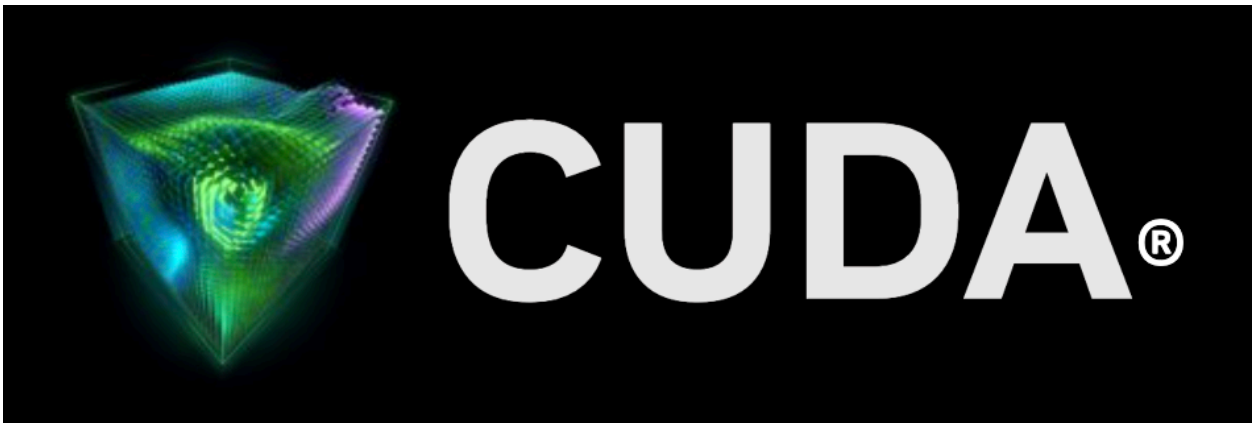

cuDLA API Reference Manual

Release 13.5

NVIDIA Corporation

Nov 02, 2026

Contents

1 APIs **1**
1.1 Data types used by cuDLA driver . . . 1
1.1.1 Macros . . . 6
1.1.2 Enumerations . . . 9
1.1.3 Typedefs . . . 13
1.2 cuDLA API . . . 14
1.2.1 Functions . . . 15

2 Structs **25**
2.1 CudlaFence . . . 25
2.2 cudlaExternalMemoryHandleDesc_t . . . 25
2.3 cudlaExternalSemaphoreHandleDesc_t . . . 26
2.4 cudlaModuleTensorDescriptor . . . 26
2.5 cudlaSignalEvents . . . 27
2.6 cudlaTask . . . 27
2.7 cudlaWaitEvents . . . 28

3 Notices **31**
3.1 Notice . . . 31
3.2 OpenCL . . . 32
3.3 Trademarks . . . 32

Chapter 1. APIs

1.1. Data types used by cuDLA driver

Macros

CUDLA_DATA_CATEGORY_BIAS

CUDLA_DATA_CATEGORY_FEATURE

CUDLA_DATA_CATEGORY_IMAGE

CUDLA_DATA_CATEGORY_PLANAR

CUDLA_DATA_CATEGORY_WEIGHT

CUDLA_DATA_FORMAT_NCHW

CUDLA_DATA_FORMAT_NCxHWx

CUDLA_DATA_FORMAT_NHWC

CUDLA_DATA_FORMAT_UNKNOWN

CUDLA_DATA_TYPE_FLOAT

CUDLA_DATA_TYPE_HALF

CUDLA_DATA_TYPE_INT16

CUDLA_DATA_TYPE_INT8

CUDLA_DATA_TYPE_UINT16

CUDLA_DATA_TYPE_UINT8

CUDLA_DATA_TYPE_UNKNOWN

CUDLA_LOADABLE_TENSOR_DESC_NUM_STRIDES

CUDLA_PIXEL_FORMAT_A16B16G16R16

CUDLA_PIXEL_FORMAT_A16B16G16R16_F

CUDLA_PIXEL_FORMAT_A16Y16U16V16

CUDLA_PIXEL_FORMAT_A16Y16U16V16_F

CUDLA_PIXEL_FORMAT_A2B10G10R10

CUDLA_PIXEL_FORMAT_A2R10G10B10

CUDLA_PIXEL_FORMAT_A2Y10U10V10

CUDLA_PIXEL_FORMAT_A8B8G8R8

CUDLA_PIXEL_FORMAT_A8R8G8B8

CUDLA_PIXEL_FORMAT_A8Y8U8V8

CUDLA_PIXEL_FORMAT_B10G10R10A2

CUDLA_PIXEL_FORMAT_B8G8R8A8

CUDLA_PIXEL_FORMAT_B8G8R8X8

CUDLA_PIXEL_FORMAT_FEATURE

CUDLA_PIXEL_FORMAT_R10

CUDLA_PIXEL_FORMAT_R10G10B10A2

CUDLA_PIXEL_FORMAT_R12

CUDLA_PIXEL_FORMAT_R16

CUDLA_PIXEL_FORMAT_R16_F

CUDLA_PIXEL_FORMAT_R16_I

CUDLA_PIXEL_FORMAT_R8

CUDLA_PIXEL_FORMAT_R8G8B8A8

CUDLA_PIXEL_FORMAT_R8G8B8X8

CUDLA_PIXEL_FORMAT_UNKNOWN

CUDLA_PIXEL_FORMAT_V10U10Y10A2

CUDLA_PIXEL_FORMAT_V16U16Y16A16

CUDLA_PIXEL_FORMAT_V8U8Y8A8

CUDLA_PIXEL_FORMAT_X16B16G16R16

CUDLA_PIXEL_FORMAT_X8B8G8R8

CUDLA_PIXEL_FORMAT_X8R8G8B8

CUDLA_PIXEL_FORMAT_Y10___U10V10_N444

CUDLA_PIXEL_FORMAT_Y10___V10U10_N444

CUDLA_PIXEL_FORMAT_Y12___U12V12_N444

CUDLA_PIXEL_FORMAT_Y12___V12U12_N444

CUDLA_PIXEL_FORMAT_Y16___U16V16_N444

CUDLA_PIXEL_FORMAT_Y16___V16U16_N444

CUDLA_PIXEL_FORMAT_Y8___U8V8_N444

CUDLA_PIXEL_FORMAT_Y8___V8U8_N444

CUDLA_PIXEL_MAPPING_BLOCK_LINEAR

CUDLA_PIXEL_MAPPING_PITCH_LINEAR

CUDLA_RUNTIME_TENSOR_DESC_NAME_MAX_LEN

CUDLA_VER_MAJOR

CUDLA_VER_MINOR

CUDLA_VER_PATCH

Enumerations

cudlaAccessPermissionFlags
Access permission flags for importing NvSciBuffers.

cudlaDevAttributeType
Device attribute type.

cudlaFenceType
Supported fence types.

cudlaMode
Device creation modes.

cudlaModuleAttributeType
Module attribute types.

cudlaModuleLoadFlags
Module load flags for *cudlaModuleLoadFromMemory* .

cudlaNvSciSyncAttributes
cuDLA NvSciSync attributes.

cudlaScratchMemoryConfig
Scratch-memory configuration for cudlaCreateDevice, OR-ed with the device mode.

cudlaStatus
Error codes.

cudlaSubmissionFlags
Task submission flags for *cudlaSubmitTask* .

