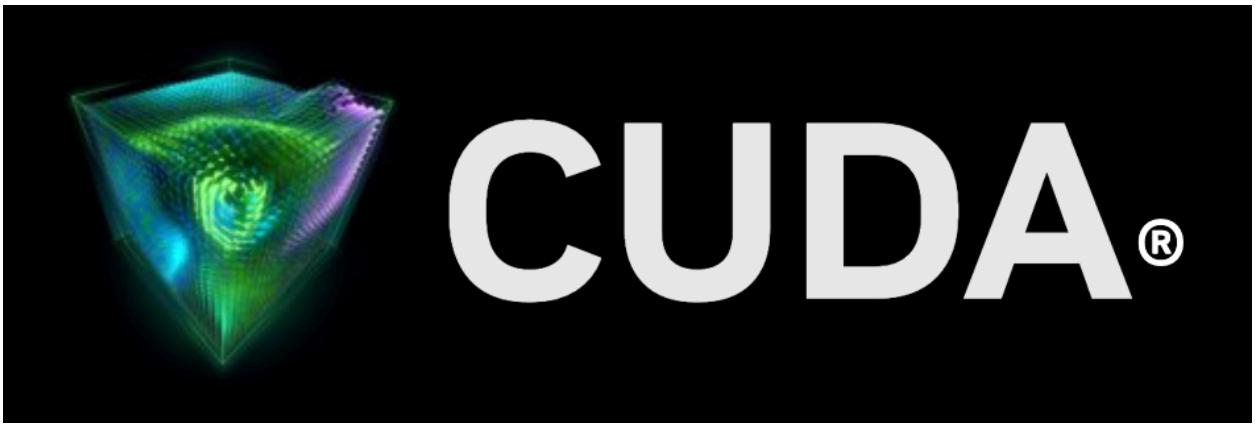

GPUDirect RDMA

Release 13.4

NVIDIA Corporation

Sep 09, 2026

Contents

1 How GPUDirect RDMA Works **3**

2 Standard DMA Transfer **5**

3 GPUDirect RDMA Transfers **7**

4 Change log **9**
4.1 Changes in CUDA 13.0 . . . 9
4.1.1 Deprecation Notice: NV-P2P API Support . . . 9
4.1.2 Device Attribute Updates . . . 9
4.1.3 2.1.3. Platform Notes: Orin (L4T) . . . 10
4.1.4 Known Issues . . . 10
4.2 Changes in CUDA 12.2 . . . 10
4.3 Changes in CUDA 11.4 . . . 11
4.4 Changes in CUDA 11.2 . . . 11
4.5 Changes in CUDA 10.1 . . . 11
4.6 Changes in CUDA 8.0 . . . 12
4.7 Changes in CUDA 7.0 . . . 12
4.8 Changes in CUDA 6.0 . . . 13

5 Design Considerations **15**
5.1 Lazy Unpinning Optimization . . . 15
5.2 Registration Cache . . . 15
5.3 Unpin Callback . . . 16
5.4 Supported Systems . . . 17
5.5 PCI BAR sizes . . . 17
5.6 Tokens Usage . . . 18
5.7 Synchronization and Memory Ordering . . . 19
5.8 CUDA Features Not Supported on Tegra . . . 20

6 How to Perform Specific Tasks **21**
6.1 Displaying GPU BAR space . . . 21
6.2 Pinning GPU memory . . . 21
6.3 Unpinning GPU memory . . . 22
6.4 Handling the free callback . . . 23
6.5 Buffer ID Tag Check for A Registration Cache . . . 23
6.6 Linking a Kernel Module against nvidia.ko . . . 24
6.7 Using nvidia-peermem . . . 25

7 References **27**
7.1 Basics of UVA CUDA Memory Management . . . 27
7.2 Userspace API . . . 28
7.3 Kernel API . . . 30
7.4 Porting to Tegra . . . 32

7.4.1 Changing the allocator . . . 33
7.4.2 Modification to Kernel API . . . 33
7.4.3 Other highlights . . . 34
7.5 Porting strategy for device drivers based on nv-p2p . . . 35

8 Notices **37**
8.1 Notice . . . 37
8.2 OpenCL . . . 38
8.3 Trademarks . . . 38

Developing a Linux Kernel Module using GPUDirect RDMA

The API reference guide for enabling GPUDirect RDMA connections to NVIDIA GPUs.

GPUDirect RDMA is a technology introduced in Kepler-class GPUs and CUDA 5.0 that enables a direct path for data exchange between the GPU and a third-party peer device using standard features of PCI Express. Examples of third-party devices are: network interfaces, video acquisition devices, storage adapters.

GPUDirect RDMA is available on both Tesla and Quadro GPUs.

A number of limitations can apply, the most important being that the two devices must share the same upstream PCI Express root complex. Some of the limitations depend on the platform used and could be lifted in current/future products.

A few straightforward changes must be made to device drivers to enable this functionality with a wide range of hardware devices. This document introduces the technology and describes the steps necessary to enable an GPUDirect RDMA connection to NVIDIA GPUs on Linux.

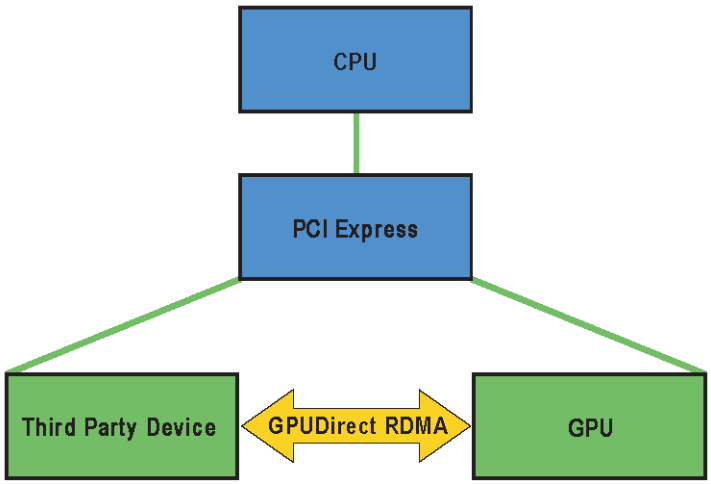

Fig. 1: GPUDirect RDMA within the Linux Device Driver Model

Chapter 1. How GPUDirect RDMA Works

When setting up GPUDirect RDMA communication between two peers, all physical addresses are the same from the PCI Express devices' point of view. Within this physical address space are linear windows called PCI BARs. Each device has six BAR registers at most, so it can have up to six active 32bit BAR regions. 64bit BARs consume two BAR registers. The PCI Express device issues reads and writes to a peer device's BAR addresses in the same way that they are issued to system memory.

Traditionally, resources like BAR windows are mapped to user or kernel address space using the CPU's MMU as memory mapped I/O (MMIO) addresses. However, because current operating systems don't have sufficient mechanisms for exchanging MMIO regions between drivers, the NVIDIA kernel driver exports functions to perform the necessary address translations and mappings.

To add GPUDirect RDMA support to a device driver, a small amount of address mapping code within the kernel driver must be modified. This code typically resides near existing calls to `get_user_pages()`.

The APIs and control flow involved with GPUDirect RDMA are very similar to those used with standard DMA transfers.

Refer to *Supported Systems* and *PCI BAR sizes* for more hardware details.

Chapter 2. Standard DMA Transfer

First, we outline a standard DMA Transfer initiated from userspace. In this scenario, the following components are present:

- Userspace program
- Userspace communication library
- Kernel driver for the device interested in doing DMA transfers

The general sequence is as follows:

1. The userspace program requests a transfer via the userspace communication library. This operation takes a pointer to data (a virtual address) and a size in bytes.
2. The communication library must make sure the memory region corresponding to the virtual address and size is ready for the transfer. If this is not the case already, it has to be handled by the kernel driver (next step).
3. The kernel driver receives the virtual address and size from the userspace communication library. It then asks the kernel to translate the virtual address range to a list of physical pages and make sure they are ready to be transferred to or from. We will refer to this operation as pinning the memory.
4. The kernel driver uses the list of pages to program the physical device's DMA engine(s).
5. The communication library initiates the transfer.
6. After the transfer is done, the communication library should eventually clean up any resources used to pin the memory. We will refer to this operation as unpinning the memory.

