DEVELOPING A LINUX KERNEL MODULE USING RDMA FOR GPUDIRECT

v5.0 | October 2012

Application Guide

TABLE OF CONTENTS

LIST OF FIGURES

Chapter 1. OVERVIEW

RDMA for GPUDirect is a feature introduced in Kepler-class GPUs and CUDA 5.0 that enables a direct path for communication between the GPU and a peer device using standard features of PCI Express. The devices must share the same upstream root complex. 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 RDMA for GPUDirect connection to NVIDIA GPUs on Linux.

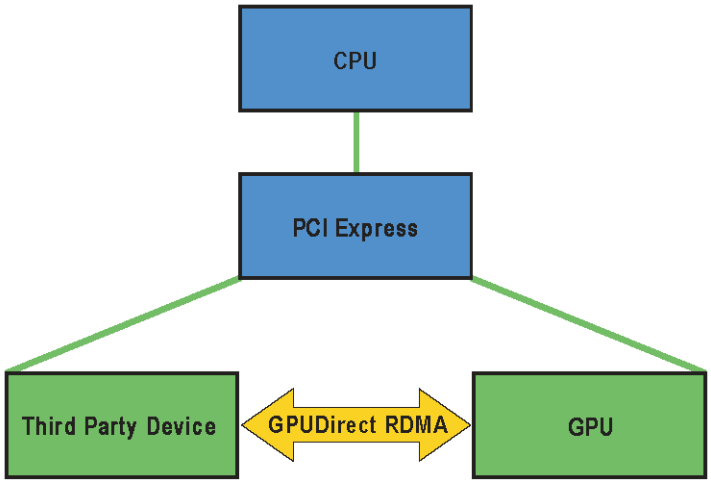

Figure 1 RDMA for GPUDirect within the Linux Device Driver Model

1.1 How RDMA for GPUDirect Works

When setting up RDMA for GPUDirect 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 can have up to six). 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 RDMA for GPUDirect 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 RDMA for GPUDirect are very similar to those used with standard DMA transfers.

See Supported Systems and PCI BAR sizes for more hardware details.

1.2 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.

1.3 RDMA for GPUDirect Transfers

For the communication to support RDMA for GPUDirect 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. See 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 2. DESIGN CONSIDERATIONS

2.1 Lazy Unpinning Optimization

The most straightforward implementation using RDMA for GPUDirect 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 (see PCI BAR sizes).

2.2 Supported Systems

Even though the only theoretical requirement for RDMA for GPUDirect 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. Finally, the third situation, where the path traverses a QPI/HT link, doesn't work reliably.

Tip lspci can be used to check the PCI topology:

```
$ lspci -t
```

IOMMUs

RDMA for GPUDirect currently relies upon all physical addresses being the same from the PCI devices' point of view. This makes it incompatible with IOMMUs and hence they must be disabled for RDMA for GPUDirect to work.

2.3 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 can back their BARs with arbitrary device memory, making RDMA for GPUDirect possible.

The maximum BAR size available for RDMA for GPUDirect differs from GPU to GPU. The main limitation is BIOS bugs that render systems unbootable (or the GPU unusable) if the requested BAR size is too big.

The smallest available BAR size on Kepler class GPUs is 256 MB. Of that, 32MB are reserved for internal use.

2.4 Tokens Usage

As can be seen in Userspace API and Kernel API, 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.

Chapter 3.
HOW TO PERFORM SPECIFIC TASKS

3.1 Pinning GPU memory

1. Invoke `cuPointerGetAttribute()` on the address to check whether it points to GPU memory.

```
void transfer_to_address(void *address, size_t size)
{
    CUDA_POINTER_ATTRIBUTE_P2P_TOKENS tokens;
    CUresult status = cuPointerGetAttribute(&tokens,
 CU_POINTER_ATTRIBUTE_P2P_TOKENS, address);
    if (CUDA_SUCCESS == status) {
        // GPU path
        pass_to_kernel_driver(tokens, address, size);
    }
    else {
        // CPU path
        // ...
    }
}
```

   If the function succeeds, the address points to GPU memory and the result can be passed to the kernel driver. See Userspace API for details on `cuPointerGetAttribute()`.

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

```
void pin_memory(CUDA_POINTER_ATTRIBUTE_P2P_TOKENS *tokens, void *address,
 size_t size)
{
    nvidia_p2p_page_table_t *page_table;

    int ret = nvidia_p2p_get_pages(tokens->p2pToken, tokens->vaSpaceToken,
 address, size,
            &page_table,
            free_callback, NULL);
    if (ret == 0) {
        // Succesfully pinned, page_table can be accessed
    }
    else {
        // Pinning failed
```

```
    }
}
```

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

3.2 Unpinning GPU memory

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

```
void unpin_memory(CUDA_POINTER_ATTRIBUTE_P2P_TOKENS *tokens, void *address,
 size_t size, nvidia_p2p_page_table_t *page_table)
{
    nvidia_p2p_put_pages(tokens->p2pToken, tokens->vaSpaceToken, address,
 size, page_table);
}
```

See Kernel API for details on `nvidia_p2p_put_pages()`.

