cuDLA API

API Reference Manual

vRelease Version | January 2022

Table of Contents

Chapter 1. Modules

Here is a list of all modules:

- Data types used by cuDLA driver
- cuDLA API

1.1. Data types used by cuDLA driver

union cudlaDevAttribute

struct cudlaExternalMemoryHandleDesc_t

struct cudlaExternalSemaphoreHandleDesc_t

struct CudlaFence

union cudlaModuleAttribute

struct cudlaModuleTensorDescriptor

struct cudlaSignalEvents

struct cudlaTask

struct cudlaWaitEvents

enum cudlaAccessPermissionFlags

Access permission flags for importing NvSciBuffers

Values

CUDLA_READ_WRITE_PERM = 0
Flag to import memory with read-write permission

CUDLA_READ_ONLY_PERM = 1
Flag to import memory with read-only permission

CUDLA_TASK_STATISTICS = 1<<1
Flag to indicate buffer as layerwise statistics buffer.

enum cudlaDevAttributeType

Device attribute type.

Values

CUDLA_UNIFIED_ADDRESSING = 0
Flag to check for support for UVA.

CUDLA_DEVICE_VERSION = 1

Flag to check for DLA HW version.

enum cudlaFenceType

Supported fence types.

Values

CUDLA_NVSCISYNC_FENCE = 1
NvSciSync fence type for EOF.
CUDLA_NVSCISYNC_FENCE_SOF = 2

enum cudlaMode

Device creation modes.

Values

CUDLA_CUDA_DLA = 0
Hyrbid mode.
CUDLA_STANDALONE = 1
Standalone mode.

enum cudlaModuleAttributeType

Module attribute types.

Values

CUDLA_NUM_INPUT_TENSORS = 0
Flag to retrieve number of input tensors.
CUDLA_NUM_OUTPUT_TENSORS = 1
Flag to retrieve number of output tensors.
CUDLA_INPUT_TENSOR_DESCRIPTORS = 2
Flag to retrieve all the input tensor descriptors.
CUDLA_OUTPUT_TENSOR_DESCRIPTORS = 3
Flag to retrieve all the output tensor descriptors.
CUDLA_NUM_OUTPUT_TASK_STATISTICS = 4
Flag to retrieve total number of output task statistics buffer.
CUDLA_OUTPUT_TASK_STATISTICS_DESCRIPTORS = 5
Flag to retrieve all the output task statistics descriptors.

enum cudlaModuleLoadFlags

Module load flags for cudlaModuleLoadFromMemory.

Values

CUDLA_MODULE_DEFAULT = 0
Default flag.

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

CUDLA_NVSCISYNC_ATTR_WAIT = 1
Wait attribute.

CUDLA_NVSCISYNC_ATTR_SIGNAL = 2
Signal attribute.

enum cudlaStatus

Error codes.

Values

cudlaSuccess = 0
The API call returned with no errors.

cudlaErrorInvalidParam = 1
This indicates that one or more parameters passed to the API is/are incorrect.

cudlaErrorOutOfResources = 2
This indicates that the API call failed due to lack of underlying resources.

cudlaErrorCreationFailed = 3
This indicates that an internal error occurred during creation of device handle.

cudlaErrorInvalidAddress = 4
This indicates that the memory object being passed in the API call has not been registered before.

cudlaErrorOs = 5
This indicates that an OS error occurred.

cudlaErrorCuda = 6
This indicates that there was an error in a CUDA operation as part of the API call.

cudlaErrorUmd = 7
This indicates that there was an error in the DLA runtime for the API call.

cudlaErrorInvalidDevice = 8
This indicates that the device handle passed to the API call is invalid.

cudlaErrorInvalidAttribute = 9
This indicates that an invalid attribute is being requested.

cudlaErrorIncompatibleDlaSWVersion = 10
This indicates that the underlying DLA runtime is incompatible with the current cuDLA version.
cudlaErrorMemoryRegistered = 11
This indicates that the memory object is already registered.
cudlaErrorInvalidModule = 12
This indicates that the module being passed is invalid.
cudlaErrorUnsupportedOperation = 13
This indicates that the operation being requested by the API call is unsupported.
cudlaErrorNvSci = 14
This indicates that the NvSci operation requested by the API call failed.
cudlaErrorDlaErrInvalidInput = 0x40000001
DLA HW Error.
cudlaErrorDlaErrInvalidPreAction = 0x40000002
DLA HW Error.
cudlaErrorDlaErrNoMem = 0x40000003
DLA HW Error.
cudlaErrorDlaErrProcessorBusy = 0x40000004
DLA HW Error.
cudlaErrorDlaErrTaskStatusMismatch = 0x40000005
DLA HW Error.
cudlaErrorDlaErrEngineTimeout = 0x40000006
DLA HW Error.
cudlaErrorDlaErrDataMismatch = 0x40000007
DLA HW Error.
cudlaErrorUnknown = 0x7fffffff
This indicates that an unknown error has occurred.

enum cudlaSubmissionFlags

Task submission flags for cudlaSubmitTask.

