Optimizing FPGA-GPU data transfers

Maximiliano Garrone ten Brink

August 18, 2017

MSc in High Performance Computing

The University of Edinburgh

Year of Presentation: 2017

Abstract

GPUs and FPGAs are two types of accelerators which can be combined in a system to exploit the strengths of both. A common problem of such hybrid solutions comes from the transfer of data from one device to the other. Using host memory as an intermediary is simple, but inefficient in terms of bandwidth and latency. One alternative is to use direct memory access (DMA) from one device to the other.

In this dissertation, two possible DMA implementations are described and analysed. One leverages the DMA engines already present in GPUs. It was found to work on Windows but not on Linux, and did not provide any significant performance improvement. The other has the FPGA directly access GPU memory through Nvidia's GPUDirect RDMA API, which is only available on Linux for select GPU brands. Transfers from the FPGA to the GPU achieved a 30% increase in bandwidth, but in the opposite direction, some kind of throttling was encountered that limited throughput for transfers larger than 512 KiB. The inefficiency of the latter can be partially explained by inherent traits of PCIe read operations, but why this increases with larger transfer sizes requires further research. Performance of reads in both implementations was found to vary significantly depending on which PCIe slots the accelerators were connected to, the cause of which was also not found.

While in some cases DMA provided some performance improvements, it comes at the cost of additional complexity that should be considered before implementation in production systems. A simpler solution based on concurrent indirect transfers might be able to produce similar results.

Contents

List of Tables

List of Figures

List of Algorithms

List of Code Listings

Glossary

API	Application Programming Interface
BAR	Base Address Register, a PCIe mechanism to assign bus addresses to device memory
CplD	Completion with Data, a type of PCIe TLP
CPU	Central Processing Unit
CUDA	Compute Unified Device Architecture, a parallel programming framework by Nvidia
DMA	Direct Memory Access, a technique to accelerate data transfers
DMA engine	A part of a device that can initiate PCIe bus transactions
FPGA	Field-Programmable Gate Array, a kind of accelerator
Gen*N*	PCIe generation *N*
GPGPU	General-purpose computing on graphics processing units
GPU	Graphics Processing Unit, a kind of accelerator
HDL	Hardware Description Language, used to describe FPGA designs
Host	The computer to which an accelerator is attached
IP Core	Intellectual Property core, a reusable component for FPGA designs
Lane	In the context of PCIe, a pair of signals, one in each direction, that can form a link individually or through aggregation with other lanes
Link	In the context of PCIe, a bidirectional connection between two devices, composed of one or more lanes
MMIO	Memory-mapped I/O, a technique to access device memory as regular host memory
MRd	Memory Read request, a type of PCIe TLP
MWr	Memory Write request, a type of PCIe TLP
PCIe	Peripheral Component Interconnect Express, the typical interconnect used by GPUs
Pin	To establish a semi-permanent mapping between memory address spaces
Register	In the context of CUDA, to instruct the CUDA runtime that a memory region is safe for DMA
TLP	Transaction-Level Packet, the unit of data transferred between two PCIe devices
Verilog	A hardware description language
WinDriver	A third-party driver development kit for Windows

Acknowledgements

I would like to thank Alistair Gorman, my supervisor from Optos plc, for his guidance in selecting a research topic and providing all the necessary hardware and software licences.

Thanks are also due to Mark Tucker, my supervisor from the University of Edinburgh's EPCC, for his invaluable feedback and help navigating the waters of academic writing.

Chapter 1

Introduction

Accelerators can be a faster and more efficient alternative than traditional CPU architectures for some computational problems. However, the additional performance usually comes at the cost of increased hardware and software complexity and loss of portability. Most importantly, accelerator are specialised devices, well-suited for some kinds of problems and not at all for others.

Graphics Processing Units (GPUs), originally intended for real-time 3D graphics processing, have become common accelerators in scientific computation. The same features that allow them to outperform CPUs for graphics, namely high levels of parallelism and high memory bandwidth, can be exploited by other kinds of programs such as simulations, image processing and neural network training and inference.

Field-Programmable Gate Arrays (FPGAs) are a different kind of accelerator which are programmed at a lower level of abstraction than CPUs or GPUs. By customising the behaviour of individual logic gates, hardware-level parallelism for operations such as integer logic and bit manipulation can be expressed more naturally than through a processor's instruction set. In addition, FPGAs offer flexible I/O options, allowing them to interface with non-standard devices. They are, however, not efficient for floating-point arithmetic.