3.3 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. See 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.

3.4 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
```

Chapter 4. REFERENCES

4.1 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 RDMA for GPUDirect. 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.

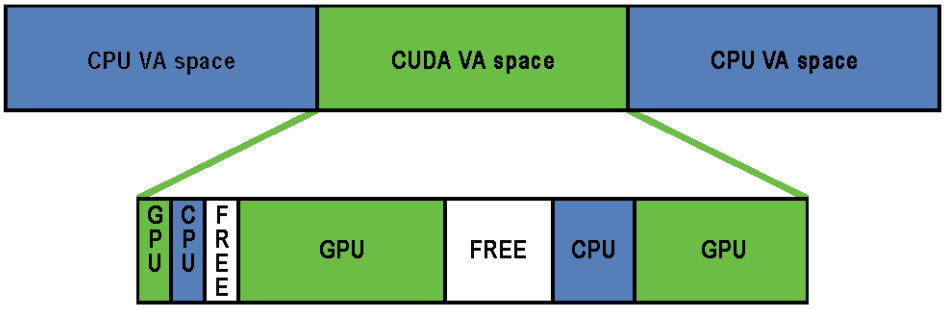

Figure 2 CUDA VA Space Addressing

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. RDMA for GPUDirect operates exclusively on GPU pages (created by `cudaMalloc()`) that are within this CUDA VA space.

4.2 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 `cuPointerGetAttribute()`

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

In RDMA for GPUDirect scope, the specialization of interest is:

```
CUresult cuPointerGetAttribute(CUDA_POINTER_ATTRIBUTE_P2P_TOKENS *tokens,
 CU_POINTER_ATTRIBUTE_P2P_TOKENS, CUdeviceptr pointer)
```

This function returns two tokens for use with the Kernel API.

Parameters

tokens [out]

Struct `CUDA_POINTER_ATTRIBUTE_P2P_TOKENS` with the two 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.

`cuPointerGetAttribute()` is a CUDA driver API function. This is because the required functionality is not incorporated into the CUDA runtime API function `cudaPointerGetAttributes()`. This in no way limits the scope where RDMA for GPUDirect may be used as `cuPointerGetAttribute()` is compatible with CUDA runtime API.

No runtime API equivalent to `cuPointerGetAttribute()` is provided, as the additional overhead associated with the CUDA runtime API to driver API call sequence would introduce unneeded overhead and `cuPointerGetAttribute()` is on the critical path in many communication libraries.

Note that values set in `tokens` can be different for the same `pointer` value during a lifetime of a userspace program. See 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.

4.3 Kernel API

Following declarations can be found in the `nv-p2p.h` header that is distributed in the NVIDIA Driver package.

Data structures

```
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 `nvidia_p2p_page` only the `phyisical_address` is relevant to RDMA for GPUDirect.

```
#define NVIDIA_P2P_PAGE_TABLE_VERSION   0x00010001

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

Fields

`version`
the version of the page table; should be compared to `NVIDIA_P2P_PAGE_TABLE_VERSION` before accessing the other fields

`page_size`
the GPU page size

pages
the page table entries
entries
number of the page table entries

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.

Parameters
p2p_token [in]
A token that uniquely identifies the P2P mapping.
va_space [in]
A GPU virtual address space qualifier.
virtual_address [in]
The start address in the specified virtual address space.
length [in]
The length of the requested P2P mapping.
page_table [out]
A pointer to an array of structures with P2P PTEs.
free_callback [in]
A pointer to the function to be invoked if the pages underlying the virtual address range are freed implicitly.
data [in]
An opaque pointer to private data to be passed to the callback function.

Returns
0
upon successful completion.
-EINVAL
if an invalid argument was supplied.
-ENOTSUPP
if the requested operation is not supported.
-ENOMEM
if the driver failed to allocate memory or if insufficient resources were available to complete the operation.
-EIO
if an unknown error occurred.

This is an expensive operation and should be performed as infrequently as possible - see Lazy Unpinning Optimization.

Function `nvidia_p2p_put_pages()`

```
int nvidia_p2p_put_pages(uint64_t p2p_token, uint32_t va_space,
        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.

Parameters

p2p_token [in]
A token that uniquely identifies the P2P mapping.

va_space [in]
A GPU virtual address space qualifier.

virtual_address [in]
The start address in the specified virtual address space.

page_table [in]
A pointer to an array of structures with P2P PTEs.

Returns

`0`
upon successful completion.

`-EINVAL`
if an invalid argument was supplied.

`-EIO`
if an unknown error occurred.

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.

Parameters

page_table [in]
A pointer to an array of structures with P2P PTEs.

Returns

`0`
upon successful completion.

`-EINVAL`
if an invalid argument was supplied.

4.4 Unpin Callback

When the 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 time to clean up and remove any references to the pages in question (i.e., wait for DMA 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 for the GPU to do anything is not safe within the callback.

Note that the access will be revoked only if the userspace program deallocates the corresponding GPU memory (either explicitly or by exiting early) before the kernel driver has a chance to unpin the memory with `nvidia_p2p_put_pages()`.