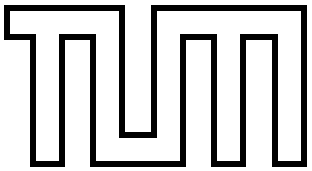


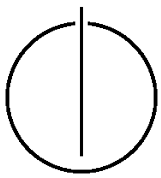
DEPARTMENT OF INFORMATICS

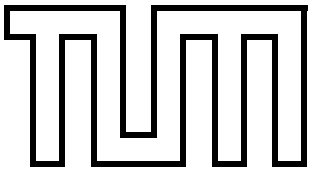
TECHNISCHE UNIVERSITÄT MÜNCHEN

Master's Thesis in Informatics

Direct GPU-FPGA Communication

Alexander Gillert





DEPARTMENT OF INFORMATICS

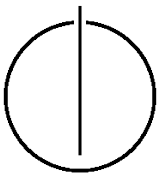
TECHNISCHE UNIVERSITÄT MÜNCHEN

Master's Thesis in Informatics

Direct GPU-FPGA Communication

Direkte GPU-FPGA Kommunikation

Author:	Alexander Gillert
Supervisor:	Prof. Dr.-Ing. habil. Alois Knoll
Advisors:	Dr. Kai Huang Biao Hu, M.Sc.
Date:	April 15, 2015



I confirm that this master's thesis is my own work and I have documented all sources and material used.

München, 12. April 2015 Alexander Gillert

Abstract

Heterogeneous computing systems consisting of CPUs, GPUs and FPGAs currently suffer from a comparatively low bandwidth and high latency for data transfers between the GPU and the FPGA. So far, no standard or vendor-provided method exists for direct communication between these two devices. Indirect communication with a round-trip via the CPU is required.

This thesis describes an example effort to enable this missing link for use with the popular computing platform OpenCL. As expected, a significant increase in bandwidth has been achieved. However, only the direction from the FPGA to the GPU could be realized. More investigation or a different approach is still required to enable the opposite direction as well.

Contents

List of Abbreviations

ATT	Address Translation Table in the Altera PCIe core
BAR	PCI Base Address Register
BSP	Board Support Package, IP stack for Altera OpenCL
CPU	Central Processing Unit
DDR	Double Data Rate, a type of memory
DMA	Direct Memory Access
DPU	Double-precision Floating Point Unit
FIFO	First-In First-Out Queue
FPGA	Field Programmable Gate Array
GPGPU	General Purpose Computing on Graphics Processing Units
GPU	Graphics Processing Unit
HDL	Hardware Description Language
HPC	High Performance Computing
ICD	Installable Client Driver, library that acts as a proxy between different OpenCL implementations
IOMMU	Input/Output Memory Management Unit
IP	Intellectual Property, usually refers to HDL code
IPC	Inter-Process Communication
LUT	Look-Up Table, refers to the basic FPGA building block
MMIO	Memory Mapped Input/Output
MMU	Memory Management Unit
OpenCL	Open Computing Language, a popular HPC platform
OS	Operating System
PCIe	Peripheral Component Interconnect Express Bus
RAM	Random Access Memory
RDMA	Remote Direct Memory Access
SDK	Software Development Kit
SMX	NVIDIA Streaming Multiprocessor

1. Introduction

1.1. Background

Steadily rising processor clock frequencies were the driving force behind computational performance gains throughout the previous century. However, this frequency scaling came to an end around the year 2005 due to its side effect of very high power consumption, known as the power wall. This has forced computer science to develop new techniques to maintain the increase of computational speed. The focus of research has shifted towards parallelism, which enables computations to be performed simultaneously instead of sequentially. Previously employed mainly in supercomputers, parallel computing has become mainstream with the development of parallel processor architectures like multi-core CPUs, GPUs and FPGAs.

GPUs and FPGAs are typically used as accelerators or co-processors in addition to a CPU. Such a *heterogeneous* computing system can combine the advantages of its individual components. The CPU is best suited for sequential and control tasks, whereas data-parallel computations are best to be performed on the GPU or FPGA accelerators. Exploiting the differences among the different accelerator architectures can result in even higher performance gains. FPGAs are hard to beat in bit shifting operations, integer arithmetic and interfacing peripheral devices (such as cameras) but are deficient on floating point operations for which GPUs can accommodate [19].

Typical applications for heterogeneous computing systems include computer vision, physical simulations or scientific visualization. Specifically a CPU-GPU-FPGA system has been used at the TU Munich for lane detection in the context of automated driving [25].

1.2. Problem Statement

High bandwidth and low latency data transfer between individual components are vital for the performance of a heterogeneous computing system. Methods for the communication between the CPU and the accelerator devices are usually provided by the corresponding vendors. Communication between accelerators from different vendors however, has to take the cumbersome and slow approach of a round trip via the CPU.

The goal of this thesis is to enable the still missing direct link between the GPU and the FPGA. A holistic framework for direct GPU-FPGA transfers based on the OpenCL computing platform should be developed. Specifically, a NVIDIA graphics card and an Altera

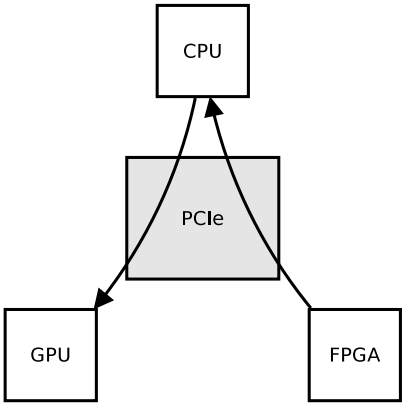


Figure 1.1.: *Indirect* GPU-FPGA transfer

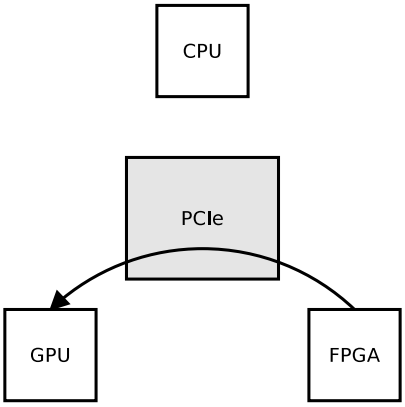


Figure 1.2.: *Direct* GPU-FPGA transfer

FPGA board shall communicate directly over the PCIe bus with minimal coordination by the CPU. A significant bandwidth improvement compared to the indirect method is to be expected.

This thesis is structured as follows:

- Chapter 2 provides a brief overview over the technologies that are required to fully understand the later parts of this thesis.
- In chapter 3 previous developments in this direction are presented.
- Chapters 4 and 5 document the efforts during the actual development of the framework.
- Chapter 6 describes an alternative to the direct GPU-FPGA communication that may be suitable in many cases.
- The evaluation of the developed methods and a comparison with the previous approaches are presented in chapter 7.
- Lastly, chapter 8 discusses the results and suggests further developments that could not be finished during this thesis.

2. Technology Overview

This chapter provides a brief overview over the technologies that will be used in this thesis.

2.1. Accelerator Architectures

Since the release of NVIDIA CUDA general purpose computing on graphics processors has become mainstream. In recent years, FPGA vendors Altera and Xilinx are pushing into the HPC field as well. This section describes the architectures of these two accelerators.

2.1.1. Graphics Processors

Graphics processing units (GPUs) were originally developed for graphics and visualization applications. As such, they are designed to handle mounds of data like vertices or pixels in a short amount of time. These capabilities are also useful for other types of data. For this reason, GPUs are today used also for high performance general purpose computing, known as *GPGPU*.

GPUs differ from traditional general purpose processors (CPUs) primarily through a much higher degree of parallelism. CPUs are built to be easy to use and versatile, being able to handle many different tasks. Graphics processors in contrast, are meant to deal with large amounts of data as fast as possible. Around 1998 the number of transistors on GPUs overtook that of CPUs because most of the die area for CPUs is dedicated to cache and control logic, whereas the die area of GPUs is mostly dedicated to computational logic, thus providing more computing power [36]. Additionally this makes them more energy efficient, offering more performance per Watt. As an example, a modern GPU like the GK110 class from NVIDIA (released 2012) features 7.1 billion transistors [13].

The disadvantages of GPUs include a more demanding programming model because of comparatively little die area dedicated to control logic. Optimally, the task should be *embarrassingly parallel* and require little synchronization. NVIDIA graphics cards can be programmed with the CUDA [15] or OpenCL [22] platforms.

For high performance computing, graphics processors are typically used as accelerators in conjunction with a host CPU system where the main application is running. This adds the drawback of communication delay between the host and the GPU which very often constitutes a bottleneck for an application.

In the following paragraphs, a brief overview of the GPU architecture using the example of the NVIDIA Kepler (GK110) class is provided. Other GPU architectures, also from other vendors, are organized in a similar fashion.

The GK110 employs a deep hierarchy of components. Up to 15 *Streaming Multiprocessors* (SMX) constitute the centerpiece of the GK110. Each of the SMX units consists of 192 *CUDA cores*. Each CUDA core in turn is composed of a floating point unit and an integer arithmetic logic unit. The cores are pipe-lined and can execute one operation per clock cycle. In addition to CUDA cores, each SMX also contains 64 double-precision floating point units (DPU), 32 load/store units for the computation of source and destination memory addresses and 32 special function units (SFUs) for fast approximations of transcendental operations such as sine, square root or interpolation. A SMX organizes the instruction execution in groups of 32 threads (*warps*). 4 *warp schedulers* per SMX can issue 2 instructions to a warp in each clock cycle, theoretically utilizing all cores to full capacity. [13]

A 256KB large register file per SMX provides 256 32-bit registers for each CUDA core and DPU. Each SMX employs additional 16 texture filtering units, 48KB read-only data cache and 64KB memory that can be split up in either L1 cache or shared memory for communication between threads. The 1.5MB L2 cache can be accessed by all SMX units. Finally 6 memory controllers for GDDR6 DRAM complement the memory hierarchy. [13]

For the communication with the host, typically the PCIe bus is used. Chapter 2.4 provides a brief overview over this protocol.

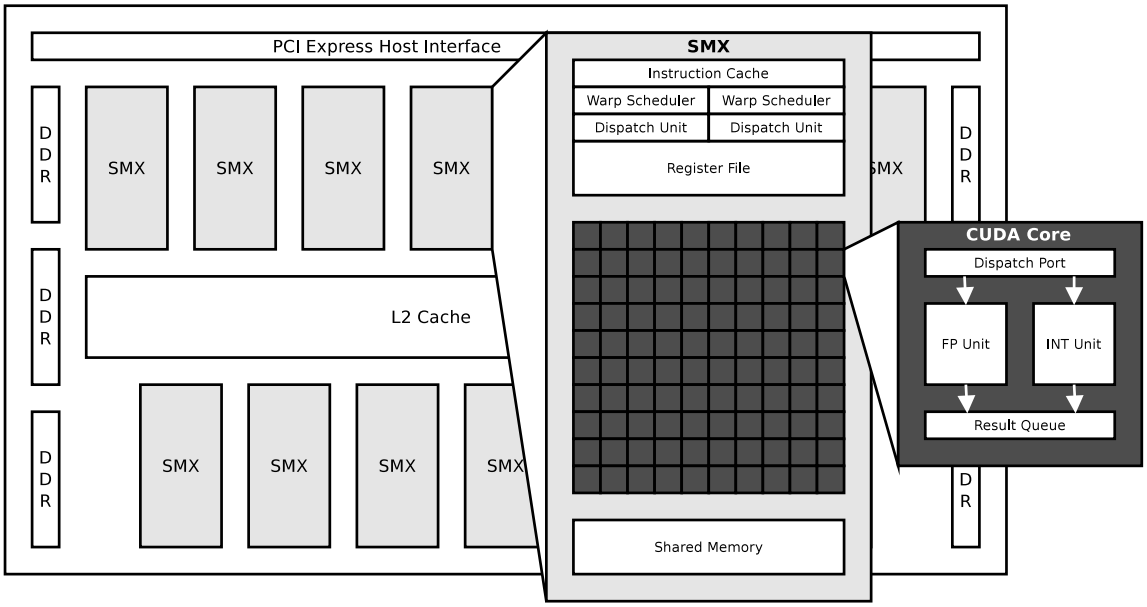
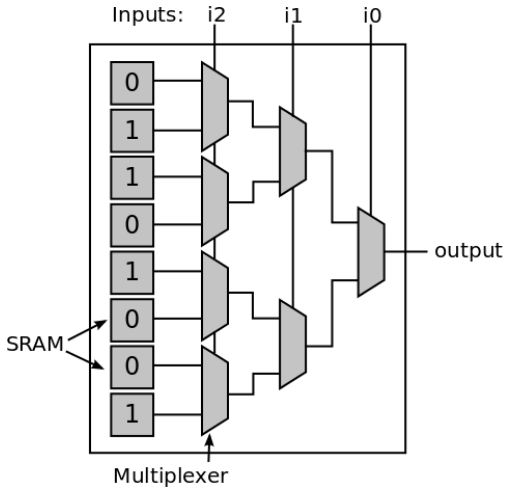


Figure 2.1.: Simplified schematic diagram of the NVIDIA GK110 class GPU architecture. (Image based on [13])

2.1.2. Field Programmable Gate Arrays

Field Programmable Gate Arrays (FPGAs) are fundamentally different from CPUs and GPUs. In contrast to processors, FPGAs are not programmed by specifying a list of sequential instructions to execute, but by constructing digital electronic circuits. These circuits can execute in parallel and do not have the overhead of instruction fetching and decoding. This can result in hundreds of times faster performance in addition to lower power consumption compared to software-based designs [17].

Very generally speaking FPGAs consist of a large number of *look-up tables* (LUTs) that can be connected together to implement any digital circuit. A LUT is a block of memory that stores a small number of bits. It can be implemented with SRAM cells and multiplexers that select which cell to route to the output. An arbitrary Boolean function can be implemented by storing the truth table of the function in the SRAM cells [17]. Figure 2.2 shows an example LUT that implements the 3-input XOR function or a 1-bit full adder.



i0	i1	i2	output
0	0	0	0
0	0	1	1
0	1	0	1
0	1	1	0
1	0	0	1
1	0	1	0
1	1	0	0
1	1	1	1

Figure 2.2.: Left: A look-up table loaded with the values of the 3-input XOR function which can also be seen as a 1-bit full adder without the carry output (Image based on [16]). Center: A truth table describing the same function. Right: An equivalent digital circuit consisting of logic gates.

More complex functions can be achieved by combining multiple look-up tables. For example the addition of two 8-bit unsigned integers can be implemented using 8 1-bit full adders from figure 2.2 and 8 LUTs storing the carry logic. They are then connected as shown in figure 2.3. Actually, this design can be optimized to use less LUTs.

Look-up tables can only act as logic functions, not able to store a state. Therefore LUTs are connected to a *D-type flip-flop* which acts as an output buffer. Together they form a *logic block* or *cell*. A multiplexer selects whether the logic block's output comes from the LUT directly or from the flip-flop. The multiplexer in turn is configured by an additional SRAM cell [16]. In reality logic blocks often include additional components such as carry logic or full adders because they are often used [1].

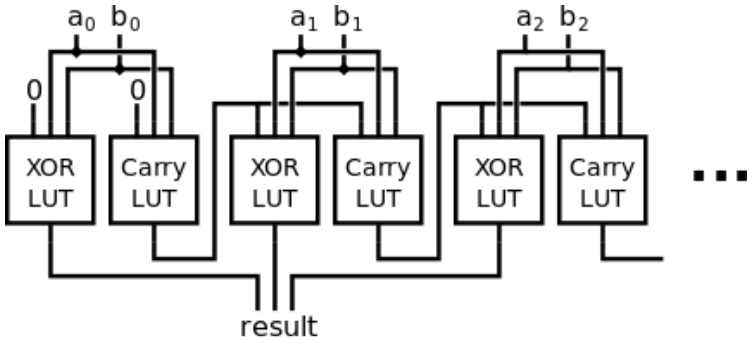


Figure 2.3.: A ripple-carry adder implemented using only 3-input LUTs

A logic block on its own can only perform very simple logic functions. For more complicated computations the cells have to be connected together. Usually, these interconnections constitute up to 90% of the whole FPGA's area [16]. This is required for a high degree of flexibility to be able to implement any digital circuit. The most common architecture is the *island style* interconnect [17] which is outlined in figure 2.4. Instead of connecting the logic blocks directly with each other, they are separated by horizontal and vertical multi-lane signal channels. On intersections configurable *switch boxes* control which direction a signal takes. The logic blocks are connected via *connection blocks* to the channels. Again, connection blocks can also be configured to allow to connect any lane to the cell's input or output. To also improve the signal delay from one logic block to another, additional long-distance lanes in the channels can be used. Most commercial FPGAs (e.g. Altera Stratix and Xilinx Virtex families [1]) employ this concept as the basis for their routing architectures.