Given the complementary strengths and weaknesses of the two mentioned accelerator types, some workloads have requirements mixed in such a way that part of them could benefit from acceleration on GPUs, and a different part on FPGAs. Doing so, however, not only further increases overall system complexity and required skillset to develop it, but it also exacerbates on one of the main weaknesses of accelerator architectures: transferring data to and from the accelerators. In a single-accelerator architecture, input data is sent from the host to the accelerator and output data in the opposite direction. If these transfers take a long time compared to their processing on the accelerator, they may offset the performance benefits of the accelerator architecture.

When adding a second type of accelerator to the system, some data generated on one accelerator may need to be processed by the other, without any intermediate step required on the host. One possible solution to these device-to-device transfers is to conceptually divide the system into two halves (e.g. host-FPGA and host-GPU), and use whatever method is available to transfer data between the host and each of the accelerators. In this way, if data needs to get from one accelerator to the other, it will go through the host. Since this re-uses existing transfer methods, it is relatively simple to implement, but it most likely does not achieve optimal transfer efficiency.

While the inefficiency of such indirect transfers could be ignored if the processing of the data takes much longer than its transfer, or masked by executing many transfers in parallel such as is commonly done for host-to-GPU transfers, in some cases a more efficient alternative is required. Some such cases were proposed by Optos, an industrial collaborator for this project.

Optos is a medical technology company that produces retinal imaging devices and their associated software. Retinal imaging is a non-invasive process in which multiple pictures are taken of the inside of a patient's eye, then combined into one single picture through an algorithm called image registration. The resulting picture allows medical professionals to diagnose and control diseases, both of the eye itself as well as of the body as a whole.

To generate these pictures, Optos uses a hybrid accelerator architecture as described above. The output from cameras and various sensors are aggregated and pre-processed by an FPGA, and the resulting image data is sent to a GPU that executes the image registration algorithm. This process allows Optos to combine the I/O flexibility and low-latency integer processing offered by FPGAs with the high floating-point throughput of GPUs. While so far, the transfer of data from the FPGA to the GPU has not been a significant issue, projected increases in image resolution could saturate the bandwidth of existing applications. Furthermore, lower processing latency could allow more accurate eye tracking, leading to faster image acquisition.

The goal of this project is to research, implement and evaluate two possible solutions to transfer data directly between FPGAs and GPUs, without going through the host. The evaluation is focused on the potential improvements to throughput and latency. One of the solutions is based on Bitner and Ruf's work [1], and the other one on Nvidia's API for Direct Memory Access [2].

1.1 Dissertation structure

This dissertation consists of six chapters, with this first one serving as introduction. Chapter 2 describes the two accelerator architectures, the typical interconnect between them, and how direct memory access can be implemented to improve data transfer performance. It also summarises previous work done on this topic.

Since each of the two solutions analysed in this dissertation is available on different operating systems and for different hardware, it is likely that the choice between them will be based more on availability than on technical merit. For this reason, they are covered separately on different chapters. Chapter 3 covers all software that was developed for the project and used for the implementation of both solutions, as well as a description of the test hardware, which was the same in both cases. Chapter 4 describes the solution implemented for Windows, the principles behind it and the obtained results. Chapter 5 does the same for the Linux solution.

Chapter 6 acts as the project's conclusion, summarising the work presented and the results obtained. It also describes possible avenues for future research on the same topic.

Finally, Appendix A gives instructions to build and run the code created for this project, and Appendix B offers a review of the project.

Chapter 2

Background Theory

This chapter briefly discusses the fundamentals of the technologies and concepts that were applied in this project. Sections 2.1 and 2.2 describe the two relevant accelerator architectures, GPUs and FPGAs, respectively. Section 2.3 gives an overview of the typical interconnect between those accelerators and their host, Section 2.4 describes a common technique to improve the performance of transfers through that interconnect, and Section 2.5 lists two ways in which that technique can be applied specifically to GPUs and FPGAs. Sections 2.6 and 2.7 cover the multiple memory address spaces present in modern computer architectures and how devices interact with operating systems, both topics which were fundamental to the development of the software presented in this dissertation. Finally, Section 2.8 discusses previous work done on the topic of FPGA-GPU data transfers.

