecated as of CUDA 3.0.

Maps the `count` Direct3D resources in `ppResource` for access by CUDA.

The resources in `ppResource` 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 cuD3D9MapResources() will complete before any CUDA kernels issued after cuD3D9MapResources() begin.

If any of `ppResource` have not been registered for use with CUDA or if `ppResource` contains any duplicate entries, then CUDA_ERROR_INVALID_HANDLE is returned. If any of `ppResource` are presently mapped for access by CUDA, then CUDA_ERROR_ALREADY_MAPPED is returned.

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

See also:

cuGraphicsMapResources

CUresult cuD3D9RegisterResource (IDirect3DResource9 *pResource, unsigned int Flags)

Register a Direct3D resource for access by CUDA.

Parameters

pResource
   - Resource to register for CUDA access
Flags
   - Flags for resource registration

Returns

CUDA_SUCCESS, CUDA_ERROR_DEINITIALIZED,
CUDA_ERROR_NOT_INITIALIZED, CUDA_ERROR_INVALID_CONTEXT,
CUDA_ERROR_INVALID_VALUE, CUDA_ERROR_INVALID_HANDLE,
CUDA_ERROR_OUT_OF_MEMORY, CUDA_ERROR_UNKNOWN

Description

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 cuD3D9UnregisterResource(). 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 cuD3D9UnregisterResource().

This call is potentially high-overhead and should not be called every frame in interactive applications.

The type of `pResource` must be one of the following.

▶ IDirect3DVertexBuffer9: Cannot be used with `Flags` set to
   CU_D3D9_REGISTER_FLAGS_ARRAY.
▶ IDirect3DIndexBuffer9: Cannot be used with `Flags` set to
   CU_D3D9_REGISTER_FLAGS_ARRAY.
▶ 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. For restrictions on the `Flags`
   parameter, see type IDirect3DBaseTexture9.
▶ IDirect3DBaseTexture9: When a texture is registered, all surfaces associated with the
   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 values are allowed.

▶ CU_D3D9_REGISTER_FLAGS_NONE: Specifies that CUDA will access
   this resource through a CUdeviceptr. The pointer, size, and (for textures),
   pitch for each subresource of this allocation may be queried through
   cuD3D9ResourceGetMappedPointer(), cuD3D9ResourceGetMappedSize(), and
   cuD3D9ResourceGetMappedPitch() respectively. This option is valid for all resource
   types.
▶ CU_D3D9_REGISTER_FLAGS_ARRAY: Specifies that CUDA will access
   this resource through a CUarray queried on a sub-resource basis through
   cuD3D9ResourceGetMappedArray(). This option is only valid for resources of type
   IDirect3DSurface9 and subtypes of IDirect3DBaseTexture9.

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 CUDA_ERROR_INVALID_CONTEXT is returned. If `pResource` is of incorrect type (e.g. is a non-stand-alone IDirect3DSurface9) or is already registered, then CUDA_ERROR_INVALID_HANDLE is returned. If `pResource` cannot be registered then CUDA_ERROR_UNKNOWN is returned.

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

See also:

cuGraphicsD3D9RegisterResource

CUresult cuD3D9ResourceGetMappedArray (CUarray *pArray, 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.

Parameters

pArray
- Returned array corresponding to subresource
pResource
- Mapped resource to access
Face
- Face of resource to access
Level
- Level of resource to access

Returns

CUDA_SUCCESS, CUDA_ERROR_DEINITIALIZED,
CUDA_ERROR_NOT_INITIALIZED, CUDA_ERROR_INVALID_CONTEXT,
CUDA_ERROR_INVALID_VALUE, CUDA_ERROR_INVALID_HANDLE,
CUDA_ERROR_NOT_MAPPED

Description

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 CUDA_ERROR_INVALID_HANDLE is returned. If `pResource` was not registered with usage flags CU_D3D9_REGISTER_FLAGS_ARRAY then CUDA_ERROR_INVALID_HANDLE is returned. If `pResource` is not mapped then CUDA_ERROR_NOT_MAPPED is returned.

For usage requirements of `Face` and `Level` parameters, see cuD3D9ResourceGetMappedPointer().

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

See also:

cuGraphicsSubResourceGetMappedArray

CUresult cuD3D9ResourceGetMappedPitch (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.

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

CUDA_SUCCESS, CUDA_ERROR_DEINITIALIZED,
CUDA_ERROR_NOT_INITIALIZED, CUDA_ERROR_INVALID_CONTEXT,
CUDA_ERROR_INVALID_VALUE, CUDA_ERROR_INVALID_HANDLE,
CUDA_ERROR_NOT_MAPPED

Description

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 CU_D3D9_REGISTER_FLAGS_NONE, then CUDA_ERROR_INVALID_HANDLE is returned. If `pResource` is not mapped for access by CUDA then CUDA_ERROR_NOT_MAPPED is returned.

For usage requirements of `Face` and `Level` parameters, see cuD3D9ResourceGetMappedPointer().

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

See also:

cuGraphicsSubResourceGetMappedArray

CUresult cuD3D9ResourceGetMappedPointer (CUdeviceptr *pDevPtr, IDirect3DResource9 *pResource, unsigned int Face, unsigned int Level)

Get the pointer through which to access a subresource of a Direct3D resource which has been mapped for access by CUDA.

Parameters

pDevPtr
   - Returned pointer corresponding to subresource
pResource
   - Mapped resource to access
Face
   - Face of resource to access
Level
   - Level of resource to access

Returns

CUDA_SUCCESS, CUDA_ERROR_DEINITIALIZED, CUDA_ERROR_NOT_INITIALIZED, CUDA_ERROR_INVALID_CONTEXT, CUDA_ERROR_INVALID_VALUE, CUDA_ERROR_INVALID_HANDLE, CUDA_ERROR_NOT_MAPPED

Description

Deprecated This function is deprecated as of CUDA 3.0.

Returns in `*pDevPtr` the base pointer of the subresource of the mapped Direct3D resource `pResource`, which corresponds to `Face` and `Level`. The value set in `pDevPtr` may change every time that `pResource` is mapped.

If `pResource` is not registered, then CUDA_ERROR_INVALID_HANDLE is returned. If `pResource` was not registered with usage flags CU_D3D9_REGISTER_FLAGS_NONE, then CUDA_ERROR_INVALID_HANDLE is returned. If `pResource` is not mapped, then CUDA_ERROR_NOT_MAPPED 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 CUDA_ERROR_INVALID_VALUE is returned.

If `pResource` is of type IDirect3DBaseTexture9, then `Level` must correspond to a valid mipmap level. At present only mipmap level 0 is supported. For all other types `Level` must be 0. If `Level` is invalid, then CUDA_ERROR_INVALID_VALUE is returned.

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

See also:

cuGraphicsResourceGetMappedPointer

CUresult cuD3D9ResourceGetMappedSize (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.

Parameters

pSize
  - Returned size of subresource
pResource
  - Mapped resource to access
Face
  - Face of resource to access
Level
  - Level of resource to access

Returns

CUDA_SUCCESS, CUDA_ERROR_DEINITIALIZED, CUDA_ERROR_NOT_INITIALIZED, CUDA_ERROR_INVALID_CONTEXT, CUDA_ERROR_INVALID_VALUE, CUDA_ERROR_INVALID_HANDLE, CUDA_ERROR_NOT_MAPPED

Description

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 CUDA_ERROR_INVALID_HANDLE is returned. If `pResource` was not

registered with usage flags CU_D3D9_REGISTER_FLAGS_NONE, then
CUDA_ERROR_INVALID_HANDLE is returned. If pResource is not mapped for
access by CUDA, then CUDA_ERROR_NOT_MAPPED is returned.

For usage requirements of `Face` and `Level` parameters, see
cuD3D9ResourceGetMappedPointer.

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

See also:

cuGraphicsResourceGetMappedPointer

CUresult cuD3D9ResourceGetSurfaceDimensions (size_t *pWidth, size_t *pHeight, size_t *pDepth, IDirect3DResource9 *pResource, unsigned int Face, unsigned int Level)

Get the dimensions of a registered surface.

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

CUDA_SUCCESS, CUDA_ERROR_DEINITIALIZED,
CUDA_ERROR_NOT_INITIALIZED, CUDA_ERROR_INVALID_CONTEXT,
CUDA_ERROR_INVALID_VALUE, CUDA_ERROR_INVALID_HANDLE

Description

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`.