Structs

CudlaFence
Fence description.

cudlaExternalMemoryHandleDesc_t
External memory handle descriptor.

cudlaExternalSemaphoreHandleDesc_t
External semaphore handle descriptor.

cudlaModuleTensorDescriptor
Tensor descriptor.

cudlaSignalEvents
Signal events for *cudlaSubmitTask* .

cudlaTask
Structure of Task.

cudlaWaitEvents
Wait events for *cudlaSubmitTask* .

Typedefs

cudlaAccessPermissionFlags

cudlaDevAttributeType

cudlaDevHandle
cuDLA Device Handle

cudlaFenceType

cudlaMode

cudlaModule
cuDLA Module Handle

cudlaModuleAttributeType

cudlaModuleLoadFlags

cudlaNvSciSyncAttributes

cudlaScratchMemoryConfig

cudlaStatus

cudlaSubmissionFlags

Unions

cudlaDevAttribute
Device attribute.

cudlaModuleAttribute
Module attribute.

1.1.1. Macros

CUDLA_DATA_CATEGORY_BIAS 4U

CUDLA_DATA_CATEGORY_FEATURE 2U

CUDLA_DATA_CATEGORY_IMAGE 0U

CUDLA_DATA_CATEGORY_PLANAR 3U

CUDLA_DATA_CATEGORY_WEIGHT 1U

CUDLA_DATA_FORMAT_NCHW 1U

CUDLA_DATA_FORMAT_NCxHWx 3U

CUDLA_DATA_FORMAT_NHWC 2U

CUDLA_DATA_FORMAT_UNKNOWN 0U

CUDLA_DATA_TYPE_FLOAT 1U

CUDLA_DATA_TYPE_HALF 2U

CUDLA_DATA_TYPE_INT16 3U

CUDLA_DATA_TYPE_INT8 4U

CUDLA_DATA_TYPE_UINT16 6U

CUDLA_DATA_TYPE_UINT8 5U

CUDLA_DATA_TYPE_UNKNOWN 0U

CUDLA_LOADABLE_TENSOR_DESC_NUM_STRIDES 8U

CUDLA_PIXEL_FORMAT_A16B16G16R16 6U

CUDLA_PIXEL_FORMAT_A16B16G16R16_F 8U

CUDLA_PIXEL_FORMAT_A16Y16U16V16 9U

CUDLA_PIXEL_FORMAT_A16Y16U16V16_F 11U

CUDLA_PIXEL_FORMAT_A2B10G10R10 20U

CUDLA_PIXEL_FORMAT_A2R10G10B10 21U

CUDLA_PIXEL_FORMAT_A2Y10U10V10 24U

CUDLA_PIXEL_FORMAT_A8B8G8R8 12U

CUDLA_PIXEL_FORMAT_A8R8G8B8 13U

CUDLA_PIXEL_FORMAT_A8Y8U8V8 26U

CUDLA_PIXEL_FORMAT_B10G10R10A2 22U

CUDLA_PIXEL_FORMAT_B8G8R8A8 14U

CUDLA_PIXEL_FORMAT_B8G8R8X8 18U

CUDLA_PIXEL_FORMAT_FEATURE 36U

CUDLA_PIXEL_FORMAT_R10 1U

CUDLA_PIXEL_FORMAT_R10G10B10A2 23U

CUDLA_PIXEL_FORMAT_R12 2U

CUDLA_PIXEL_FORMAT_R16 3U

CUDLA_PIXEL_FORMAT_R16_F 5U

CUDLA_PIXEL_FORMAT_R16_I 4U

CUDLA_PIXEL_FORMAT_R8 0U

CUDLA_PIXEL_FORMAT_R8G8B8A8 15U

CUDLA_PIXEL_FORMAT_R8G8B8X8 19U

CUDLA_PIXEL_FORMAT_UNKNOWN 37U

CUDLA_PIXEL_FORMAT_V10U10Y10A2 25U

CUDLA_PIXEL_FORMAT_V16U16Y16A16 10U

CUDLA_PIXEL_FORMAT_V8U8Y8A8 27U

CUDLA_PIXEL_FORMAT_X16B16G16R16 7U

CUDLA_PIXEL_FORMAT_X8B8G8R8 16U

CUDLA_PIXEL_FORMAT_X8R8G8B8 17U

CUDLA_PIXEL_FORMAT_Y10___U10V10_N444 30U

CUDLA_PIXEL_FORMAT_Y10___V10U10_N444 31U

CUDLA_PIXEL_FORMAT_Y12___U12V12_N444 32U

CUDLA_PIXEL_FORMAT_Y12___V12U12_N444 33U

CUDLA_PIXEL_FORMAT_Y16___U16V16_N444 34U

CUDLA_PIXEL_FORMAT_Y16___V16U16_N444 35U

CUDLA_PIXEL_FORMAT_Y8___U8V8_N444 28U

CUDLA_PIXEL_FORMAT_Y8___V8U8_N444 29U

CUDLA_PIXEL_MAPPING_BLOCK_LINEAR 1U

CUDLA_PIXEL_MAPPING_PITCH_LINEAR 0U

CUDLA_RUNTIME_TENSOR_DESC_NAME_MAX_LEN 80U

CUDLA_VER_MAJOR 1U

CUDLA_VER_MINOR 5U

CUDLA_VER_PATCH 0U

1.1.2. Enumerations

enum **cudlaAccessPermissionFlags**

Access permission flags for importing NvSciBuffers.

Values:

enumerator **CUDLA_READ_WRITE_PERM** = 0

Flag to import memory with read-write permission.

enumerator **CUDLA_READ_ONLY_PERM** = 1

Flag to import memory with read-only permission.

enumerator **CUDLA_TASK_STATISTICS** = 1 << 1

Flag to indicate buffer as layerwise statistics buffer.

enum **cudlaDevAttributeType**

Device attribute type.

Values:

enumerator **CUDLA_UNIFIED_ADDRESSING** = 0

Flag to check for support for UVA.

enumerator **CUDLA_DEVICE_VERSION** = 1

Flag to check for DLA HW version.

enum **cudlaFenceType**

Supported fence types.