2.1 Graphics Processing Units

Traditional CPUs are general purpose devices, designed to handle many different use cases with reasonable efficiency, but not necessarily to excel at all of them. Thus, a processor tailored for a specific use case can be more efficient than a traditional CPU for that use case, at the cost of not handling others as well.

Graphics Processing Units are an example of this. As the name implies, they were originally designed to process 3D computer graphics in real time. This use case is peculiar because of its high level of data parallelism: the same relatively simple algorithms heavy on floating-point operations are applied independently to millions of vertices and pixels, multiple times per second. Under such circumstances, some of the resources of CPUs, such as integer units, caches and branch predictors, become unnecessary. GPUs thus avoid dedicating transistors to such resources, and use them instead to add cores to allow higher levels of parallelism. GPU cores are then simpler, and thus more power efficient, than CPU cores for their intended purpose.

The increased compute capability of GPUs leads to data being consumed and generated at higher rates. To cope with this, GPUs are typically equipped with higher-performance memory, such as GDDR5 and HBM, which offer higher bandwidth than memory technologies used by traditional CPUs, such as DDR3 or DDR4.

Despite their increased floating-point compute performance and memory bandwidth, GPUs are not well suited for general purpose computing. They are not capable of hosting an operating system, and their high degree of specialization comes at the expense of performance on other use cases.

For this reason, they are used as *accelerators*. The operating system and user processes run on a CPU, called the *host*, but can offload their graphics processing to an external GPU. To achieve this, there must be an interconnect between the host and the GPU, so that the necessary resources can be moved from one to the other so they can be processed. Although Nvidia introduced a custom interconnect for their high-end compute cards, the traditional one is PCI Express [3].

Although GPUs were originally designed for graphics, other workloads such as scientific simulations and image processing have similar performance profiles, and thus can benefit from being run on GPUs. This is known as General Purpose GPU (GPGPU) computing, and led to the development of GPGPU-specific languages such as CUDA and OpenCL. These languages attempt to simplify the programming of GPUs by making the process and the resulting code similar to programming a CPU. However, achieving good performance is not trivial, and requires that the programmer avoids many possible pitfalls, as described in the CUDA C Best Practices Guide [4].

One such performance pitfall is the transfer of data to and from the GPU. In graphics processing, assets such as textures and meshes tend to be reused for multiple consecutive frames, so the cost of transferring them to the GPU can be amortized. In GPGPU computing, this is not necessarily the case, and these transfers through the relatively slow interconnect can offset the performance gains obtained by offloading the computation to the GPU.

2.2 Field-Programmable Gate Arrays

As described in Section 2.1, special-purpose processors can be more efficient than a traditional CPU for a given use case. If there is no commercial processor specialized for that use case, it is possible to design an *Application-Specific Integrated Circuit* (ASIC) for it. However, circuit design and fabrication is a complex, costly and lengthy endeavour. If the demand for such circuits is not large enough to benefit from economies of scale, ASICs can be prohibitively expensive.

An intermediate option is to employ *Field-Programmable Gate Arrays* (FPGAs). These devices can achieve many of the performance and efficiency benefits of ASICs, but at a lower design and implementation cost for small volumes. At a high level, an FPGA is a digital circuit whose logic gates can be arbitrarily reconfigured any number of times to implement different logical circuits. In this way, the same physical chip can behave as a given circuit at one point in time, then be reconfigured to a completely different one. [5]

The main building block of an FPGA is the *logic cell*, consisting of a lookup table, a flip-flop and I/O wires. Both the output of the lookup table and whether the flip-flop is used in each logic cell are configured by volatile memory internal to the logic cell. Logic cells are arranged in a mesh, and connected to each other via an interconnect formed by wires and multiplexers. By configuring multiple interconnected logic cells, complex digital circuits can be implemented, limited only by the number of logic cells in the FPGA and the connections between them. [6]

To connect the FPGA to other devices, I/O wires are placed at the boundaries of the mesh and connected to the same interconnect as the logic gates. These I/O wires can interface with many different types of devices, such as external memory (volatile or otherwise), buses or LEDs.