Chapter 3. GPUDirect RDMA Transfers

For the communication to support GPUDirect RDMA transfers some changes to the sequence above have to be introduced. First of all, two new components are present:

- Userspace CUDA library
- NVIDIA kernel driver

As described in *Basics of UVA CUDA Memory Management*, programs using the CUDA library have their address space split between GPU and CPU virtual addresses, and the communication library has to implement two separate paths for them.

The userspace CUDA library provides a function that lets the communication library distinguish between CPU and GPU addresses. Moreover, for GPU addresses it returns additional metadata that is required to uniquely identify the GPU memory represented by the address. Refer to *Userspace API* for details.

The difference between the paths for CPU and GPU addresses is in how the memory is pinned and unpinned. For CPU memory this is handled by built-in Linux Kernel functions (`get_user_pages()` and `put_page()`). However, in the GPU memory case the pinning and unpinning has to be handled by functions provided by the NVIDIA Kernel driver. See *Pinning GPU memory* and *Unpinning GPU memory* for details.

Some hardware caveats are explained in *Supported Systems* and *PCI BAR sizes*.

Chapter 4. Change log

4.1. Changes in CUDA 13.0

4.1.1. Deprecation Notice: NV-P2P API Support

The nv-p2p APIs will no longer be supported on NVIDIA Tegra platforms starting with Nvidia Blackwell-based SoCs. Users are encouraged to migrate to alternative APIs for peer-to-peer communication on future Tegra platforms. One such alternative is the Linux upstream kernel API.

A mapping between the `nv-p2p` APIs and the corresponding Linux kernel APIs is provided in the *Porting strategy for device drivers based on nv-p2p*.

Only NVIDIA Orin platforms running L4T (Linux for Tegra) will continue to support the `nv-p2p` APIs. However, this support is also scheduled for deprecation (which indicates that the feature will continue to function in the current release but will no longer receive enhancements or bug fixes) with the CUDA 13.0 release.

4.1.2. Device Attribute Updates

- Added device attribute:
  - `CU_DEVICE_ATTRIBUTE_HOST_ALLOC_DMA_BUF_SUPPORTED`
    - `True`: Page-locked host memory supports `dma_buf` export.
    - `False`: It does not.
    - Check this attribute before using `cuMemAllocHost()` or `cuMemHostAlloc()` with `cuMemGetHandleForAddressRange()` for `CU_MEM_RANGE_HANDLE_TYPE_DMA_BUF_FD`.
- Updated device attributes:
  - `CU_DEVICE_ATTRIBUTE_DMA_BUF_SUPPORTED`
    - Prior to CUDA 13.0:
      - `True`: Indicates that `dma_buf` export is supported.
      - `False`: Indicates that `dma_buf` export is not supported.

- Applications should check this attribute to determine device support for requesting a handle of type `CU_MEM_RANGE_HANDLE_TYPE_DMA_BUF_FD` from `cuMemGetHandleForAddressRange()`.

- As of CUDA 13.0 and later:
  - `True`: Device memory obtained from `cuMemAlloc()` or `cuMemAddressReserve()` supports `dma_buf` export.
  - `False`: Device memory obtained from `cuMemAlloc()` or `cuMemAddressReserve()` does not support `dma_buf` export.
  - Applications should check this attribute to determine device support for requesting a handle of type `CU_MEM_RANGE_HANDLE_TYPE_DMA_BUF_FD` from `cuMemGetHandleForAddressRange()`.

- `CU_DEVICE_ATTRIBUTE_GPU_DIRECT_RDMA_SUPPORTED`
  - `True`: Set only on Orin platforms running L4T or Embedded Linux through CUDA 13.x releases.
  - `False`: Set on all Blackwell-based SoCs and on platforms that do not support GPUDirect RDMA.
  - Applications should check this attribute to determine whether the device supports GPUDirect RDMA APIs, such as `nvidia_p2p_get_pages()`.

4.1.3. 2.1.3. Platform Notes: Orin (L4T)

- `nv-p2p` APIs are deprecated starting with CUDA 13.0 and will be removed in CUDA 14.0. Deprecated APIs remain functional but will no longer receive enhancements or bug fixes.

4.1.4. Known Issues

- Symbol conflicts may occur with `nv-p2p` APIs. - To avoid these issues, unload the NVIDIA display driver before using `nv-p2p`.

4.2. Changes in CUDA 12.2

In drivers released from the R515 up to the R535 branches, except for newer R525 and R535 releases mentioned below, there is a race bug which may show up as a kernel null-pointer dereference. This happens when the GPU invokes the (hereby I/O) kernel driver invalidation callback, the one which was registered during the call to `nvidia_p2p_get_pages`, concurrently with the I/O driver calling `nvidia_p2p_put_pages`.

The race bug does not affect the persistent mapping case, as in that case an invalidation callback is not supported nor needed.

The bug fix required the following API change:

- `nvidia_p2p_get_pages` no longer accepts a NULL callback pointer.

- Instead, `nvidia_p2p_put_pages_persistent` and `nvidia_p2p_get_pages_persistent` have been introduced and should be used instead when requesting a persistent mapping.
- The use of those new persistent APIs can be guarded by the `NVIDIA_P2P_CAP_GET_PAGES_PERSISTENT_API` preprocessor macro, for example when writing portable drivers.
- The `nvidia-peermem` kernel module has been updated accordingly.
- Although deprecated when running GPU drivers from the R470 branch and newer, customers still using the off-tree `nv_peer_mem module` (https://github.com/Mellanox/nv_peer_memory) and needing the persistent mapping feature will have to switch to `nvidia-peermem`.

Note that I/O drivers, which do not need persistent mappings, do not require source code changes.

The API changes described above are deployed in the R535 branch, specifically in release 535.14 and later, and have also been back-ported to the R525 branch, for TeslaRD3 (525.105.17) and later.

4.3. Changes in CUDA 11.4

Added a new a kernel module, `nvidia-peermem`, which provides Mellanox InfiniBand-based HCAs (Host Channel Adapters) direct peer-to-peer read and write access to the NVIDIA GPU's video memory. Refer to *Using nvidia-peermem* for details.

GPUDirect RDMA is supported on Jetson Orin platform. Refer to the *Porting to Tegra* section for details.

Known Issue:

Currently, there is no service to automatically load `nvidia-peermem`. Users need to load the module manually.

4.4. Changes in CUDA 11.2

GPUDirect RDMA is supported on Drive AGX Xavier Linux based platform. Refer to the *Porting to Tegra* section for details.

4.5. Changes in CUDA 10.1

GPUDirect RDMA is supported on Jetson AGX Xavier platform. Refer to the *Porting to Tegra* section for details.

4.6. Changes in CUDA 8.0

In this section we briefly list the changes that are available in CUDA 8.0:

- The nvidia_p2p_page_table struct has been extended to include a new member, without breaking binary compatibility. The minor version in the NVIDIA_P2P_PAGE_TABLE_VERSION macro has been updated accordingly.
- The `nvidia_p2p_dma_mapping` structure, the `nvidia_p2p_dma_map_pages()` and `nvidia_p2p_dma_unmap_pages()` APIs, the `NVIDIA_P2P_DMA_MAPPING_VERSION` macro have been introduced. These APIs can be used by third party device drivers to map and unmap the GPU BAR pages into their device's I/O address space. The main use case is on platforms where the I/O addresses of PCIe resources, used for PCIe peer-to-peer transactions, are different from the physical addresses used by the CPU to access those same resources. See this link for an example of code using these new APIs.
- The `NVIDIA_P2P_PAGE_TABLE_VERSION_COMPATIBLE` and `NVIDIA_P2P_DMA_MAPPING_VERSION_COMPATIBL` macros have been introduced. These are meant to be called by third-party device drivers to check for runtime binary compatibility, for example in case of changes to the data structure's layout.
- On the IBM POWER8 platform, when using the above APIs, GPUDirect RDMA is reported to work correctly restricted to the case where the GPU and the third party device are connected through a supported PCIe switch.