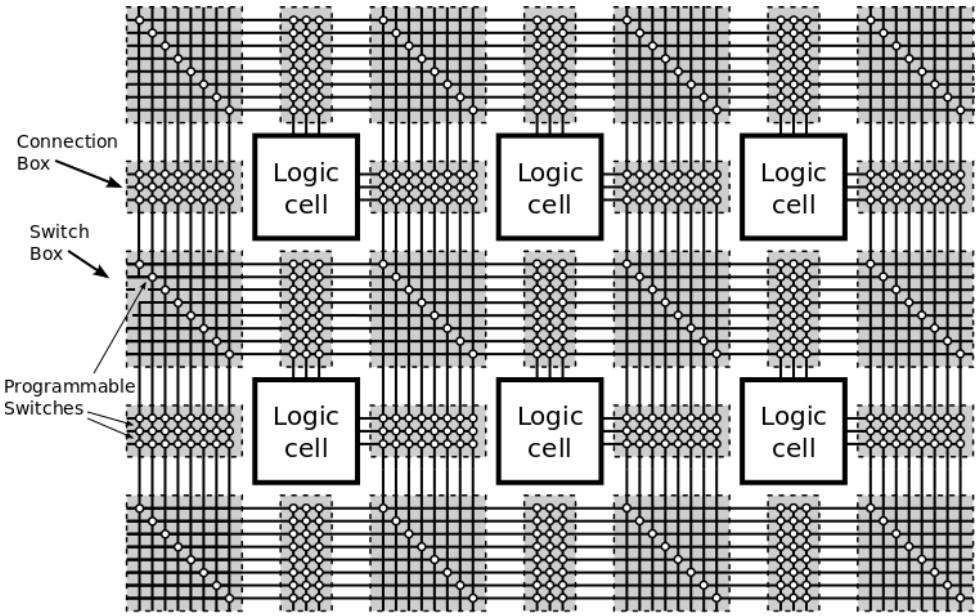


Figure 2.4.: Example of an island style connectivity (Components are not to scale. Image based on [17]

In theory a FPGA can consist only of configurable logic blocks, interconnections and I/O pins. However, to achieve higher performance vendors usually employ dedicated non-configurable (*hardened*) hardware for often used and expensive operations. For instance, multipliers, DSP blocks and dedicated SRAM cells, distributed across the chip are very common in modern FPGAs. Moreover some models may include complete hard CPUs for computations that are inherently sequential or to run a whole operating system. Altera's Cyclone V SoC series for example includes an ARM Cortex-A9 core [1].

In contrast to compiling a software program, FPGA configuration is not a straight-forward process, as outlined in figure 2.5. For one, yielding decent results almost always requires optimization towards one of two often contradicting goals: minimal resource usage vs. performance. Furthermore, after each step several constraints have to be met, most important one being that the design does not require more resources than are available on the target chip. If one of those constraints cannot be maintained the procedure needs to be restarted with a different optimization strategy. This contributes to rather long build process.

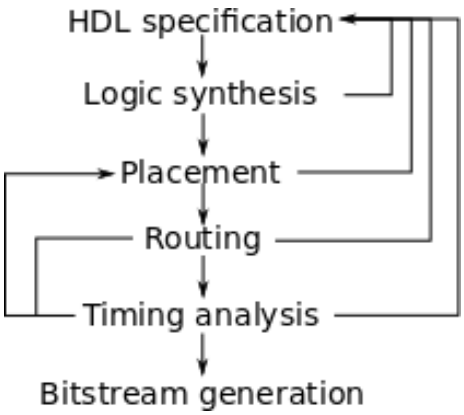


Figure 2.5.: The major phases involved in the configuration procedure. Falling back to a previous step may be required if a constraint cannot be met. (Image based on [17])

The most commonly applied method to configure FPGAs is to use a *hardware description language* (HDL) such as *VHDL* or *Verilog*. HDLs differ fundamentally from imperative programming languages like C or Fortran because of their concurrent semantics [17]. A logic synthesizer converts the HDL code into a set of primitives, called a *netlist* which basically consists of logic gates, flipflops and the interconnections between them. These netlist primitives are then mapped to the target device's primitives (i.e. LUTs of certain sizes, various DSP units). Next, every element of the new technology-mapped netlist is assigned to a real physical building block of the corresponding type on the chip and a the connections between them are established. In the end, if the timing simulation satisfies the target clock rate, a bitstream containing the contents of the memory cells of the LUTs, multiplexers and switches is produced.

HDLs provide a great flexibility and the ability to control the flow of every digital signal in every clock cycle. However, this demands a high level of expertise and long develop-

ment time. In 2013, Altera released an SDK for the configuration of their FPGAs with the OpenCL standard [4]. This enables to program FPGAs in the same way as GPUs. This standard is described in more detail in section 2.2.

It is difficult to compare the performance of FPGAs to GPUs because of the completely different architectures. Very generally speaking, FPGAs are better suited for integer operations, GPUs on the other hand achieve better results with floating point calculations. Moreover, FPGAs are very flexible and can interface a variety of other devices [19].

2.2. OpenCL

OpenCL (Open Computing Language)[22] is a standard for heterogeneous high performance computing managed by the Khronos consortium. It has been implemented on a wide range of devices, most importantly multi-core CPUs, GPUs and FPGAs. It defines a high level abstraction layer for low level hardware instructions. This enables to scale computations from general purpose processors to massively parallel devices without changing the source code.

The OpenCL specification resembles in many aspects the NVIDIA CUDA platform and can be roughly summarized as follows[22]:

An OpenCL application runs on a *host* system which is connected to one or more accelerator *devices*. A device, divided into *compute units* and *processing elements*, is usually able to execute compute *kernels* in a SIMD or sometimes SPMD fashion. The kernels are mostly written in the *OpenCL C* programming language, a dialect of the C99 standard and compiled with a vendor-specific compiler, but native kernels are optionally supported as well. They describe the sequence of instructions within a single execution instance, called a *work-item*. Work-items that are grouped in the same *work-group* are executed concurrently.

The memory hierarchy consists of four distinct regions for the kernels:

- *Global* memory that can be written and read by all work-items in all work-groups
- *Constant* memory, a region of global memory that is initialized by the host and does not change during the execution
- *Local* memory that is only accessible to work-items within the same work-group
- *Private* memory owned by a single work-item and not visible by others

The functions `clEnqueueWriteBuffer` and `clEnqueueReadBuffer` provide a convenient way to transfer data between host and device memory. Methods for data transfers between multiple devices are not specified by the standard. This is only possible with vendor-specific extensions. Specifically for GPU-FPGA transfers, no extensions are available. The only portable workaround is to read a memory region from the first device into CPU memory and then write it to the second device. Obviously, this approach is rather slow, due to the overhead of the two transfers.

The `cl_khr_icd` extension[20] allows multiple OpenCL implementations from different vendors to co-exist on the same system. It defines an *installable client driver (ICD) loader*, a unifying library that acts as a mediator between the different platforms. This enables the use of multiple heterogeneous devices in a single process without the different implementations interfering with each other. Without this mechanism, the overhead of multiple processes and inter-process communication (IPC) inbetween them is required. As of the time of this writing, Altera's latest SDK version 14.1 does not support this extension yet. Section 4 describes an implementation of an incomplete yet usable ICD for Altera OpenCL.

2.3. Linux Device Drivers

Hardly any two hardware devices provide the same control interface to the host system. As a consequence either an application program has to know how to interface every single device (which is impossible for those that are not yet available) or an abstraction layer between the application software and the actual device is needed. The role of a *device driver* is to provide this abstraction layer [29]. A driver is built for a specific piece of hardware and knows its internal logic. The user program may use a set of standardized calls that are independent of the hardware and the device driver will map these calls to the hardware specific commands.

A device driver communicates with the peripheral device through its I/O registers. Using hardware registers is very similar to main memory: Every register has an address which can be accessed with the `read` and `write` system calls. The CPU is then asserting electrical signals on the address bus and control bus and reading from or writing to the data bus [29]. The most important difference is that I/O regions can and do have side effects whereas memory usually does not.

Direct I/O access to a hardware device may cause physical damage if operated incorrectly. Therefore in Linux, this is only allowed for the privileged code running in kernel space. Though the Linux kernel is largely monolithic, it allows *modules* to be built separately from the rest of the kernel and inserted or removed at runtime when needed, without having to reboot the whole system. Device drivers are usually constructed as kernel modules [29].

A kernel module does not have a `main` function and is completely event-driven [29]. It has to provide an initialization function which may initialize the hardware and register callbacks for hardware interrupts or user space communication calls. A minimal kernel module written in C looks like this:

Listing 2.1: Minimal kernel module

```
#include <linux/init.h>
#include <linux/module.h>

static int hello_init(void)
{ printk(KERN_ALERT "Hello, world\n"); return 0; }
```

```
static void hello_exit(void)
{ printk(KERN_ALERT "Goodbye\n"); }

module_init(hello_init);
module_exit(hello_exit);
```

To insert a module into the kernel at runtime, the following command can be used:

```
insmod path/to/example_module.ko
```

The following command will remove the module from the kernel again:

```
rmmod example_module
```

In Linux, the main method for a user space program to communicate with a kernel module is through file operations on a file exposed by the module. Such a *device file* is usually located in the `/dev/` directory of the file system. The module can register callback functions that are executed when system calls like `open`, `read` or `ioctl` are performed on this file.

Both, Altera and NVIDIA provide Linux drivers for their chips. Altera released its PCIe driver under an open source license, available for modification, whereas the NVIDIA driver is split into two parts: the kernel module which is also open source and the actual closed source driver. The kernel module acts only as an mediator between the user programs and the actual driver.

2.4. PCIe

Peripheral Component Interconnect Express (PCIe) is a very commonly used bus for the connection of peripheral devices to a host system. It is an extension of the PCI bus and is completely compatible with its predecessor up to the point that PCI-only devices can communicate over PCIe without modifications [11].

A PCIe interconnect that connects two devices together is referred to as a *link*. A link consists of either x1, x2, x4, x8, x12, x16 or x32 signal pairs in each direction, called *lanes*. Data can be transmitted in both directions simultaneously on a transmit and receive lane. The signal pairs are *differential*, that means that 2 electrical signals with opposing voltages (`D+` and `D−`) are used for each logical signal. A positive voltage difference between `D+` and `D−` implies a logical `1` and logical `0` for a negative difference. Differential signals have the advantage that they are more robust towards electrical noise. Each PCIe lane thus totals to overall 4 electrical signals. [11]

The PCI Express 2.0 specification supports 5.0 Gbits/second/lane/direction transfer rate. For an additional degree of robustness during data transfer, each byte of data transmitted is converted into a 10-bit code (*8b/10b encoding*) i.e. for every byte of data, 10-bits are actually transmitted, resulting in 25% overhead. The 8b/10b encoding moreover guarantees one-zero transitions in every symbol, which eliminates the need of a clock signal. The receiving device can recover the sender's clock through a *PLL* from the rising and falling

edges of the signal [11]. The following table shows the maximum possible transfer speeds with PCIe:

PCIe link width	Theoretical bandwidth in one direction
x1	500 MBytes/second
x8	4 GBytes/second
x32	16 GBytes/second

The PCI Express hierarchy centers around a *root complex*. It connects the CPU and the memory subsystem to the PCIe fabric consisting of endpoints and *switches* which may extend the hierarchy. Its functionality includes the generation of transactions on the behalf of the CPU and routing of packets from one of its ports to another. [11] An example topology is illustrated in figure 2.6.

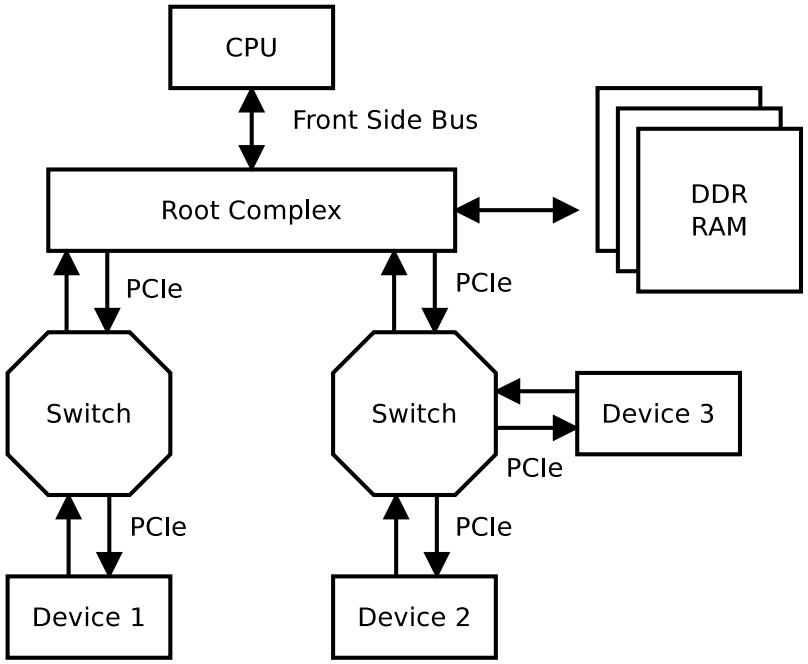


Figure 2.6.: Example topology of a system employing PCIe (Image based on [11])

In PCI Express data is transferred from one device to another via *transactions*. A transaction is a series of one or more *packets* required to complete an information transfer between a sender and a receiver. [11] Transactions can be divided into 4 address spaces:

- *Memory*: Data transfer to or from a location to the system memory or a device.
- *IO*: Data transfer to or from a location to the system IO map. PCIe devices do not initiate IO transactions. They are intended for legacy support.
- *Configuration*: Read from or write into the configuration space of a PCIe device. Only initiated by the host.
- *Message*: General messaging and event reporting

The *configuration space* of a device contains a set of standardized registers used by the host to discover the existence of a device function and to configure it for normal operation. A device may implement multiple functions. Each function's configuration space is 4KB large of which the first 256 Bytes are occupied by the legacy PCI header. This header includes the *Vendor ID* and *Device ID* registers to identify a device, *Status* and *Command* registers used to report and control certain features of the device as well as up to 6 32-bit *Base Address Registers (BARs)*. [11] Two BARs can be merged into one 64-bit BAR. A device may expose a linear window of its memory into one of the BARs. The host or another PCIe device may then use the value programmed into the BAR as a physical address to write from or read into the device's memory in the same way as to system memory, i.e. via Memory Read or Write Transactions. [14]

A PCI Express device may deliver interrupts to its device driver on the host via *Message Signaled Interrupts (MSI)* [11]. A MSI is a Memory Write transaction to a specific address of the system memory which is reserved for interrupt delivery. Moreover legacy interrupts employing dedicated physical pins are supported. Interrupts are useful for a high degree of performance. Instead of the host polling the state of a device, the device can notify the host when an event, such as a completed data transfer, occurs.

2.5. Direct Memory Access

Direct Memory Access (DMA) is a hardware mechanism that allows peripheral components to transfer data directly to and from main memory without involving the system processor [29]. DMA is useful when the CPU is not able to keep up with the transfer rate or has to perform other operations inbetween. This can increase not only the throughput to and from a device but also the overall system performance.

DMA is carried out by a *DMA controller* built into the device. DMA transfers can be classified as *synchronous* or *asynchronous*. In the asynchronous case the device initiates the accesses to the system memory. This is mostly done by acquisition devices or network cards. A synchronous transfer is initiated by the software and contains roughly the following steps [29]:

1. The driver sends a command to the device's DMA controller containing the memory address to read from or to write into. The associated process is put to sleep so that the CPU can process other tasks.

2. The DMA controller performs the transfer and sends an interrupt to the driver when it is completed.

3. The interrupt handler acknowledges the interrupt and wakes up the process.

It is important to note that a DMA controller requires an address from the *bus address space* whereas whereas software operates on the *virtual address space*. The virtual address space is managed by the operating system and the *memory management unit (MMU)*, a hardware component usually built into the processor chip [32]. The virtual address space

is segmented into *pages* of equal size[1]. A *page table* on the MMU maps from pages to the real physical memory. Every process gets a set of pages assigned by the OS. When a process needs to access the main memory, the CPU sends the virtual address to the MMU, which translates it through the page table to the actual *physical address* of the system memory. In case there is no page table entry for the specified virtual address a *page fault* exception is signaled to the OS. This can happen due to the software accessing a memory area it is not allowed to or if the memory is full and the requested page has been moved from the RAM to the swap partition on the hard drive. In the latter case the OS loads it from the swap and puts it back to the RAM, replacing another page.

This mechanism frees programs from the need to manage shared memory, allows processes to use more memory than might be physically available and increases the security by disallowing a process to access memory outside of its address space [32].