Because 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 CUDA_ERROR_INVALID_HANDLE is returned.

For usage requirements of `Face` and `Level` parameters, see cuD3D9ResourceGetMappedPointer().

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

See also:

cuGraphicsSubResourceGetMappedArray

CUresult cuD3D9ResourceSetMapFlags (IDirect3DResource9 *pResource, unsigned int Flags)

Set usage flags for mapping a Direct3D resource.

Parameters

pResource
  - Registered resource to set flags for
Flags
  - Parameters for resource mapping

Returns

CUDA_SUCCESS, CUDA_ERROR_DEINITIALIZED, CUDA_ERROR_NOT_INITIALIZED, CUDA_ERROR_INVALID_CONTEXT, CUDA_ERROR_INVALID_VALUE, CUDA_ERROR_INVALID_HANDLE, CUDA_ERROR_ALREADY_MAPPED

Description

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:

▸ CU_D3D9_MAPRESOURCE_FLAGS_NONE: 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.

▸ CU_D3D9_MAPRESOURCE_FLAGS_READONLY: Specifies that CUDA kernels which access this resource will not write to this resource.

▸ CU_D3D9_MAPRESOURCE_FLAGS_WRITEDISCARD: 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 CUDA_ERROR_INVALID_HANDLE is returned. If `pResource` is presently mapped for access by CUDA, then CUDA_ERROR_ALREADY_MAPPED is returned.

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

See also:

cuGraphicsResourceSetMapFlags

CUresult cuD3D9UnmapResources (unsigned int count, IDirect3DResource9 **ppResource)

Unmaps Direct3D resources.

Parameters

count
   - Number of resources to unmap for CUDA
ppResource
   - Resources to unmap for CUDA

Returns

CUDA_SUCCESS, CUDA_ERROR_DEINITIALIZED, CUDA_ERROR_NOT_INITIALIZED, CUDA_ERROR_INVALID_CONTEXT, CUDA_ERROR_INVALID_HANDLE, CUDA_ERROR_NOT_MAPPED, CUDA_ERROR_UNKNOWN

Description

Deprecated This function is deprecated as of CUDA 3.0.

Unmaps the `count` Direct3D resources in `ppResource`.

This function provides the synchronization guarantee that any CUDA kernels issued before cuD3D9UnmapResources() will complete before any Direct3D calls issued after cuD3D9UnmapResources() begin.

If any of `ppResource` have not been registered for use with CUDA or if `ppResource` contains any duplicate entries, then CUDA_ERROR_INVALID_HANDLE is returned. If any of `ppResource` are not presently mapped for access by CUDA, then CUDA_ERROR_NOT_MAPPED is returned.

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

See also:

cuGraphicsUnmapResources

CUresult cuD3D9UnregisterResource (IDirect3DResource9 *pResource)

Unregister a Direct3D resource.

Parameters

pResource
   - Resource to unregister

Returns

CUDA_SUCCESS, CUDA_ERROR_DEINITIALIZED, CUDA_ERROR_NOT_INITIALIZED, CUDA_ERROR_INVALID_CONTEXT, CUDA_ERROR_INVALID_HANDLE, CUDA_ERROR_UNKNOWN

Description

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 CUDA_ERROR_INVALID_HANDLE is returned.

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

See also:

cuGraphicsUnregisterResource

3.27. Direct3D 10 Interoperability

This section describes the Direct3D 10 interoperability functions of the low-level CUDA driver application programming interface. Note that mapping of Direct3D 10 resources is performed with the graphics API agnostic, resource mapping interface described in Graphics Interoperability.

Direct3D 10 Interoperability [DEPRECATED]

enum CUd3d10DeviceList

CUDA devices corresponding to a D3D10 device

Values

CU_D3D10_DEVICE_LIST_ALL = 0x01
  The CUDA devices for all GPUs used by a D3D10 device
CU_D3D10_DEVICE_LIST_CURRENT_FRAME = 0x02
  The CUDA devices for the GPUs used by a D3D10 device in its currently rendering frame
CU_D3D10_DEVICE_LIST_NEXT_FRAME = 0x03
  The CUDA devices for the GPUs to be used by a D3D10 device in the next frame

CUresult cuD3D10GetDevice (CUdevice *pCudaDevice, IDXGIAdapter *pAdapter)

Gets the CUDA device corresponding to a display adapter.

Parameters

pCudaDevice
  - Returned CUDA device corresponding to `pAdapter`
pAdapter
  - Adapter to query for CUDA device

Returns

CUDA_SUCCESS, CUDA_ERROR_DEINITIALIZED,
CUDA_ERROR_NOT_INITIALIZED, CUDA_ERROR_INVALID_VALUE,
CUDA_ERROR_NOT_FOUND, CUDA_ERROR_UNKNOWN

Description

Returns in `*pCudaDevice` the CUDA-compatible device corresponding to the adapter `pAdapter` obtained from IDXGIFactory::EnumAdapters.

If no device on `pAdapter` is CUDA-compatible then the call will fail.

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

See also:

cuD3D10GetDevices

CUresult cuD3D10GetDevices (unsigned int *pCudaDeviceCount, CUdevice *pCudaDevices, unsigned int cudaDeviceCount, ID3D10Device *pD3D10Device, CUd3d10DeviceList deviceList)

Gets the CUDA devices corresponding to a Direct3D 10 device.

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 CU_D3D10_DEVICE_LIST_ALL for all devices, CU_D3D10_DEVICE_LIST_CURRENT_FRAME for the devices used to render the current frame (in SLI), or CU_D3D10_DEVICE_LIST_NEXT_FRAME for the devices used to render the next frame (in SLI).

Returns

CUDA_SUCCESS, CUDA_ERROR_DEINITIALIZED, CUDA_ERROR_NOT_INITIALIZED, CUDA_ERROR_NO_DEVICE, CUDA_ERROR_INVALID_VALUE, CUDA_ERROR_NOT_FOUND, CUDA_ERROR_UNKNOWN

Description

Returns in `*pCudaDeviceCount` the number of CUDA-compatible device corresponding to the Direct3D 10 device `pD3D10Device`. Also returns in `*pCudaDevices` at most `cudaDeviceCount` of 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 CUDA_ERROR_NO_DEVICE.

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

See also:

cuD3D10GetDevice

CUresult cuGraphicsD3D10RegisterResource (CUgraphicsResource *pCudaResource, ID3D10Resource *pD3DResource, unsigned int Flags)

Register a Direct3D 10 resource for access by CUDA.

Parameters

pCudaResource
   - Returned graphics resource handle
pD3DResource
   - Direct3D resource to register
Flags
   - Parameters for resource registration

Returns

CUDA_SUCCESS, CUDA_ERROR_DEINITIALIZED, CUDA_ERROR_NOT_INITIALIZED, CUDA_ERROR_INVALID_CONTEXT, CUDA_ERROR_INVALID_VALUE, CUDA_ERROR_INVALID_HANDLE, CUDA_ERROR_OUT_OF_MEMORY, CUDA_ERROR_UNKNOWN

Description

Registers the Direct3D 10 resource `pD3DResource` for access by CUDA and returns a CUDA handle to `pD3Dresource` in `pCudaResource`. The handle returned in `pCudaResource` may be used to map and unmap this resource until it is unregistered. On success this call will increase the internal reference count on `pD3DResource`.

This reference count will be decremented when this resource is unregistered through cuGraphicsUnregisterResource().

This call is potentially 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 through 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

▸ CU_GRAPHICS_REGISTER_FLAGS_NONE: Specifies no hints about how this resource will be used.
▸ CU_GRAPHICS_REGISTER_FLAGS_SURFACE_LDST: Specifies that CUDA will bind this resource to a surface reference.
▸ CU_GRAPHICS_REGISTER_FLAGS_TEXTURE_GATHER: 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 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 CUDA_ERROR_INVALID_HANDLE is returned. If `pD3DResource` cannot be registered then CUDA_ERROR_UNKNOWN is returned. If `Flags` is not one of the above specified value then CUDA_ERROR_INVALID_VALUE is returned.