4.7. Changes in CUDA 7.0

In this section we briefly list the changes that are available in CUDA 7.0:

- On the IBM POWER8 platform, GPUDirect RDMA is not supported, though it is not explicitly disabled.
- GPUDirect RDMA is not guaranteed to work on any given ARM64 platform.
- Management of GPU BAR mappings has been improved with respect to CUDA 6.0. Now when a device memory region is pinned, GPU BAR space might be shared with pre-existing mappings. This is the case for example when pinning overlapping or duplicate device memory ranges. As a consequence, when unpinning a region, its whole BAR space will not be returned if even only a subset of its BAR space is shared.
- The new `cuPointerGetAttributes()` API has been introduced. It can be useful when retrieving multiple attributes for the same buffer, e.g. in MPI when examining a new buffer.
- `cudaPointerGetAttributes()` is now faster since it leverages `cuPointerGetAttributes()` internally.
- A new sample code, `samples/7_CUDALibraries/cuHook`, has been added in CUDA 6.5. It can be used as a template for implementing an interception framework for CUDA memory de/allocation APIs.

4.8. Changes in CUDA 6.0

In this section we briefly list the changes that are available in CUDA 6.0:

- CUDA peer-to-peer tokens are no longer mandatory. For memory buffers owned by the calling process (which is typical) tokens can be replaced by zero (0) in the kernel-mode function `nvidia_p2p_get_pages()`. This new feature is meant to make it easier for existing third party software stacks to adopt RDMA for GPUDirect.
- As a consequence of the change above, a new API `cuPointerSetAttribute()` has been introduced. This API must be used to register any buffer for which no peer-to-peer tokens are used. It is necessary to ensure correct synchronization behavior of the CUDA API when operation on memory which may be read by RDMA for GPUDirect. Failing to use it in these cases may cause data corruption. See changes in *Tokens Usage*.
- `cuPointerGetAttribute()` has been extended to return a globally unique numeric identifier, which in turn can be used by lower-level libraries to detect buffer reallocations happening in user-level code (Refer to *Userspace API*). It provides an alternative method to detect reallocations when intercepting CUDA allocation and deallocation APIs is not possible.
- The kernel-mode memory pinning feature has been extended to work in combination with Multi-Process Service (MPS).

Caveats as of CUDA 6.0:

- CUDA Unified Memory is not explicitly supported in combination with GPUDirect RDMA. While the page table returned by `nvidia_p2p_get_pages()` is valid for managed memory buffers and provides a mapping of GPU memory at any given moment in time, the GPU device copy of that memory may be incoherent with the writable copy of the page which is not on the GPU. Using the page table in this circumstance may result in accessing stale data, or data loss, because of a DMA write access to device memory that is subsequently overwritten by the Unified Memory run-time. `cuPointerGetAttribute()` may be used to determine if an address is being managed by the Unified Memory runtime.
- Every time a device memory region is pinned, new GPU BAR space is allocated unconditionally, even when pinning overlapping or duplicate device memory ranges, i.e. there is no attempt at reusing mappings. This behavior has been changed since CUDA 7.0.

Chapter 5. Design Considerations

When designing a system to utilize GPUDirect RDMA, there a number of considerations which should be taken into account.

5.1. Lazy Unpinning Optimization

Pinning GPU device memory in BAR is an expensive operation, taking up to milliseconds. Therefore the application should be designed in a way to minimize that overhead.

The most straightforward implementation using GPUDirect RDMA would pin memory before each transfer and unpin it right after the transfer is complete. Unfortunately, this would perform poorly in general, as pinning and unpinning memory are expensive operations. The rest of the steps required to perform an RDMA transfer, however, can be performed quickly without entering the kernel (the DMA list can be cached and replayed using MMIO registers/command lists).

Hence, lazily unpinning memory is key to a high performance RDMA implementation. What it implies, is keeping the memory pinned even after the transfer has finished. This takes advantage of the fact that it is likely that the same memory region will be used for future DMA transfers thus lazy unpinning saves pin/unpin operations.

An example implementation of lazy unpinning would keep a set of pinned memory regions and only unpin some of them (for example the least recently used one) if the total size of the regions reached some threshold, or if pinning a new region failed because of BAR space exhaustion (refer to *PCI BAR sizes*).

5.2. Registration Cache

Communication middleware often employs an optimization called a registration cache, or pin-down cache, to minimize pinning overhead. Typically it already exists for host memory, implementing lazy unpinning, LRU de-registration, etc. For networking middleware, such caches are usually implemented in user-space, as they are used in combination with hardware capable of user-mode message injection. CUDA UVA memory address layout enables GPU memory pinning to work with these caches by taking into account just a few design considerations. In the CUDA environment, this is even more important as the amount of memory which can be pinned may be significantly more constrained than for host memory.

As the GPU BAR space is typically mapped using 64KB pages, it is more resource efficient to maintain a cache of regions rounded to the 64KB boundary. Even more so, as two memory areas which are in the same 64KB boundary would allocate and return the same BAR mapping.

Registration caches usually rely on the ability to intercept deallocation events happening in the user application, so that they can unpin the memory and free important HW resources, e.g. on the network card. To implement a similar mechanism for GPU memory, an implementation has two options:

- Instrument all CUDA allocation and deallocation APIs.
- Use a tag check function to track deallocation and reallocation. Refer to *Buffer ID Tag Check for A Registration Cache*.

There is a sample application, `7_CUDALibraries/cuHook`, showing how to intercept calls to CUDA APIs at run-time, which can be used to detect GPU memory de/allocations.

While intercepting CUDA APIs is beyond the scope of this document, an approach to performing tag checks is available starting with CUDA 6.0. It involves the usage of the `CU_POINTER_ATTRIBUTE_BUFFER_ID` attribute in `cuPointerGetAttribute()` (or `cuPointerGetAttributes()` if more attributes are needed) to detect memory buffer deallocations or reallocations. The API will return a different ID value in case of reallocation or an error if the buffer address is no longer valid. Refer to *Userspace API* for API usage.

Note: Using tag checks introduces an extra call into the CUDA API on each memory buffer use, so this approach is most appropriate when the additional latency is not a concern.

5.3. Unpin Callback

When a third party device driver pins the GPU pages with `nvidia_p2p_get_pages()` it must also provide a callback function that the NVIDIA driver will call if it needs to revoke access to the mapping. **This callback occurs synchronously**, giving the third party driver the opportunity to clean up and remove any references to the pages in question (i.e., wait for outstanding DMAs to complete). **The user callback function may block for a few milliseconds**, although it is recommended that the callback complete as quickly as possible. Care has to be taken not to introduce deadlocks as waiting within the callback for the GPU to do anything is not safe.

The callback must call `nvidia_p2p_free_page_table()` (not `nvidia_p2p_put_pages()`) to free the memory pointed to by `page_table`. The corresponding mapped memory areas will only be unmapped by the NVIDIA driver after returning from the callback.

Note that the callback will be invoked in two scenarios:

- If the userspace program explicitly deallocates the corresponding GPU memory, e.g. `cuMemFree`, `cuCtxDestroy`, etc. before the third party kernel driver has a chance to unpin the memory with `nvidia_p2p_put_pages()`.
- As a consequence of an early exit of the process.

In the latter case there can be tear-down ordering issues between closing the file descriptor of the third party kernel driver and that of the NVIDIA kernel driver. In the case the file descriptor for the NVIDIA kernel driver is closed first, the `nvidia_p2p_put_pages()` callback will be invoked.