Bus addresses are used between a peripheral bus and the system memory. An *I/O memory management unit (IOMMU)* may play the same role as the MMU and translate from the bus address space to the physical address space. However, often the IOMMU is not present in the system or deactivated, in which case the bus addresses are equivalent to physical addresses [29]. This scenario is assumed throughout the rest of this thesis as it represents a majority of today's systems and for the sake of convenience. Figure 2.7 illustrates the relations between the address spaces.

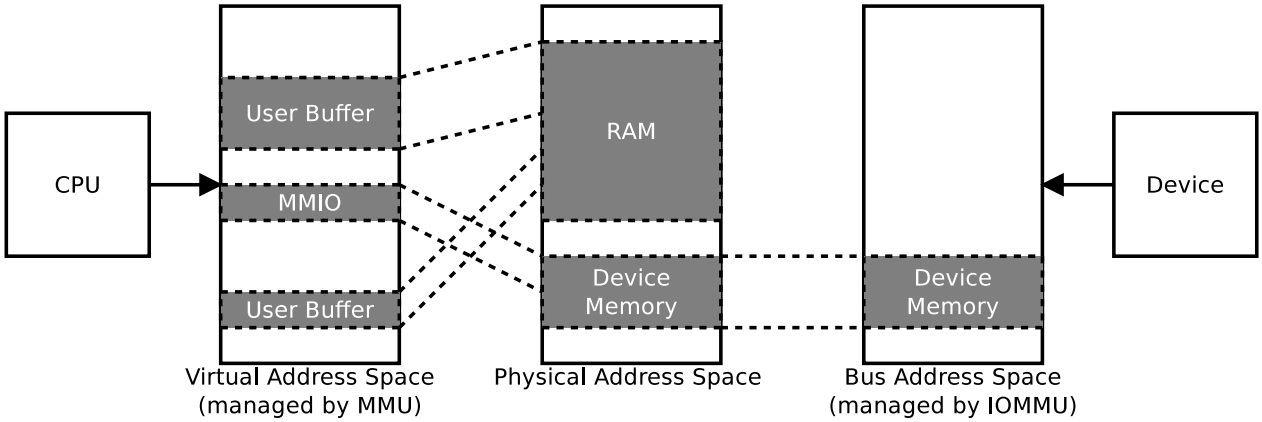


Figure 2.7.: Relationship between the different address spaces with example mappings. In this case the IOMMU performs a 1:1 mapping. (Image based on [32, 29])

Physical addresses are usually not visible neither to the user application nor to the driver. To get the physical address of a buffer that was allocated in user space, an operation called *pinning* is required [29]. Pinning marks pages as not swappable to ensure that they are not moved out to the swap partition. This ensures that the virtual address of the page always maps to the same physical address. The `get_user_pages(...)` function can be used within the driver to perform this operation. It also provides the physical addresses to each page it pins.

[1] in Linux by default 4KB

Unfortunately, especially for large buffers, the acquired physical addresses do not form a contiguous block, but are distributed across the memory. A DMA controller that supports *scatter-gather* operations can accept a scatter-list, an array of addresses and lengths, and transfer them in one DMA operation [29]. For controllers that to not support this feature, either a contiguous buffer has to be allocated (for the write direction) or multiple transfers are required.

DMA is indispensable in the HPC field and most accelerator cards have a built-in DMA controller. For example in Altera OpenCL a modular Scatter-Gather-DMA (SGDMA) IP core [2] is used for transfers of at least 1KB size.

Starting with the Kepler class architecture, all NVIDIA Quadro and Tesla cards additionally employ a technology called *GPUDirect RDMA* [14]. With RDMA, the GPU is able to pin a buffer in its internal memory and expose it into one of the PCIe BARs. This BAR can be used as a bus address for other devices within the bus to directly write into or read from the GPU memory without the overhead of transferring the data to system memory first.

3. Previous Work

Parts of the issues addressed in this thesis have already been tackled in the past. This chapter gives a brief overview of the previous research. A detailed comparison of this thesis and these other approaches is presented in section 7.5.

3.1. Ray Bittner & Erik Ruf

Ray Bittner and Erik Ruf from Microsoft Research implement direct GPU-FPGA communication with a custom IP core, named *Speedy PCIe* [9, 8]. The IP is written for Xilinx FPGAs and designed to exploit the full bandwidth potential of the PCIe bus. A Microsoft Windows driver is also supplied for the core.

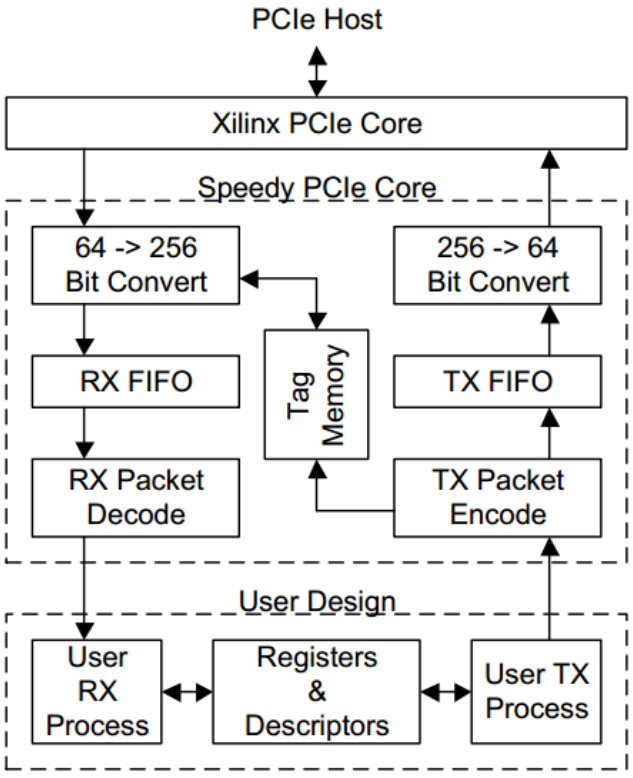


Figure 3.1.: Speedy PCIe IP core design (Image from [8])

Bittner and Ruf discovered that several CUDA operations that are intended for GPU-CPU data transfers can be used for GPU-FPGA transfers as well. Specifically, the function

`cudaHostRegister` pins a contiguous block of memory in the virtual address space of the CPU and maps it for the CUDA device. Afterwards the `cudaMemcpy` function can be used to transfer data to and from this buffer by the DMA controller on the GPU. These functions do not differentiate whether the buffer is mapping to CPU memory or to FPGA memory. Therefore, by mapping FPGA memory with the Speedy PCIe driver to the CPU virtual address space, direct GPU-FPGA communication can be achieved.

In the GPU to FPGA case, the performance improved by 34.5% for large transfers compared to the GPU to CPU to FPGA data path. The other direction on the other hand suffered a penalty of 52.6% compared to the indirect path. According to the authors this was mainly due to the testing procedure Verilog code. [9]

In contrast to this thesis, Bittner and Ruf do not use OpenCL, neither on the GPU side nor on the FPGA. The Speedy PCIe core cannot be used in this thesis as it is not compatible with Altera's OpenCL which uses an own IP.

3.2. Yann Thoma, Alberto Dassatti & Daniel Molla

Yann Thoma, Alberto Dassatti and Daniel Molla from the University of Applied Sciences Western Switzerland developed an open source framework for direct GPU-FPGA PCIe communication, called FPGA^2 [33]. Similarly to the approach used by Bittner and Ruf, they use a custom IP stack, designed for Xilinx FPGAs. The largest difference is, that the transfer is driven by the custom DMA controller on the FPGA and not by the GPU.

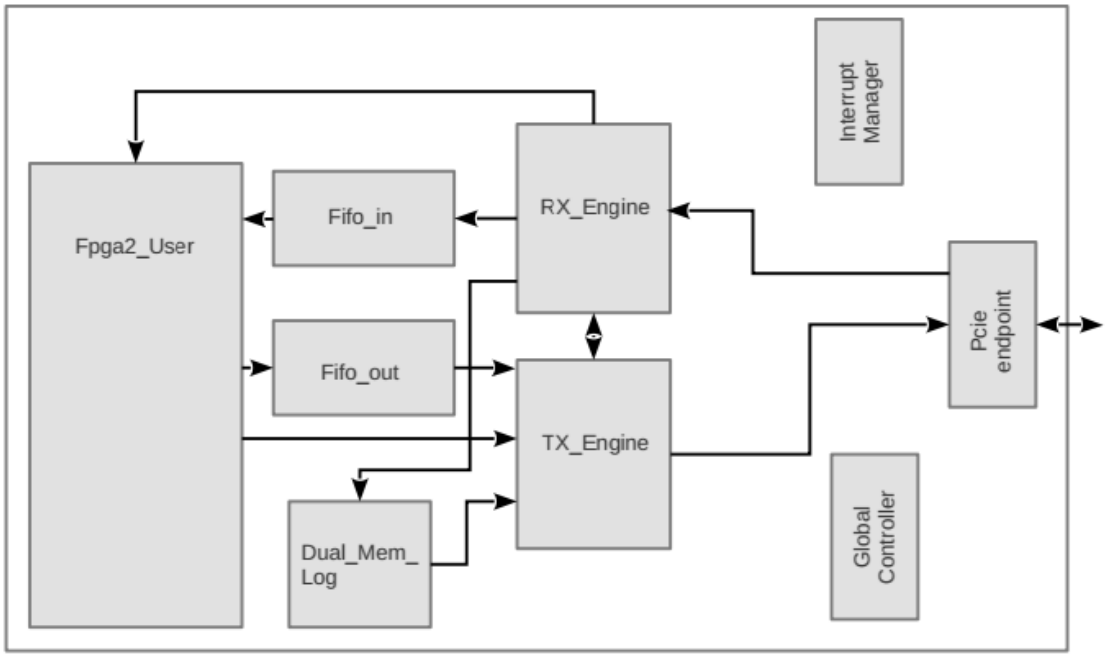


Figure 3.2.: FPGA^2 IP stack design (Image from [33])

Instead of the official NVIDIA software, FPGA^2 uses the open source GPU driver *nouveau* [34] and *gdev* [30], an open source CUDA implementation, to retrieve the physical address of a buffer allocated on the GPU. To do this, the buffer has to be copied within the GPU memory into a window exposed by the PCIe BAR. According to the authors, the overhead of this copy operation is negligible due to very fast copy bandwidth of more than 5GB/s. After that, the physical address is passed to the DMA controller on the FPGA, which initiates the actual direct transfer, with minimal interaction from the CPU.

For the evaluation, the team uses only a single PCIe lane for the direct transfer and compares it with an indirect approach using the official CUDA implementation. In the latter case, 8 PCIe lanes are used and the data is scaled with a constant factor to make it comparable with the single-lane direct transfer data. Moreover, only transfer sizes of up to 8MB have been evaluated. Because of these issues the results should be interpreted carefully. Overall, the direct communication method outperformed the indirect solution for small transfer sizes. For larger transfers, the data fluctuates too much for a clear interpretation.

The limiting factors of this approach are mainly nouveau and gdev. As the authors themself state, the performance of those open source projects is often worse than that of the official vendor-provided software, and lack several features and support for some devices [33]. The development of the OpenCL port for example, has been stalled by the nouveau team [35].

The difference to this thesis is that OpenCL is not used by FPGA^2, the FPGA has to be programmed with a HDL instead. This also means that the IP stack cannot be used as it is not compatible with Altera OpenCL. Nouveau and gdev will not be used, in favor of the official NVIDIA driver, though some modifications will be required.

3.3. David Susanto

In his master's thesis, David Susanto implements and evaluates three methods for GPU-FPGA communication for use in heterogeneous HPC [31]. Initially, he uses OpenCL for both GPU and FPGA, however due to the lack of an ICD in Altera's OpenCL implementation, he is forced to split his application into two processes and use inter-process communication inbetween. His first two methods employ indirect device communication and shared memory or a message queue for IPC.

The third method uses direct device communication. The physical address of a GPU buffer is retrieved via the GPUDirect RDMA technology and passed to the DMA controller on the FPGA. Extension of Altera's PCIe device driver is required. CUDA has to be used in the GPU process because this technology is not directly supported in OpenCL. He reports a performance improvement of ca 38% for direct FPGA to GPU case and ca 34% for the opposite direction.

This thesis continues Susanto's work. The need for two processes and IPC will be removed by implementing an ICD for Altera OpenCL as well as the requirement of CUDA. Unfortunately his device driver code is not available and has to be re-implemented.

4. Implementation of an Installable Client Driver for Altera OpenCL

As of SDK version 13.1 (which is used in this thesis) all Altera OpenCL functions are implemented in the dynamic library `libalteracl.so`. Using this library directly inhibits the use of a second OpenCL implementation from a different vendor in the same application. Trying to link both libraries during compilation will fail due to conflicting multiple symbol definitions.

The `cl_khr_icd` OpenCL extension [20] defines the *Installable Client Driver* (ICD) and the *ICD Loader* libraries that act as a proxy between the user program and the actual implementations. With this extension, the vendor implementations are loaded at run time, avoiding the issue of symbol conflicts. The application is then linked to the ICD Loader instead of the individual vendor libraries. Figure 4.1 illustrates the relationships between the libraries. Currently, Altera does not provide an ICD for its SDK. A minimal ICD was implemented during this thesis and it is documented in this section.

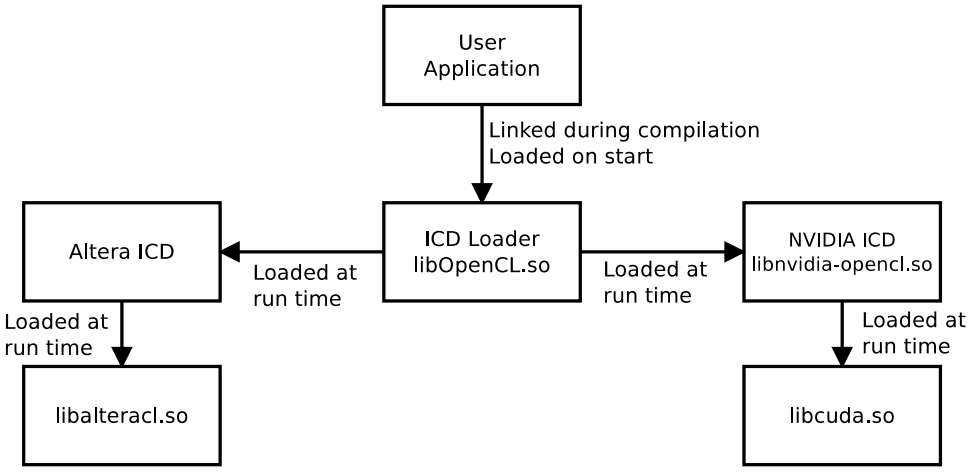


Figure 4.1.: Relationship between the dynamic libraries

To enumerate vendor ICDs on Linux, the ICD Loader scans the files located in the path `/etc/OpenCL/vendors` [20]. These files have to contain a single line with the absolute or relative path to the actual vendor ICD dynamic library. The loader will then attempt to load this library using `dlopen`.

The extension re-defines all OpenCL functions and objects. When the user application calls an OpenCL function it actually calls one of those re-defined functions from the ICD Loader. The first function, of an OpenCL application is usually `clGetPlatformIDs`.

When it is called, the ICD Loader iterates over all known vendor ICDs and in turn calls their `clGetPlatformIDs`. The vendor ICD then returns a pointer to a struct which has to contain the new `KHRicdVendorDispatch` struct, as in listing 4.1.

Listing 4.1: Definition of `struct _cl_platform_id` in the Altera ICD

```
struct _cl_platform_id
{
   KHRicdVendorDispatch *dispatch;
   cl_platform_id actual_altera_platform_id;
};
```

All other OpenCL objects have to contain the `KHRicdVendorDispatch` struct too. The `KHRicdVendorDispatch` contains a list of pointers to all remaining vendor ICD wrapper functions. The vendor ICD has to fill in this struct. When the user application calls another OpenCL function, the ICD Loader calls the appropriate function pointer from the dispatch struct. This way the correct vendor is automatically inferred. For clarification, the implementation of the `clFinish` function inside the ICD Loader is shown in listing 4.2.

Listing 4.2: Implementation of `clFinish` in the ICD Loader [21]

```
clFinish(cl_command_queue command_queue)
{
   return command_queue->dispatch->clFinish(command_queue);
}
```

The wrapper function in the vendor ICD has then to call the actual OpenCL function and pass it the real object, i.e. without the dispatch struct. Unfortunately this cannot be automated because of functions which accept more than one OpenCL object or those that create new objects. Every OpenCL function has to be re-implemented again manually. The listing 4.3 shows how this can be realized, using again the example of `clFinish`. Here, `private_dispatch` is another internally used dispatch which stores the real Altera function pointers. The diagram in figure 4.2 illustrates the complete procedure.