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

See also:

cuGraphicsUnregisterResource, cuGraphicsMapResources, cuGraphicsSubResourceGetMappedArray, cuGraphicsResourceGetMappedPointer

3.27.1. Direct3D 10 Interoperability [DEPRECATED]

Direct3D 10 Interoperability

This section describes deprecated Direct3D 10 interoperability functionality.

enum CUD3D10map_flags

Flags to map or unmap a resource

Values

CU_D3D10_MAPRESOURCE_FLAGS_NONE = 0x00
CU_D3D10_MAPRESOURCE_FLAGS_READONLY = 0x01
CU_D3D10_MAPRESOURCE_FLAGS_WRITEDISCARD = 0x02

enum CUD3D10register_flags

Flags to register a resource

Values

CU_D3D10_REGISTER_FLAGS_NONE = 0x00
CU_D3D10_REGISTER_FLAGS_ARRAY = 0x01

CUresult cuD3D10CtxCreate (CUcontext *pCtx, CUdevice *pCudaDevice, unsigned int Flags, ID3D10Device *pD3DDevice)

Create a CUDA context for interoperability with Direct3D 10.

Parameters

pCtx
- Returned newly created CUDA context

pCudaDevice
- Returned pointer to the device on which the context was created

Flags
- Context creation flags (see cuCtxCreate() for details)

pD3DDevice
- Direct3D device to create interoperability context with

Returns

CUDA_SUCCESS, CUDA_ERROR_DEINITIALIZED, CUDA_ERROR_NOT_INITIALIZED, CUDA_ERROR_INVALID_VALUE, CUDA_ERROR_OUT_OF_MEMORY, CUDA_ERROR_UNKNOWN

Description

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 context with a D3D10 device in order to achieve maximum interoperability performance.

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

See also:

cuD3D10GetDevice, cuGraphicsD3D10RegisterResource

CUresult cuD3D10CtxCreateOnDevice (CUcontext *pCtx, unsigned int flags, ID3D10Device *pD3DDevice, CUdevice cudaDevice)

Create a CUDA context for interoperability with Direct3D 10.

Parameters

pCtx
- Returned newly created CUDA context

flags

- Context creation flags (see cuCtxCreate() for details)

pD3DDevice

- Direct3D device to create interoperability context with

cudaDevice

- The CUDA device on which to create the context. This device must be among the devices returned when querying CU_D3D10_DEVICES_ALL from cuD3D10GetDevices.

Returns

CUDA_SUCCESS, CUDA_ERROR_DEINITIALIZED, CUDA_ERROR_NOT_INITIALIZED, CUDA_ERROR_INVALID_VALUE, CUDA_ERROR_OUT_OF_MEMORY, CUDA_ERROR_UNKNOWN

Description

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 context with a D3D10 device in order to achieve maximum interoperability performance.

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

See also:

cuD3D10GetDevices, cuGraphicsD3D10RegisterResource

CUresult cuD3D10GetDirect3DDevice (ID3D10Device **ppD3DDevice)

Get the Direct3D 10 device against which the current CUDA context was created.

Parameters

ppD3DDevice

- Returned Direct3D device corresponding to CUDA context

Returns

CUDA_SUCCESS, CUDA_ERROR_DEINITIALIZED, CUDA_ERROR_NOT_INITIALIZED, CUDA_ERROR_INVALID_CONTEXT

Description

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 context with a D3D10 device in order to achieve maximum interoperability performance.

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

See also:

cuD3D10GetDevice

CUresult cuD3D10MapResources (unsigned int count, ID3D10Resource **ppResources)

Map Direct3D resources for access by CUDA.

Parameters

count
   - Number of resources to map for CUDA
ppResources
   - Resources to map for CUDA

Returns

CUDA_SUCCESS, CUDA_ERROR_DEINITIALIZED, CUDA_ERROR_NOT_INITIALIZED, CUDA_ERROR_INVALID_CONTEXT, CUDA_ERROR_INVALID_HANDLE, CUDA_ERROR_ALREADY_MAPPED, CUDA_ERROR_UNKNOWN

Description

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 cuD3D10MapResources() will complete before any CUDA kernels issued after cuD3D10MapResources() begin.

If any of `ppResources` have not been registered for use with CUDA or if `ppResources` contains any duplicate entries, then

CUDA_ERROR_INVALID_HANDLE is returned. If any of `ppResources` are presently mapped for access by CUDA, then CUDA_ERROR_ALREADY_MAPPED is returned.

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

See also:

cuGraphicsMapResources

CUresult cuD3D10RegisterResource (ID3D10Resource *pResource, unsigned int Flags)

Register a Direct3D resource for access by CUDA.

Parameters

pResource
  - Resource to register
Flags
  - Parameters for resource registration

Returns

CUDA_SUCCESS, CUDA_ERROR_DEINITIALIZED, CUDA_ERROR_NOT_INITIALIZED, CUDA_ERROR_INVALID_CONTEXT, CUDA_ERROR_INVALID_VALUE, CUDA_ERROR_INVALID_HANDLE, CUDA_ERROR_OUT_OF_MEMORY, CUDA_ERROR_UNKNOWN

Description

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 cuD3D10UnregisterResource(). 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 cuD3D10UnregisterResource().

This call is potentially 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 CU_D3D10_REGISTER_FLAGS_ARRAY.

- 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.

- CU_D3D10_REGISTER_FLAGS_NONE: Specifies that CUDA will access this resource through a CUdeviceptr. The pointer, size, and (for textures), pitch for each subresource of this allocation may be queried through cuD3D10ResourceGetMappedPointer(), cuD3D10ResourceGetMappedSize(), and cuD3D10ResourceGetMappedPitch() respectively. This option is valid for all resource types.
- CU_D3D10_REGISTER_FLAGS_ARRAY: Specifies that CUDA will access this resource through a CUarray queried on a sub-resource basis through cuD3D10ResourceGetMappedArray(). 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 CUDA_ERROR_INVALID_CONTEXT is returned. If `pResource` is of incorrect type or is already registered, then CUDA_ERROR_INVALID_HANDLE is returned. If `pResource` cannot be registered, then CUDA_ERROR_UNKNOWN is returned.

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

See also:

cuGraphicsD3D10RegisterResource

CUresult cuD3D10ResourceGetMappedArray (CUarray *pArray, ID3D10Resource *pResource, unsigned int SubResource)

Get an array through which to access a subresource of a Direct3D resource which has been mapped for access by CUDA.

Parameters

pArray
   - Returned array corresponding to subresource
pResource
   - Mapped resource to access
SubResource
   - Subresource of pResource to access

Returns

CUDA_SUCCESS, CUDA_ERROR_DEINITIALIZED,
CUDA_ERROR_NOT_INITIALIZED, CUDA_ERROR_INVALID_CONTEXT,
CUDA_ERROR_INVALID_VALUE, CUDA_ERROR_INVALID_HANDLE,
CUDA_ERROR_NOT_MAPPED

Description

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 `SubResource` may be accessed. The value set in `pArray` may change every time that `pResource` is mapped.

If `pResource` is not registered, then CUDA_ERROR_INVALID_HANDLE is returned. If `pResource` was not registered with usage flags CU_D3D10_REGISTER_FLAGS_ARRAY, then CUDA_ERROR_INVALID_HANDLE is returned. If `pResource` is not mapped, then CUDA_ERROR_NOT_MAPPED is returned.

For usage requirements of the `SubResource` parameter, see cuD3D10ResourceGetMappedPointer().

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

See also:

cuGraphicsSubResourceGetMappedArray

CUresult cuD3D10ResourceGetMappedPitch (size_t *pPitch, size_t *pPitchSlice, ID3D10Resource *pResource, unsigned int SubResource)

Get the pitch of a subresource of a Direct3D resource which has been mapped for access by CUDA.

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

CUDA_SUCCESS, CUDA_ERROR_DEINITIALIZED, CUDA_ERROR_NOT_INITIALIZED, CUDA_ERROR_INVALID_CONTEXT, CUDA_ERROR_INVALID_VALUE, CUDA_ERROR_INVALID_HANDLE, CUDA_ERROR_NOT_MAPPED

Description

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 IDirect3DBaseTexture10 or one of its sub-types or if `pResource` has not been registered for use with CUDA, then

CUDA_ERROR_INVALID_HANDLE is returned. If `pResource` was not registered with usage flags CU_D3D10_REGISTER_FLAGS_NONE, then CUDA_ERROR_INVALID_HANDLE is returned. If `pResource` is not mapped for access by CUDA, then CUDA_ERROR_NOT_MAPPED is returned.