Values

CUDLA_SUBMIT_NOOP = 1
Flag to specify that the submitted task must be bypassed for execution.
CUDLA_SUBMIT_SKIP_LOCK_ACQUIRE = 1<<1
Flag to specify that the global lock acquire must be skipped.
CUDLA_SUBMIT_DIAGNOSTICS_TASK = 1<<2
Flag to specify that the submitted task is to run permanent fault diagnostics for DLA HW.

typedef cudlaDevHandle_t *cudlaDevHandle

cuDLA Device Handle

typedef cudlaModule_t *cudlaModule

cuDLA Module Handle

1.2. cuDLA API

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

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

Create a device handle.

Parameters

device

- Device number (can be 0 or 1).

devHandle

- Pointer to hold the created cuDLA device handle.

flags

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.

- Flags controlling device creation. Valid values for `flags` are:

Returns

cudlaSuccess, cudlaErrorOutOfResources, cudlaErrorInvalidParam, cudlaErrorIncompatibleDlaSWVersion, cudlaErrorCreationFailed, cudlaErrorCuda, cudlaErrorUmd, cudlaErrorUnsupportedOperation

- 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.

Description

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.

cudlaStatus cudlaDestroyDevice (const cudlaDevHandle devHandle)

Destroy device handle.

Parameters

devHandle
- A valid device handle.

Returns

cudlaSuccess, cudlaErrorInvalidDevice, cudlaErrorCuda, cudlaErrorUmd

Description

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.

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

Get cuDLA device attributes.

Parameters

devHandle
- The input cuDLA device handle.

attrib
- The attribute that is being requested.

pAttribute
- The output pointer where the attribute will be available.

Returns

cudlaSuccess, cudlaErrorInvalidParam, cudlaErrorInvalidDevice, cudlaErrorUmd, cudlaErrorInvalidAttribute

Description

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.

cudlaStatus cudlaDeviceGetCount (const uint64_t *pNumDevices)

Get device count.

Parameters

pNumDevices

- The number of DLA devices will be available in this variable upon successful completion.

Returns

cudlaSuccess, cudlaErrorInvalidParam, cudlaErrorUmd, cudlaErrorIncompatibleDlaSWVersion

Description

Get number of DLA devices available to use.

cudlaStatus cudlaGetLastError (const cudlaDevHandle devHandle)

Gets the last asynchronous error in task execution.

Parameters

devHandle

- A valid device handle.

Returns

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

Description

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.

cudlaStatus cudlaGetNvSciSyncAttributes (uint64_t *attrList, const uint32_t flags)

Get cuDLA's NvSciSync attributes.

Parameters

attrList

- Attribute list created by the application.

flags

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.

- 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):

Returns

cudlaSuccess, cudlaErrorInvalidParam, cudlaErrorUnsupportedOperation, cudlaErrorInvalidAttribute, cudlaErrorNvSci

- 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.

Description

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

cuDLA supports two types of NvSciSync object primitives -

- Sync point
- Deterministic semaphore cuDLA prioritizes sync point primitive over deterministic semaphore primitive 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.

cudlaStatus cudlaGetVersion (const uint64_t *version)

Returns the version number of the library.

Parameters

version

- cuDLA library version will be available in this variable upon successful execution.

Returns

cudlaSuccess, cudlaErrorInvalidParam

Description

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

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

Imports external memory into cuDLA.

Parameters

devHandle

- A valid device handle.

desc

- Contains description about allocated external memory.

devPtr

- The output pointer where the mapping will be available.

flags

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.

- 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:

Returns

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

- 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.

Description

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 will result in an undefined behaviour.

Note:

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

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

Imports external semaphore into cuDLA.

Parameters

devHandle
- A valid device handle.

desc
- Contains sempahore object.

devPtr
- The output pointer where the mapping will be available.

flags
- Reserved for future. Must be set to 0.

Returns

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

Description

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.

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

Registers the CUDA memory to DLA engine.