Listing 4.3: Implementation of the wrapper function `_icd_clFinish` in the Altera ICD

```
cl_int CL_API_CALL
_icd_clFinish(cl_command_queue command_queue)
{
      cl_command_queue
      real_queue = ((struct _cl_command_queue*)command_queue)->queue;
      return( private_dispatch.clFinish( real_queue ) );
}
```

The Altera ICD cannot be linked to the main library `libalteracl.so` which defines the actual OpenCL functions during compile time. This would again result in symbol conflicts, this time with the ICD Loader. Instead, it must be loaded with `dlopen` during the run time. `libalteracl.so` depends on `libelf.so`, `libalterammdpcie.so` and `libalterahalmmd.so`, which means these libraries have to be loaded beforehand with

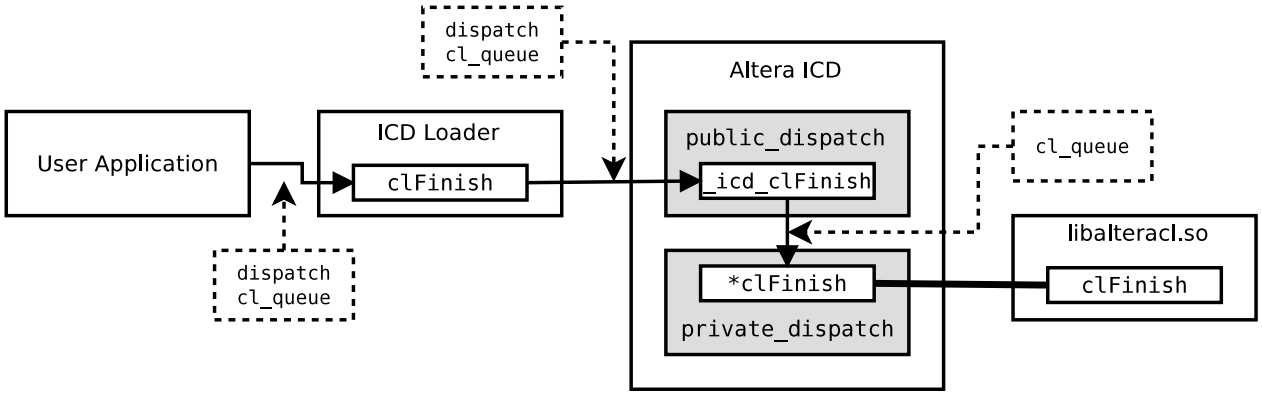


Figure 4.2.: Call graph for the function `clFinish`. The application calls the ICD loader function, which forwards the call to the wrapper function in the vendor ICD stored in the dispatch struct. The wrapper function then calls the real OpenCL function. The dashed boxes illustrate the contents of the arguments.

the `RTLD_GLOBAL` flag. This flag allows subsequently loaded libraries to use the symbols defined in the library to be loaded [23]. While the `libelf.so` can be loaded without problems, the other two libraries in turn depend on `libalteracl.so` itself, creating a circular dependency. A solution to this problem is to open `libalteracl.so` first with the `RTLD_LAZY` flag and afterwards the other libraries with the opposite flag `RTLD_NOW`. With `RTLD_LAZY`, a lazy binding is performed, that means the symbols are only resolved when needed, whereas `RTLD_NOW` resolves all symbols before `dlopen` returns [23].

Moreover to avoid conflicts with already loaded OpenCL symbols (defined by the ICD loader itself), the flag `RTLD_DEEPBIND` is required. This flag specifies that the library to be loaded should use its own symbols in preference to already loaded global symbols with the same name [23]. The correct sequence of `dlopen` operations and flags is shown in listing 4.4.

Listing 4.4: Dynamically loading Altera OpenCL libraries

```
dlopen( "libelf.so", RTLD_NOW | RTLD_GLOBAL | RTLD_DEEPBIND );
dlopen( "libalteracl.so", RTLD_LAZY | RTLD_GLOBAL | RTLD_DEEPBIND );
dlopen( "libalterammdpcie.so", RTLD_NOW | RTLD_GLOBAL | RTLD_DEEPBIND );
dlopen( "libalterahalmmd.so", RTLD_NOW | RTLD_GLOBAL | RTLD_DEEPBIND );
```

Loading the actual functions is accomplished with the `dlsym` function as in listing 4.5. They will be stored in a second dispatch struct, only for use within the ICD.

Listing 4.5: Dynamically loading the original `clFinish` and filling in the dispatch

```
private_dispatch.clFinish
   = (KHRpfn_clFinish)dlsym(libalteracl_handle, "clFinish");
errmsg=dlerror();
if(errmsg!=NULL){return(-1);}
public_dispatch.clFinish=&_icd_clFinish;
```

The OpenCL specification defines more than 100 functions, most of them are very rarely used. Therefore only the following most common OpenCL functions have been wrapped for this thesis:

`clGetPlatformIDs`	`clGetPlatformInfo`	`clGetDeviceIDs`
`clGetDeviceInfo`	`clCreateCommandQueue`	`clCreateContext`
`clCreateBuffer`	`clCreateProgramWithBinary`	`clBuildProgram`
`clCreateKernel`	`clEnqueueNDRangeKernel`	`clSetKernelArg`
`clEnqueueReadBuffer`	`clEnqueueWriteBuffer`	`clFinish`

Trying to call a function not in this list will result in a segmentation fault. Moreover, none of the `clEnqueue*` functions listed above support OpenCL *events*. Events can be sometimes useful for asynchronous operations, i.e. those that do not block. To support events, a complex memory management system that tracks the lifetime of the event objects is required.

Though incomplete, this set of functions allows a fully functional OpenCL application to be built and used together with implementations from other vendors within the same process. Additional functions can be added, as described above.

5. Implementation of Direct GPU-FPGA Communication

In this thesis, a similar approach as described by Susanto [31] will be used for the direct GPU-FPGA communication. The transfer will be handled mainly by the DMA controller on the FPGA. The CPU will only coordinate the transfer. On the GPU side the NVIDIA GPUDirect RDMA mechanism is required.

5.1. Altera PCIe Driver and IP Overview

Altera's OpenCL implementation consists mainly of a dynamic Linux library, a PCIe driver and a *Board Support Package* (BSP) containing the Intellectual Property (IP) cores to be deployed on the FPGA [3]. The dynamic library is proprietary but the BSP and the driver are open source and can be analyzed and modified.

The components of the Altera OpenCL IP stack are interconnected by the *Avalon Memory-Mapped* (Avalon-MM) interconnect [3, 7]. Avalon-MM is a multi-master/multi-slave bus. An address range is assigned to each slave connected to the bus. The master can then initiate transactions to specific slaves by writing to or reading from the corresponding slave addresses, similarly to memory-mapped I/O on the Linux operating system.

The most important cores for this thesis are the PCIe core, the DMA controller core and the DDR memory interface. Figure 5.1 provides an overview of the connections between them. When the driver writes to the FPGA's PCIe BAR with some additional offset, the PCIe core forwards the data from its RX port to the Avalon interconnect. Depending on the offset the message is routed to one of the slaves. In this specific example, to communicate with the DMA controller, the driver has to write to the address `0xc800` within the FPGA BAR.

The DMA controller is a *Modular Scatter-Gather DMA* (mSGDMA) IP core [5]. It is subdivided into a read master, write master and a dispatcher which coordinates the other two components. A transfer is initiated by writing a 32-byte *descriptor* to the dispatcher. The descriptor contains all the required information about the desired transfer, including the source and destination memory addresses and the number of bytes to be transmitted. Then, the dispatcher pushes the descriptors into a FIFO and processes them sequentially.

The PCIe core contains an *Address Translation Table* (ATT) which stores physical PCIe addresses [6]. With an ATT, the DMA controller does not need to differentiate between physical addresses or addresses pointing to on-board DDR memory. Instead, it can simply write the address it received in the descriptor from the device driver to the Avalon

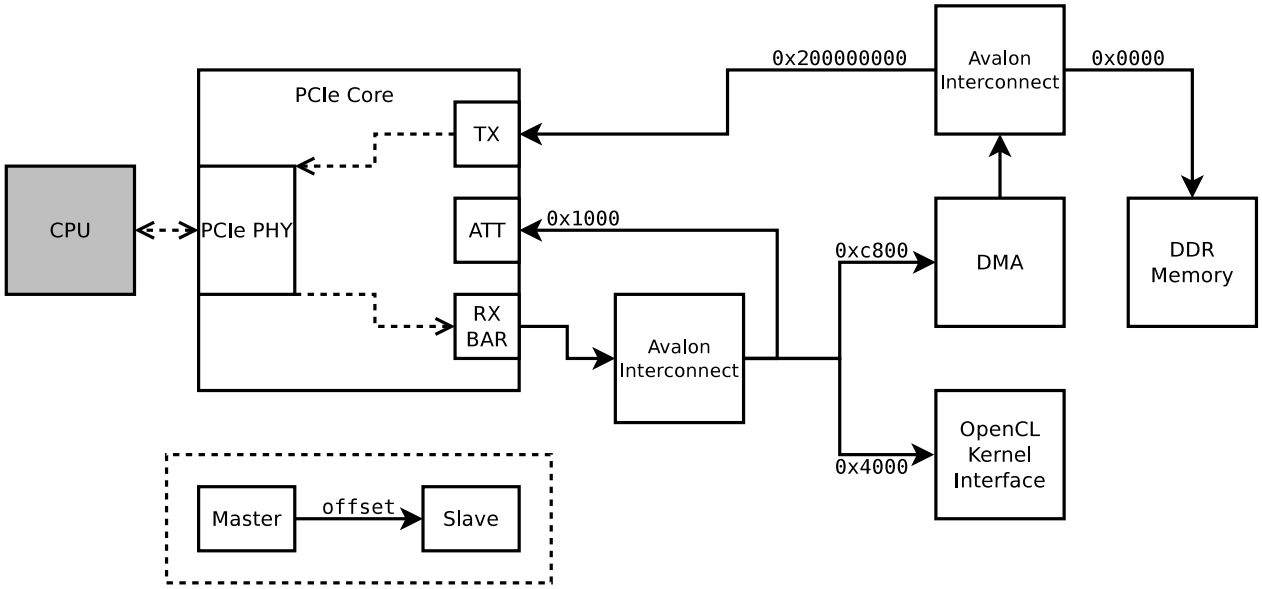


Figure 5.1.: Simplified configuration of the Altera OpenCL IP stack on the FPGA. Note that the solid arrows represent master/slave relationships and not necessarily the data flow direction.

master port. The Avalon interconnect will then route the signals to either the PCIe core or the FPGA memory. The PCIe core would then select the required physical address from the ATT. The differentiation has to made by the device driver. To specify a physical address it has first to store it in the ATT by writing to offset `0x1000` (plus the ATT row number). Then it writes to the offset `0xc800` (the DMA Controller) the address of the PCIe TX port as seen from the DMA. i.e. an offset of `0x200000000` plus the ATT row number. Listing 5.1 shows this calculation in a more portable manner with the macros already defined in the Altera PCIe driver code.

Listing 5.1: Calculation of the PCIe address as seen from the DMA controller

```
unsigned long
pcietxs_addr = (size_t) ACL_PCIE_TX_PORT
             | (att_row * ACL_PCIE_DMA_MAX_ATT_PAGE_SIZE)
             | (physical_addr & (ACL_PCIE_DMA_MAX_ATT_PAGE_SIZE - 1));
```

The direction of the transfer, i.e. whether data is to be read from or written into the FPGA memory, is specified implicitly by the `read_address` and `write_address` fields of the descriptor similar to listing 5.2.

Listing 5.2: Selection of the transfer direction

```
if(direction == READING)
{
      //reading from FPGA memory to PCIe bus
      dmadescriptor.read_address = fpga_address;
      dmadescriptor.write_address = pcietxs_address;
```

```
}
else
{
      //writing to FPGA memory from PCIe bus
      dmadescriptor.read_address = pcietxs_address;
      dmadescriptor.write_address = fpga_address;
}
```

5.2. GPUDirect RDMA Overview

NVIDIA GPUDirect [12] is a family of technologies for fast data transfers in high performance computing systems with multiple devices. Its key features include an accelerated pipeline for video capture devices, storage devices, peer to peer memory access between GPUs and RDMA (Remote Direct Memory Access). RDMA, as the name implies, is mainly intended to transfer data over network to or from other nodes in a compute cluster. However it can also be used to transfer data to other third party PCIe devices within the same root complex. One limitation is that it can only be used for NVIDIA Quadro and Tesla graphics cards. Furthermore, systems which employ an IOMMU are not compatible.

The main idea behind GPUDirect RDMA is that the GPU can expose a region of its global memory into a PCIe BAR [14]. This can then be used by third party devices to access the memory region directly without the round-trip via the CPU.

Since version 4.0, the CUDA platform, on which NVIDIA's OpenCL implementation is based, uses a memory address management system called *Unified Virtual Addressing* (UVA) [15, 14]. With UVA the GPU memory pages are mapped into the system's virtual address space providing a single address space instead of multiple address spaces, one for the CPU and one for each GPU. For the CUDA platform this simplifies the programming interface, but on the other hand requires pinning for DMA transfers.

The NVIDIA device driver provides the function `nvidia_p2p_get_pages` to pin a GPU memory region [14]. This function must be called from within the kernel space, i.e. from the third party device driver. It accepts a virtual GPU address and, if the pinning is successful, returns a page table struct containing the physical addresses to each GPU memory page. The virtual GPU address must be aligned to a 64KB boundary. The listing 5.3 provides an example of this process. `nvidia_p2p_put_pages` is the corresponding function to unpin the pages. Additionally, a callback function has to be registered which has to call the function `nvidia_p2p_free_page_table` to release resources. This callback is invoked by the NVIDIA driver when it has to revoke the mapping for some reason. This is mostly the case when the associated user process terminates.

Listing 5.3: Pinning GPU memory [14]

```
// do proper alignment, as required by NVIDIA kernel driver
u64 virt_start = ((u64)address) & GPU_BOUND_MASK;
u64 pin_size = ((u64)address) + size - virt_start;
```

```
struct nvidia_p2p_page_table page_table;
nvidia_p2p_get_pages( 0, 0, virt_start, pin_size,
                      &page_table, free_callback, &page_table );
```

For Kepler class GPUs the pages typically have a size of 64KB. The PCIe BAR is up to 256MB large of which 32MB are reserved for internal use. This means that, in theory, up to 224MB can be pinned at a time [14]. However, during development, this number has been found to be slightly smaller, around 200MB.

5.3. Extension of the Altera PCIe Driver

The DMA component of the PCIe device driver provided by Altera expects a virtual address that maps to the CPU memory. It tries to pin the buffer with the `get_user_pages` function to get a list of the pages and their physical addresses. Simply passing a GPU address to the module will thus result in an error. Neither will it work by just replacing `get_user_pages` with `nvidia_p2p_get_pages`, the corresponding GPU memory pinning function, due to assumptions related to the address space (for example about the page size) and differences in the pinning work flow. A new RDMA component, managing only the direct GPU-FPGA communication, will thus extend the driver. Transfers to and from CPU memory will be handled by the original code which shall remain untouched as far as possible.

New driver commands `ACLPCI_CMD_RDMA` and `ACLPCI_CMD_RDMA_UPDATE` are added to differentiate between the original DMA and the new RDMA transfers. They are issued via file I/O to the `/dev/acl0` device file, the same way as the original commands. Section 5.5 describes in detail how to use the new commands from user space.

5.3.1. Basic Version

To avoid unnecessary complications during development only a very basic RDMA mechanism has been implemented first. Its limitations include:

- Only one active transfer at a time
- Only one active descriptor at a time. The next descriptor is sent only after the previous one is finished.
- The size of descriptors fixed to 4KB. This is equivalent to the maximum size of an ATT entry.
- Overall transfer size limited to 192MB. As noted in section 5.2 this is roughly the maximum that can be pinned at a time.

Section 5.3.2 provides an overview over some improvements to this version.