A properly designed circuit implemented on an FPGA can achieve higher levels of parallelism than a CPU because it is not bound by the abstraction presented by the latter, where a list of instructions is executed serially. Instead, each part of the circuit is running concurrently. Furthermore, the possibility of configuring the I/O wires gives FPGAs great flexibility to interface

with different devices and leading to them being used as adaptors between non-standard devices. [5]

The downside is that since each conceptual logic gate is implemented in an FPGA in terms of one or more relatively complex logic cells consisting of several transistors, an FPGA is significantly less efficient that implementing the same circuit in dedicated hardware. Moreover, the massive levels of parallelism in an FPGA limit the attainable clock speeds. All of this means that if dedicated hardware exists to carry out a given task, it will most likely be more efficient than an FPGA for that task. A relevant example of this are floating-point-intensive algorithms, which per se are an inherent weakness of FPGAs, and thus more efficiently executed on GPUs, which were designed for that purpose. [7]

When an FPGA is chosen as the tool to implement a given algorithm, a conceptual digital circuit needs to be designed. This is typically done in a *Hardware Description Language* (HDL), such as Verilog or VHDL. An HDL syntactically resembles a programming language like C, but its usage is significantly different because of the massively parallel execution that happens in an FPGA [8]. HDL code is more about defining the inputs and outputs of component circuits than about executing serial instructions. They are thus limited in the data types and control structures they offer.

After describing a circuit in an HDL, it can by *synthesized* by a tool, translating it to the configuration that logic cells in the FPGA should take to implement it. A synthesized design then needs to be *placed and routed* to a specific FPGA model, mapping it to the physical cells and I/O wires. This generates a binary file that can be *downloaded* to that FPGA model to implement the circuit.

Analogously to the libraries provided for programming languages, *Intellectual Property (IP) cores* are reusable components that can be bundled for distribution, then added to an FPGA design. A *soft* IP core is one that is distributed as synthesizable HDL code, while a *hard* one is distributed as an unmodifiable binary file. FPGA vendors such as Altera [9] and Xilinx [10] provide IP cores to access their devices' functionality, such as memory controllers, bus interfaces and efficient custom implementations of algorithms like FFTs.

2.3 PCI Express Fundamentals

As the interconnect between the major components of the computer architecture under study, the Peripheral Component Interface Express (PCIe) is one of the main factors that determine the performance of transfers between these components. This subsection, based on *PCI Express Technology* [11], covers the fundamentals of this interconnect, which will serve as the basis for the implementation and results discussion in the following chapters.

Unlike its predecessors PCI and PCI-X, PCIe is a serial bus with a packet-based protocol and a tree topology. The root of the tree, called the *root complex*, is typically the interface between the CPU and the bus. The leaves are called *endpoints*, and are the other devices connected to the bus, such as GPUs and FPGAs. Intermediate nodes in the topology are switches and bridges, but they are not always present.

The physical connection between two PCIe devices is a full-duplex, bidirectional *link*, composed of at least one *lane*. A lane is a pair of signals, one in either direction, that is sufficient for all communication between the two devices, meaning that there are no out-of-band communications

in PCIe. However, multiple lanes can be aggregated to form a single link with higher bandwidth, at the cost of more pins and power. The number of lanes in a link is called the *link width*, and is usually represented as the link width preceded by the letter x (e.g. x4 for a link width of 4).

The original version of the PCIe standard, called *generation 1 (Gen1)*, specifies a transfer frequency of 2.5 GT/s per lane in either direction, where 1 T/s means the transfer of one bit per second. However, 20% of this bandwidth is consumed by the physical protocol encoding, known as *8b/10b* and which uses 10 bits for every 8 bits of useful data. With 8 bits per byte, this results in a usable bandwidth of 250 MB/s in either direction per Gen1 lane.

The PCIe standard has been subject to two main revisions in which the bandwidth per lane was increased. *Gen2* simply doubles the transfer frequency to 5 GT/s, effectively doubling the bandwidth to 500 MB/s per lane. *Gen3* raises the transfer frequency again to 8 GT/s, and replaces the 8b/10b encoding with a much more efficient *128b/130b*, increasing the bandwidth per lane to almost 1GB/s.