A proper software design is important as the NVIDIA kernel driver will protect itself from reentrancy issues with locks before invoking the callback. The third party kernel driver will almost certainly take similar actions, so dead-locking or live-locking scenarios may arise if careful consideration is not taken.

5.4. Supported Systems

General remarks

Even though the only theoretical requirement for GPUDirect RDMA to work between a third-party device and an NVIDIA GPU is that they share the same root complex, there exist bugs (mostly in chipsets) causing it to perform badly, or not work at all in certain setups.

We can distinguish between three situations, depending on what is on the path between the GPU and the third-party device:

- PCIe switches only
- single CPU/IOH
- CPU/IOH <-> QPI/HT <-> CPU/IOH

The first situation, where there are only PCIe switches on the path, is optimal and yields the best performance. The second one, where a single CPU/IOH is involved, works, but yields worse performance (especially peer-to-peer read bandwidth has been shown to be severely limited on some processor architectures). Finally, the third situation, where the path traverses a QPI/HT link, may be extremely performance-limited or even not work reliably.

Tip: lspci can be used to check the PCI topology:

```
$ lspci -t
```

Platform support

For IBM POWER8 platform, GPUDirect RDMA and P2P are not supported, but are not explicitly disabled. They may not work at run-time.

GPUDirect RDMA is supported on Jetson AGX Xavier platform starting from CUDA 10.1 and on Drive AGX Xavier Linux based platforms from CUDA 11.2. Refer to *Porting to Tegra* for details. On ARM64, the necessary peer-to-peer functionality depends on both the hardware and the software of the particular platform. So while GPUDirect RDMA is not explicitly disabled on non-Jetson and non-Drive platforms, there are no guarantees that it will be fully functional.

IOMMUs

GPUDirect RDMA currently relies upon all physical addresses being the same from the different PCI devices' point of view. This makes it incompatible with IOMMUs performing any form of translation other than 1:1, hence they must be disabled or configured for pass-through translation for GPUDirect RDMA to work.

5.5. PCI BAR sizes

PCI devices can ask the OS/BIOS to map a region of physical address space to them. These regions are commonly called BARs. NVIDIA GPUs currently expose multiple BARs, and some of them can back arbitrary device memory, making GPUDirect RDMA possible. The maximum BAR size available for GPUDirect RDMA differs from GPU to GPU. For example, currently the smallest available BAR size on Kepler class GPUs is 256 MB. Of that, 32MB are currently reserved for internal use. These sizes may change.

On some Tesla-class GPUs a large BAR feature is enabled, e.g. BAR1 size is set to 16GB or larger. Large BARs can pose a problem for the BIOS, especially on older motherbords, related to compatibility support for 32bit operating systems. On those motherboards the bootstrap can stop during the early POST phase, or the GPU may be misconfigured and so unusable. If this appears to be occurring it might be necessary to enable some special BIOS feature to deal with the large BAR issue. Please consult your system vendor for more details regarding large BAR support.

5.6. Tokens Usage

Warning: Starting in CUDA 6.0, tokens should be considered deprecated, though they are still supported.

As can be seen in *Userspace API* and *Kernel API*, one method for pinning and unpinning memory requires two tokens in addition to the GPU virtual address.

These tokens, `p2pToken` and `vaSpaceToken`, are necessary to uniquely identify a GPU VA space. A process identifier alone does not identify a GPU VA space.

The tokens are consistent within a single CUDA context (i.e., all memory obtained through `cudaMalloc()` within the same CUDA context will have the same `p2pToken` and `vaSpaceToken`). However, a given GPU virtual address need not map to the same context/GPU for its entire lifetime. As a concrete example:

```
cudaSetDevice(0)
ptr0 = cudaMalloc();
cuPointerGetAttribute(&return_data, CU_POINTER_ATTRIBUTE_P2P_TOKENS, ptr0);
// Returns [p2pToken = 0xabcd, vaSpaceToken = 0x1]
cudaFree(ptr0);
cudaSetDevice(1);
ptr1 = cudaMalloc();
assert(ptr0 == ptr1);
// The CUDA driver is free (although not guaranteed) to reuse the VA,
// even on a different GPU
cuPointerGetAttribute(&return_data, CU_POINTER_ATTRIBUTE_P2P_TOKENS, ptr0);
// Returns [p2pToken = 0x0123, vaSpaceToken = 0x2]
```

That is, the same address, when passed to `cuPointerGetAttribute`, may return different tokens at different times during the program's execution. Therefore, the third party communication library must call `cuPointerGetAttribute()` for every pointer it operates on.

Security implications

The two tokens act as an authentication mechanism for the NVIDIA kernel driver. If you know the tokens, you can map the address space corresponding to them, and the NVIDIA kernel driver doesn't perform any additional checks. The 64bit `p2pToken` is randomized to prevent it from being guessed by an adversary.

When no tokens are used, the NVIDIA driver limits the *Kernel API* to the process which owns the memory allocation.

5.7. Synchronization and Memory Ordering

GPUDirect RDMA introduces a new independent GPU data flow path exposed to third party devices and it is important to understand how these devices interact with the GPU's relaxed memory model.

- Properly registering a BAR mapping of CUDA memory is required for that mapping to remain consistent with CUDA APIs operations on that memory.
- Only CUDA synchronization and work submission APIs provide memory ordering of GPUDirect RDMA operations.

Registration for CUDA API Consistency

Registration is necessary to ensure the CUDA API memory operations visible to a BAR mapping happen before the API call returns control to the calling CPU thread. This provides a consistent view of memory to a device using GPUDirect RDMA mappings when invoked after a CUDA API in the thread. This is a strictly more conservative mode of operation for the CUDA API and disables optimizations, thus it may negatively impact performance.

This behavior is enabled on a per-allocation granularity either by calling `cuPointerSetAttribute()` with the `CU_POINTER_ATTRIBUTE_SYNC_MEMOPS` attribute, or p2p tokens are retrieved for a buffer when using the legacy path. Refer to *Userspace API* for more details.

An example situation would be Read-after-Write dependency between a `cuMemcpyDtoD()` and subsequent GPUDirect RDMA read operation on the destination of the copy. As an optimization the device-to-device memory copy typically returns asynchronously to the calling thread after queuing the copy to the GPU scheduler. However, in this circumstance that will lead to inconsistent data read via the BAR mapping, so this optimization is disabled an the copy completed before the CUDA API returns.

CUDA APIs for Memory Ordering

Only CPU initiated CUDA APIs provide ordering of GPUDirect memory operations as observed by the GPU. That is, despite a third party device having issued all PCIE transactions, a running GPU kernel or copy operation may observe stale data or data that arrives out-of-order until a subsequent CPU initiated CUDA work submission or synchronization API. To ensure that memory updates are visible to CUDA kernels or copies, an implementation should ensure that all writes to the GPU BAR happen before control is returned to the CPU thread which will invoke the dependent CUDA API.

An example situation for a network communication scenario is when a network RDMA write operation is completed by the third party network device and the data is written to the GPU BAR mapping. Though reading back the written data either through GPU BAR or a CUDA memory copy operation, will return the newly written data, a concurrently running GPU kernel to that network write might observe stale data, the data partially written, or the data written out-of-order.

In short, a GPU kernel is wholly inconsistent with concurrent RDMA for GPUDirect operations and accessing the memory overwritten by the third party device in such a situation would be considered a data race. To resolve this inconsistency and remove the data race the DMA write operation must complete with respect to the CPU thread which will launch the dependent GPU kernel.

5.8. CUDA Features Not Supported on Tegra

All core features of CUDA are supported on Tegra platforms. The exceptions are listed below.