Parameters

devHandle

- A valid cuDLA device handle create by a previous call to cudlaCreateDevice.

ptr

- The CUDA pointer to be registered.

size

- The size of the mapping i.e the number of bytes from ptr that must be mapped.

devPtr

- The output pointer where the mapping will be available.

flags

0, default CUDLA_TASK_STATISTICS, specifies that the external memory needs to be registered with DLA for layerwise statistics.

- 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):

Returns

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

- 0, default
- CUDLA_TASK_STATISTICS, specifies that the external memory needs to be registered with DLA for layerwise statistics.

Description

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.

Note:

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

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

Unregisters the input memory from DLA engine.

Parameters

devHandle

- A valid cuDLA device handle create by a previous call to cudlaCreateDevice.

devPtr

- The pointer to be unregistered.

Returns

cudlaSuccess, cudlaErrorInvalidDevice, cudlaErrorInvalidAddress, cudlaErrorUmd

Description

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.

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

Get DLA module attributes.

Parameters

hModule

- The input DLA module.

attrType

- The attribute type that is being requested.

attribute

- The output pointer where the attribute will be available.

Returns

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

Description

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

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

Load a DLA module.

Parameters

devHandle

- The input cuDLA device handle. The module will be loaded in the context of this handle.

pModule

- A pointer to an in-memory module.

moduleSize

- The size of the module.

hModule

- The address in which the loaded module handle will be available upon successful execution.

flags

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.

- 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:

Returns

cudlaSuccess, cudlaErrorInvalidDevice, cudlaErrorInvalidParam, cudlaErrorOutOfResources, cudlaErrorUnsupportedOperation, cudlaErrorUmd

- 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.

Description

Loads the module into the current device handle. Currently, DLA supports only 1 loadable per device handle. So, attempting to load another loadable in the same device handle would return with an error code of cudlaErrorUnsupportedOperation.

cudlaStatus cudlaModuleUnload (const cudlaModule hModule, const uint32_t flags)

Unload a DLA module.

Parameters

hModule
- Handle to the loaded module.

flags
- Reserved for future. Must be set to 0.

Returns

cudlaSuccess, cudlaErrorInvalidParam, cudlaErrorInvalidDevice, cudlaErrorInvalidModule, cudlaErrorUmd

Description

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.

cudlaStatus cudlaSetTaskTimeoutInMs (const cudlaDevHandle devHandle, const uint32_t timeout)

Set task timeout in millisecond.

Parameters

devHandle
- A valid device handle.

timeout
- task timeout value in ms.

Returns

cudlaSuccess, cudlaErrorInvalidParam

Description

Set task timeout in ms for each device handle

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

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

Submits the inference operation on DLA.

Parameters

devHandle

- A valid cuDLA device handle.

ptrToTasks

- A list of inferencing tasks.

numTasks

- The number of tasks.

stream

- The stream on which the DLA task has to be submitted.

flags

0, default CUDLA_SUBMIT_NOOP, specifies that the submitted task must be skipped during execution on the DLA. However, all the waitEvents and signalEvents dependencies must be satisfied. This flag is ignored when NULL data submissions are being done as in that case only the wait and signal events are internally stored for the next task submission. CUDLA_SUBMIT_SKIP_LOCK_ACQUIRE, specifies that the submitted task is being enqueued in a device handle and that no other task is being enqueued in that device handle at that time in any other thread. This is a flag that apps can use as an optimization. Ordinarily, the cuDLA APIs acquire a global lock internally to guarantee thread safety. However, this lock causes unwanted serialization in cases where the the applications are submitting tasks to different device handles. If an application was submitting one or more tasks in multiple threads and if these submissions are to different device handles and if there is no shared data being provided as part of the task information in the respective submissions then applications can specify this flag during submission so that the internal lock acquire is skipped. Shared data also includes the input stream in hybrid mode operation. Therefore, if the same stream is being used to submit two different tasks and even if the two device handles are different, the usage of this flag is invalid. CUDLA_SUBMIT_DIAGNOSTICS_TASK, specifies that the submitted task is to run permanent fault diagnostics for DLA HW. User can use this task to probe the state of DLA HW. With this flag set, in standalone mode user is not allowed to do event only submissions, where tensor information is NULL and only events (wait/signal or both) are present in task. This is because the task always runs on a internally loaded diagnostic module. This diagnostic module does not expect any input tensors and so input tensor memory, however user is

expected to query no. of output tensors, allocate the output tensor memory and pass the same while using the submit task.