For usage requirements of the `SubResource` parameter, see cuD3D10ResourceGetMappedPointer().

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

See also:

cuGraphicsSubResourceGetMappedArray

CUresult cuD3D10ResourceGetMappedPointer (CUdeviceptr *pDevPtr, ID3D10Resource *pResource, unsigned int SubResource)

Get a pointer through which to access a subresource of a Direct3D resource which has been mapped for access by CUDA.

Parameters

pDevPtr
   - Returned pointer corresponding to subresource
pResource
   - Mapped resource to access
SubResource
   - Subresource of pResource to access

Returns

CUDA_SUCCESS, CUDA_ERROR_DEINITIALIZED, CUDA_ERROR_NOT_INITIALIZED, CUDA_ERROR_INVALID_CONTEXT, CUDA_ERROR_INVALID_VALUE, CUDA_ERROR_INVALID_HANDLE, CUDA_ERROR_NOT_MAPPED

Description

Deprecated This function is deprecated as of CUDA 3.0.

Returns in `*pDevPtr` the base pointer of the subresource of the mapped Direct3D resource `pResource`, which corresponds to `SubResource`. The value set in `pDevPtr` may change every time that `pResource` is mapped.

If `pResource` is not registered, then CUDA_ERROR_INVALID_HANDLE is returned. If `pResource` was not registered with usage flags

CU_D3D10_REGISTER_FLAGS_NONE, then CUDA_ERROR_INVALID_HANDLE is returned. If `pResource` is not mapped, then CUDA_ERROR_NOT_MAPPED 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().

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

See also:

cuGraphicsResourceGetMappedPointer

CUresult cuD3D10ResourceGetMappedSize (size_t *pSize, ID3D10Resource *pResource, unsigned int SubResource)

Get the size of a subresource of a Direct3D resource which has been mapped for access by CUDA.

Parameters

pSize
   - Returned size of subresource
pResource
   - Mapped resource to access
SubResource
   - Subresource of pResource to access

Returns

CUDA_SUCCESS, CUDA_ERROR_DEINITIALIZED, CUDA_ERROR_NOT_INITIALIZED, CUDA_ERROR_INVALID_CONTEXT, CUDA_ERROR_INVALID_VALUE, CUDA_ERROR_INVALID_HANDLE, CUDA_ERROR_NOT_MAPPED

Description

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 CUDA_ERROR_INVALID_HANDLE is returned. If `pResource` was not

registered with usage flags CU_D3D10_REGISTER_FLAGS_NONE, then CUDA_ERROR_INVALID_HANDLE is returned. If pResource is not mapped for access by CUDA, then CUDA_ERROR_NOT_MAPPED is returned.

For usage requirements of the SubResource parameter, see cuD3D10ResourceGetMappedPointer().

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

See also:

cuGraphicsResourceGetMappedPointer

CUresult cuD3D10ResourceGetSurfaceDimensions (size_t *pWidth, size_t *pHeight, size_t *pDepth, ID3D10Resource *pResource, unsigned int SubResource)

Get the dimensions of a registered surface.

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

CUDA_SUCCESS, CUDA_ERROR_DEINITIALIZED, CUDA_ERROR_NOT_INITIALIZED, CUDA_ERROR_INVALID_CONTEXT, CUDA_ERROR_INVALID_VALUE, CUDA_ERROR_INVALID_HANDLE

Description

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.

Because 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 IDirect3DBaseTexture10 or IDirect3DSurface10 or if `pResource` has not been registered for use with CUDA, then CUDA_ERROR_INVALID_HANDLE is returned.

For usage requirements of the `SubResource` parameter, see cuD3D10ResourceGetMappedPointer().

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

See also:

cuGraphicsSubResourceGetMappedArray

CUresult cuD3D10ResourceSetMapFlags (ID3D10Resource *pResource, unsigned int Flags)

Set usage flags for mapping a Direct3D resource.

Parameters

pResource
   - Registered resource to set flags for
Flags
   - Parameters for resource mapping

Returns

CUDA_SUCCESS, CUDA_ERROR_DEINITIALIZED, CUDA_ERROR_NOT_INITIALIZED, CUDA_ERROR_INVALID_CONTEXT, CUDA_ERROR_INVALID_VALUE, CUDA_ERROR_INVALID_HANDLE, CUDA_ERROR_ALREADY_MAPPED

Description

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.

▸ CU_D3D10_MAPRESOURCE_FLAGS_NONE: 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.

▸ CU_D3D10_MAPRESOURCE_FLAGS_READONLY: Specifies that CUDA kernels which access this resource will not write to this resource.

▸ CU_D3D10_MAPRESOURCE_FLAGS_WRITEDISCARD: 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 CUDA_ERROR_INVALID_HANDLE is returned. If `pResource` is presently mapped for access by CUDA then CUDA_ERROR_ALREADY_MAPPED is returned.

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

See also:

cuGraphicsResourceSetMapFlags

CUresult cuD3D10UnmapResources (unsigned int count, ID3D10Resource **ppResources)

Unmap Direct3D resources.

Parameters

count
- Number of resources to unmap for CUDA

ppResources
- Resources to unmap for CUDA

Returns

CUDA_SUCCESS, CUDA_ERROR_DEINITIALIZED, CUDA_ERROR_NOT_INITIALIZED, CUDA_ERROR_INVALID_CONTEXT, CUDA_ERROR_INVALID_VALUE, CUDA_ERROR_INVALID_HANDLE, CUDA_ERROR_NOT_MAPPED, CUDA_ERROR_UNKNOWN

Description

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 cuD3D10UnmapResources() will complete before any Direct3D calls issued after cuD3D10UnmapResources() begin.

If any of ppResources have not been registered for use with CUDA or if ppResources contains any duplicate entries, then CUDA_ERROR_INVALID_HANDLE is returned. If any of ppResources are not presently mapped for access by CUDA, then CUDA_ERROR_NOT_MAPPED is returned.

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

See also:

cuGraphicsUnmapResources

CUresult cuD3D10UnregisterResource (ID3D10Resource *pResource)

Unregister a Direct3D resource.

Parameters

pResource
 - Resources to unregister

Returns

CUDA_SUCCESS, CUDA_ERROR_DEINITIALIZED, CUDA_ERROR_NOT_INITIALIZED, CUDA_ERROR_INVALID_CONTEXT, CUDA_ERROR_INVALID_HANDLE, CUDA_ERROR_UNKNOWN

Description

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 CUDA_ERROR_INVALID_HANDLE is returned.

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

See also:

cuGraphicsUnregisterResource

3.28. Direct3D 11 Interoperability

This section describes the Direct3D 11 interoperability functions of the low-level CUDA driver application programming interface. Note that mapping of Direct3D 11 resources is performed with the graphics API agnostic, resource mapping interface described in Graphics Interoperability.

Direct3D 11 Interoperability [DEPRECATED]

enum CUd3d11DeviceList

CUDA devices corresponding to a D3D11 device

Values

CU_D3D11_DEVICE_LIST_ALL = 0x01
  The CUDA devices for all GPUs used by a D3D11 device
CU_D3D11_DEVICE_LIST_CURRENT_FRAME = 0x02
  The CUDA devices for the GPUs used by a D3D11 device in its currently rendering frame
CU_D3D11_DEVICE_LIST_NEXT_FRAME = 0x03
  The CUDA devices for the GPUs to be used by a D3D11 device in the next frame

CUresult cuD3D11GetDevice (CUdevice *pCudaDevice, IDXGIAdapter *pAdapter)

Gets the CUDA device corresponding to a display adapter.

Parameters

pCudaDevice
  - Returned CUDA device corresponding to `pAdapter`
pAdapter
  - Adapter to query for CUDA device

Returns

CUDA_SUCCESS, CUDA_ERROR_DEINITIALIZED, CUDA_ERROR_NOT_INITIALIZED, CUDA_ERROR_NO_DEVICE, CUDA_ERROR_INVALID_VALUE, CUDA_ERROR_NOT_FOUND, CUDA_ERROR_UNKNOWN

Description

Returns in `*pCudaDevice` the CUDA-compatible device corresponding to the adapter `pAdapter` obtained from IDXGIFactory::EnumAdapters.