- The `cudaHostRegister()` function is not supported on QNX systems. This is due to the limitations on QNX OS. It is supported in Linux systems with compute capability greater than or equal to 7.2.
- System wide atomics are not supported on Tegra devices with compute capability less than 7.2.
- Unified memory is not supported on dGPU attached to Tegra.
- `cudaMemPrefetchAsync()` function is not supported if `cudaDeviceProp::concurrentManagedAccess` is 0, which is the case on some Tegra platforms.
- NVIDIA management library (NVML) library is only supported on Thor or later Tegra devices running L4T. Where this is not supported, as an alternative to monitor the resource utilization, `tegrastats` can be used.
- Since CUDA 11.5, events-sharing IPC APIs have been supported on L4T and embedded Linux Tegra devices with compute capability 7.x and higher.

  Starting CUDA 13.0, memory-sharing CUDA IPC APIs would be supported on Tegra platforms with open-source GPU driver (i.e. Jetson Thor and beyond) GPU drivers. For all other Tegra platforms, EGLStream, NvSci, or the `cuMemExportToShareableHandle()` / `cuMemImportFromShareableHandle()` APIs can be used to communicate between CUDA contexts in two processes.
- Remote direct memory access (RDMA) is supported only on Tegra devices running L4T or embedded-linux.
- JIT compilation might require a considerable amount of CPU and bandwidth resources, potentially interfering with other workloads in the system. Thus, JIT compilations such as PTX-JIT and NVRTC JIT are not recommended for deterministic embedded applications and can be bypassed completely by compiling for specific GPU targets. For example: If you are compiling for SM version 87, use this nvcc flag `--generate-code arch=compute_87,code=sm_87` to create a CUDA binary for this device. This avoids JIT compilation during the first run and improves the runtime performance. JIT compilation is not supported on Tegra devices in the safe context.
- Peer to peer (P2P) communication calls are not supported on Tegra.
- The cuSOLVER library is not supported on Tegra systems running QNX.
- The nvGRAPH library is not supported.
- CUB is experimental on Tegra products.

More information on some of these features can be found at the following web sites:

IPC:

https://docs.nvidia.com/cuda/cuda-c-programming-guide/index.html#interprocess-communication

NVSCI:

https://docs.nvidia.com/cuda/cuda-c-programming-guide/index.html#nvidia-softwarcommunication-interface-interc

RDMA:

https://docs.nvidia.com/cuda/gpudirect-rdma/index.html

P2P:

https://docs.nvidia.com/cuda/cuda-c-programming-guide/index.html#peer-to-peer-memory-access

Chapter 6. How to Perform Specific Tasks

6.1. Displaying GPU BAR space

Starting in CUDA 6.0 the NVIDIA SMI utility provides the capability to dump BAR1 memory usage. It can be used to understand the application usage of BAR space, the primary resource consumed by GPUDirect RDMA mappings.

```
$ nvidia-smi -q
...
      BAR1 Memory Usage
         Total                       : 256 MiB
         Used                        : 2 MiB
         Free                        : 254 MiB
...
```

GPU memory is pinned in fixed size chunks, so the amount of space reflected here might be unexpected. In addition, a certain amount of BAR space is reserved by the driver for internal use, so not all available memory may be usable via GPUDirect RDMA. Note that the same ability is offered programmatically through the `nvmlDeviceGetBAR1MemoryInfo()` NVML API.

6.2. Pinning GPU memory

1. Correct behavior requires using `cuPointerSetAttribute()` on the memory address to enable proper synchronization behavior in the CUDA driver. Refer to *Synchronization and Memory Ordering*.

```
void pin_buffer(void *address, size_t size)
{
    unsigned int flag = 1;
    CUresult status = cuPointerSetAttribute(&flag, CU_POINTER_ATTRIBUTE_SYNC_
↪MEMOPS, address);
    if (CUDA_SUCCESS == status) {
        // GPU path
        pass_to_kernel_driver(address, size);
    } else {
        // CPU path
```

(continues on next page)

(continued from previous page)

```
        // ...
    }
}
```

This is required so that the GPU memory buffer is treated in a special way by the CUDA driver, so that CUDA memory transfers are guaranteed to always be synchronous with respect to the host. Refer to *Userspace API* for details on `cuPointerSetAttribute()`.

2. In the kernel driver, invoke `nvidia_p2p_get_pages()`.

```
// for boundary alignment requirement
#define GPU_BOUND_SHIFT   16
#define GPU_BOUND_SIZE    ((u64)1 << GPU_BOUND_SHIFT)
#define GPU_BOUND_OFFSET  (GPU_BOUND_SIZE-1)
#define GPU_BOUND_MASK    (~GPU_BOUND_OFFSET)

struct kmd_state {
    nvidia_p2p_page_table_t *page_table;
    // ...
};

void kmd_pin_memory(struct kmd_state *my_state, void *address, size_t size)
{
    // do proper alignment, as required by NVIDIA kernel driver
    u64 virt_start = address & GPU_BOUND_MASK;
    size_t pin_size = (address + size - virt_start + GPU_BOUND_SIZE - 1) & GPU_
↪BOUND_MASK;
    if (!size)
        return -EINVAL;
    int ret = nvidia_p2p_get_pages(0, 0, virt_start, pin_size, &my_state->page_
↪table, free_callback, &my_state);
    if (ret == 0) {
        // Succesfully pinned, page_table can be accessed
    } else {
        // Pinning failed
    }
}
```

Note how the start address is aligned to a 64KB boundary before calling the pinning functions.

If the function succeeds the memory has been pinned and the `page_table` entries can be used to program the device's DMA engine. Refer to *Kernel API* for details on `nvidia_p2p_get_pages()`.

6.3. Unpinning GPU memory

In the kernel driver, invoke `nvidia_p2p_put_pages()`.

```
void unpin_memory(void *address, size_t size, nvidia_p2p_page_table_t *page_table)
{
    nvidia_p2p_put_pages(0, 0, address, size, page_table);
}
```

Refer to *Kernel API* for details on `nvidia_p2p_put_pages()`.

Starting CUDA 6.0 zeros should be used as the token parameters. Note that `nvidia_p2p_put_pages()` must be called from within the same process context as the one from which the corresponding `nvidia_p2p_get_pages()` has been issued.

6.4. Handling the free callback

1. The NVIDIA kernel driver invokes `free_callback(data)` as specified in the `nvidia_p2p_get_pages()` call if it needs to revoke the mapping. Refer to *Kernel API* and *Unpin Callback* for details.
2. The callback waits for pending transfers and then cleans up the page table allocation.

```
void free_callback(void *data)
{
   my_state *state = data;
   wait_for_pending_transfers(state);
   nvidia_p2p_free_pages(state->page_table);
}
```

3. The NVIDIA kernel driver handles the unmapping so `nvidia_p2p_put_pages()` should not be called.

6.5. Buffer ID Tag Check for A Registration Cache

Remember that a solution built around Buffer ID tag checking is not recommended for latency sensitive implementations. Instead, instrumentation of CUDA allocation and deallocation APIs to provide callbacks to the registration cache is recommended, removing tag checking overhead from the critical path.