Values:

enumerator **CUDLA_NVSCISYNC_FENCE** = 1

NvSciSync fence type for EOF.

enumerator **CUDLA_NVSCISYNC_FENCE_SOF** = 2

enum **cudlaMode**

Device creation modes.

Values:

enumerator **CUDLA_CUDA_DLA** = 0

Hyrbid mode.

enumerator **CUDLA_STANDALONE** = 1

Standalone mode.

enum **cudlaModuleAttributeType**

Module attribute types.

Values:

enumerator **CUDLA_NUM_INPUT_TENSORS** = 0

Flag to retrieve number of input tensors.

enumerator **CUDLA_NUM_OUTPUT_TENSORS** = 1

Flag to retrieve number of output tensors.

enumerator **CUDLA_INPUT_TENSOR_DESCRIPTORS** = 2

Flag to retrieve all the input tensor descriptors.

enumerator **CUDLA_OUTPUT_TENSOR_DESCRIPTORS** = 3

Flag to retrieve all the output tensor descriptors.

enumerator **CUDLA_NUM_OUTPUT_TASK_STATISTICS** = 4

Flag to retrieve total number of output task statistics buffer.

enumerator **CUDLA_OUTPUT_TASK_STATISTICS_DESCRIPTORS** = 5

Flag to retrieve all the output task statistics descriptors.

enum **cudlaModuleLoadFlags**

Module load flags for *cudlaModuleLoadFromMemory*.

Values:

enumerator **CUDLA_MODULE_DEFAULT** = 0

Default flag.

enumerator **CUDLA_MODULE_ENABLE_FAULT_DIAGNOSTICS** = 1

Flag to load a module that is used to perform permanent fault diagnostics for DLA HW.

enum **cudlaNvSciSyncAttributes**

cuDLA NvSciSync attributes.

Values:

enumerator **CUDLA_NVSCISYNC_ATTR_WAIT** = 1

Wait attribute.

enumerator **CUDLA_NVSCISYNC_ATTR_SIGNAL** = 2

Signal attribute.

enum **cudlaScratchMemoryConfig**

Scratch-memory configuration for cudlaCreateDevice, OR-ed with the device mode.

Bit 0 of 'flags' selects the cudlaMode; the scratch-memory configuration occupies bit 1 onward.

Values:

enumerator **CUDLA_SCRATCH_MEMORY_DEFAULT** = 0U << 1

enumerator **CUDLA_SCRATCH_MEMORY_SHARED_STATIC** = 1U << 1

enumerator **CUDLA_SCRATCH_MEMORY_CONFIG_MAX** = 0x7FFFFFFF

enum **cudlaStatus**

Error codes.

Values:

enumerator **cudlaSuccess** = 0

The API call returned with no errors.

enumerator **cudlaErrorInvalidParam** = 1

This indicates that one or more parameters passed to the API is/are incorrect.

enumerator **cudlaErrorOutOfResources** = 2

This indicates that the API call failed due to lack of underlying resources.

enumerator **cudlaErrorCreationFailed** = 3

This indicates that an internal error occurred during creation of device handle.

enumerator **cudlaErrorInvalidAddress** = 4

This indicates that the memory object being passed in the API call has not been registered before.

enumerator **cudlaErrorOs** = 5

This indicates that an OS error occurred.

enumerator **cudlaErrorCuda** = 6

This indicates that there was an error in a CUDA operation as part of the API call.

enumerator **cudlaErrorUmd** = 7

This indicates that there was an error in the DLA runtime for the API call.

enumerator **cudlaErrorInvalidDevice** = 8

This indicates that the device handle passed to the API call is invalid.

enumerator **cudlaErrorInvalidAttribute** = 9

This indicates that an invalid attribute is being requested.

enumerator **cudlaErrorIncompatibleDlaSWVersion** = 10
This indicates that the underlying DLA runtime is incompatible with the current cuDLA version.

enumerator **cudlaErrorMemoryRegistered** = 11
This indicates that the memory object is already registered.

enumerator **cudlaErrorInvalidModule** = 12
This indicates that the module being passed is invalid.

enumerator **cudlaErrorUnsupportedOperation** = 13
This indicates that the operation being requested by the API call is unsupported.

enumerator **cudlaErrorNvSci** = 14
This indicates that the NvSci operation requested by the API call failed.

enumerator **cudlaErrorDriverNotFound** = 15
This indicates that the dependent driver is not found or loaded.

enumerator **cudlaErrorDlaErrInvalidInput** = 0x40000001
DLA HW Error.

enumerator **cudlaErrorDlaErrInvalidPreAction** = 0x40000002
DLA HW Error.

enumerator **cudlaErrorDlaErrNoMem** = 0x40000003
DLA HW Error.

enumerator **cudlaErrorDlaErrProcessorBusy** = 0x40000004
DLA HW Error.

enumerator **cudlaErrorDlaErrTaskStatusMismatch** = 0x40000005
DLA HW Error.

enumerator **cudlaErrorDlaErrEngineTimeout** = 0x40000006
DLA HW Error.

enumerator **cudlaErrorDlaErrDataMismatch** = 0x40000007
DLA HW Error.

enumerator **cudlaErrorUnknown** = 0x7fffffff
This indicates that an unknown error has occurred.

enum **cudlaSubmissionFlags**
Task submission flags for *cudlaSubmitTask*.

Values:

enumerator **CUDLA_SUBMIT_NOOP** = 1
Flag to specify that the submitted task must be bypassed for execution.

enumerator **CUDLA_SUBMIT_SKIP_LOCK_ACQUIRE** = 1 << 1
Flag to specify that the global lock acquire must be skipped.

enumerator **CUDLA_SUBMIT_DIAGNOSTICS_TASK** = 1 << 2
Flag to specify that the submitted task is to run permanent fault diagnostics for DLA HW.

1.1.3. Typedefs

typedef enum *cudlaAccessPermissionFlags* **cudlaAccessPermissionFlags**

typedef enum *cudlaDevAttributeType* **cudlaDevAttributeType**

typedef struct cudlaDevHandle_t ***cudlaDevHandle**
cuDLA Device Handle

typedef enum *cudlaFenceType* **cudlaFenceType**

typedef enum *cudlaMode* **cudlaMode**

typedef struct cudlaModule_t ***cudlaModule**
cuDLA Module Handle

typedef enum *cudlaModuleAttributeType* **cudlaModuleAttributeType**

typedef enum *cudlaModuleLoadFlags* **cudlaModuleLoadFlags**

typedef enum *cudlaNvSciSyncAttributes* **cudlaNvSciSyncAttributes**

typedef enum *cudlaScratchMemoryConfig* **cudlaScratchMemoryConfig**

typedef enum *cudlaStatus* **cudlaStatus**

typedef enum *cudlaSubmissionFlags* **cudlaSubmissionFlags**

1.2. cuDLA API

This section describes the application programming interface of the cuDLA driver.