If no device on `pAdapter` is CUDA-compatible the call will return CUDA_ERROR_NO_DEVICE.

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

See also:

cuD3D11GetDevices

CUresult cuD3D11GetDevices (unsigned int *pCudaDeviceCount, CUdevice *pCudaDevices, unsigned int cudaDeviceCount, ID3D11Device *pD3D11Device, CUd3d11DeviceList deviceList)

Gets the CUDA devices corresponding to a Direct3D 11 device.

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 CU_D3D11_DEVICE_LIST_ALL for all devices, CU_D3D11_DEVICE_LIST_CURRENT_FRAME for the devices used to render the current frame (in SLI), or CU_D3D11_DEVICE_LIST_NEXT_FRAME for the devices used to render the next frame (in SLI).

Returns

CUDA_SUCCESS, CUDA_ERROR_DEINITIALIZED, CUDA_ERROR_NOT_INITIALIZED, CUDA_ERROR_NO_DEVICE, CUDA_ERROR_INVALID_VALUE, CUDA_ERROR_NOT_FOUND, CUDA_ERROR_UNKNOWN

Description

Returns in `*pCudaDeviceCount` the number of CUDA-compatible device corresponding to the Direct3D 11 device `pD3D11Device`. Also returns in `*pCudaDevices` at most `cudaDeviceCount` of 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 CUDA_ERROR_NO_DEVICE.

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

See also:

cuD3D11GetDevice

CUresult cuGraphicsD3D11RegisterResource (CUgraphicsResource *pCudaResource, ID3D11Resource *pD3DResource, unsigned int Flags)

Register a Direct3D 11 resource for access by CUDA.

Parameters

pCudaResource
   - Returned graphics resource handle
pD3DResource
   - Direct3D resource to register
Flags
   - Parameters for resource registration

Returns

CUDA_SUCCESS, CUDA_ERROR_DEINITIALIZED,
CUDA_ERROR_NOT_INITIALIZED, CUDA_ERROR_INVALID_CONTEXT,
CUDA_ERROR_INVALID_VALUE, CUDA_ERROR_INVALID_HANDLE,
CUDA_ERROR_OUT_OF_MEMORY, CUDA_ERROR_UNKNOWN

Description

Registers the Direct3D 11 resource `pD3DResource` for access by CUDA and returns a CUDA handle to `pD3Dresource` in `pCudaResource`. The handle returned in `pCudaResource` may be used to map and unmap this resource until it is unregistered. On success this call will increase the internal reference count on `pD3DResource`.

This reference count will be decremented when this resource is unregistered through cuGraphicsUnregisterResource().

This call is potentially 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 through 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

▸ CU_GRAPHICS_REGISTER_FLAGS_NONE: Specifies no hints about how this resource will be used.
▸ CU_GRAPHICS_REGISTER_FLAGS_SURFACE_LDST: Specifies that CUDA will bind this resource to a surface reference.
▸ CU_GRAPHICS_REGISTER_FLAGS_TEXTURE_GATHER: 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 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 CUDA_ERROR_INVALID_HANDLE is returned. If `pD3DResource` cannot be registered then CUDA_ERROR_UNKNOWN is returned. If `Flags` is not one of the above specified value then CUDA_ERROR_INVALID_VALUE is returned.

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

See also:

cuGraphicsUnregisterResource, cuGraphicsMapResources, cuGraphicsSubResourceGetMappedArray, cuGraphicsResourceGetMappedPointer

3.28.1. Direct3D 11 Interoperability [DEPRECATED]

Direct3D 11 Interoperability

This section describes deprecated Direct3D 11 interoperability functionality.

CUresult cuD3D11CtxCreate (CUcontext *pCtx, CUdevice *pCudaDevice, unsigned int Flags, ID3D11Device *pD3DDevice)

Create a CUDA context for interoperability with Direct3D 11.

Parameters

pCtx
 - Returned newly created CUDA context
pCudaDevice
 - Returned pointer to the device on which the context was created
Flags
 - Context creation flags (see cuCtxCreate() for details)
pD3DDevice
 - Direct3D device to create interoperability context with

Returns

CUDA_SUCCESS, CUDA_ERROR_DEINITIALIZED, CUDA_ERROR_NOT_INITIALIZED, CUDA_ERROR_INVALID_VALUE, CUDA_ERROR_OUT_OF_MEMORY, CUDA_ERROR_UNKNOWN

Description

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 context with a D3D11 device in order to achieve maximum interoperability performance.

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

See also:

cuD3D11GetDevice, cuGraphicsD3D11RegisterResource

CUresult cuD3D11CtxCreateOnDevice (CUcontext *pCtx, unsigned int flags, ID3D11Device *pD3DDevice, CUdevice cudaDevice)

Create a CUDA context for interoperability with Direct3D 11.

Parameters

pCtx
  - Returned newly created CUDA context
flags
  - Context creation flags (see cuCtxCreate() for details)
pD3DDevice
  - Direct3D device to create interoperability context with
cudaDevice
  - The CUDA device on which to create the context. This device must be among the devices returned when querying CU_D3D11_DEVICES_ALL from cuD3D11GetDevices.

Returns

CUDA_SUCCESS, CUDA_ERROR_DEINITIALIZED, CUDA_ERROR_NOT_INITIALIZED, CUDA_ERROR_INVALID_VALUE, CUDA_ERROR_OUT_OF_MEMORY, CUDA_ERROR_UNKNOWN

Description

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 context with a D3D11 device in order to achieve maximum interoperability performance.

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

See also:

cuD3D11GetDevices, cuGraphicsD3D11RegisterResource

CUresult cuD3D11GetDirect3DDevice (ID3D11Device **ppD3DDevice)

Get the Direct3D 11 device against which the current CUDA context was created.

Parameters

ppD3DDevice
   - Returned Direct3D device corresponding to CUDA context

Returns

CUDA_SUCCESS, CUDA_ERROR_DEINITIALIZED,
CUDA_ERROR_NOT_INITIALIZED, CUDA_ERROR_INVALID_CONTEXT

Description

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 context with a D3D11 device in order to achieve maximum interoperability performance.

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

See also:

cuD3D11GetDevice

3.29. VDPAU Interoperability

This section describes the VDPAU interoperability functions of the low-level CUDA driver application programming interface.

CUresult cuGraphicsVDPAURegisterOutputSurface (CUgraphicsResource *pCudaResource, VdpOutputSurface vdpSurface, unsigned int flags)

Registers a VDPAU VdpOutputSurface object.

Parameters

pCudaResource
 - Pointer to the returned object handle
vdpSurface
 - The VdpOutputSurface to be registered
flags
 - Map flags

Returns

CUDA_SUCCESS, CUDA_ERROR_INVALID_HANDLE,
CUDA_ERROR_ALREADY_MAPPED, CUDA_ERROR_INVALID_CONTEXT,

Description

Registers the VdpOutputSurface specified by `vdpSurface` for access by CUDA. A handle to the registered object is returned as `pCudaResource`. The surface's intended usage is specified using `flags`, as follows:

▶ CU_GRAPHICS_MAP_RESOURCE_FLAGS_NONE: 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.

▶ CU_GRAPHICS_MAP_RESOURCE_FLAGS_READ_ONLY: Specifies that CUDA will not write to this resource.

▶ CU_GRAPHICS_MAP_RESOURCE_FLAGS_WRITE_DISCARD: 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 VdpOutputSurface is presented as an array of subresources that may be accessed using pointers returned by cuGraphicsSubResourceGetMappedArray. The exact number of valid `arrayIndex` values depends on the VDPAU surface format. The mapping is shown in the table below. `mipLevel` must be 0.

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

See also:

cuCtxCreate, cuVDPAUCtxCreate, cuGraphicsVDPAURegisterVideoSurface, cuGraphicsUnregisterResource, cuGraphicsResourceSetMapFlags, cuGraphicsMapResources, cuGraphicsUnmapResources, cuGraphicsSubResourceGetMappedArray, cuVDPAUGetDevice

CUresult cuGraphicsVDPAURegisterVideoSurface (CUgraphicsResource *pCudaResource, VdpVideoSurface vdpSurface, unsigned int flags)

Registers a VDPAU VdpVideoSurface object.