1. The first time a device memory buffer is encountered and recognized as not yet pinned, the pinned mapping is created and the associated buffer ID is retrieved and stored together in the cache entry. The `cuMemGetAddressRange()` function can be used to obtain the size and starting address for the whole allocation, which can then be used to pin it. As `nvidia_p2p_get_pages()` will need a pointer aligned to 64K, it is useful to directly align the cached address. Also, as the BAR space is currently mapped in chunks of 64KB, it is more resource efficient to round the whole pinning to 64KB.

```
// struct buf represents an entry of the registration cache
struct buf {
    CUdeviceptr pointer;
    size_t      size;
    CUdeviceptr aligned_pointer;
    size_t      aligned_size;
    int         is_pinned;
    uint64_t    id; // buffer id obtained right after pinning
};
```

2. Once created, every time a registration cache entry will be used it must be first checked for validity. One way to do this is to use the Buffer ID provided by CUDA as a tag to check for deallocation or reallocation.

```
int buf_is_gpu_pinning_valid(struct buf* buf) {
    uint64_t buffer_id;
    int retcode;
    assert(buf->is_pinned);
    // get the current buffer id
    retcode = cuPointerGetAttribute(&buffer_id, CU_POINTER_ATTRIBUTE_BUFFER_ID,
↪buf->pointer);
    if (CUDA_ERROR_INVALID_VALUE == retcode) {
        // the device pointer is no longer valid
        // it could have been deallocated
        return ERROR_INVALIDATED;
    } else if (CUDA_SUCCESS != retcode) {
        // handle more serious errors here
        return ERROR_SERIOUS;
    }
    if (buf->id != buffer_id)
        // the original buffer has been deallocated and the cached mapping should
↪be invalidated and the buffer re-pinned
        return ERROR_INVALIDATED;
    return 0;
}
```

When the buffer identifier changes the corresponding memory buffer has been reallocated so the corresponding kernel-space page table will not be valid anymore. In this case the kernel-space `nvidia_p2p_get_pages()` callback would have been invoked. Thus the Buffer IDs provide a tag to keep the pin-down cache consistent with the kernel-space page table without requiring the kernel driver to up-call into the user-space.

If `CUDA_ERROR_INVALID_VALUE` is returned from `cuPointerGetAttribute()`, the program should assume that the memory buffer has been deallocated or is otherwise not a valid GPU memory buffer.

3. In both cases, the corresponding cache entry must be invalidated.

```
// in the registration cache code
if (buf->is_pinned && !buf_is_gpu_pinning_valid(buf)) {
  regcache_invalidate_entry(buf);
  pin_buffer(buf);
}
```

6.6. Linking a Kernel Module against nvidia.ko

1. Run the extraction script:

```
./NVIDIA-Linux-x86_64-<version>.run -x
```

This extracts the NVIDA driver and kernel wrapper.

2. Navigate to the output directory:

```
cd <output directory>/kernel/
```

3. Within this directory, build the NVIDIA module for your kernel:

```
make module
```

After this is done, the `Module.symvers` file under your kernel build directory contains symbol information for `nvidia.ko`.

4. Modify your kernel module build process with the following line:

```
KBUILD_EXTRA_SYMBOLS := <path to kernel build directory>/Module.symvers
```

6.7. Using nvidia-peermem

The NVIDIA GPU driver package provides a kernel module, `nvidia-peermem`, which provides NVIDIA InfiniBand based HCAs (Host Channel Adapters) direct peer-to-peer read and write access to the NVIDIA GPU's video memory. It allows GPUDirect RDMA-based applications to use GPU computing power with the RDMA interconnect without needing to copy data to host memory.

This capability is supported with NVIDIA ConnectX®-3 VPI or newer adapters. It works with both InfiniBand and RoCE (RDMA over Converged Ethernet) technologies.

NVIDIA OFED (Open Fabrics Enterprise Distribution), or MLNX_OFED, introduces an API between the InfiniBand Core and peer memory clients such as NVIDIA GPUs. The `nvidia-peermem` module registers the NVIDIA GPU with the InfiniBand subsystem by using peer-to-peer APIs provided by the NVIDIA GPU driver.

The kernel must have the required support for RDMA peer memory either through additional patches to the kernel or via MLNX_OFED as a prerequisite for loading and using `nvidia-peermem`.

It is possible that the `nv_peer_mem` module from the GitHub project may be installed and loaded on the system. Installation of `nvidia-peermem` will not affect the functionality of the existing `nv_peer_mem` module. But, to load and use `nvidia-peermem`, users must disable the `nv_peer_mem` service. Additionally, it is encouraged to uninstall the `nv_peer_mem` package to avoid any conflict with `nvidia-peermem` since only one module can be loaded at any time.

To stop the `nv_peer_mem` service:

```
# service nv_peer_mem stop</screen>
```

Check if `nv_peer_mem.ko` is still loaded after stopping the service:

```
# lsmod | grep nv_peer_mem
```

If `nv_peer_mem.ko` is still loaded, unload it using:

```
# rmmod nv_peer_mem
```

Uninstall the `nv_peer_mem` package:

For DEB-based OS:

```
# dpkg -P nvidia-peer-memory
```

```
# dpkg -P nvidia-peer-memory-dkms
```

For RPM-based OS:

```
# rpm -e nvidia_peer_memory
```

After ensuring kernel support and installing the GPU driver, `nvidia-peermem` can be loaded with the following command with root privileges in a terminal window:

```
# modprobe nvidia-peermem
```

Note: If the NVIDIA GPU driver is installed before MLNX_OFED, the GPU driver must be uninstalled and installed again to make sure `nvidia-peermem` is compiled with the RDMA APIs that are provided by MLNX_OFED.

Chapter 7. References

7.1. Basics of UVA CUDA Memory Management

Unified virtual addressing (UVA) is a memory address management system enabled by default in CUDA 4.0 and later releases on Fermi and Kepler GPUs running 64-bit processes. The design of UVA memory management provides a basis for the operation of GPUDirect RDMA. On UVA-supported configurations, when the CUDA runtime initializes, the virtual address (VA) range of the application is partitioned into two areas: the CUDA-managed VA range and the OS-managed VA range. All CUDA-managed pointers are within this VA range, and the range will always fall within the first 40 bits of the process's VA space.

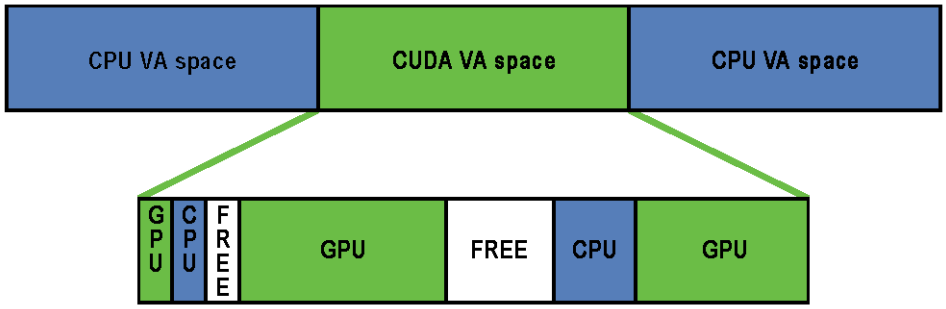

Fig. 1: CUDA VA Space Addressing

Subsequently, within the CUDA VA space, addresses can be subdivided into three types:

GPU
A page backed by GPU memory. This will not be accessible from the host and the VA in question will never have a physical backing on the host. Dereferencing a pointer to a GPU VA from the CPU will trigger a segfault.

CPU
A page backed by CPU memory. This will be accessible from both the host and the GPU at the same VA.

FREE
These VAs are reserved by CUDA for future allocations.

This partitioning allows the CUDA runtime to determine the physical location of a memory object by its pointer value within the reserved CUDA VA space.

Addresses are subdivided into these categories at page granularity; all memory within a page is of the same type. Note that GPU pages may not be the same size as CPU pages. The CPU pages are usually

4KB and the GPU pages on Kepler-class GPUs are 64KB. GPUDirect RDMA operates exclusively on GPU pages (created by `cudaMalloc()`) that are within this CUDA VA space.

7.2. Userspace API

Data structures

```
typedef struct CUDA_POINTER_ATTRIBUTE_P2P_TOKENS_st {
    unsigned long long p2pToken;
    unsigned int vaSpaceToken;
} CUDA_POINTER_ATTRIBUTE_P2P_TOKENS;
```

Function cuPointerSetAttribute()

```
CUresult cuPointerSetAttribute(void *data, CUpointer_attribute attribute, CUdeviceptr
↪pointer);
```

In GPUDirect RDMA scope, the interesting usage is when `CU_POINTER_ATTRIBUTE_SYNC_MEMOPS` is passed as the `attribute`:

```
unsigned int flag = 1;
cuPointerSetAttribute(&flag, CU_POINTER_ATTRIBUTE_SYNC_MEMOPS, pointer);
```

Parameters

data [in]
A pointer to a `unsigned int` variable containing a boolean value.

attribute [in]
In GPUDirect RDMA scope should always be `CU_POINTER_ATTRIBUTE_SYNC_MEMOPS`.

pointer [in]
A pointer.