Functions

cudlaStatus *cudlaCreateDevice*(uint64_t const device, cudlaDevHandle *const devHandle, uint32_t const flags)
Create a device handle.

cudlaStatus *cudlaDestroyDevice*(cudlaDevHandle const devHandle)
Destroy device handle.

cudlaStatus *cudlaDeviceGetAttribute*(cudlaDevHandle const devHandle, cudlaDevAttributeType const attrib, cudlaDevAttribute *const pAttribute)
Get cuDLA device attributes.

cudlaStatus *cudlaDeviceGetCount*(uint64_t *const pNumDevices)
Get device count.

cudlaStatus *cudlaGetLastError*(cudlaDevHandle const devHandle)
Gets the last asynchronous error in task execution.

cudlaStatus *cudlaGetNvSciSyncAttributes*(uint64_t *attrList, uint32_t const flags)
Get cuDLA's NvSciSync attributes.

cudlaStatus *cudlaGetVersion*(uint64_t *const version)
Returns the version number of the library.

cudlaStatus *cudlaImportExternalMemory*(cudlaDevHandle const devHandle, const cudlaExternalMemoryHandleDesc *const desc, uint64_t **const devPtr, uint32_t const flags)
Imports external memory into cuDLA.

cudlaStatus *cudlaImportExternalSemaphore*(cudlaDevHandle const devHandle, const cudlaExternalSemaphoreHandleDesc *const desc, uint64_t **const devPtr, uint32_t const flags)
Imports external semaphore into cuDLA.

cudlaStatus *cudlaMemRegister*(cudlaDevHandle const devHandle, const uint64_t *const ptr, size_t const size, uint64_t **const devPtr, uint32_t const flags)
Registers the CUDA memory to DLA engine.

cudlaStatus *cudlaMemUnregister*(cudlaDevHandle const devHandle, const uint64_t *const devPtr)
Unregisters the input memory from DLA engine.

cudlaStatus *cudlaModuleGetAttributes*(cudlaModule const hModule, cudlaModuleAttributeType const attrType, cudlaModuleAttribute *const attribute)
Get DLA module attributes.

cudlaStatus *cudlaModuleLoadFromMemory*(cudlaDevHandle const devHandle, const uint8_t *const pModule, size_t const moduleSize, cudlaModule *const hModule, uint32_t const flags)
Load a DLA module.

cudlaStatus *cudlaModuleUnload*(cudlaModule const hModule, uint32_t const flags)
Unload a DLA module.

cudlaStatus *cudlaSetTaskTimeoutInMs*(cudlaDevHandle const devHandle, uint32_t const timeout)
Set task timeout in millisecond.

cudlaStatus *cudlaSubmitTask*(cudlaDevHandle const devHandle, const cudlaTask *const ptrToTasks, uint32_t const numTasks, void *const stream, uint32_t const flags)
Submits the inference operation on DLA.

1.2.1. Functions

cudlaStatus **cudlaCreateDevice**(uint64_t const device, *cudlaDevHandle* *const devHandle, uint32_t const flags)

Create a device handle.

Creates an instance of a cuDLA device which can be used to submit DLA operations. The application can create the handle in hybrid or standalone mode. In hybrid mode, the current set GPU device is used by this API to decide the association of the created DLA device handle. This function returns *cudlaErrorUnsupportedOperation* if the current set GPU device is a dGPU as cuDLA is not supported on dGPU presently. cuDLA supports 16 cuDLA device handles per DLA HW instance.

A scratch-memory configuration is OR-ed with the mode:

- CUDLA_SCRATCH_MEMORY_SHARED_STATIC - Modules loaded on this handle share a single fixed-size pinned Local DRAM scratch pool instead of each allocating a private buffer. Loadables that do not fit the pool fall back to a private buffer. This flag must be OR-ed with a device mode (*CUDLA_CUDA_DLA* or *CUDLA_STANDALONE*).

Parameters

- **device** – **[in]** - Device number (can be 0 or 1).
- **devHandle** – **[out]** - Pointer to hold the created cuDLA device handle.
- **flags** – **[in]** - Flags controlling device creation. Valid values for `flags` are:
  - *CUDLA_CUDA_DLA* - In this mode, cuDLA serves as a programming model extension of CUDA wherein DLA work can be submitted using CUDA constructs.
  - *CUDLA_STANDALONE* - In this mode, cuDLA works standalone without any interaction with CUDA.

Returns
cudlaSuccess, *cudlaErrorOutOfResources*, *cudlaErrorInvalidParam*, *cudlaErrorIncompatibleDlaSWVersion*, *cudlaErrorCreationFailed*, *cudlaErrorCuda*, *cudlaErrorUmd*, *cudlaErrorUnsupportedOperation*, *cudlaErrorDriverNotFound*

cudlaStatus **cudlaDestroyDevice**(*cudlaDevHandle* const devHandle)

Destroy device handle.

Destroys the instance of the cuDLA device which was created with cudlaCreateDevice. Before destroying the handle, it is important to ensure that all the tasks submitted previously to the device are completed. Failure to do so can lead to application crashes.

In hybrid mode, cuDLA internally performs memory allocations with CUDA using the primary context. As a result, before destroying or resetting a CUDA primary context, it is mandatory that all cuDLA device initializations are destroyed.

Note: This API can return task execution errors from previous DLA task submissions.

Parameters
devHandle – **[in]** - A valid device handle.

Returns
cudlaSuccess, *cudlaErrorInvalidDevice*, *cudlaErrorCuda*, *cudlaErrorUmd*, *cudlaErrorDriverNotFound*

cudlaStatus **cudlaDeviceGetAttribute**(*cudlaDevHandle* const devHandle, *cudlaDevAttributeType* const attrib, cudlaDevAttribute *const pAttribute)

Get cuDLA device attributes.

UVA addressing between CUDA and DLA requires special support in the underlying kernel mode drivers. Applications are expected to query the cuDLA runtime to check if the current version of cuDLA supports UVA addressing.

Note: This API can return task execution errors from previous DLA task submissions.

Parameters

- **devHandle** – **[in]** - The input cuDLA device handle.
- **attrib** – **[in]** - The attribute that is being requested.
- **pAttribute** – **[out]** - The output pointer where the attribute will be available.

Returns
cudlaSuccess, *cudlaErrorInvalidParam*, *cudlaErrorInvalidDevice*, *cudlaErrorUmd*, *cudlaErrorInvalidAttribute*, *cudlaErrorDriverNotFound*

cudlaStatus **cudlaDeviceGetCount**(uint64_t *const pNumDevices)

Get device count.

Get number of DLA devices available to use.

Parameters
pNumDevices – **[out]** - The number of DLA devices will be available in this variable upon successful completion.