Parameters

pCudaResource
 - Pointer to the returned object handle
vdpSurface
 - The VdpVideoSurface to be registered
flags
 - Map flags

Returns

CUDA_SUCCESS, CUDA_ERROR_INVALID_HANDLE, CUDA_ERROR_ALREADY_MAPPED, CUDA_ERROR_INVALID_CONTEXT,

Description

Registers the VdpVideoSurface specified by `vdpSurface` for access by CUDA. A handle to the registered object is returned as `pCudaResource`. The surface's intended usage is specified using `flags`, as follows:

▶ CU_GRAPHICS_MAP_RESOURCE_FLAGS_NONE: 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.

▶ CU_GRAPHICS_MAP_RESOURCE_FLAGS_READ_ONLY: Specifies that CUDA will not write to this resource.

▶ CU_GRAPHICS_MAP_RESOURCE_FLAGS_WRITE_DISCARD: 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 VdpVideoSurface is presented as an array of subresources that may be accessed using pointers returned by cuGraphicsSubResourceGetMappedArray. The exact number of valid `arrayIndex` values depends on the VDPAU surface format. The mapping is shown in the table below. `mipLevel` must be 0.

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

See also:

cuCtxCreate, cuVDPAUCtxCreate, cuGraphicsVDPAURegisterOutputSurface, cuGraphicsUnregisterResource, cuGraphicsResourceSetMapFlags, cuGraphicsMapResources, cuGraphicsUnmapResources, cuGraphicsSubResourceGetMappedArray, cuVDPAUGetDevice

CUresult cuVDPAUCtxCreate (CUcontext *pCtx, unsigned int flags, CUdevice device, VdpDevice vdpDevice, VdpGetProcAddress *vdpGetProcAddress)

Create a CUDA context for interoperability with VDPAU.

Parameters

pCtx
   - Returned CUDA context
flags
   - Options for CUDA context creation
device
   - Device on which to create the context
vdpDevice
   - The VdpDevice to interop with
vdpGetProcAddress
   - VDPAU's VdpGetProcAddress function pointer

Returns

CUDA_SUCCESS, CUDA_ERROR_DEINITIALIZED, CUDA_ERROR_NOT_INITIALIZED, CUDA_ERROR_INVALID_CONTEXT, CUDA_ERROR_INVALID_VALUE, CUDA_ERROR_OUT_OF_MEMORY

Description

Creates a new CUDA context, initializes VDPAU interoperability, and associates the CUDA context with the calling thread. It must be called before performing any other VDPAU interoperability operations. It may fail if the needed VDPAU driver facilities are not available. For usage of the `flags` parameter, see cuCtxCreate().

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

See also:

cuCtxCreate, cuGraphicsVDPAURegisterVideoSurface, cuGraphicsVDPAURegisterOutputSurface, cuGraphicsUnregisterResource, cuGraphicsResourceSetMapFlags, cuGraphicsMapResources, cuGraphicsUnmapResources, cuGraphicsSubResourceGetMappedArray, cuVDPAUGetDevice

CUresult cuVDPAUGetDevice (CUdevice *pDevice, VdpDevice vdpDevice, VdpGetProcAddress *vdpGetProcAddress)

Gets the CUDA device associated with a VDPAU device.

Parameters

pDevice
   - Device associated with vdpDevice
vdpDevice
   - A VdpDevice handle
vdpGetProcAddress
   - VDPAU's VdpGetProcAddress function pointer

Returns

CUDA_SUCCESS, CUDA_ERROR_DEINITIALIZED, CUDA_ERROR_NOT_INITIALIZED, CUDA_ERROR_INVALID_CONTEXT, CUDA_ERROR_INVALID_VALUE

Description

Returns in `*pDevice` the CUDA device associated with a `vdpDevice`, if applicable.

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

See also:

cuCtxCreate, cuVDPAUCtxCreate, cuGraphicsVDPAURegisterVideoSurface, cuGraphicsVDPAURegisterOutputSurface, cuGraphicsUnregisterResource,

cuGraphicsResourceSetMapFlags, cuGraphicsMapResources,
cuGraphicsUnmapResources, cuGraphicsSubResourceGetMappedArray

Chapter 4.
DATA STRUCTURES

Here are the data structures with brief descriptions:

CUDA_ARRAY3D_DESCRIPTOR
CUDA_ARRAY_DESCRIPTOR
CUDA_MEMCPY2D
CUDA_MEMCPY3D
CUDA_MEMCPY3D_PEER
CUDA_POINTER_ATTRIBUTE_P2P_TOKENS
CUDA_RESOURCE_DESC
CUDA_RESOURCE_VIEW_DESC
CUDA_TEXTURE_DESC
CUdevprop
CUipcEventHandle
CUipcMemHandle

4.1. CUDA_ARRAY3D_DESCRIPTOR Struct Reference

3D array descriptor

size_t CUDA_ARRAY3D_DESCRIPTOR::Depth

Depth of 3D array

unsigned int CUDA_ARRAY3D_DESCRIPTOR::Flags

Flags

CUarray_format CUDA_ARRAY3D_DESCRIPTOR::Format

Array format

size_t CUDA_ARRAY3D_DESCRIPTOR::Height

Height of 3D array

unsigned int CUDA_ARRAY3D_DESCRIPTOR::NumChannels

Channels per array element

size_t CUDA_ARRAY3D_DESCRIPTOR::Width

Width of 3D array

4.2. CUDA_ARRAY_DESCRIPTOR Struct Reference

Array descriptor

CUarray_format CUDA_ARRAY_DESCRIPTOR::Format

Array format

size_t CUDA_ARRAY_DESCRIPTOR::Height

Height of array

unsigned int CUDA_ARRAY_DESCRIPTOR::NumChannels

Channels per array element

size_t CUDA_ARRAY_DESCRIPTOR::Width

Width of array

4.3. CUDA_MEMCPY2D Struct Reference

2D memory copy parameters

CUarray CUDA_MEMCPY2D::dstArray

Destination array reference

CUdeviceptr CUDA_MEMCPY2D::dstDevice

Destination device pointer

void *CUDA_MEMCPY2D::dstHost

Destination host pointer

CUmemorytype CUDA_MEMCPY2D::dstMemoryType

Destination memory type (host, device, array)

size_t CUDA_MEMCPY2D::dstPitch

Destination pitch (ignored when dst is array)

size_t CUDA_MEMCPY2D::dstXInBytes

Destination X in bytes

size_t CUDA_MEMCPY2D::dstY

Destination Y

size_t CUDA_MEMCPY2D::Height

Height of 2D memory copy

CUarray CUDA_MEMCPY2D::srcArray

Source array reference

CUdeviceptr CUDA_MEMCPY2D::srcDevice

Source device pointer

const void *CUDA_MEMCPY2D::srcHost

Source host pointer

CUmemorytype CUDA_MEMCPY2D::srcMemoryType

Source memory type (host, device, array)

size_t CUDA_MEMCPY2D::srcPitch

Source pitch (ignored when src is array)

size_t CUDA_MEMCPY2D::srcXInBytes

Source X in bytes

size_t CUDA_MEMCPY2D::srcY

Source Y

size_t CUDA_MEMCPY2D::WidthInBytes

Width of 2D memory copy in bytes

4.4. CUDA_MEMCPY3D Struct Reference

3D memory copy parameters

size_t CUDA_MEMCPY3D::Depth

Depth of 3D memory copy

CUarray CUDA_MEMCPY3D::dstArray

Destination array reference

CUdeviceptr CUDA_MEMCPY3D::dstDevice

Destination device pointer

size_t CUDA_MEMCPY3D::dstHeight

Destination height (ignored when dst is array; may be 0 if Depth==1)

void *CUDA_MEMCPY3D::dstHost

Destination host pointer

size_t CUDA_MEMCPY3D::dstLOD

Destination LOD

CUmemorytype CUDA_MEMCPY3D::dstMemoryType

Destination memory type (host, device, array)

size_t CUDA_MEMCPY3D::dstPitch

Destination pitch (ignored when dst is array)