- Applications can use this flag to control several aspects of the submission process. The valid values of `flags` can be one of the following (or an OR of these values):

Returns

cudlaSuccess, cudlaErrorInvalidParam, cudlaErrorInvalidDevice, cudlaErrorInvalidModule, cudlaErrorCuda, cudlaErrorUmd, cudlaErrorOutOfResources, cudlaErrorInvalidAddress, cudlaErrorUnsupportedOperation, cudlaErrorInvalidAttribute, cudlaErrorNvSci cudlaErrorOs

- 0, default
- CUDLA_SUBMIT_NOOP, specifies that the submitted task must be skipped during execution on the DLA. However, all the waitEvents and signalEvents dependencies must be satisfied. This flag is ignored when NULL data submissions are being done as in that case only the wait and signal events are internally stored for the next task submission.
- CUDLA_SUBMIT_SKIP_LOCK_ACQUIRE, specifies that the submitted task is being enqueued in a device handle and that no other task is being enqueued in that device handle at that time in any other thread. This is a flag that apps can use as an optimization. Ordinarily, the cuDLA APIs acquire a global lock internally to guarantee thread safety. However, this lock causes unwanted serialization in cases where the the applications are submitting tasks to different device handles. If an application was submitting one or more tasks in multiple threads and if these submissions are to different device handles and if there is no shared data being provided as part of the task information in the respective submissions then applications can specify this flag during submission so that the internal lock acquire is skipped. Shared data also includes the input stream in hybrid mode operation. Therefore, if the same stream is being used to submit two different tasks and even if the two device handles are different, the usage of this flag is invalid.
- CUDLA_SUBMIT_DIAGNOSTICS_TASK, specifies that the submitted task is to run permanent fault diagnostics for DLA HW. User can use this task to probe the state of DLA HW. With this flag set, in standalone mode user is not allowed to do event only submissions, where tensor information is NULL and only events (wait/signal or both) are present in task. This is because the task always runs on a internally loaded diagnostic module. This diagnostic module does not expect any input tensors and so input tensor memory, however user is expected to query no. of output tensors, allocate the output tensor memory and pass the same while using the submit task.

Description

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(). 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.

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 submissions. 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 wait and signal events considered are effectively what are specified in the last submit call with NULL data fields. 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.

For wait events, applications are expected to

- register their synchronization objects using cudlaImportExternalSemaphore.
- create the required number of fence placeholders using CudlaFence.
- fill in the placeholders with the relevant fences from the application.
- list out all the fences in cudlaWaitEvents.

For signal events, applications are expected to

- register their synchronization objects using cudlaImportExternalSemaphore.
- create the required number of placeholder fences using CudlaFence. cuDLA supports 2 kinds of Fences, SOF and EOF Fence.
  - SOF(Start Of Frame) Fence is the type of fence which is signaled before the task execution on DLA starts. Use cudlaFenceType as CUDLA_NVSCISYNC_FENCE_SOF to mark a fence as SOF fence.

    - EOF(End Of Frame) Fence is the type of fence which is signaled after the task execution on DLA is complete. Use cudlaFenceType as CUDLA_NVSCISYNC_FENCE to mark a fence as EOF fence.
- place the registered objects and the corresponding fences in cudlaSignalEvents. In case ofdeterministic semaphore, fence is not required to be passed in cudlaSignalEvents.

  When cudlaSubmitTask returns successfully, the fences present in cudlaSignalEvents can be used to wait for the particular task to be completed. cuDLA supports 1 sync point and any number of semaphores as part of cudlaSignalEvents. If more than 1 sync point is specified, cudlaErrorInvalidParam is returned.

During submission, users have an option to enable layerwise statistics profiling for the individual layers of the network. This option needs to be exercised by specifying additional output buffers that would contain the profiling information. Specifically,

- "cudlaTask::numOutputTensors" should be the sum of value returned by cudlaModuleGetAttributes(...,CUDLA_NUM_OUTPUT_TENSORS,...) and cudlaModuleGetAttributes(...,CUDLA_NUM_OUTPUT_TASK_STATISTICS,...)
- "cudlaTask::outputTensor" should contain the array of output tensors appended with array of statistics output buffer.

This function can return cudlaErrorUnsupportedOperation if

- stream being used in hybrid mode is in capturing state.
- application attempts to use NvSci functionalities in hybrid mode.
- loading of NvSci libraries failed for a particular platform.
- fence type other than CUDLA_NVSCISYNC_FENCE is specified.
- waitEvents or signaEvents is not NULL in hybrid mode.
- inputTensor or outputTensor is NULL in hybrid mode and the flags are not CUDLA_SUBMIT_DIAGNOSTICS_TASK.
- inputTensor is NULL and outputTensor is not NULL and vice versa in standalone mode and the flags are not CUDLA_SUBMIT_DIAGNOSTICS_TASK.
- inputTensor and outputTensor is NULL and number of tasks is not equal to 1 in standalone mode and the flags are not CUDLA_SUBMIT_DIAGNOSTICS_TASK.
- inputTensor is not NULL or output tensor is NULL and the flags are CUDLA_SUBMIT_DIAGNOSTICS_TASK.
- the effective signal events list has multiple sync points to signal.
- if layerwise feature is unsupported.