Returns
cudlaSuccess, *cudlaErrorInvalidParam*, *cudlaErrorUmd*, *cudlaErrorIncompatibleDlaSWVersion*, *cudlaErrorDriverNotFound*

cudlaStatus **cudlaGetLastError**(*cudlaDevHandle* const devHandle)

Gets the last asynchronous error in task execution.

The DLA tasks execute asynchronously on the DLA HW. As a result, the status of the task execution is not known at the time of task submission. The status of the task executed by the DLA HW most recently for the particular device handle can be queried using this interface.

Note that a return code of *cudlaSuccess* from this function does not necessarily imply that most recent task executed successfully. Since this function returns immediately, it can only report the status of the tasks at the snapshot of time when it is called. To be guaranteed of task completion,

applications must synchronize on the submitted tasks in hybrid or standalone modes and then call this API to check for errors.

Parameters

devHandle – **[in]** - A valid device handle.

Returns

cudlaSuccess, *cudlaErrorInvalidDevice*, *cudlaErrorDlaErrInvalidInput*, *cudlaErrorDlaErrInvalidPreAction*, *cudlaErrorDlaErrNoMem*, *cudlaErrorDlaErrProcessorBusy*, *cudlaErrorDlaErrTaskStatusMismatch*, *cudlaErrorDlaErrEngineTimeout*, *cudlaErrorDlaErrDataMismatch*, *cudlaErrorUnknown*, *cudlaErrorDriverNotFound*

cudlaStatus **cudlaGetNvSciSyncAttributes**(uint64_t *attrList, uint32_t const flags)

Get cuDLA's NvSciSync attributes.

Gets the NvSciSync's attributes in the attribute list created by the application.

cuDLA supports three types of NvSciSync object primitives -

- Sync point
- Regular semaphore
- Deterministic semaphore cuDLA prioritizes sync point primitive over regular and deterministic semaphore primitives by default and sets these priorities in the NvSciSync attribute list.

For Deterministic semaphore, NvSciSync attribute list used to create the NvSciSync object must have value of NvSciSyncAttrKey_RequireDeterministicFences key set to true.

cuDLA also supports Timestamp feature on NvSciSync objects. Waiter can request for this by setting NvSciSync attribute "NvSciSyncAttrKey_WaiterRequireTimestamps" as true.

In the event of failed NvSci initialization this function would return *cudlaErrorUnsupportedOperation*. This function can return *cudlaErrorNvSci* or *cudlaErrorInvalidAttribute* in certain cases when the underlying NvSci operation fails.

This API updates the input nvSciSyncAttrList with values equivalent to the following public attribute key-values:

NvSciSyncAttrKey_RequiredPerm is set to

- NvSciSyncAccessPerm_SignalOnly if value of flag is set to CUDLA_NVSCISYNC_ATTR_WAIT.
- NvSciSyncAccessPerm_WaitOnly if value of flag is set to CUDLA_NVSCISYNC_ATTR_SIGNAL.
- NvSciSyncAccessPerm_WaitSignal if value of flag is set to CUDLA_NVSCISYNC_ATTR_SIGNAL | CUDLA_NVSCISYNC_ATTR_WAIT.

As NvSciSyncAttrKey_RequiredPerm is internally set by cuDLA, setting this value by the application is disallowed.

Note: Users of cuDLA can only append attributes to output `attrList` using NvSci API, modifying already populated values of the output `attrList` can result in undefined behavior.

Parameters

- **attrList** – **[out]** - Attribute list created by the application.

- **flags – [in]** - Applications can use this flag to specify how they intend to use the NvSciSync object created from the `attrList`. The valid values of `flags` can be one of the following (or an OR of these values):
  - *CUDLA_NVSCISYNC_ATTR_WAIT*, specifies that the application intend to use the NvSciSync object created using this attribute list as a waiter in cuDLA and therefore needs cuDLA to fill waiter specific NvSciSyncAttr.
  - *CUDLA_NVSCISYNC_ATTR_SIGNAL*, specifies that the application intend to use the NvSciSync object created using this attribute list as a signaler in cuDLA and therefore needs cuDLA to fill signaler specific NvSciSyncAttr.

Returns

- *cudlaSuccess*, The API call returned with no errors.
- *cudlaErrorInvalidParam*, This API call failed because invalid parameter attrList was passed.
- *cudlaErrorUnsupportedOperation*, This error code indicates that the API call failed because the operation is not supported in hybrid mode.
- *cudlaErrorInvalidAttribute*, The API call failed as parameter attrList has invalid values.
- *cudlaErrorNvSci*, This error code indicates error in the NvSci operation as part of the API call.
- ::cudlaErrorNotPermittedOperation, This error code indicates that the API call is not permitted when DRIVE OS is in Operational state.
- *cudlaErrorUnknown*, This error code indicates that an unknown error has occurred.
- *cudlaErrorDriverNotFound*, This indicates that the cuDLA driver library cannot be loaded.

cudlaStatus **cudlaGetVersion**(uint64_t *const version)

Returns the version number of the library.

cuDLA is semantically versioned. This function will return the version as 1000000*major + 1000*minor + patch.

Parameters

version – [out] - cuDLA library version will be available in this variable upon successful execution.

Returns

cudlaSuccess, *cudlaErrorInvalidParam*, *cudlaErrorDriverNotFound*

cudlaStatus **cudlaImportExternalMemory**(*cudlaDevHandle* const devHandle, const cudlaExternalMemoryHandleDesc *const desc, uint64_t **const devPtr, uint32_t const flags)

Imports external memory into cuDLA.

Imports the allocated external memory by registering it with DLA. After successful registration, the returned pointer can be used in a task submit.

On Tegra, cuDLA supports importing NvSciBuf objects in standalone mode only. In the event of failed NvSci initialization (either due to usage of this API in hybrid mode or an issue in the NvSci library initialization), this function would return *cudlaErrorUnsupportedOperation*. This function

can return *cudlaErrorNvSci* or *cudlaErrorInvalidAttribute* in certain cases when the underlying NvSci operation fails.

Note: cuDLA only supports importing NvSciBuf objects of type NvSciBufType_RawBuffer or NvSciBufType_Tensor. Importing NvSciBuf object of any other type can result in an undefined behaviour.

Note: This API may result in undefined behavior if the address being registered is not 32-byte aligned. The input pointer `devPtr` must always satisfy the condition ((devPtr & 0x1F) == 0)

Note: This API can return task execution errors from previous DLA task submissions.