Returns

CUDA_SUCCESS
if pointer points to GPU memory and the CUDA driver was able to set the new behavior for the whole device memory allocation.

anything else
if pointer points to CPU memory.

It is used to explicitly enable a strictly synchronizing behavior on the whole memory allocation pointed to by `pointer`, and by doing so disabling all data transfer optimizations which might create problems with concurrent RDMA and CUDA memory copy operations. This API has CUDA synchronizing behavior, so it should be considered expensive and possibly invoked only once per buffer.

Function cuPointerGetAttribute()

```
CUresult cuPointerGetAttribute(const void *data, CUpointer_attribute attribute,
↪CUdeviceptr pointer);
```

This function has two different attributes related to GPUDirect RDMA: `CU_POINTER_ATTRIBUTE_P2P_TOKENS` and `CU_POINTER_ATTRIBUTE_BUFFER_ID`.

Warning: CU_POINTER_ATTRIBUTE_P2P_TOKENS has been deprecated in CUDA 6.0

When `CU_POINTER_ATTRIBUTE_P2P_TOKENS` is passed as the `attribute`, `data` is a pointer to `CUDA_POINTER_ATTRIBUTE_P2P_TOKENS`:

```
CUDA_POINTER_ATTRIBUTE_P2P_TOKENS tokens;
cuPointerGetAttribute(&tokens, CU_POINTER_ATTRIBUTE_P2P_TOKENS, pointer);
```

In this case, the function returns two tokens for use with the *Kernel API*.

Parameters

data [out]
Struct `CUDA_POINTER_ATTRIBUTE_P2P_TOKENS` with the two tokens.

attribute [in]
In GPUDirect RDMA scope should always be `CU_POINTER_ATTRIBUTE_P2P_TOKENS`.

pointer [in]
A pointer.

Returns

CUDA_SUCCESS
if pointer points to GPU memory.

anything else
if pointer points to CPU memory.

This function may be called at any time, including before CUDA initialization, and it has CUDA synchronizing behavior, as in `CU_POINTER_ATTRIBUTE_SYNC_MEMOPS`, so it should be considered expensive and should be invoked only once per buffer.

Note that values set in `tokens` can be different for the same `pointer` value during a lifetime of a user-space program. Refer to *Tokens Usage* for a concrete example.

Note that for security reasons the value set in `p2pToken` will be randomized, to prevent it from being guessed by an adversary.

In CUDA 6.0, a new attribute has been introduced that is useful to detect memory reallocations.

When `CU_POINTER_ATTRIBUTE_BUFFER_ID` is passed as the `attribute`, `data` is expected to point to a 64bit unsigned integer variable, like `uint64_t`.

```
uint64_t buf_id;
cuPointerGetAttribute(&buf_id, CU_POINTER_ATTRIBUTE_BUFFER_ID, pointer);
```

Parameters

data [out]
A pointer to a 64 bits variable where the buffer id will be stored.

attribute [in]
The `CU_POINTER_ATTRIBUTE_BUFFER_ID` enumerator.

pointer [in]
A pointer to GPU memory.

Returns

CUDA_SUCCESS
if pointer points to GPU memory.

anything else
if pointer points to CPU memory.

Some general remarks follow:

- `cuPointerGetAttribute()` and `cuPointerSetAttribute()` are CUDA driver API functions only.
- In particular, `cuPointerGetAttribute()` is not equivalent to `cudaPointerGetAttributes()`, as the required functionality is only present in the former function. This in no way limits the scope where GPUDirect RDMA may be used as `cuPointerGetAttribute()` is compatible with the CUDA Runtime API.
- No runtime API equivalent to `cuPointerGetAttribute()` is provided. This is so as the additional overhead associated with the CUDA runtime API to driver API call sequence would introduce unneeded overhead and `cuPointerGetAttribute()` can be on the critical path, e.g. of communication libraries.
- Whenever possible, we suggest to combine multiple calls to `cuPointerGetAttribute` by using `cuPointerGetAttributes`.

Function ``cuPointerGetAttributes()``

```
CUresult cuPointerGetAttributes(unsigned int numAttributes, CUpointer_attribute
↪*attributes, void **data, CUdeviceptr ptr);
```

This function can be used to inspect multiple attributes at once. The one most probably related to GPUDirect RDMA are `CU_POINTER_ATTRIBUTE_BUFFER_ID`, `CU_POINTER_ATTRIBUTE_MEMORY_TYPE` and `CU_POINTER_ATTRIBUTE_IS_MANAGED`.

7.3. Kernel API

The following declarations can be found in the `nv-p2p.h` header that is distributed in the NVIDIA Driver package. Please refer to the inline documentation contained in that header file for a detailed description of the parameters and the return values of the functions described below.

Preprocessor macros

`NVIDIA_P2P_PAGE_TABLE_VERSION_COMPATIBLE()` and `NVIDIA_P2P_DMA_MAPPING_VERSION_COMPATIBLE()` preprocessor macros are meant to be called by third-party device drivers to check for runtime binary compatibility.

Structure nvidia_p2p_page

```
typedef
struct nvidia_p2p_page {
    uint64_t physical_address;
    union nvidia_p2p_request_registers {
        struct {
            uint32_t wreqmb_h;
            uint32_t rreqmb_h;
            uint32_t rreqmb_0;
            uint32_t reserved[3];
        } fermi;
    } registers;
} nvidia_p2p_page_t;
```

In the nvidia_p2p_page structure only the physical_address field is relevant to GPUDirect RDMA.

Structure nvidia_p2p_page_table

```
typedef
struct nvidia_p2p_page_table {
    uint32_t version;
    uint32_t page_size;
    struct nvidia_p2p_page **pages;
    uint32_t entries;
    uint8_t *gpu_uuid;
} nvidia_p2p_page_table_t;
```

The version field of the page table should be checked by using NVIDIA_P2P_PAGE_TABLE_VERSION_COMPATIBLE() before accessing the other fields.

The page_size field is encoded according to the nvidia_p2p_page_size_type enum.

Structure nvidia_p2p_dma_mapping

```
typedef
struct nvidia_p2p_dma_mapping {
    uint32_t  version;
    enum nvidia_p2p_page_size_type page_size_type;
    uint32_t  entries;
    uint64_t *dma_addresses;
} nvidia_p2p_dma_mapping_t;
```

The version field of the dma mapping should be passed to NVIDIA_P2P_DMA_MAPPING_VERSION_COMPATIBLE() before accessing the other fields.

Function nvidia_p2p_get_pages()

```
int nvidia_p2p_get_pages(uint64_t p2p_token, uint32_t va_space_token,
                uint64_t virtual_address,
                uint64_t length,
                struct nvidia_p2p_page_table **page_table,
                void (*free_callback)(void *data),
                void *data);
```

This function makes the pages underlying a range of GPU virtual memory accessible to a third-party device.

Warning: This is an expensive operation and should be performed as infrequently as possible - refer to *Lazy Unpinning Optimization*.

Function nvidia_p2p_put_pages()

```
int nvidia_p2p_put_pages(uint64_t p2p_token, uint32_t va_space_token,
        uint64_t virtual_address,
        struct nvidia_p2p_page_table *page_table);
```

This function releases a set of pages previously made accessible to a third-party device. Warning: it is not meant to be called from within the nvidia_p2p_get_pages() callback.

Function nvidia_p2p_free_page_table()

```
int nvidia_p2p_free_page_table(struct nvidia_p2p_page_table *page_table);
```

This function frees a third-party P2P page table and is meant to be invoked during the execution of the `nvidia_p2p_get_pages()` callback.