The new RDMA component consists of five main functions. Figure 5.2 illustrates their relationship. Their semantics are as follows:

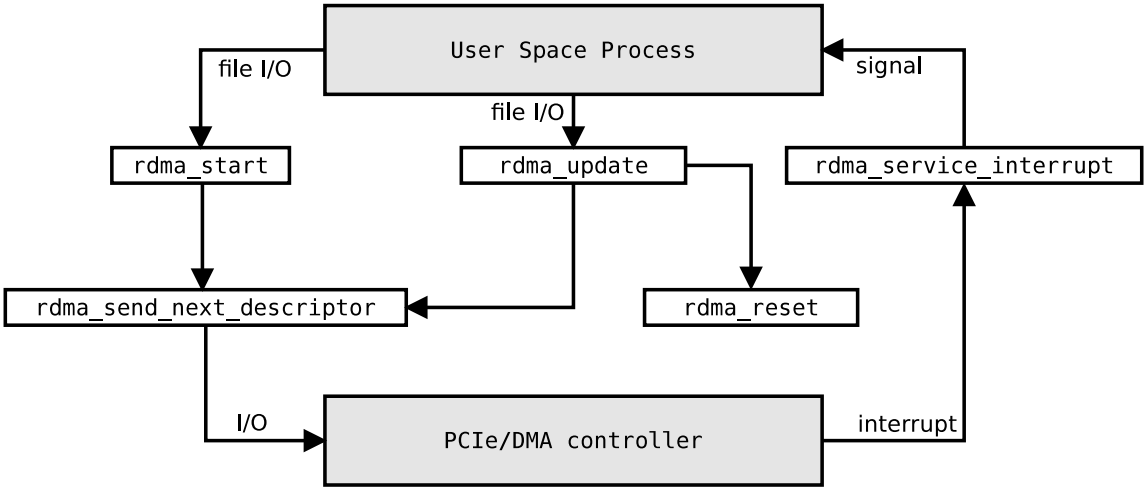


Figure 5.2.: The main functions of the new RDMA component

- **`rdma_start`**: The entry point for all RDMA transfers, invoked when the user space program issues the new `ACLPCI_CMD_RDMA` driver command. If there is no other active RDMA transfer, it tries to pin the supplied virtual GPU address as in listing 5.3. If successful, the actual transfer is started by calling the `rdma_send_next_descriptor` function.
- **`rdma_send_next_descriptor`**: This function performs the actual communication with the DMA controller on the FPGA. It has to write the physical GPU address to an ATT entry of the PCI core and send a descriptor with the addresses and the size of the transfer. After this function, the driver terminates the execution and returns the control to the user space.
- **`rdma_update`**: This function is invoked from user space with the new driver command `ACLPCI_CMD_RDMA_UPDATE` to continue with the transfer. If the previous descriptor has finished, it calls `rdma_send_next_descriptor` to send the next one. It returns `1` if there is still data remaining to be transferred and `0` when the transfer finished to inform the user program.
- **`rdma_service_interrupt`**: This is the new DMA interrupt handler for the device driver. It replaces the previous interrupt handler `aclpci_dma_service_interrupt` from the original DMA code. It is called whenever the DMA controller signals an interrupt, i.e. when a descriptor is finished. Its first task is to acknowledge the interrupt by clearing the IRQ bit as soon as possible, to avoid multiple interrupts in a row. An interrupt does not differentiate whether a descriptor is from the RDMA or the original DMA component. Therefore the new handler has to relay to the original one if no RDMA transfer is active.

Listing 5.4: Acknowledging the interrupt and relaying to the original handler if needed

```
// Clear the IRQ bit
dma_write32 (aclpci, DMA_CSR_STATUS, ACL_PCIE_GET_BIT(DMA_STATUS_IRQ) );
```

```
//check whether the interrupt is for RDMA or DMA
if( !rdma_active )
{
      //this is an interrupt for the original DMA component
      return( aclpci_dma_service_interrupt( aclpci ) );
}
```

An interrupt handler has to return as fast as possible. Therefore it should not directly call `rdma_update` to continue with the transfer. Instead, it issues the new `SIG_INT_RDMA` signal to the user process, which in turn drives the transfer with the `rdma_update` function.

At this point, one of the few interactions to the driver's original DMA component has to be made: this component keeps track of the number of completed descriptors to be able to calculate how much data has been transferred. This value has to be kept updated, otherwise it will not function correctly. This update procedure is shown in listing 5.5.

Listing 5.5: Updating the number of completed descriptors in the original code

```
// Read the DMA status register
dma_status = dma_read32 (aclpci, DMA_CSR_STATUS );
// Compute how many descriptors completed since last reset
done_count = ACL_PCIE_READ_BIT_RANGE(dma_status,
                                     DMA_STATUS_COUNT_HI,
                                     DMA_STATUS_COUNT_LO );
//have to update the dma_data struct
//so that the original DMA module does not get confused
aclpci->dma_data.m_old_done_count = done_count;
```

- **`rdma_reset`**: Unpins the GPU buffer and removes the `rdma_active` flag again, so that future hardware interrupts are relayed to the original DMA component.

5.3.2. Optimizations

The version presented above describes only a very basic implementation with the purpose to prevent errors and to be intuitive to understand. Maximal bandwidth should not be expected from it. This section presents three main optimizations for higher performance.

Larger Descriptor Sizes

Descriptors of size 4KB, as used above, are convenient because this corresponds to the maximal size of an ATT page. By increasing the descriptor size, overall less descriptors are required to complete a transfer. This in turn, reduces the number of interrupts and the delays caused by the software to react to a hardware interrupt. Larger descriptors can be constructed by simply incrementing the `bytes` count register sent to the DMA controller. After the transfer of the first 4KB of an ATT entry is finished, the consecutive ATT entry

is used. However the addresses have to be continuous. Up to 128 ATT entries can be covered by one descriptor [3]. This corresponds to a size of up to 512KB.

Listing 5.6: Setting the size of a descriptor as large as possible

```
for( i=0; i < ACL_PCIE_DMA_MAX_ATT_PER_DESCRIPTOR; i++ )
{
      unsigned offset_i = i * ACL_PCIE_DMA_MAX_ATT_PAGE_SIZE;
      unsigned page_nr_i = offset_i / GPU_PAGE_SIZE;

      //check whether the transfer size has already been reached
      if( bytes_sent + dmadesc.bytes >= transfer_size )
      { break; }

      //check whether the next gpu page is consecutive to previous one
      if( page_table->pages[page_nr + page_nr_i]->physical_address
          != phymem+page_nr_i * GPU_PAGE_SIZE )
      { break; }

      set_att_entry( aclpci, phymem + offset_i, next_free_att_row );
      dmadesc.bytes += ACL_PCIE_DMA_MAX_ATT_PAGE_SIZE;
      next_free_att_row = (next_free_att_row+1)%ACL_PCIE_DMA_MAX_ATT_SIZE;
}
```

Multiple Descriptors

Additionally to larger descriptors, the DMA controller on the FPGA also accepts multiple descriptors at once. They are buffered in a FIFO queue and processed one after the other. As with the optimization above, this can reduce the number of delays from an interrupt until the next descriptor. The size of the FIFO queue is fixed in the hardware to a maximum of 128 descriptors [3]. However, the actual number depends even more on the size of the ATT which is limited to 256 entries or 1MB. Only two 512KB descriptors already span the whole table. Sending a third one would require to overwrite the ATT entries of the first descriptor which would result in incorrectly transmitted data.

Instead of calling `rdma_send_next_descriptor` directly, the functions `rdma_update` and `rdma_start` will now call the new function `rdma_send_many_descriptors`.

Listing 5.7: Sending as many descriptors as possible or needed

```
//maximum of two descriptors because this would override ATT
for( i = 0; i < 2; i++ )
{
      //check whether all data has been sent already
      if(bytes_sent >= transfer_size){ break; }
      rdma_send_next_descriptor();
}
```

Since the descriptors may be of different sizes, two additional values have to be stored between the transfers: The number of descriptors sent to the FPGA which is incremented

in `rdma_send_next_descriptor` and the number of completed descriptors, updated in the interrupt handler. Due to possible race conditions, the number of sent descriptors cannot be accessed in the asynchronous interrupt handler: an interrupt may arrive while the driver is still busy sending new descriptors to the DMA controller and thus incrementing this value. Access synchronization with mutexes or semaphores is not possible because an interrupt handler is not allowed to wait. The handler will only update the done count to the value read out from the DMA status register. The `rdma_update` function which is called via a command from the user space will then compare those two values. If equal, then the next descriptors can be issued to the DMA controller.

Lazy Unpinning

Lazy Unpinning is an optimization technique that is recommended by NVIDIA in the GPUDirect RDMA documentation [14]. Pinning GPU memory pages with the function `nvidia_p2p_get_pages` is an expensive operation that may take several milliseconds to complete. It should be performed as rarely as possible. A naive implementation, as the one described above, that pins a buffer before every transfer and unpins it afterwards will not perform to full potential.

For a higher degree of performance, the memory region should be kept pinned after the transfer is finished. For the first transfer to or from a buffer, the driver still has to pin the memory and the optimization will not affect it. All following transfers however, will benefit.

To realize this behavior, a look-up table will store the virtual GPU addresses and the corresponding page tables containing the physical addresses between the transfers. The size of the look-up table is fixed to 6. Section 5.5 explains why this number was chosen. When a transfer command is initiated, the requested virtual GPU address has to be looked up in the table first. Actual pinning is then only required if the table does not contain this address.

A buffer should be only unpinned when the table is full to make room for a new entry. The decision which entry to remove is not straightforward. A strategy like *least recently used* (LRU) can be beneficial if many buffers (more than 6) have to be accessed in a non-sequential order. On the other hand, if the access is strictly consecutive it is better to remove always one specific entry and leave the other 5 pinned. This strategy was selected also because it is beneficial for very large transfers (>200MB) and easier to implement. In future, if a higher degree of control is desired, it may be reasonable to leave this decision to the user application.

When the user space process terminates, the NVIDIA driver revokes the mapping. The look-up table has to be cleaned up accordingly in the callback function.

The look-up table also indirectly removes the limitation of the maximum transfer size of 192MB. This is explained in section 5.5.

5.4. GPUDirect RDMA for OpenCL

The GPUDirect RDMA technology that allows direct PCIe communication between a NVIDIA GPU and a third-party device is only suppported for the CUDA toolkit and not for OpenCL [14]. The central function `nvidia_p2p_get_pages` returns the error code `-EINVAL` if it is called with an address from a buffer allocated by OpenCL. This thesis proves that it is nevertheless possible to make it work with OpenCL despite the lack of official support. This section documents the actions that were taken to achieve this. In subsection 5.4.1, the communication calls to the NVIDIA driver from the CUDA and OpenCL libraries are analyzed using reverse engineering techniques. Subsection 5.4.2 describes the modifications to the NVIDIA kernel module that are necessary to imitate CUDA buffer allocation from within OpenCL.

5.4.1. Reverse Engineering the NVIDIA Driver Communication

The CUDA and OpenCL dynamic libraries, as provided by NVIDIA, communicate with the NVIDIA kernel module via the `ioctl` system call [28]. The `ioctl` call takes 3 arguments [24]:

- an open file descriptor (in this case referring to `/dev/nvidia0`)
- a 32-bit request code of which bits 7 to 0 are the *command* number.
- a pointer to a parameter stored in the user address space. Its size is encoded in the bits 29 to 16 of the request code mentioned above.

Inside the module, the function `nvidia_ioctl(...)` is registered as a callback function for this kind of calls. Besides thread synchronization and several sanity checks the function itself only performs very simple operations like returning device information or the driver version. For all other commands it relays the `ioctl` parameters to the `rm_ioctl(...)` function which is defined in the proprietary binary driver. The semantics of the `ioctl` commands are not documented by the vendor.

The GPUDirect RDMA function `nvidia_p2p_get_pages` is defined in the NVIDIA kernel module and thus has itself no way of knowing whether CUDA or OpenCL is used in the user program. As a consequence, there must be some differences in the way the two libraries communicate with the module. Of special interest are the commands for buffer allocation.

In several experiments, the module has been modified to save all incoming `ioctl` commands and their parameter data to files on the hard drive to analyze them later. Two minimalistic applications, one for CUDA and one for OpenCL, that perform as little initialization as needed and allocate one or more memory buffers provided the stimuli. The data files were then compared for differences.

Some observations from the resulting data are that both libraries perform internal device memory management to some degree. Small buffer allocations up to a size of 1MB are sometimes not relayed to the driver. Additionally, NVIDIA's OpenCL implementation

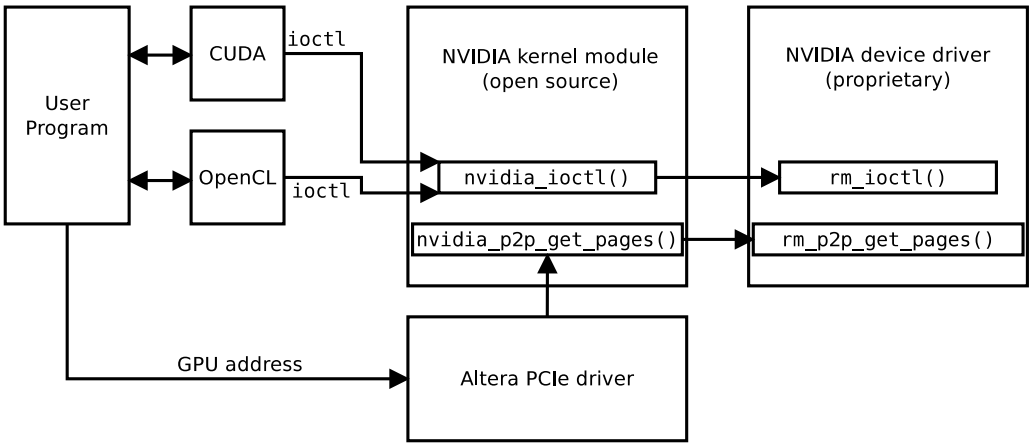


Figure 5.3.: Communication between CUDA/OpenCL and the NVIDIA driver

seems to utilize lazy buffer allocation, i.e. no communication to the driver happens until the buffer is actually used. To circumvent this, at least one byte of data had to be written to the buffer for the experiments.

The key finding from these experiments is that CUDA sends 3 `ioctl` messages to the kernel module to allocate a buffer, OpenCL on the other hand sends only 2. Their request codes are as follows (command numbers highlighted):

Order	CUDA	OpenCL
1	0xc0a046**4a**	0xc0a046**4a**
2	0xc03846**57**	0xc03846**57**
3	0xc02046**2a**	-

Libcudest [10] is a project aimed at reverse engineering the CUDA platform. Its documentation pages provide explanations to some of the command codes. The `0x4a` and `0x57` commands are not listed but the `0x2a` command, that is missing in OpenCL, is described there as a *GPU method invocation*. The `ioctl` parameter is 32 (`0x20`) bytes large and specifies the GPU method type in the second and third words, an address to the GPU method parameter in the fifth and sixth words and its size in the seventh and eighth words. No information about the first word of the parameter is provided there. The analysis of the data from several experiment trials showed that this word always contains the value `0xc1d00001` for the first process that is started after the NVIDIA driver is loaded. It is then incremented for each following process that uses CUDA or OpenCL. Therefore this word will be called *User Identifier* in this thesis. The following table provides an example of the `0x2a ioctl` parameter that is missing in OpenCL:

Offset	Content	Description
0x00	0xc1d00001	*User / Process Identifier*
0x04	0x5c000007	GPU method type [10]
0x08	0x503c0104	
0x0c	0x00000000	unknown / always zero
0x10	0xcbfe9bc0	Address to a GPU method parameter [10]
0x14	0x00007fff	(user space)
0x18	0x00000004	Size of the GPU method parameter [10]
0x1c	0x00000000	(4 bytes)

This GPU method (`0x5c000007:503c0104`) is not documented by Libcudest. The GPU method parameter in this case is 4 bytes large. Again, after data analysis it has been observed that, similarly to the User Identifier, this parameter value starts with a fixed number for the first device buffer allocation within a process and increments for each of the following buffer allocations. It will be called *Buffer Identifier* from now on. For completeness, a GPU method parameter of the first user buffer allocated in CUDA is shown in this table:

Offset	Content	Description
0x00	0x5c000029	*Buffer Identifier*

Both, the User Identifier as well as the Buffer Identifier are also present in the parameters of the preceding `0x4a` and `0x57` `ioctl` commands (also in OpenCL). The following table shows a partial example of the `0x57` command parameter. The identifiers are located at offsets `0x00` and `0x0c`.

Offset	Content	Description
0x00	0xc1d00001	*User Identifier*
0x04	0x5c000001	unknown
0x08	0x5c000003	unknown
0x0c	0x5c000029	*Buffer Identifier*
0x10	0x00000000	unknown
0x14	0x00000000	unknown
0x18	0x04000000	Buffer Size
0x1c	0x00000000	(64MB)
...	...	unknown

After sending the missing `0x2a` `ioctl` command with the correct identifiers manually, following to a call to `clCreateBuffer` in OpenCL, the `nvidia_p2p_get_pages` will accept the OpenCL buffer and return its physical address. This allows to emulate CUDA behavior in OpenCL for GPUDirect RDMA transfers.

5.4.2. Extension of the NVIDIA Kernel Module

Neither the User Identifier nor the Buffer Identifier are accessible to the user application because they are managed internally by the CUDA and OpenCL libraries. A possible way

to retrieve them is to intercept the `0x4a` or `0x57` `ioctl` messages. Again, modification of the NVIDIA kernel module is required.