Parameters

- **`devHandle`** – **[in]** - A valid device handle.
- **`desc`** – **[in]** - Contains description about allocated external memory.
- **`devPtr`** – **[out]** - The output pointer where the mapping will be available.
- **`flags`** – **[in]** - Application can use this flag to specify the memory access permissions of the memory that needs to be registered with DLA. The valid values of `flags` can be one of the following:
  - *CUDLA_READ_WRITE_PERM*, specifies that the external memory needs to be registered with DLA as read-write memory.
  - *CUDLA_READ_ONLY_PERM*, specifies that the external memory needs to be registered with DLA as read-only memory.
  - *CUDLA_TASK_STATISTICS*, specifies that the external memory needs to be registered with DLA for layerwise statistics.

Returns

cudlaSuccess, *cudlaErrorInvalidParam*, *cudlaErrorInvalidDevice*, *cudlaErrorUnsupportedOperation*, *cudlaErrorNvSci*, *cudlaErrorInvalidAttribute*, *cudlaErrorMemoryRegistered*, *cudlaErrorUmd*, *cudlaErrorDriverNotFound*

cudlaStatus **`cudlaImportExternalSemaphore`**(*cudlaDevHandle* const devHandle, const cudlaExternalSemaphoreHandleDesc *const desc, uint64_t **const devPtr, uint32_t const flags)

Imports external semaphore into cuDLA.

Imports the allocated external semaphore by registering it with DLA. After successful registration, the returned pointer can be used in a task submission to signal synchronization objects.

On Tegra, cuDLA supports importing NvSciSync objects in standalone mode only. NvSciSync object primitives that cuDLA supports are sync point and deterministic semaphore.

cuDLA also supports Timestamp feature on NvSciSync objects, using which the user can get a snapshot of DLA clock at which a particular fence is signaled. At any point in time there are only 512 valid timestamp buffers that can be associated with fences. For example, If User has created 513 fences from a single NvSciSync object with timestamp enabled then the timestamp buffer associated with 1st fence is same as with 513th fence.

In the event of failed NvSci initialization (either due to usage of this API in hybrid mode or an issue in the NvSci library initialization), this function would return *cudlaErrorUnsupportedOperation*. This function can return *cudlaErrorNvSci* or *cudlaErrorInvalidAttribute* in certain cases when the underlying NvSci operation fails.

Note: This API can return task execution errors from previous DLA task submissions.

Parameters

- **devHandle** – **[in]** - A valid device handle.
- **desc** – **[in]** - Contains sempahore object.
- **devPtr** – **[out]** - The output pointer where the mapping will be available.
- **flags** – **[in]** - Reserved for future. Must be set to 0.

Returns

cudlaSuccess, *cudlaErrorInvalidParam*, *cudlaErrorInvalidDevice*, *cudlaErrorUnsupportedOperation*, *cudlaErrorNvSci*, *cudlaErrorInvalidAttribute*, *cudlaErrorMemoryRegistered*, *cudlaErrorDriverNotFound*

cudlaStatus **cudlaMemRegister**(*cudlaDevHandle* const devHandle, const uint64_t *const ptr, size_t const size, uint64_t **const devPtr, uint32_t const flags)

Registers the CUDA memory to DLA engine.

As part of registration, a system mapping is created whereby the DLA HW can access the underlying CUDA memory. The resultant mapping is available in devPtr and applications must use this mapping while referring this memory in submit operations.

This function will return *cudlaErrorInvalidAddress* if the pointer or size to be registered is invalid. In addition, if the input pointer was already registered, then this function will return *cudlaErrorMemoryRegistered*. Attempting to re-register memory does not cause any irrecoverable error in cuDLA and applications can continue to use cuDLA APIs even after this error has occurred.

In hybrid mode, CUDA memory may be registered implicitly as part of implicit CUDA context integration with memory sharing. When such implicit registration is in effect, an explicit call to this API for that memory (including a repeat registration) is a benign no-op that returns *cudlaSuccess* rather than *cudlaErrorMemoryRegistered*.

Note: This API may result in undefined behavior if the address being registered is not 32-byte aligned. The input pointer `ptr` must always satisfy the condition ((ptr & 0x1F) == 0)

Note: This API can return task execution errors from previous DLA task submissions.

Parameters

- **devHandle** – **[in]** - A valid cuDLA device handle create by a previous call to *cudlaCreateDevice*.
- **ptr** – **[in]** - The CUDA pointer to be registered.

- **size – [in]** - The size of the mapping i.e the number of bytes from ptr that must be mapped.
- **devPtr – [out]** - The output pointer where the mapping will be available.
- **flags – [in]** - Applications can use this flag to control several aspects of the registration process. The valid values of `flags` can be one of the following (or an OR of these values):
  - 0, default
  - *CUDLA_TASK_STATISTICS*, specifies that the external memory needs to be registered with DLA for layerwise statistics.

Returns

cudlaSuccess, *cudlaErrorInvalidDevice*, *cudlaErrorInvalidParam*, *cudlaErrorInvalidAddress*, *cudlaErrorCuda*, *cudlaErrorUmd*, *cudlaErrorOutOfResources*, *cudlaErrorMemoryRegistered*, *cudlaErrorUnsupportedOperation*, *cudlaErrorDriverNotFound*

cudlaStatus **cudlaMemUnregister**(*cudlaDevHandle* const devHandle, const uint64_t *const devPtr)

Unregisters the input memory from DLA engine.

The system mapping that enables the DLA HW to access the memory is removed. This mapping could have been created by a previous call to *cudlaMemRegister* , *cudlaImportExternalMemory* or *cudlaImportExternalSemaphore*.

Note: This API can return task execution errors from previous DLA task submissions.

Parameters

- **devHandle – [in]** - A valid cuDLA device handle create by a previous call to *cudlaCreateDevice*.
- **devPtr – [in]** - The pointer to be unregistered.

Returns

cudlaSuccess, *cudlaErrorInvalidDevice*, *cudlaErrorInvalidAddress*, *cudlaErrorUmd*, *cudlaErrorDriverNotFound*

cudlaStatus **cudlaModuleGetAttributes**(*cudlaModule* const hModule, *cudlaModuleAttributeType* const attrType, cudlaModuleAttribute *const attribute)

Get DLA module attributes.

Get module attributes from the loaded module. This API returns *cudlaErrorInvalidDevice* if the module is not loaded in any device.

Note: This API can return task execution errors from previous DLA task submissions.

Parameters

- **hModule – [in]** - The input DLA module.
- **attrType – [in]** - The attribute type that is being requested.
- **attribute – [out]** - The output pointer where the attribute will be available.

Returns

cudlaSuccess, *cudlaErrorInvalidParam*, *cudlaErrorInvalidModule*, *cudlaErrorInvalidDevice*, *cudlaErrorUmd*, *cudlaErrorInvalidAttribute*, *cudlaErrorUnsupportedOperation*, *cudlaErrorDriverNotFound*

cudlaStatus **cudlaModuleLoadFromMemory**(*cudlaDevHandle* const devHandle, const uint8_t *const pModule, size_t const moduleSize, *cudlaModule* *const hModule, uint32_t const flags)

Load a DLA module.