Table 1 summarizes the maximum physical bandwidth provided by different PCIe configurations.

	x1	x2	x4	x8	x16
Gen1	0.250	0.500	1.00	2.00	4.00
Gen2	0.500	1.00	2.00	4.00	8.00
Gen3	0.985	1.97	3.94	7.88	15.8

Table 1. PCIe aggregate bandwidth in GB/s for different link widths

Due to the lack of any out-of-band channels, all control data must also be transferred through the same links. This introduces some additional overhead that limits the bandwidth usable by applications. Each transfer is executed in terms of at least one *Transaction-Level Packet (TLP)*, which carries a variable-sized header plus the data payload. There are many different types of TLPs, but the only three relevant for this report are *Memory Write Request (MWr)*, *Memory Read Request (MRd)*, and *Completion with Data (CplD)*.

A MWr request initiates a *posted transaction*. This means that the TLP carries all data necessary for the destination to carry out the request, and no completion TLP is expected.

On the other hand, a MRd request TLP specifies a range of data to read, and requires that the destination replies with a CplD TLP containing the requested data. This is known as a *non-posted transaction*, and requires the requester keeps track of all outstanding requests so they can be matched to the eventual completion. The extra TLPs and bookkeeping involved lead to non-posted transactions usually having worse performance than the equivalent posted ones.

Each node in the bus needs to specify the *maximum payload size* it can handle. This can be up to 4096 bytes, but it is typically 128 or 256 bytes. No device should issue a TLP with a larger payload than the maximum specified by the TLP's destination. However, a second per-device parameter is the *maximum read size*, which specifies the maximum amount of data that can be requested by a single MRd TLP. If this amount is larger than the destination's TLP, the destination will reply with multiple completion TLPs. This can ameliorate the performance difference between reads and writes, but only up to a point, because the maximum read size can be no larger than 4096 bytes.

These TLPs specify the origin and destination of TLPs, as well as the transaction type, memory ranges, transfer size and many other parameters in the *TLP header*. Each header takes 12 or 16 bytes, depending on whether 64-bit addresses are used or not. The shorter headers must be used if the requested address fits in 32 bits; otherwise, the longer 16-byte header must be used instead. If the longer header is used for addresses that fit in 32 bits, the result is unspecified by the standard, but most devices can process them without issues. Lower layers of the PCIe protocol add their own headers taking 8 additional bytes, for a total of 20 to 28 bytes worth of headers.

The memory addresses specified in TLPs are *bus addresses*, and need not be mapped to actual memory of any kind. Any device is free to respond to read requests to a given address with data from any source (e.g. generated on the fly, or retrieved from on-device memory), and nothing is expected after a write request (e.g. a read request following a write request need not respond with the data previously written).

Bus addresses are configured by the BIOS or OS at boot time or when a device is plugged in. Each endpoint can request up to six memory regions of 4 GiB each through one of the six available *Base Address Registers (BARs)*. The BIOS or OS then assign to the device the base bus address corresponding to each of the requested regions. If regions of more than 4 GiB are required, two contiguous BARs can be combined to form a 64-bit BAR.

2.4 Direct Memory Access

To transfer data from one device to another, one of them must issue the necessary MRd or MWr TLPs. CPUs can do this through their root complex, by what is called Memory-Mapped IO (MMIO). The target device's BARs are mapped to the CPU's physical address space, so that a read or write by the CPU to those addresses triggers the root complex to issue the corresponding TLPs. This enables a convenient programming model, because it allows programs to treat device memory as normal system memory. However, transferring large amounts of data in this manner means that the CPU is kept busy while the data moves along the relatively slower PCIe bus. [11]

Alternatively, a peripheral device can initiate the same memory transactions from the other end, by issuing the opposite TLP to bus addresses corresponding to host memory. The root complex can then read or write that memory accordingly, without involving the CPU. This principle is what is known as *Direct Memory Access (DMA)*. [11]

While the PCIe standard allows this general principle by letting any device act as *bus master* (i.e. as initiator of a PCIe transaction), there is no standard way to implement DMA. Instead, DMA is considered an umbrella term for any memory transaction not initiated by the CPU. [12]

Although the CPU does not initiate every PCIe transaction in a DMA operation, there is typically a need for software running on the CPU to specify the necessary transfers to the device. This can be accomplished by transferring a small control data structure to the device through simple MMIO. Some part of the device, the *DMA engine*, receives this data and starts issuing the corresponding TLPs. Due to this multi-step process, simple MMIO might be faster than DMA for small transfers. [12]

2.5 Direct Memory Access between FPGAs and GPUs

While there is no standard way to implement DMA, the proprietary nature and complex memory model of GPUs limit the ways in which third party devices can access their memory. Since the

typical partner for transfers to or from GPU memory is the host's root complex, which usually does not have a DMA engine [11], GPUs use their own DMA engines which are programmed by their drivers. In their paper [1], further discussed in Section 2.8, Bitner and Ruf propose a way to repurpose the GPU's DMA engine for transfers to or from an FPGA instead of the host.