The new module will simply extract the identifiers from the `0x57` messages and store them in a static variable. A new `ioctl` command will then return the intercepted identifiers to the user application. The the new request code should be chosen carefully, so that it does not collide with an existing NVIDIA `ioctl` command. Unfortunately, a complete list of these codes is not available. Out of observation and from the Libcudest project [10] the assumption was made that all original NVIDIA commands use read and write parameters as encoded in the bits 31 and 30 of the 32-bit request code. In this case, the parameter can be write-only, thus the read bit has been chosen to be `0`. At the same time, this should guarantee the new code to be collision-free. Since two 32-bit values have to be retrieved, the parameter size (bits 29 to 16) should be `0x08`. The module type code (bits 15 to 8) used by NVIDIA is `0x46` and was adopted. Finally, the command number (bits 7 to 0) was chosen to be `0x00` resulting in the complete request code `0x40084600`.

The changes applied to the module can be summarized as in listing 5.8.

Listing 5.8: Changes to `nvidia_ioctl(..)` in the NVIDIA kernel module

```
if( arg_cmd == 0x57 )
{
      last_buffer_id = *(unsigned*)(arg_copy + 12);
      last_user_id = *(unsigned*)(arg_copy + 0);
}
if( cmd == 0x40084600 )
{
      ((unsigned*)arg_copy)[0] = last_buffer_id;
      ((unsigned*)arg_copy)[1] = last_user_id;
      goto done;
}
```

It should be noted that these modifications are not thread-safe, i.e. allocating multiple buffers in different threads or processes at the same time will cause race conditions. However, since buffer allocation is usually performed only once during the initialization, this issue should not be of large concern. Further development is needed if thread safety is desired.

5.5. User Space Invocation

A user space application cannot simply use the standard OpenCL data transfer functions to make use of the new capabilities of the modified driver. Peer to peer transfer functions for example, are only provided by vendor specific extensions. The small library `cl_rdma.c` will thus provide an interface to the new RDMA mechanism. An example minimal application is shown in appendix A. This section is mainly concerned with the inner workings of this library.

The initialization procedure involves opening both of the device driver files located in the path `/dev/*`. All of the communication to the drivers will happen will happen through the `read`, `write` and `ioctl` system calls on the resulting file descriptors. An example of how to issue a command to the Altera driver, in this case the retrieval of the driver version, is shown in the listing 5.9. The modified driver appends the string `"with_rdma"` to the version which should be checked to ensure that the correct driver is loaded. Moreover, a signal handler has to be installed to catch the signals that are issued when a RDMA descriptor is finished.

Listing 5.9: Initialization procedure

```
//open the device drivers
int nvidia0fd = open( "/dev/nvidia0", O_RDWR );
int acl0fd = open( "/dev/acl0", O_RDWR );

//check whether the modified altera driver is loaded
char cbuf[1024] = { 0 };
struct acl_cmd cmd = { ACLPCI_CMD_BAR, ACLPCI_CMD_GET_DRIVER_VERSION,
                       NULL, &cbuf, sizeof(cbuf) };
read( acl0fd, &cmd, sizeof(cbuf) );
if( strstr( cbuf, "with_rdma" ) == NULL ){ return(1); }

//setup signal handler for interrupts from the driver
struct sigaction sa = { 0 };
sa.sa_sigaction = signal_handler;
sigemptyset( &sa.sa_mask );
sigaction( SIG_INT_RDMA, &sa, NULL );
```

The signal handler (listing 5.10) is very simple. Its only task is to wake up the main thread which is waiting for a descriptor to complete. This is done through a global semaphore.

Listing 5.10: Signal handler function

```
static void signal_handler(int signum, siginfo_t* siginfo, void* uctx)
{
      sem_post( &semaphore );
}
```

An address from each buffer is needed for the RDMA transfers. Unfortunately, the OpenCL buffer allocation function `clCreateBuffer` returns an object of the type `cl_mem` and not directly the address to the memory on the device. In the header file `<CL/cl.h>`, the type `cl_mem` is defined as

```
typedef struct _cl_mem* cl_mem;
```

with `struct _cl_mem` not defined, which means that the struct is defined externally in the proprietary NVIDIA or Altera OpenCL dynamic libraries and its layout is unknown. Specifically, the buffer address cannot be simply extracted from the pointer. A solution is to launch the kernel that is shown in listing 5.11 with the desired buffer as parameter `a`.

Listing 5.11: OpenCL C kernel to retrieve the address from a `cl_mem` object

```
__kernel void get_address(__global unsigned long* a)
{
      a[0] = (unsigned long) a;
}
```

The kernel writes the address of the buffer into the buffer itself. This value can then be retrieved by reading the first 8 bytes from the buffer with the `clEnqueueReadBuffer` function. This procedure has to be performed only once for each buffer during the initialization of a program.

As noted in section 5.4, normal NVIDIA OpenCL buffers cannot be simply pinned by the `nvidia_p2p_get_pages` function. The `0x2a ioctl` command has to be sent manually with the Buffer and User Identifiers retrieved from the modified NVIDIA module. Moreover, the size of the buffer should be at least 1MB and one byte has to be written into it to ensure it is actually allocated. The complete procedure is shown in listing 5.12

Listing 5.12: Creating a pinnable buffer in OpenCL

```
//a buffer will not always be allocated if its smaller than 1MB
if( size < 1024 * 1024 ){ size = 1024 * 1024; }
cl_mem buffer = clCreateBuffer(context,CL_MEM_READ_WRITE,size,NULL,NULL);

//make sure the buffer actually gets allocated by writing a byte into it
char dummydata = 0xff;
clEnqueueWriteBuffer(queue,buffer,CL_TRUE,0,1,&dummydata,0,NULL,NULL);

//get the buffer and user identifiers from the modified NVIDIA module
unsigned ioctl_param[2];
ioctl( fd, 0x40084600, ioctl_param );

//send the missing 0x2a ioctl to emulate CUDA behavior
unsigned gpu_method_param = ioctl_param[0];
unsigned gpu_method[8] = { ioctl_param[1],
                           0x5c000007u,
                           0x503c0104u,
                           0x00000000u,
                           0x00000000u,
                           0x00000000u,
                           0x00000004u,
                           0x00000000u };
((unsigned long*) gpu_method)[2] = (unsigned long) &gpu_method_param;
ioctl( fd, 0xc020462a, gpu_method);
```

The actual transfer is started by sending the new `ACLPCI_CMD_RDMA` command to the Altera driver including the FPGA and GPU addresses. This will invoke the function `rdma_start` inside the driver. As mentioned in section 5.2, the GPU is not able to pin more than ca. 200MB at a time. Therefore, for large buffers, the transfer has to be partitioned into smaller blocks which are processed sequentially. A rather small block size of 32MB was chosen. This allows the look-up table (from the Lazy Unpinning optimization

in section 5.3.2) to be used in a more efficient way: when frequent unpinning is required, only these 32 MB will be unpinned. With a maximum size of 6, the table can thus hold up to 192MB pinned GPU blocks, which is almost the maximum amount that can be pinned at a time. The block size of 32MB is still large enough to have no measurable impact on the performance.

Listing 5.13: RDMA transfer initation

```
unsigned blocksize=1024*1024*32; //32MB
for(int i = 0; i < size; i += blocksize)
{
      if(i+blocksize> size){blocksize=size-i;}
      struct acl_cmd cmd={ACLPCI_DMA_BAR, ACLPCI_CMD_RDMA,
                          (void*)(fpga_address+i), (void*)(gpu_address+i),
                           blocksize, 0};
      read(acl0fd, &cmd, blocksize);
      clrdma_wait_for_transfer(timeout_seconds);
}
```

As noted in section 2.3, kernel modules are always completely event-driven, i.e. do not have a main loop. As a consequence, the RDMA component must be triggered from outside to continue with the transfer after a descriptor is finished. This is done by sending the new driver command `ACLPCI_CMD_RDMA_UPDATE` to the driver which will invoke the driver function `rdma_update`. To avoid busy waiting, a wait on a semaphore is used, which gets interrupted by the signal handler from listing 5.10. In case of an error a timeout can be specified to avoid a deadlock.

Listing 5.14: Driving the RDMA transfer

```
struct timespec ts;
struct timeval tp;
gettimeofday(&tp, NULL);
// Convert from timeval to timespec
ts.tv_sec = tp.tv_sec;
ts.tv_nsec = tp.tv_usec * 1000;
ts.tv_sec += timeout_seconds;

while(rdma_result != 0)
{
      //request transfer update from driver
      struct acl_cmd cmd={ACLPCI_DMA_BAR, ACLPCI_CMD_RDMA_UPDATE,
                          NULL, NULL, 0};
      rdma_result=read(acl0fd, &cmd, sizeof(cmd));
      if(rdma_result == 0){ break; } //transfer done

      //wait for interrupt from signal handler
      while(1)
      {
            int rc=sem_timedwait(&semaphore, &ts);
            if(rc && errno == EINTR){ continue; } //spurious wakeup
            else{ break; } //actual interrupt or timeout } }
```

6. Implementation of Concurrent Indirect GPU-FPGA Communication

While direct device to device communication is in theory always the fastest possible method, it is more cumbersome to use, requires unofficial modified drivers and is only available for NVIDIA Quadro and Tesla graphics cards. The indirect method that involves a round-trip via the CPU on the other hand can be simply implemented in standard OpenCL, also with the more popular GeForce cards. This section describes a naive version and an optimization that can be applied for large transfers

The simplest possible procedure of an indirect transfer can be summarized as follows:

1. Read data from the first device into a CPU buffer
2. Wait until the transfer is done
3. Send data from the CPU buffer to the second device

In OpenCL this can easily be accomplished in only two lines of code (error checking omitted):

Listing 6.1: Sequential indirect device to device transfer in OpenCL

```
clEnqueueReadBuffer (queue_device1, buffer_device1, CL_TRUE,
                     0, size, cpu_buffer, 0, NULL, NULL);
clEnqueueWriteBuffer(queue_device2, buffer_device2, CL_TRUE,
                     0, size, cpu_buffer, 0, NULL, NULL);
```

However, this is highly inefficient because at any time one of the two devices is not doing anything.

Both, the GPU as well as the FPGA have one DMA controller. Both can operate independently of each other. By splitting up a large transfer into multiple small transfers, one DMA controller can read from a portion of the temporary CPU buffer while the other one is simultaneously writing to the next portion. This results in an overall faster transfer, as illustrated in figure 6.1.

An important parameter is the size and number of the smaller transfers. If their size is too large, hardly any benefit from the concurrency is gained. With a lot of transfers on the other hand, too much time is wasted for synchronization and in the worst case may be even slower than the sequential transfer strategy. Different parameters were evaluated and the results are presented in section 7.3.

During the implementation of this transfer strategy, several problems appeared that are documented in the following paragraphs:

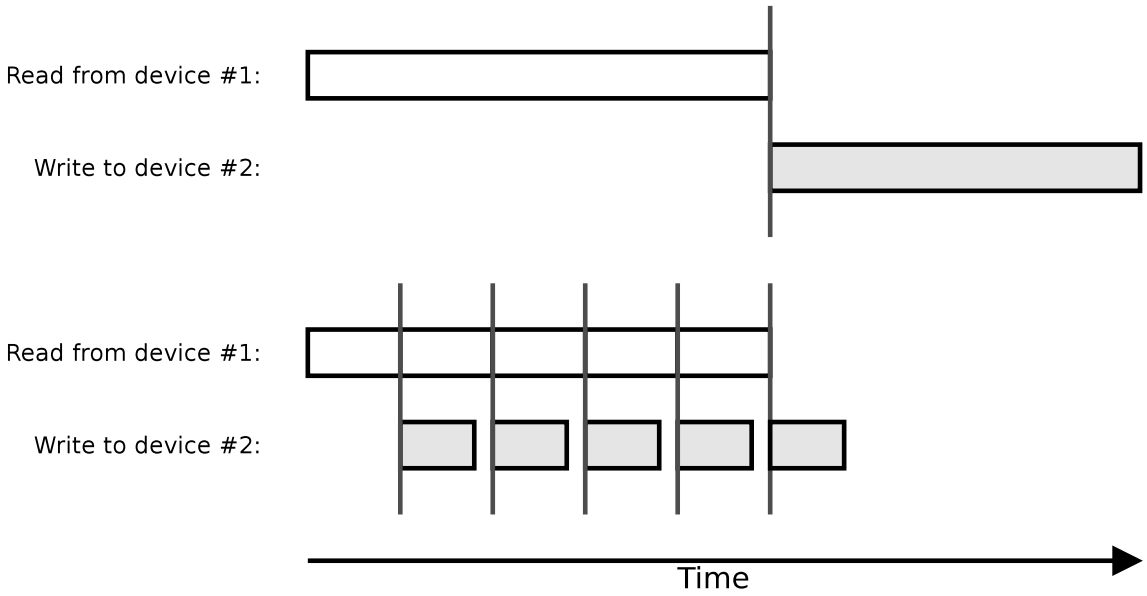


Figure 6.1.: Comparison of sequential (top) and concurrent (bottom) indirect transfer strategies. The two large transfers are splitted into 10 smaller transfers, 8 of them can be performed simultaneously. The gray vertical lines represent synchronization barriers.

The first attempt was to utilize the *asynchronous* memory transfer capabilities of OpenCL. A function call that initiates an asynchronous transfer does not wait until the transfer is done and returns immediately. This can be used to perform other operations in the meantime. In OpenCL, the `blocking_write` argument of the `clEnqueueWriteBuffer` can be set to `CL_FALSE` to start an asynchronous write operation. The complete but simplified code is presented in listing 6.2 (error checking omitted).

Listing 6.2: Asynchronous indirect device to device transfer

```
unsigned bytes_done = 0;
while(bytes_done < size)
{
      if(blocksize > size - bytes_done ){blocksize = size - bytes_done;}

      //synchronous read
      clEnqueueReadBuffer (queue_device1, buffer_device1, CL_TRUE,
                           bytes_done, blocksize, cpu_buffer+bytes_done,
                           0, NULL, NULL);
      //asynchronous write
      clEnqueueWriteBuffer(queue_device2, buffer_device2, CL_FALSE,
                           bytes_done, blocksize, cpu_buffer+bytes_done,
                           0, NULL, NULL);
      bytes_done += blocksize;
}
//wait for the asynchronous write transfers to finish
clFinish(queue_device2);
```

The first observation from the execution of this code was, that NVIDIA's implementation of the `clEnqueueWriteBuffer` function does not return immediately despite passing `CL_FALSE` as the `blocking_write` argument. Instead it blocks until the write is completed, as if it was a synchronous transfer. The solution to this issue can be found in the *NVIDIA OpenCL Best Practices Guide* [27]. A *pinned* CPU memory buffer is required as the source for asynchronous writes. It can be created as shown in listing 6.3 and simply used in the previous code example.

Listing 6.3: Creation of a pinned CPU buffer in OpenCL

```
cl_mem pinned_buffer =
   clCreateBuffer(gpu_context,
                  CL_MEM_READ_ONLY | CL_MEM_ALLOC_HOST_PTR,
                  size, NULL, NULL);

unsigned char* cpu_buffer =
   (unsigned char*) clEnqueueMapBuffer(gpu_queue, pinned_buffer, CL_TRUE,
                                       CL_MAP_WRITE, 0, size, 0,
                                       NULL, NULL, NULL);
```

The larger issue stems from the Altera side. Though its asynchronous transfer initiation call does not block (as expected), the `clFinish` function, required for the final synchronization, blocks for a surprisingly long time. In fact, this time is roughly equivalent to the time that is needed for a complete synchronous transfer. This indicates that no data is actually transferred, until this function is called. This suspicion is substantiated when taking into account the architecture of the Altera PCIe driver: Its DMA component does not notify the user space process when a transfer (or a portion of it) is completed. Instead, it relies on *polling* from the user space to drive the transfer. In all likelihood this polling is performed by `clFinish`.

In an attempt to circumvent this problem, the code has been modified to start a second thread which calls the `clFinish` function while the main thread continues to issue the asynchronous transfers as before. Though at first glance, this seemed to work, a new unexpected behavior appeared: Approximately 2% of the time, `clFinish` did not terminate at all (or resulted in a segmentation fault after some time) while CPU usage was constantly at 100% workload. This behavior is of course not acceptable in a real application. The reasons for this are unknown. One can only speculate that Altera's OpenCL implementation is maybe not thread-safe.

The final solution that proved to work, was to use only synchronous transfers, but in two threads. While one thread is only concerned with reading from device 1, the second thread waits until a read transfer is finished and then performs a blocking write to device 2. The simplified code for both threads is shown in listings 6.4 and 6.5.