Function nvidia_p2p_dma_map_pages()

```
int nvidia_p2p_dma_map_pages(struct pci_dev *peer,
        struct nvidia_p2p_page_table *page_table,
        struct nvidia_p2p_dma_mapping **dma_mapping);
```

This function makes the physical pages retrieved using `nvidia_p2p_get_pages()` accessible to a third-party device.

It is required on platforms where the I/O addresses of PCIe resources, used for PCIe peer-to-peer transactions, are different from the physical addresses used by the CPU to access those same resources.

On some platforms, this function relies on a correct implementation of the `dma_map_resource()` Linux kernel function.

Function nvidia_p2p_dma_unmap_pages()

```
int nvidia_p2p_dma_unmap_pages(struct pci_dev *peer,
        struct nvidia_p2p_page_table *page_table,
        struct nvidia_p2p_dma_mapping *dma_mapping);
```

This function unmaps the physical pages previously mapped to the third-party device by `nvidia_p2p_dma_map_pages()`.

It is not meant to be called from within the `nvidia_p2p_get_pages()` invalidation callback.

Function nvidia_p2p_free_dma_mapping()

```
int nvidia_p2p_free_dma_mapping(struct nvidia_p2p_dma_mapping *dma_mapping);
```

This function is meant to be called from within the `nvidia_p2p_get_pages()` invalidation callback.

Note that the deallocation of the I/O mappings may be deferred, for example after returning from the invalidation callback.

7.4. Porting to Tegra

GPUDirect RDMA is supported on Jetson Orin platform from CUDA 11.4. Note that this document will collectively refer Jetson and DRIVE platforms as Tegra. Owing to hardware and software specific divergence of Tegra vis-a-vis Linux-Desktop, already developed applications needs to be slightly modified in order to port them to Tegra. The following sub-sections (5.4.1-5.4.3) briefs over the necessary changes.

7.4.1. Changing the allocator

GPUDirect RDMA on Desktop allows applications to operate exclusively on GPU pages allocated using `cudaMalloc()`. On Tegra, applications will have to change the memory allocator from `cudaMalloc()` to `cudaHostAlloc()`. Applications can either:

1. Treat the returned pointer as if it is a device pointer, provided that the iGPU supports UVA or `cudaDevAttrCanUseHostPointerForRegisteredMem` device attribute is a non-zero value when queried using `cudaDeviceGetAttribute()` for iGPU.
2. Get the device pointer corresponding to the host memory allocated using `cudaHostGetDevicePointer()`. Once the application has the device pointer, all the rules that are applicable to the standard GPUDirect solution also apply to Tegra.

7.4.2. Modification to Kernel API

The declarations under Tegra API column of the following table can be found in the nv-p2p.h header that is distributed in the NVIDIA Driver package. Refer to the inline documentation contained in that header file for a detailed description of the parameters and the return values. The table below represents the Kernel API changes on Tegra vis-a-vis Desktop.

Desktop API	Tegra API
`int nvidia_p2p_get_pages(uint64_t p2p_token, uint32_t va_space_token, uint64_t virtual_address, uint64_t length, struct nvidia_p2p_page_table \*\*page_table, void (\*free_callback)(void \*data), void \*data);`	`int nvidia_p2p_get_pages(u64 virtual_address, u64 length, struct nvidia_p2p_page_table \*\*page_table, void (\*free_callback)(void \*data), void \*data);`
`int nvidia_p2p_put_pages(uint64_t p2p_token, uint32_t va_space_token, uint64_t virtual_address, struct nvidia_p2p_page_table \*page_table);`	`int nvidia_p2p_put_pages(struct nvidia_p2p_page_table \*page_table);`
`int nvidia_p2p_dma_map_pages(struct pci_dev \*peer, struct nvidia_p2p_page_table \*page_table, struct nvidia_p2p_dma_mapping \*\*dma_mapping);`	`int nvidia_p2p_dma_map_pages(struct device \*dev, struct nvidia_p2p_page_table \*page_table, struct nvidia_p2p_dma_mapping \*\*dma_mapping, enum dma_data_direction direction);`
`int nvidia_p2p_dma_unmap_pages(struct pci_dev \*peer, struct nvidia_p2p_page_table \*page_table, struct nvidia_p2p_dma_mapping \*dma_mapping);`	`int nvidia_p2p_dma_unmap_pages(struct nvidia_p2p_dma_mapping \*dma_mapping);`
`int nvidia_p2p_free_page_table(struct nvidia_p2p_page_table \*page_table);`	`int nvidia_p2p_free_page_table(struct nvidia_p2p_page_table \*page_table);`
`int nvidia_p2p_free_dma_mapping(struct nvidia_p2p_dma_mapping \*dma_mapping);`	`int nvidia_p2p_free_dma_mapping(struct nvidia_p2p_dma_mapping \*dma_mapping);`

7.4.3. Other highlights

1. The length of the requested mapping and base address must be a multiple of 4KB, failing which leads to an error.
2. Unlike the Desktop version, callback registered at `nvidia_p2p_get_pages()` will always be triggered when `nvidia_p2p_put_pages()` is invoked. It is the reponsibilty of the kernel driver to free the page_table allocated by calling `nvidia_p2p_free_page_table()`. Note that, similar to the Desktop version, the callback will also triggered in scenarios explained in *Unpin Callback*.
3. Since `cudaHostAlloc()` can be allocated with `cudaHostAllocWriteCombined` flag or default flag, applications are expected to excercise caution when mapping the memory to userspace, for example using standard linux `mmap()`. In this regard:
    a. When GPU memory is allocated as writecombined, the userspace mapping should also be done as writecombined by passing the `vm_page_prot` member of `vm_area_struct` to the standard linux interface: `` `pgprot_writecombine() <https://elixir.bootlin.com/linux/latest/source/arch/arm64/include/asm/pgtable.h#L403>`__.

b. When GPU memory is allocated as default, no modifcations to the `vm_page_prot` member of `vm_area_struct` should be done.

Incompatible combination of map and allocation attributes will lead to undefined behavior.

7.5. Porting strategy for device drivers based on nv-p2p

The porting strategy for I/O device drivers (e.g., video cameras or sensors) that rely on `nv-p2p` APIs, such as `cuMemHostAlloc` memory, differs based on the target platform:

Orin (L4T):

- Support for `nv-p2p` APIs is scheduled for deprecation (which indicates that the feature will continue to function in the current release but will no longer receive enhancements or bug fixes) with the CUDA 13.0 release.
- Applications are encouraged to adopt the Linux upstream kernel APIs, specifically the `pin_user_pages` / `get_user_pages` family.

Thor and Thor+ Platforms:

- Applications targeting Thor and Thor+ should exclusively use Linux upstream kernel APIs: `pin_user_pages` / `get_user_pages`.

API Porting Table

The following table lists the recommended Linux kernel API replacements for `nv-p2p` functions:

nv-p2p API	Linux Kernel API	Notes
`nvidia_p2p_get_pages`	`pin_user_pages` (with `FOLL_LONGTERM`)	Click here for notes on: pin_user_pages()
`nvidia_p2p_put_pages`	`unpin_user_page`	Click here for notes on: unpin_user_page()
`nvidia_p2p_dma_map_pages`	`dma_map_sgtable`	Click here for notes on: dma_map_sgtable()
`nvidia_p2p_dma_unmap_pages`	`dma_unmap_sgtable`	Click here for notes on: dma_unmap_sgtable()
`nvidia_p2p_free_page_table`	`unpin_user_page`	On dGPU, called when RM frees the physical allocation. On iGPU, triggered when user-space mapping is torn down.
`nvidia_p2p_free_dma_mapping`	`dma_unmap_sgtable`	
`struct nvidia_p2p_page_table`	`int nr_pages, struct page **pages`	
`struct nvidia_p2p_dma_mapping`	`struct sg_table`	

Chapter 8. Notices

8.1. 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.

8.2. OpenCL

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

8.3. 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

©2012-2026, NVIDIA Corporation & affiliates. All rights reserved