Loads the module into the current device handle.

- Multiple loadables are not allowed to load onto single cuDLA device handle.
- A Loadable can only be loaded once in cuDLA device handle lifecycle.

Note: This API can return task execution errors from previous DLA task submissions.

Parameters

- **devHandle** – **[in]** - The input cuDLA device handle. The module will be loaded in the context of this handle.
- **pModule** – **[in]** - A pointer to an in-memory module.
- **moduleSize** – **[in]** - The size of the module.
- **hModule** – **[out]** - The address in which the loaded module handle will be available upon successful execution.
- **flags** – **[in]** - Applications can use this flag to specify how the module is going to be used. The valid values of `flags` can be one of the following:
  - *CUDLA_MODULE_DEFAULT*, Default value which is 0.
  - *CUDLA_MODULE_ENABLE_FAULT_DIAGNOSTICS*, Application can specify this flag to load a module that is used for performing fault diagnostics for DLA HW. With this flag set, the `pModule` and `moduleSize` parameters shall be NULL and 0 as the diagnostics module is loaded internally.

Returns

cudlaSuccess, *cudlaErrorInvalidDevice*, *cudlaErrorInvalidParam*, *cudlaErrorOutOfResources*, *cudlaErrorUnsupportedOperation*, *cudlaErrorUmd*, *cudlaErrorDriverNotFound*

cudlaStatus **cudlaModuleUnload**(*cudlaModule* const hModule, uint32_t const flags)

Unload a DLA module.

Unload the module from the device handle that it was loaded into. This API returns *cudlaErrorInvalidDevice* if the module is not loaded into a valid device.

Note: This API can return task execution errors from previous DLA task submissions.

Parameters

- **hModule** – **[in]** - Handle to the loaded module.
- **flags** – **[in]** - Reserved for future. Must be set to 0.

Returns

cudlaSuccess, *cudlaErrorInvalidParam*, *cudlaErrorInvalidDevice*, *cudlaErrorInvalidModule*, *cudlaErrorUmd*, *cudlaErrorDriverNotFound*

cudlaStatus **cudlaSetTaskTimeoutInMs**(*cudlaDevHandle* const devHandle, uint32_t const timeout)

Set task timeout in millisecond.

Set task timeout in ms for each device handle. cuDLA sets 30 seconds as default timeout value if user doesn't explicitly set the timeout.

In case , device handle is invalid or timeout is 0 or timeout is greater than 1000 sec, this function would return cudlaErrorInvalidParam otherwise cudlaSuccess.

Note: This API can return task execution errors from previous DLA task submissions.

Parameters

- **devHandle** – **[in]** - A valid device handle.
- **timeout** – **[in]** - task timeout value in ms.

Returns

cudlaSuccess, *cudlaErrorInvalidParam*, *cudlaErrorDriverNotFound*

cudlaStatus **cudlaSubmitTask**(*cudlaDevHandle* const devHandle, const *cudlaTask* *const ptrToTasks, uint32_t const numTasks, void *const stream, uint32_t const flags)

Submits the inference operation on DLA.

This operation takes in a sequence of tasks and submits them to the DLA HW for execution in the same sequence as they appear in the input task array. The input and output tensors (and statistics buffer if used) are assumed to be pre-registered using *cudlaMemRegister* (in hybrid mode) or *cudlaImportExternalMemory* (in standalone mode). Failure to do so can result in this function returning *cudlaErrorInvalidAddress*.

The `stream` parameter must be specified as the CUDA stream on which the DLA task is submitted for execution in hybrid mode. In standalone mode, this parameter must be passed as NULL and failure to do so will result in this function returning *cudlaErrorInvalidParam*.

The *cudlaTask* structure has a provision to specify wait and signal events that cuDLA must wait on and signal respectively as part of *cudlaSubmitTask()*. In hybrid mode, the waitEvents and signalEvents fields of the *cudlaTask* structure must be set to NULL. Wait and signal events are only supported in standalone mode. Failure to set these fields to NULL in hybrid mode will result in this function returning *cudlaErrorUnsupportedOperation*.

In standalone mode, each submitted task will wait for all its wait events to be signaled before beginning execution and will provide a signal event (if one is requested for during *cudlaSubmitTask*) that the application (or any other entity) can wait on to ensure that the submitted task has completed execution. In cuDLA 1.0, only NvSciSync fences are supported as part of wait events. Furthermore, only NvSciSync objects (registered as part of *cudlaImportExternalSemaphore*) can

be signaled as part of signal events and the fence corresponding to the signaled event is returned as part of *cudlaSubmitTask*.

Event-Only submissions - In standalone mode, if inputTensor and outputTensor fields are set to NULL inside the *cudlaTask* structure, the task submission is interpreted as an enqueue of wait and signal events that must be considered for subsequent task submission. No actual task submission is done. Multiple such subsequent task submissions with NULL fields in the input/outputTensor fields will overwrite the list of wait and signal events to be considered. In other words, the latest non-null wait events and/or latest non-null signal events before a non-null submission are considered for subsequent actual task submission. During an actual task submit in standalone mode, the effective wait events and signal events that will be considered are what the application sets using NULL data submissions and what is set for that particular task submission in the waitEvents and signalEvents fields. The wait events set as part of NULL data submission are considered as dependencies for only the first task and the signal events set as part of NULL data submission are signaled when the last task of task list is complete. All constraints that apply to waitEvents and signalEvents individually (as described below) are also applicable to the combined list.

Configurations allowed in cuDLA for a task submission

Data types used by cuDLA driver

cuDLA API

This section describes the application programming interface of the cuDLA driver.

Chapter 2. Structs

2.1. CudlaFence

Defined in cudla.h

struct **CudlaFence**

Fence description.

Public Members

void ***fence**

Fence.

cudlaFenceType **type**

Fence type.

2.2. cudlaExternalMemoryHandleDesc_t

Defined in cudla.h

struct **cudlaExternalMemoryHandleDesc_t**

External memory handle descriptor.

Public Members

const void ***extBufObject**

A handle representing an external memory object.

unsigned long long **size**

Size of the memory allocation.

2.3. cudlaExternalSemaphoreHandleDesc_t

Defined in cudla.h

struct **cudlaExternalSemaphoreHandleDesc_t**

External semaphore handle descriptor.

Public Members

const void ***extSyncObject**

A handle representing an external synchronization object.

2.4. cudlaModuleTensorDescriptor

Defined in cudla.h

struct **cudlaModuleTensorDescriptor**

Tensor descriptor.