size_t CUDA_MEMCPY3D::dstXInBytes

Destination X in bytes

size_t CUDA_MEMCPY3D::dstY

Destination Y

size_t CUDA_MEMCPY3D::dstZ

Destination Z

size_t CUDA_MEMCPY3D::Height

Height of 3D memory copy

void *CUDA_MEMCPY3D::reserved0

Must be NULL

void *CUDA_MEMCPY3D::reserved1

Must be NULL

CUarray CUDA_MEMCPY3D::srcArray

Source array reference

CUdeviceptr CUDA_MEMCPY3D::srcDevice

Source device pointer

size_t CUDA_MEMCPY3D::srcHeight

Source height (ignored when src is array; may be 0 if Depth==1)

const void *CUDA_MEMCPY3D::srcHost

Source host pointer

size_t CUDA_MEMCPY3D::srcLOD

Source LOD

CUmemorytype CUDA_MEMCPY3D::srcMemoryType

Source memory type (host, device, array)

size_t CUDA_MEMCPY3D::srcPitch

Source pitch (ignored when src is array)

size_t CUDA_MEMCPY3D::srcXInBytes

Source X in bytes

size_t CUDA_MEMCPY3D::srcY

Source Y

size_t CUDA_MEMCPY3D::srcZ

Source Z

size_t CUDA_MEMCPY3D::WidthInBytes

Width of 3D memory copy in bytes

4.5. CUDA_MEMCPY3D_PEER Struct Reference

3D memory cross-context copy parameters

size_t CUDA_MEMCPY3D_PEER::Depth

Depth of 3D memory copy

CUarray CUDA_MEMCPY3D_PEER::dstArray

Destination array reference

CUcontext CUDA_MEMCPY3D_PEER::dstContext

Destination context (ignored with dstMemoryType is CU_MEMORYTYPE_ARRAY)

CUdeviceptr CUDA_MEMCPY3D_PEER::dstDevice

Destination device pointer

size_t CUDA_MEMCPY3D_PEER::dstHeight

Destination height (ignored when dst is array; may be 0 if Depth==1)

void *CUDA_MEMCPY3D_PEER::dstHost

Destination host pointer

size_t CUDA_MEMCPY3D_PEER::dstLOD

Destination LOD

CUmemorytype CUDA_MEMCPY3D_PEER::dstMemoryType

Destination memory type (host, device, array)

size_t CUDA_MEMCPY3D_PEER::dstPitch

Destination pitch (ignored when dst is array)

size_t CUDA_MEMCPY3D_PEER::dstXInBytes

Destination X in bytes

size_t CUDA_MEMCPY3D_PEER::dstY

Destination Y

size_t CUDA_MEMCPY3D_PEER::dstZ

Destination Z

size_t CUDA_MEMCPY3D_PEER::Height

Height of 3D memory copy

CUarray CUDA_MEMCPY3D_PEER::srcArray

Source array reference

CUcontext CUDA_MEMCPY3D_PEER::srcContext

Source context (ignored with srcMemoryType is CU_MEMORYTYPE_ARRAY)

CUdeviceptr CUDA_MEMCPY3D_PEER::srcDevice

Source device pointer

size_t CUDA_MEMCPY3D_PEER::srcHeight

Source height (ignored when src is array; may be 0 if Depth==1)

const void *CUDA_MEMCPY3D_PEER::srcHost

Source host pointer

size_t CUDA_MEMCPY3D_PEER::srcLOD

Source LOD

CUmemorytype CUDA_MEMCPY3D_PEER::srcMemoryType

Source memory type (host, device, array)

size_t CUDA_MEMCPY3D_PEER::srcPitch

Source pitch (ignored when src is array)

size_t CUDA_MEMCPY3D_PEER::srcXInBytes

Source X in bytes

size_t CUDA_MEMCPY3D_PEER::srcY

Source Y

size_t CUDA_MEMCPY3D_PEER::srcZ

Source Z

size_t CUDA_MEMCPY3D_PEER::WidthInBytes

Width of 3D memory copy in bytes

4.6. CUDA_POINTER_ATTRIBUTE_P2P_TOKENS Struct Reference

GPU Direct v3 tokens

4.7. CUDA_RESOURCE_DESC Struct Reference

CUDA Resource descriptor

CUdeviceptr CUDA_RESOURCE_DESC::devPtr

Device pointer

unsigned int CUDA_RESOURCE_DESC::flags

Flags (must be zero)

CUarray_format CUDA_RESOURCE_DESC::format

Array format

CUarray CUDA_RESOURCE_DESC::hArray

CUDA array

size_t CUDA_RESOURCE_DESC::height

Height of the array in elements

CUmipmappedArray CUDA_RESOURCE_DESC::hMipmappedArray

CUDA mipmapped array

unsigned int CUDA_RESOURCE_DESC::numChannels

Channels per array element

size_t CUDA_RESOURCE_DESC::pitchInBytes

Pitch between two rows in bytes

CUresourcetype CUDA_RESOURCE_DESC::resType

Resource type

size_t CUDA_RESOURCE_DESC::sizeInBytes

Size in bytes

size_t CUDA_RESOURCE_DESC::width

Width of the array in elements

4.8. CUDA_RESOURCE_VIEW_DESC Struct Reference

Resource view descriptor

size_t CUDA_RESOURCE_VIEW_DESC::depth

Depth of the resource view

unsigned int CUDA_RESOURCE_VIEW_DESC::firstLayer

First layer index

unsigned int CUDA_RESOURCE_VIEW_DESC::firstMipmapLevel

First defined mipmap level

CUresourceViewFormat CUDA_RESOURCE_VIEW_DESC::format

Resource view format

size_t CUDA_RESOURCE_VIEW_DESC::height

Height of the resource view

unsigned int CUDA_RESOURCE_VIEW_DESC::lastLayer

Last layer index

unsigned int CUDA_RESOURCE_VIEW_DESC::lastMipmapLevel

Last defined mipmap level

size_t CUDA_RESOURCE_VIEW_DESC::width

Width of the resource view

4.9. CUDA_TEXTURE_DESC Struct Reference

Texture descriptor

CUaddress_mode CUDA_TEXTURE_DESC::addressMode

Address modes

CUfilter_mode CUDA_TEXTURE_DESC::filterMode

Filter mode

unsigned int CUDA_TEXTURE_DESC::flags

Flags

unsigned int CUDA_TEXTURE_DESC::maxAnisotropy

Maximum anisotropy ratio

float CUDA_TEXTURE_DESC::maxMipmapLevelClamp

Mipmap maximum level clamp

float CUDA_TEXTURE_DESC::minMipmapLevelClamp

Mipmap minimum level clamp

CUfilter_mode
CUDA_TEXTURE_DESC::mipmapFilterMode

Mipmap filter mode

float CUDA_TEXTURE_DESC::mipmapLevelBias

Mipmap level bias

4.10. CUdevprop Struct Reference

Legacy device properties

int CUdevprop::clockRate

Clock frequency in kilohertz

int CUdevprop::maxGridSize

Maximum size of each dimension of a grid

int CUdevprop::maxThreadsDim

Maximum size of each dimension of a block

int CUdevprop::maxThreadsPerBlock

Maximum number of threads per block

int CUdevprop::memPitch

Maximum pitch in bytes allowed by memory copies

int CUdevprop::regsPerBlock

32-bit registers available per block

int CUdevprop::sharedMemPerBlock

Shared memory available per block in bytes

int CUdevprop::SIMDWidth

Warp size in threads

int CUdevprop::textureAlign

Alignment requirement for textures

int CUdevprop::totalConstantMemory

Constant memory available on device in bytes

4.11. CUipcEventHandle Struct Reference

CUDA IPC event handle

4.12. CUipcMemHandle Struct Reference

CUDA IPC mem handle

Chapter 5.
DATA FIELDS

Here is a list of all documented struct and union fields with links to the struct/union documentation for each field:

A
addressMode
  CUDA_TEXTURE_DESC

C
clockRate
  CUdevprop

D
Depth
  CUDA_MEMCPY3D
  CUDA_MEMCPY3D_PEER
depth
  CUDA_RESOURCE_VIEW_DESC
Depth
  CUDA_ARRAY3D_DESCRIPTOR
devPtr
  CUDA_RESOURCE_DESC
dstArray
  CUDA_MEMCPY3D_PEER
  CUDA_MEMCPY2D
  CUDA_MEMCPY3D
dstContext
  CUDA_MEMCPY3D_PEER
dstDevice
  CUDA_MEMCPY2D
  CUDA_MEMCPY3D