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

This function can return cudlaErrorOs if an internal system operation fails.

Note:

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

Chapter 2. Data Structures

Here are the data structures with brief descriptions:

2.1. cudlaDevAttribute Union Reference

Device attribute.

uint32_t cudlaDevAttribute::deviceVersion

DLA device version. Xavier has 1.0 and Orin has 2.0.

uint8_t cudlaDevAttribute::unifiedAddressingSupported

Returns 0 if unified addressing is not supported.

2.2. cudlaExternalMemoryHandleDesc_t Struct Reference

External memory handle descriptor.

const void *cudlaExternalMemoryHandleDesc_t::extBufObject

A handle representing an external memory object.

unsigned long long cudlaExternalMemoryHandleDesc_t::size

Size of the memory allocation

2.3. cudlaExternalSemaphoreHandleDesc_t Struct Reference

External semaphore handle descriptor.

const void *cudlaExternalSemaphoreHandleDesc_t::extSyncObject

A handle representing an external synchronization object.

2.4. CudlaFence Struct Reference

Fence description.

void *CudlaFence::fence

Fence.

cudlaFenceType CudlaFence::type

Fence type.

2.5. cudlaModuleAttribute Union Reference

Module attribute.

cudlaModuleTensorDescriptor *cudlaModuleAttribute::inputTensorDesc

Returns an array of input tensor descriptors.

uint32_t cudlaModuleAttribute::numInputTensors

Returns the number of input tensors.

uint32_t cudlaModuleAttribute::numOutputTensors

Returns the number of output tensors.

cudlaModuleTensorDescriptor *cudlaModuleAttribute::outputTensorDesc

Returns an array of output tensor descriptors.

2.6. cudlaModuleTensorDescriptor Struct Reference

Tensor descriptor.

2.7. cudlaSignalEvents Struct Reference

Signal events for cudlaSubmitTask

const **cudlaSignalEvents::devPtrs

Array of registered synchronization objects (via cudlaImportExternalSemaphore).

CudlaFence *cudlaSignalEvents::eofFences

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

uint32_t cudlaSignalEvents::numEvents

Total number of signal events.

2.8. cudlaTask Struct Reference

Structure of Task.

const **cudlaTask::inputTensor

Array of input tensors.

cudlaModule cudlaTask::moduleHandle

cuDLA module handle.

uint32_t cudlaTask::numInputTensors

Number of input tensors.

uint32_t cudlaTask::numOutputTensors

Number of output tensors.

const **cudlaTask::outputTensor

Array of output tensors.

cudlaSignalEvents *cudlaTask::signalEvents

Signal events.

const cudlaWaitEvents *cudlaTask::waitEvents

Wait events.

2.9. cudlaWaitEvents Struct Reference

Wait events for cudlaSubmitTask.

uint32_t cudlaWaitEvents::numEvents

Total number of wait events.

const CudlaFence *cudlaWaitEvents::preFences

Array of fence pointers for all the wait events.

Chapter 3. Data Fields

Here is a list of all documented struct and union fields with links to the struct/union documentation for each field:

deviceVersion
cudlaDevAttribute
devPtrs
cudlaSignalEvents
eofFences
cudlaSignalEvents
extBufObject
cudlaExternalMemoryHandleDesc
extSyncObject
cudlaExternalSemaphoreHandleDesc
fence
CudlaFence
inputTensor
cudlaTask
inputTensorDesc
cudlaModuleAttribute
moduleHandle
cudlaTask
numEvents
cudlaWaitEvents
cudlaSignalEvents
numInputTensors
cudlaTask
cudlaModuleAttribute
numOutputTensors
cudlaTask
cudlaModuleAttribute
outputTensor
cudlaTask
outputTensorDesc
cudlaModuleAttribute

preFences
cudlaWaitEvents
signalEvents
cudlaTask
size
cudlaExternalMemoryHandleDesc
type
CudlaFence
unifiedAddressingSupported
cudlaDevAttribute
waitEvents
cudlaTask

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.

OpenCL

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

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

NVIDIA Corporation | 2788 San Tomas Expressway, Santa Clara, CA 95051
https://www.nvidia.com