Public Members

uint64_t **c**

uint8_t **dataCategory**

uint8_t **dataFormat**

uint8_t **dataType**

uint64_t **h**

uint64_t **n**

char **name**[80U + 1]

uint8_t **pixelFormat**

uint8_t **pixelMapping**

uint64_t **size**

uint32_t **stride**[8U]

uint64_t **w**

2.5. cudlaSignalEvents

Defined in cudla.h

struct **cudlaSignalEvents**

Signal events for *cudlaSubmitTask*.

Public Members

uint64_t *const ***devPtrs**

Array of registered synchronization objects (via *cudlaImportExternalSemaphore*).

CudlaFence ***eofFences**

Array of fences pointers for all the signal events corresponding to the synchronization objects.

uint32_t **numEvents**

Total number of signal events.

2.6. cudlaTask

Defined in cudla.h

struct **cudlaTask**

Structure of Task.

Public Members

const uint64_t *const ***inputTensor**

Array of input tensors.

cudlaModule **moduleHandle**

cuDLA module handle.

uint32_t **numInputTensors**
: Number of input tensors.

uint32_t **numOutputTensors**
: Number of output tensors.

uint64_t *const ***outputTensor**
: Array of output tensors.

cudlaSignalEvents ***signalEvents**
: Signal events.

const *cudlaWaitEvents* ***waitEvents**
: Wait events.

2.7. cudlaWaitEvents

Defined in cudla.h

struct **cudlaWaitEvents**
: Wait events for *cudlaSubmitTask*.

Public Members

uint32_t **numEvents**
: Total number of wait events.

const *CudlaFence* ***preFences**
: Array of fence pointers for all the wait events.

CudlaFence
: Fence description.

cudlaExternalMemoryHandleDesc_t
: External memory handle descriptor.

cudlaExternalSemaphoreHandleDesc_t
: External semaphore handle descriptor.

cudlaModuleTensorDescriptor
: Tensor descriptor.

cudlaSignalEvents
: Signal events for *cudlaSubmitTask* .

cudlaTask
: Structure of Task.

cudlaWaitEvents

Wait events for *cudlaSubmitTask* .

Chapter 3. Notices

3.1. Notice

This document is provided for information purposes only and shall not be regarded as a warranty of a certain functionality, condition, or quality of a product. NVIDIA Corporation ("NVIDIA") makes no representations or warranties, expressed or implied, as to the accuracy or completeness of the information contained in this document and assumes no responsibility for any errors contained herein. NVIDIA shall have no liability for the consequences or use of such information or for any infringement of patents or other rights of third parties that may result from its use. This document is not a commitment to develop, release, or deliver any Material (defined below), code, or functionality.

NVIDIA reserves the right to make corrections, modifications, enhancements, improvements, and any other changes to this document, at any time without notice.

Customer should obtain the latest relevant information before placing orders and should verify that such information is current and complete.

NVIDIA products are sold subject to the NVIDIA standard terms and conditions of sale supplied at the time of order acknowledgement, unless otherwise agreed in an individual sales agreement signed by authorized representatives of NVIDIA and customer ("Terms of Sale"). NVIDIA hereby expressly objects to applying any customer general terms and conditions with regards to the purchase of the NVIDIA product referenced in this document. No contractual obligations are formed either directly or indirectly by this document.

NVIDIA products are not designed, authorized, or warranted to be suitable for use in medical, military, aircraft, space, or life support equipment, nor in applications where failure or malfunction of the NVIDIA product can reasonably be expected to result in personal injury, death, or property or environmental damage. NVIDIA accepts no liability for inclusion and/or use of NVIDIA products in such equipment or applications and therefore such inclusion and/or use is at customer's own risk.

NVIDIA makes no representation or warranty that products based on this document will be suitable for any specified use. Testing of all parameters of each product is not necessarily performed by NVIDIA. It is customer's sole responsibility to evaluate and determine the applicability of any information contained in this document, ensure the product is suitable and fit for the application planned by customer, and perform the necessary testing for the application in order to avoid a default of the application or the product. Weaknesses in customer's product designs may affect the quality and reliability of the NVIDIA product and may result in additional or different conditions and/or requirements beyond those contained in this document. NVIDIA accepts no liability related to any default, damage, costs, or problem which may be based on or attributable to: (i) the use of the NVIDIA product in any manner that is contrary to this document or (ii) customer product designs.

No license, either expressed or implied, is granted under any NVIDIA patent right, copyright, or other NVIDIA intellectual property right under this document. Information published by NVIDIA regarding third-party products or services does not constitute a license from NVIDIA to use such products or

services or a warranty or endorsement thereof. Use of such information may require a license from a third party under the patents or other intellectual property rights of the third party, or a license from NVIDIA under the patents or other intellectual property rights of NVIDIA.

Reproduction of information in this document is permissible only if approved in advance by NVIDIA in writing, reproduced without alteration and in full compliance with all applicable export laws and regulations, and accompanied by all associated conditions, limitations, and notices.

THIS DOCUMENT AND ALL NVIDIA DESIGN SPECIFICATIONS, REFERENCE BOARDS, FILES, DRAWINGS, DIAGNOSTICS, LISTS, AND OTHER DOCUMENTS (TOGETHER AND SEPARATELY, "MATERIALS") ARE BEING PROVIDED "AS IS." NVIDIA MAKES NO WARRANTIES, EXPRESSED, IMPLIED, STATUTORY, OR OTHERWISE WITH RESPECT TO THE MATERIALS, AND EXPRESSLY DISCLAIMS ALL IMPLIED WARRANTIES OF NONINFRINGEMENT, MERCHANTABILITY, AND FITNESS FOR A PARTICULAR PURPOSE. TO THE EXTENT NOT PROHIBITED BY LAW, IN NO EVENT WILL NVIDIA BE LIABLE FOR ANY DAMAGES, INCLUDING WITHOUT LIMITATION ANY DIRECT, INDIRECT, SPECIAL, INCIDENTAL, PUNITIVE, OR CONSEQUENTIAL DAMAGES, HOWEVER CAUSED AND REGARDLESS OF THE THEORY OF LIABILITY, ARISING OUT OF ANY USE OF THIS DOCUMENT, EVEN IF NVIDIA HAS BEEN ADVISED OF THE POSSIBILITY OF SUCH DAMAGES. Notwithstanding any damages that customer might incur for any reason whatsoever, NVIDIA's aggregate and cumulative liability towards customer for the products described herein shall be limited in accordance with the Terms of Sale for the product.

3.2. OpenCL

OpenCL is a trademark of Apple Inc. used under license to the Khronos Group Inc.

3.3. Trademarks

NVIDIA and the NVIDIA logo are trademarks or registered trademarks of NVIDIA Corporation in the U.S. and other countries. Other company and product names may be trademarks of the respective companies with which they are associated.

Copyright

©2007-2026, NVIDIA Corporation & affiliates. All rights reserved