Format

CUDA_ARRAY3D_DESCRIPTOR

format

CUDA_RESOURCE_VIEW_DESC

CUDA_RESOURCE_DESC

Format

CUDA_ARRAY_DESCRIPTOR

H

hArray

CUDA_RESOURCE_DESC

height

CUDA_RESOURCE_DESC

Height

CUDA_MEMCPY2D

CUDA_ARRAY_DESCRIPTOR

CUDA_MEMCPY3D_PEER

CUDA_MEMCPY3D

CUDA_ARRAY3D_DESCRIPTOR

height

CUDA_RESOURCE_VIEW_DESC

hMipmappedArray

CUDA_RESOURCE_DESC

L

lastLayer

CUDA_RESOURCE_VIEW_DESC

lastMipmapLevel

CUDA_RESOURCE_VIEW_DESC

M

maxAnisotropy

CUDA_TEXTURE_DESC

maxGridSize

CUdevprop

maxMipmapLevelClamp

CUDA_TEXTURE_DESC

maxThreadsDim

CUdevprop

maxThreadsPerBlock

CUdevprop

memPitch

CUdevprop

minMipmapLevelClamp
   CUDA_TEXTURE_DESC
mipmapFilterMode
   CUDA_TEXTURE_DESC
mipmapLevelBias
   CUDA_TEXTURE_DESC

N
NumChannels
   CUDA_ARRAY_DESCRIPTOR
numChannels
   CUDA_RESOURCE_DESC
NumChannels
   CUDA_ARRAY3D_DESCRIPTOR

P
pitchInBytes
   CUDA_RESOURCE_DESC

R
regsPerBlock
   CUdevprop
reserved0
   CUDA_MEMCPY3D
reserved1
   CUDA_MEMCPY3D
resType
   CUDA_RESOURCE_DESC

S
sharedMemPerBlock
   CUdevprop
SIMDWidth
   CUdevprop
sizeInBytes
   CUDA_RESOURCE_DESC
srcArray
   CUDA_MEMCPY2D
   CUDA_MEMCPY3D
   CUDA_MEMCPY3D_PEER
srcContext
   CUDA_MEMCPY3D_PEER

srcDevice

CUDA_MEMCPY2D

CUDA_MEMCPY3D

CUDA_MEMCPY3D_PEER

srcHeight

CUDA_MEMCPY3D

CUDA_MEMCPY3D_PEER

srcHost

CUDA_MEMCPY2D

CUDA_MEMCPY3D

CUDA_MEMCPY3D_PEER

srcLOD

CUDA_MEMCPY3D

CUDA_MEMCPY3D_PEER

srcMemoryType

CUDA_MEMCPY2D

CUDA_MEMCPY3D_PEER

CUDA_MEMCPY3D

srcPitch

CUDA_MEMCPY3D_PEER

CUDA_MEMCPY3D

CUDA_MEMCPY2D

srcXInBytes

CUDA_MEMCPY3D

CUDA_MEMCPY2D

CUDA_MEMCPY3D_PEER

srcY

CUDA_MEMCPY3D_PEER

CUDA_MEMCPY2D

CUDA_MEMCPY3D

srcZ

CUDA_MEMCPY3D_PEER

CUDA_MEMCPY3D

T

textureAlign

CUdevprop

totalConstantMemory

CUdevprop

W

Width

CUDA_ARRAY_DESCRIPTOR

CUDA_ARRAY3D_DESCRIPTOR
width
CUDA_RESOURCE_VIEW_DESC
CUDA_RESOURCE_DESC
WidthInBytes
CUDA_MEMCPY3D_PEER
CUDA_MEMCPY2D
CUDA_MEMCPY3D

Chapter 6.
DEPRECATED LIST

Global CU_CTX_BLOCKING_SYNC

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

Global CUDA_ERROR_PROFILER_NOT_INITIALIZED

This error return is deprecated as of CUDA 5.0. It is no longer an error to attempt to enable/disable the profiling via cuProfilerStart or cuProfilerStop without initialization.

Global CUDA_ERROR_PROFILER_ALREADY_STARTED

This error return is deprecated as of CUDA 5.0. It is no longer an error to call cuProfilerStart() when profiling is already enabled.

Global CUDA_ERROR_PROFILER_ALREADY_STOPPED

This error return is deprecated as of CUDA 5.0. It is no longer an error to call cuProfilerStop() when profiling is already disabled.

Global CUDA_ERROR_CONTEXT_ALREADY_CURRENT

This error return is deprecated as of CUDA 3.2. It is no longer an error to attempt to push the active context via cuCtxPushCurrent().

Global cuDeviceComputeCapability

Global cuDeviceGetProperties

Global cuCtxAttach

Global cuCtxDetach

Global cuFuncSetBlockShape

Global cuFuncSetSharedSize

Global cuLaunch

Global cuLaunchGrid

Global cuLaunchGridAsync

Global cuParamSetf

Global cuParamSeti

Global cuParamSetSize

Global cuParamSetTexRef

Global cuParamSetv

Global cuTexRefCreate

Global cuTexRefDestroy

Global cuGLCtxCreate

This function is deprecated as of Cuda 5.0.

Global cuGLInit

This function is deprecated as of Cuda 3.0.

Global cuGLMapBufferObject

This function is deprecated as of Cuda 3.0.

Global cuGLMapBufferObjectAsync

This function is deprecated as of Cuda 3.0.

Global cuGLRegisterBufferObject

This function is deprecated as of Cuda 3.0.

Global cuGLSetBufferObjectMapFlags

This function is deprecated as of Cuda 3.0.

Global cuGLUnmapBufferObject

This function is deprecated as of Cuda 3.0.

Global cuGLUnmapBufferObjectAsync

This function is deprecated as of Cuda 3.0.

Global cuGLUnregisterBufferObject

This function is deprecated as of Cuda 3.0.

Global cuD3D9MapResources

This function is deprecated as of CUDA 3.0.

Global cuD3D9RegisterResource

This function is deprecated as of CUDA 3.0.

Global cuD3D9ResourceGetMappedArray

This function is deprecated as of CUDA 3.0.

Global cuD3D9ResourceGetMappedPitch

This function is deprecated as of CUDA 3.0.

Global cuD3D9ResourceGetMappedPointer

This function is deprecated as of CUDA 3.0.

Global cuD3D9ResourceGetMappedSize

This function is deprecated as of CUDA 3.0.

Global cuD3D9ResourceGetSurfaceDimensions

This function is deprecated as of CUDA 3.0.

Global cuD3D9ResourceSetMapFlags

This function is deprecated as of Cuda 3.0.

Global cuD3D9UnmapResources

This function is deprecated as of CUDA 3.0.

Global cuD3D9UnregisterResource

This function is deprecated as of CUDA 3.0.

Global cuD3D10CtxCreate

This function is deprecated as of CUDA 5.0.

Global cuD3D10CtxCreateOnDevice

This function is deprecated as of CUDA 5.0.

Global cuD3D10GetDirect3DDevice

This function is deprecated as of CUDA 5.0.

Global cuD3D10MapResources

This function is deprecated as of CUDA 3.0.

Global cuD3D10RegisterResource

This function is deprecated as of CUDA 3.0.

Global cuD3D10ResourceGetMappedArray

This function is deprecated as of CUDA 3.0.

Global cuD3D10ResourceGetMappedPitch

This function is deprecated as of CUDA 3.0.

Global cuD3D10ResourceGetMappedPointer

This function is deprecated as of CUDA 3.0.

Global cuD3D10ResourceGetMappedSize

This function is deprecated as of CUDA 3.0.

Global cuD3D10ResourceGetSurfaceDimensions

This function is deprecated as of CUDA 3.0.

Global cuD3D10ResourceSetMapFlags

This function is deprecated as of CUDA 3.0.

Global cuD3D10UnmapResources

This function is deprecated as of CUDA 3.0.

Global cuD3D10UnregisterResource

This function is deprecated as of CUDA 3.0.

Global cuD3D11CtxCreate

This function is deprecated as of CUDA 5.0.

Global cuD3D11CtxCreateOnDevice

This function is deprecated as of CUDA 5.0.

Global cuD3D11GetDirect3DDevice

This function is deprecated as of CUDA 5.0.

www.nvidia.com