Listing 6.4: First thread for concurrent device to device transfer

```
unsigned bytes_read = 0;
pthread_t write_thread;
pthread_create( &write_thread, NULL,
                concurrent_write_thread, (void*) &bytes_read);
while(bytes_read < size)
{
      clEnqueueReadBuffer(queue_device1, buffer_device1, CL_TRUE,
                          bytes_read, blocksize, cpu_buffer+bytes_read,
                          0, NULL, NULL);

      bytes_read += blocksize;
}
pthread_join(write_thread, NULL);
```

Listing 6.5: Second thread for concurrent device to device transfer

```
unsigned bytes_sent = 0;
while(bytes_sent < size)
{
      //busy waiting until a read transfer is completed
      while(bytes_sent >= bytes_read) { sched_yield(); }

      clEnqueueWriteBuffer(queue_device2, buffer_device2, CL_TRUE,
                           bytes_sent, blocksize, cpu_buffer+bytes_sent,
                           0, NULL, NULL);

      bytes_sent += blocksize;
}
```

Note that no synchronization with mutexes is required to access the shared variable `bytes_read` because this is a *single producer single consumer* case. However, a condition variable may be useful to eliminate the busy waiting loop. Additionally for stability reasons, an *abort flag* is shared between the threads, not shown in these code snippets.

7. Evaluation

The three methods (direct, concurrent indirect and sequential indirect) have been evaluated for bandwidth. This section presents the results of these evaluations. All tests were performed 5 times and the results averaged. In each trial, the memory buffer on the first device was filled with random data. After the benchmarked transfer from device 1 to device 2, the data was read out from the second device again and compared to the original data to make sure it is correct.

The largest issue that appeared already during the development of the driver is that the direct transfer in the direction from the GPU to the FPGA does not work. Trying this direction, will make the FPGA board unresponsive or may even result in a complete freeze of the operating system. This makes the issue hard to debug, because each time a system reboot is required. Despite extensive efforts, including an analysis of the OpenCL IP system on the FPGA and contacting Altera, this issue could not be fixed during this thesis.

7.1. Hardware Configuration

A NVIDIA Quadro GPU and an Altera Stratix V FPGA were used for the development and evaluation during this thesis. The relevant features for both devices are shown in the following table:

	FPGA [26]	GPU [18]
Model	Nallatech 385-D5	Hewlett-Packard Quadro K600
Architecture	Altera Stratix V D5	NVIDIA Kepler GK 107
Original driver version	13.1	340.58
PCIe Generation	3.0	2.0
Number of PCIe lanes	8x	16x
Global memory size	8GB	1GB
Memory type	DDR3	GDDR3
Measured bandwidth to CPU	750 MB/s	1930 MB/s
Measured bandwidth from CPU	550 MB/s	1950 MB/s

7.2. Effects of RDMA Optimizations

To find out to which extent the optimizations from section 5.3.2 contribute to the final performance, each of them has been evaluated. The results are shown in figure 7.1.

The first version, that was developed mainly as a prototype performs very poorly, as expected. A maximum bandwidth of only 170 MB/s has been measured, which is even slower than the sequential indirect transfer method. Its main bottleneck is the response latency from a PCIe interrupt until the next descriptor is sent, as indicated by the measurements from the larger descriptors optimization. This optimization had the largest impact, raising the bandwidth to ca. 580 MB/s for large transfers with 512KB descriptor sizes.

The multiple descriptors optimization raises the maximum measured bandwidth to ca. 610 MB/s. The overall improvement is rather small because of the limitation of being able to process only 2 descriptors at a time which in turn is due to the small ATT.

Lazy unpinning contributes a roughly constant performance gain, independent of the transfer size. This is especially important for the smaller transfers. Note that the data for this version does not include the first transfer, which performs the actual pinning and does not benefit from this optimization. Only the subsequent transfers are considered. This is nevertheless representable, since real applications usually require many transfers.

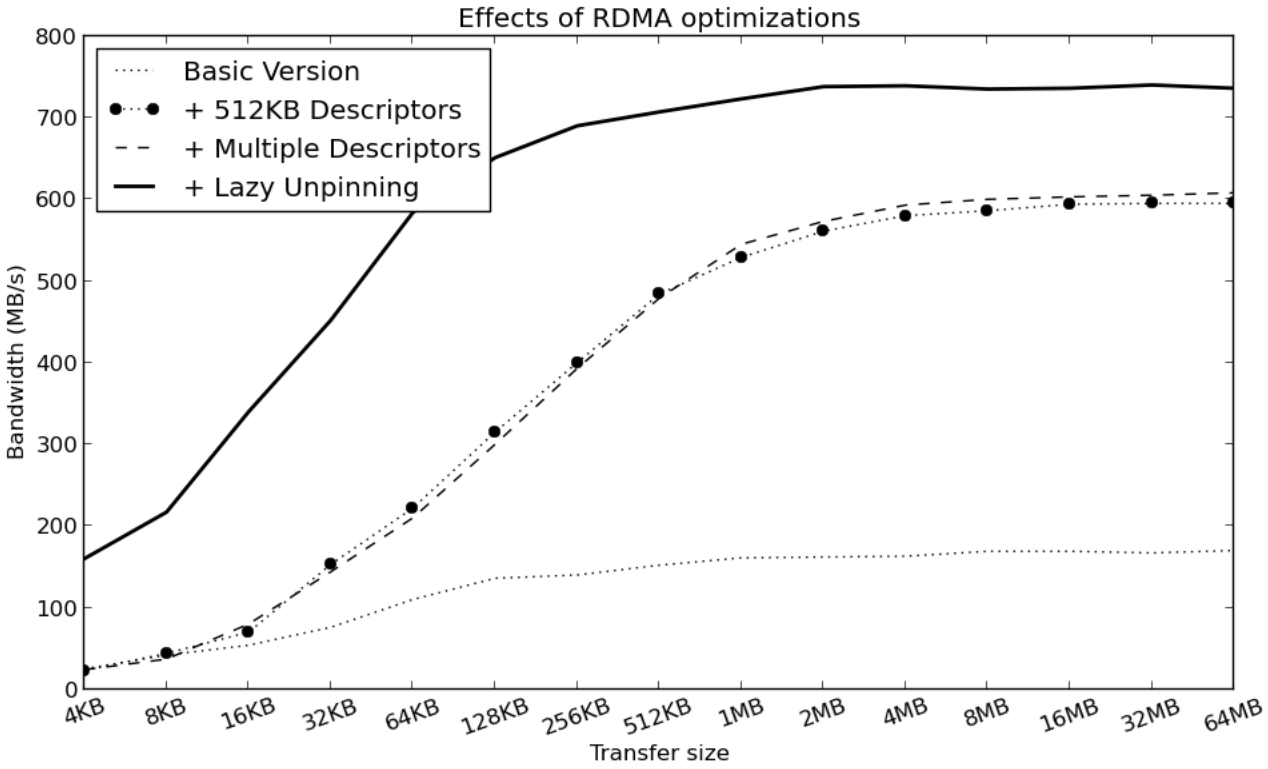


Figure 7.1.: Comparison of the optimizations described in section 5.3.2. Each optimization is added to the previous version (i.e. the Lazy Unpinning benchmark also includes the Multiple Descriptors and 512KB Descriptors optimizations). The Lazy Unpinning benchmark does not include the first transfer, which performs the actual memory pinning.

7.3. Parameter Choice for Concurrent Indirect Transfer

As already mentioned in section 6, the choice of the block size parameter is important for the concurrent indirect transfer method. Different values have been evaluated. Figure 7.2 presents the results.

The preliminary idea that too small and too large block sizes are detrimental can be confirmed. However, the extent partly depends on the transfer size and direction. For the direction from the GPU to the FPGA, a block size of one fourth of the transfer size seems to be an overall good choice. The opposite direction is not as clear. Large transfers seem to benefit from smaller block sizes of ca. one eighth of the overall size, whereas for small transfers the synchronization costs are rather large and a blocks of one half of the overall size should be chosen.

7.4. Method Comparison

Figure 7.3 presents a comparison of the sequential indirect, concurrent indirect and the direct transfer methods. For contrast, the CPU-FPGA bandwidth, which can be regarded as an upper limit for the GPU-FPGA bandwidth, is also shown. Again, the first transfer which includes the expensive pinning operation is not included in the graphs.

Different block size values were used for the concurrent indirect method: Transfers with sizes until including 1MB used a value of one half, from 2MB until 8MB a value of one fourth and from 16MB on a value of one eighth of the overall size.

As expected, the direct transfer method is indeed the fastest of the three methods, peaking at 740MB/s. For large transfers the concurrent indirect approach performs almost as well with a speed of up to 730MB/s. This results in a speed-up of ca 30% and 28% compared with the indirect transfers. For the direction from the GPU to the FPGA, for which the direct transfer could not be enabled, this is even the best solution, with a bandwidth of ca 525MB/s and a speed-up of ca 39%.

The bandwidth of the direct method deteriorates after the 192MB transfer size mark. This is due to the limitation of the graphics card not being able to pin more than ca. 200MB at a time. For larger transfers, a memory region has to be unpinned first which costs a significant amount of time.

The 512MB transfer for the concurrent indirect method failed to succeed. Presumably, the pinned CPU memory buffer (mentioned in section 6) uses up space not only on the CPU but also on the GPU. As a consequence this exceeds the memory limit of the Quadro K600 of 1GB.

7.5. Comparison with Previous Work

The following table compares the relevant features of the approaches from previous research, presented in section 3 and the two approaches from this thesis, direct and concurrent indirect (the two columns on the right). Note that the bandwidths cannot be simply compared due to different devices.

	Previous Work			This Thesis	
	Bittner & Ruf	FPGA^2	Susanto	Direct	Concurrent
Operating System	Windows	Linux	Linux	Linux	
DMA Master	GPU	FPGA	FPGA	FPGA	Both
Processes	1	1	2	1	
FPGA Vendor	Xilinx	Xilinx	Altera	Altera	
FPGA Model	Virtex 6	Virtex 5	Stratix V	Stratix V	
FPGA IP Stack	Custom	Custom	Vendor	Vendor	
FPGA Driver	Custom	Custom	Modified	Modified	Original
FPGA Programming	HDL	HDL	OpenCL	OpenCL	
GPU Vendor	NVIDIA	NVIDIA	NVIDIA	NVIDIA	
GPU Model	GeForce GTX580	GeForce 8400GS	GeForce GTX660Ti	Quadro K600	
GPU Driver	Original	Nouveau	Original	Modified	Original
GPU Programming	CUDA	gdev	CUDA	OpenCL	
Effective PCIe lanes	8	1	8	8	
Effective PCIe generation	1.0	1.1	not specified	2.0	
Maximal bandwidth FPGA to GPU	514MB/s	203MB/s	680MB/s	740MB/s	730MB/s
Maximal bandwidth GPU to FPGA	1.6GB/s	189MB/s	540MB/s	N/A	525MB/s

The approaches that utilize the FPGA as the DMA master perform better for the direction from the FPGA to the GPU. The opposite is true if the GPU is used as the DMA master. This may have to do with the PCIe protocol: an extra PCIe packet is required to read from another device, whereas writing to the other device requires only one packet.

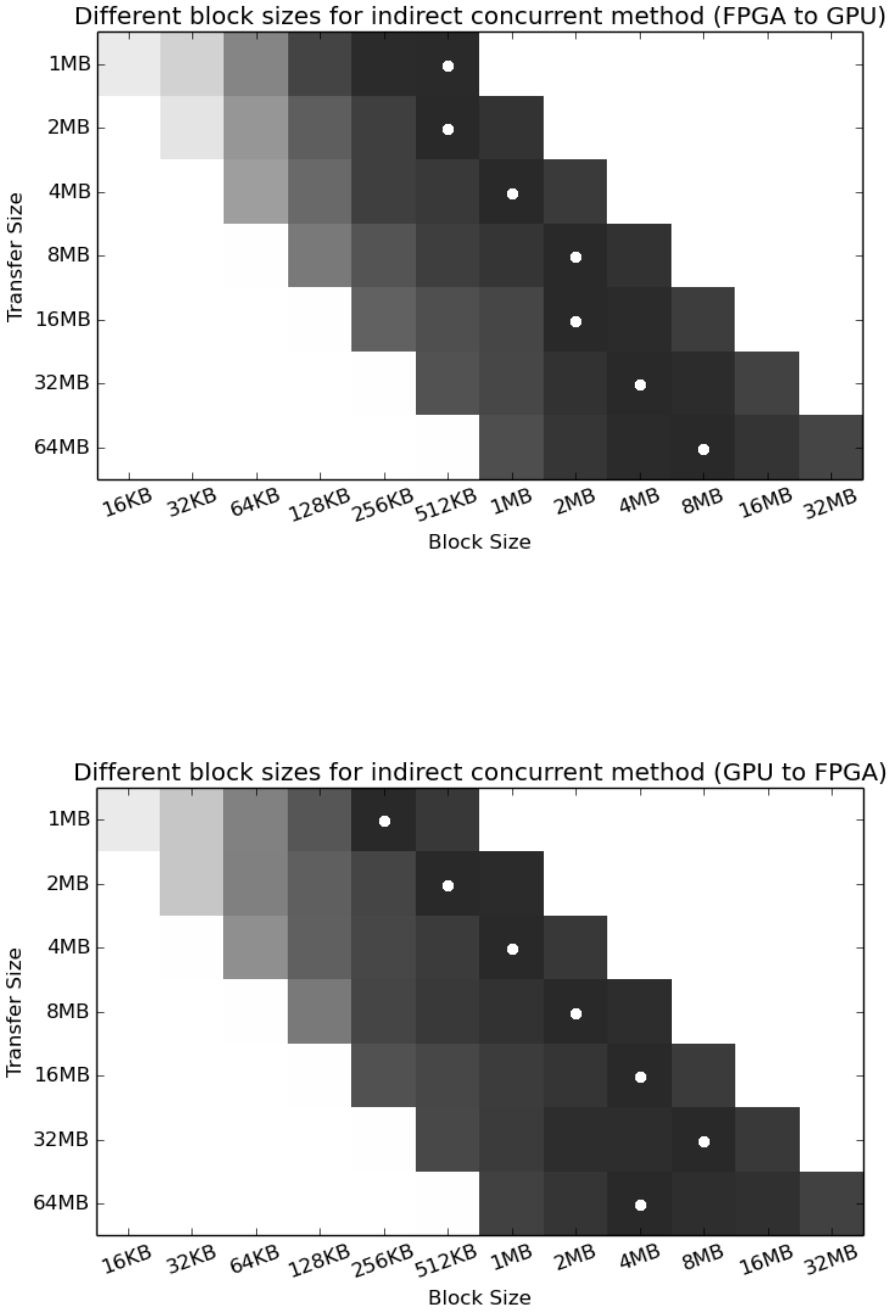


Figure 7.2.: Influence of the block size parameter on the performance for the concurrent indirect transfer method. Higher bandwidth is darker. The white dots mark the maximum bandwidth for each transfer.

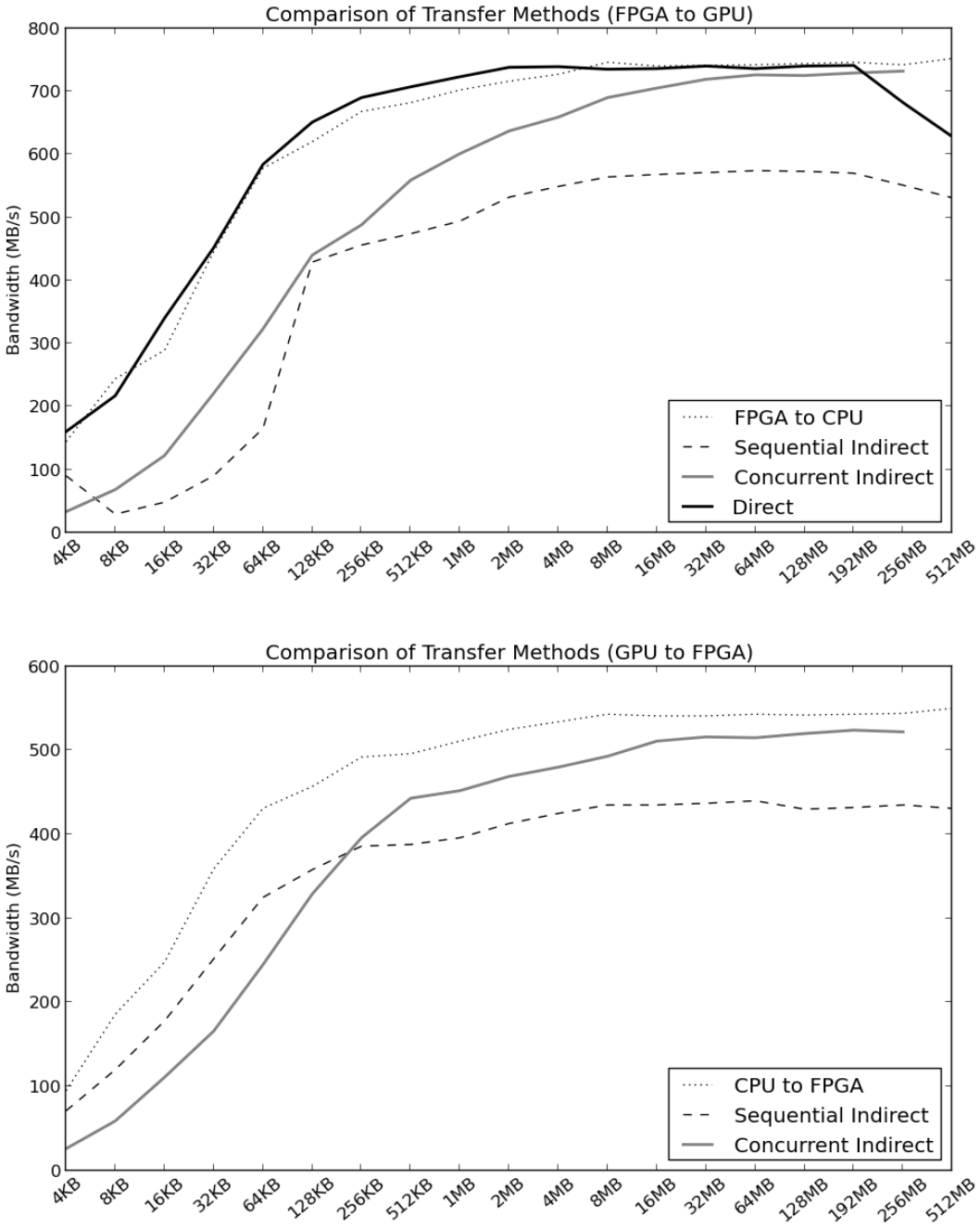


Figure 7.3.: Comparison of sequential indirect, concurrent indirect and direct transfer methods.

8. Conclusions and Future Work

This thesis shows an example implementation of direct communication between a GPU and a FPGA over PCIe for use in OpenCL. Several major problems had to be solved, e.g. the lack of official support of the GPUDirect RDMA technology for OpenCL and the lack of an ICD from Altera. Unfortunately, only the transfer direction from FPGA to GPU could be realized. This direction showed significant performance improvement compared to the indirect method with a round-trip via CPU. Moreover, an alternative to the direct method, that is simple to implement yet yields decent results and works for both directions, has been evaluated.

Several enhancements, that either could not be finished in time or go beyond the scope of this thesis, are still imaginable:

- The largest drawback of the implemented method is that the GPU-to-FPGA direction could not be enabled. Further investigation is required to find out the reasons why this direction fails. A possible approach may include a thorough analysis of the OpenCL Verilog code provided by Altera.
- The Installable Client Driver that was implemented for Altera OpenCL during the practical part of this thesis is still incomplete as it only wraps the most basic OpenCL functions. But also those that were implemented lack the support for OpenCL events, which can be useful for asynchronous transfers. Supporting events would require a rather complicated memory management system that keeps track of the event objects' lifetimes. The development of a full ICD is more time consuming than could be achieved during this thesis. However, Altera announced a full ICD implementation in the upcoming SDK version 15.0.
- The GPUDirect RDMA technology that was used as the basis for direct GPU-FPGA communication is only available for the NVIDIA Quadro and Tesla graphics cards. A possible alternative is to utilize a similar approach as Bittner and Ruf, which is described in section 3.1, however without their Speedy PCIe IP core. In theory, its functionality can be provided by the Altera's PCIe driver and PCIe core. In contrast to this thesis which utilizes the DMA controller on the FPGA, this alternative would employ the GPU's DMA controller. Not only may this enable the direct communication also for the popular GeForce cards but also the still missing GPU-to-FPGA direction.
- It may be interesting see to which extent the higher bandwidth will speed up an existing application that uses a heterogeneous computing system employing GPUs and FPGAs. A candidate could be the lane detection algorithm for automated driving, used at the TU Munich [25] or some of the examples provided in [19].

Appendix

A. Example RDMA application

```
#include <iostream>
#include <CL/opencl.h>
#include "cl_rdma.c"

using namespace std;

//transfer size 32 MB
#define N (1024 * 1024 * 32)
#define ALIGN_TO 4096

int main(int argc, char* argv[])
{
cl_uint n_platforms;
cl_context contexts[n_platforms];
cl_device_id devices[n_platforms];
cl_platform_id platformID[n_platforms];
cl_mem buffers[n_platforms];
cl_command_queue queues[n_platforms];
int altera_idx=-1;
int nvidia_idx=-1;

clGetPlatformIDs(0, NULL, &n_platforms);
if(n_platforms < 2)
{ cout<<"Found only "<<n_platforms<<" OpenCL platforms"<<endl; return(1); }

clGetPlatformIDs(n_platforms, platformID, NULL);
for(unsigned i = 0; i<n_platforms; i++)
{
      char chbuffer[1024];
      clGetPlatformInfo(platformID[i],CL_PLATFORM_NAME,1024,chbuffer,NULL);
      if(strstr(chbuffer,"NVIDIA") != NULL){nvidia_idx = i;}
      if(strstr(chbuffer,"Altera") != NULL){altera_idx = i;}

      clGetDeviceIDs(platformID[i],CL_DEVICE_TYPE_DEFAULT,1,&devices[i],NULL);
      clGetDeviceInfo(devices[i],CL_DEVICE_NAME,sizeof(chbuffer),chbuffer,NULL);
      contexts[i]=clCreateContext(0,1,&devices[i],NULL,NULL,NULL);
      queues[i]=clCreateCommandQueue(contexts[i],devices[i],0,NULL);
}
if(nvidia_idx==-1)
{cout<<"NVIDIA platform not available!"<<endl;return(1);}
if(altera_idx==-1)
{cout<<"Altera platform not available!"<<endl;return(1);}
```

```
//initialize RDMA
int rc=clrdma_init_altera(contexts[altera_idx], devices[altera_idx]);
if (rc==1){cout<<"Altera Driver is not loaded!"<<endl; return(1);}
else if(rc==2){cout<<"Wrong Altera Driver is loaded!"<<endl; return(1);}
else if(rc==4){cout<<"Could not load get_address.aocx!"<<endl; return(1);}
else if(rc){cout<<"Failed to initialize rdma for Altera."<<endl; return(1);}

rc=clrdma_init_nvidia(contexts[nvidia_idx]);
if (rc==1){cout<<"Nvidia Driver is not loaded!"<<endl; return(1);}
else if(rc){cout<<"Failed to initialize rdma for NVIDIA."<<endl;return(1);}

rc=clrdma_create_pinnable_buffer_nvidia(contexts[nvidia_idx],
                                        queues[nvidia_idx],
                                       &buffers[nvidia_idx], N);
if(rc){cout<<"Failed to create a pinnable buffer!"<<endl; return(1);}
buffers[altera_idx]
  =clCreateBuffer(contexts[altera_idx],CL_MEM_READ_WRITE,N,NULL,NULL);

unsigned long addresses[2];
rc=clrdma_get_buffer_address_altera(buffers[altera_idx],
                                    queues[altera_idx],
                                   &addresses[altera_idx]);
if(rc){cout<<"Could not get address for FPGA buffer."<<endl;return(1);}
rc=clrdma_get_buffer_address_nvidia(buffers[nvidia_idx],
                                    queues[nvidia_idx],
                                   &addresses[nvidia_idx]);
if(rc){cout<<"Could not get address for GPU buffer."<<endl;return(1);}

unsigned char* data0; unsigned char* data1;
if(posix_memalign((void**)&data0,ALIGN_TO,N))
{ cout<<"Could not allocate aligned memory!"<<endl; return(1); }
if(posix_memalign((void**)&data1,ALIGN_TO,N))
{ cout<<"Could not allocate aligned memory!"<<endl; return(1); }
for(unsigned i=0;i<N;i++){data0[i] = i%256; }
clEnqueueWriteBuffer(queues[altera_idx],buffers[altera_idx],
                     CL_TRUE,0,N,data0,0,NULL,NULL);

//RDMA transfer from FPGA to GPU of size N (timeout after 5 seconds)
rc = read_rdma(addresses[altera_idx], addresses[nvidia_idx], N, 5);
if(rc){cout<<"RDMA transfer failed. Code:"<<rc<<endl;return(1);}

clEnqueueReadBuffer (queues[1],buffers[1],CL_TRUE,0,N,data1,0,NULL,NULL);
//check whether the data is correct
for(unsigned i=0;i<N;i++)
{
      if(data0[i]!=data1[i])
      {cout<<"Transferred data is incorrect!"<<endl;break;}
}

free(data0); free(data1);
return(0);
}
```

B. Setup Instructions

This section provides an overview of the procedures needed to run the benchmarks for this thesis. The Ubuntu 14.04 OS is assumed.

The following command installs the Altera ICD on the system. Note that `<PATH_TO>` should be replaced with the actual path to the dynamic library.

```
echo "<PATH_TO>/libalteraicd.so" >> /etc/OpenCL/vendors/alteraicd
```

The modified Altera PCIe module can be compiled with the following script:

```
./make_all.sh
```

The modified NVIDIA module can be compiled with the following command:

```
make module
```

To load the modified modules, the original modules have to be removed first. Since the display manager depends on the NVIDIA module it must be stopped too. To do this, the TTY has to be switched, e.g. with the key combination `CTRL+ALT+F4`. Then, the following command sequence will replace the modules:

```
sudo service lightdm stop
sudo rmmod nvidia-uvm aclpci_drv nvidia
sudo insmod nvidia.ko
sudo insmod aclpci_drv.ko
sudo service lightdm start
```

To load the modules automatically during the boot procedure, they previous modules should be overwritten:

```
sudo cp aclpci_drv.ko /lib/modules/$(uname -r)/kernel/drivers/
sudo cp nvidia.ko /lib/modules/$(uname -r)/kernel/drivers/
```

For testing, the code from appendix A of the provided benchmark `direct_benchmark` can be used.

Bibliography

[1] Altera Corporation. FPGA Architecture - White Paper. https://www.altera.com/en_US/pdfs/literature/wp/wp-01003.pdf, 2006. Retrieved on 2015.03.13.

[2] Altera Corporation. Embedded Peripherals IP User Guide. https://www.altera.com/en_US/pdfs/literature/ug/ug_embedded_ip.pdf, 2011. Retrieved on 2015.03.24.

[3] Altera Corporation. Altera SDK for OpenCL (version 13.1). http://dl.altera.com/opencl/, 2013. Retrieved on 2015.03.28.

[4] Altera Corporation. Implementing FPGA Design with the OpenCL Standard. http://www.altera.com/literature/wp/wp-01173-opencl.pdf, 2013. Retrieved on 2015.03.13.

[5] Altera Corporation. Embedded Peripheral IP User Guide, 2014. Chapter 22: Altera Modular Scatter-Gather DMA.

[6] Altera Corporation. Stratix V Avalon-MM Interface for PCIe Solutions User Guide, 2014.

[7] Altera Corporation. Avalon Interface Specifications. https://www.altera.com/en_US/pdfs/literature/manual/mnl_avalon_spec.pdf, 2015. Retrieved on 2015.04.05.

[8] Ray Bittner. Speedy Bus Mastering PCI Express. *22nd International Conference on Field Programmable Logic and Applications (FPL 2012)*, August 2012.

[9] Ray Bittner and Erik Ruf. Direct GPU/FPGA Communication Via PCI Express. *1st International Workshop on Unconventional Cluster Architectures and Applications (UCAA 2012)*, September 2012.

[10] Nick Black. Libcudest. http://nick-black.com/dankwiki/index.php/Libcudest. Retrieved on 2015.03.18.

[11] Ravi Budruk. *PCI express system architecture*. Addison-Wesley, 2003.

[12] NVIDIA Corporation. NVIDIA GPUDirect. https://developer.nvidia.com/gpudirect, 2010. Retrieved on 2015.04.02.

[13] NVIDIA Corporation. NVIDIA's Next Generation CUDA(TM) Compute Architecture: Kepler(TM) GK110. http://www.nvidia.com/content/PDF/kepler/NVIDIA-Kepler-GK110-Architecture-Whitepaper.pdf, 2012. Retrieved on 2015.02.25.

[14] NVIDIA Corporation. Developing a Linux Kernel Module using GPUDirect RDMA. http://docs.nvidia.com/cuda/gpudirect-rdma/index.html, 2014. Retrieved on 2015.03.11.

[15] NVIDIA Corporation. CUDA C Programming Guide [version 7.0]. http://docs.nvidia.com/cuda/pdf/CUDA_C_Programming_Guide.pdf, 2015. Retrieved on 2015.04.03.

[16] Umer Farooq, Zied Marrakchi, and Habib Mehrez. *Tree-based Heterogeneous FPGA Architectures - Application Specific Exploration and Optimization*. Springer, Berlin, Heidelberg, 2012.

[17] Scott Hauck. *Reconfigurable computing the theory and practice of FPGA-based computation*. Morgan Kaufmann, Amsterdam Boston, 2008.

[18] Hewlett-Packard. Hewlett-Packard Quadro K600 Specifications. http://www8.hp.com/h20195/v2/GetDocument.aspx?docname=c04128134, 2013. Retrieved on 2015.04.07.

[19] Ra Inta, David J. Bowman, and Susan M. Scott. The "Chimera": An Off-The-Shelf CPU/GPGPU/FPGA Hybrid Computing Platform, 2012. International Journal of Reconfigurable Computing, Volume 2012.

[20] Khronos Group. OpenCL Extension #5: Installable Client Driver (ICD) Loader. https://www.khronos.org/registry/cl/extensions/khr/cl_khr_icd.txt, 2010. Retrieved on 2015.03.15.

[21] Khronos Group. OpenCL 1.2 Installable Client Driver Loader [Source]. https://www.khronos.org/registry/cl/specs/opencl-icd-1.2.11.0.tgz, 2012. Retrieved on 2015.04.01.

[22] Khronos Group. The OpenCL Specification Version 1.2. https://www.khronos.org/registry/cl/specs/opencl-1.2.pdf, 2012. Retrieved on 2015.03.15.

[23] Linux man-pages project. dlopen(3) Linux man page, 2000.

[24] Linux man-pages project. ioctl(2) Linux man page, 2000.

[25] Nikhil Madduri. Hardware Accelerated Particle Filter for Lane Detection and Tracking in OpenCL. Master's thesis, Technische Universität München, 2014.

[26] Nallatech Inc. Nallatech 385-D5 Specifications. http://www.nallatech.com/wp-content/uploads/pcie_385pb_v1_21.pdf, 2014. Retrieved on 2015.04.07.

[27] NVIDIA Corporation. NVIDIA OpenCL Best Practices Guide. http://www.nvidia.com/content/cudazone/CUDABrowser/downloads/papers/NVIDIA_OpenCL_BestPracticesGuide.pdf, 2009. Retrieved on 2015.03.21.

[28] NVIDIA Corporation. LINUX X64 (AMD64/EM64T) DISPLAY DRIVER (Version 346.47) [kernel/nv.c]. http://us.download.nvidia.com/XFree86/Linux-x86_64/346.47/NVIDIA-Linux-x86_64-346.47.run, 2015. Retrieved on 2015.03.19.

[29] Alessandro Rubini, Jonathan Corbet, and Greg Kroah-Hartman. *Linux Device Drivers, 3nd Edition*. O'Reilly Media, 2005.

[30] Shinpei Kato and Michael McThrow and Carlos Maltzahn and Scott Brandt. Gdev: First-Class GPU Resource Management in the Operating System. Tech. Rep., 2012, uSENIX Annual Technical Conference (USENIX ATC'12).

[31] David Susanto. Parallelism for Computationally Intensive Algorithm with GPU/FPGA Interleaving. Master's thesis, Technische Universität München, 2014.

[32] Andrew S Tanenbaum. *Modern Operating Systems*. Pearson, 2 edition, 2007.

[33] Yann Thoma, Alberto Dassatti, and Daniel Molla. FPGA^2: An open source framework for FPGA-GPU PCIe communication. In *2012 International Conference on Reconfigurable Computing and FPGAs, ReConFig 2013, Cancun, Mexico, December 9-11, 2013*, pages 1–6, 2013.

[34] Various Authors. Nouveau: Accelerated Open Source driver for nVidia cards. http://nouveau.freedesktop.org/wiki/, 2012. Retrieved on 2015.04.10.

[35] Various Authors. Nouveau: Feature Matrix. http://nouveau.freedesktop.org/wiki/FeatureMatrix/, 2014. Retrieved on 2015.04.11.

[36] Nicholas Wilt. *The CUDA handbook a comprehensive guide to GPU programming*. Addison-Wesley, Upper Saddle River, NJ, 2013.