A second alternative is to have the FPGA act as bus master, but to do so, it is necessary to know the mapping between the GPU's internal memory and its BARs. Nvidia provides an API called GPUDirect RDMA that allows applications to establish this mapping and keep it pinned until no longer desired [2]. With this information and an FPGA capable of bus mastering, it is straightforward to have the FPGA access the pinned memory on the GPU. However, GPUDirect RDMA is only available on Linux and for professional-grade Nvidia GPUs, sold under the Tesla and Quadro brands, limiting the availability of this option [2].

2.6 Address spaces

The memory addresses on which the PCIe bus operates are called *bus addresses*. These addresses are unique across the whole system, and are assigned through BARs at device configuration time. The meaning of the addresses assigned to each device is defined by the device itself. [11]

In a typical architecture in which the CPU is at the root complex, system memory is mapped to the bus address space so that devices can access it. The translation between the bus address space and the physical memory address space is handled by an *Input–Output Memory Management Unit (IOMMU)*. In modern x86 architectures, this translation is transparent, so that physical memory addresses are the same as bus addresses. Code that needs to know bus addresses can be simplified by making this assumption, at the cost of portability to other platforms where the translation is not straightforward. [11]

Besides these two address spaces, operating systems such as Linux or Windows introduce *virtual address spaces*. A process running on such a system gets its own private virtual memory address space, made up of a logically contiguous block of memory. However, this memory is not physically contiguous. Instead, the memory is split into *pages*, and each logical page can be placed on any position in physical memory. When the process tries to access any virtual address, the OS and hardware collaborate to translate the physical address into the corresponding physical one. [13]

The use of virtual memory has two main benefits. First, it adds a layer of security by preventing a process from accessing another process' memory. Furthermore, by having the OS kernel operate in its own virtual address space, all processes are also impeded from accessing critical system data structures. [13]

The second advantage of virtual memory is that it allows the OS to dynamically *swap* pages that are not being actively used out to disk, allowing the physical memory region to be used to hold a different page. When the page that was swapped out is referenced, the OS can swap it back into main memory, perhaps at a different physical location. [13]

Swapping effectively boosts the amount of available physical memory, but it makes it difficult for DMA operations to be carried out. Since the physical address (and the corresponding bus address) associated with a given virtual address might change at any point, or might not exist at all if the OS swaps the page out, a process cannot easily instruct a device to directly access its virtual address space. To get around this, a memory region can be *pinned*, instructing the OS to always keep it in

the same physical location. After pinning, the physical and bus addresses can be retrieved and used for DMA. [2]

2.7 Device Drivers

A system interacts with any PCIe device solely through simple mechanisms like MMIO and interrupts. This means that, for example, to program a DMA engine on a device, the system must know exactly what data structures need to be written to which addresses on the device. Instead of having every application implement the details of communication with any relevant device, operating systems use a *driver* architecture to abstract this complexity. Under a driver architecture, the specifics of communicating with a device are programmed as a driver that the OS loads when the device is configured. The OS then exposes a common set of APIs to applications, so they can interact with a given class of device without knowing the details of how it is performed. [12]

How drivers are developed and the functions they need to perform varies between different operating systems. In the case of Linux, drivers run as part of the kernel. They can be statically linked to it, which means that the whole kernel needs to be recompiled, or they can be compiled into an independent *module* that the kernel can dynamically load at run time. In any case, an application interacts with a Linux device driver, and the underlying hardware, through a special file created to represent the device. After opening this file, the application can use standard file API calls to read or write to it, map it to memory, or send control instructions, and the driver implements these operations in the manner most appropriate for the device. [12]

For simple interactions, Linux provides a basic set of drivers that are loaded for every PCIe device, and allow a process to access the device's BARs. [12] This is useful for development purposes, but relying on this functionality defeats the purpose of having drivers: applications end up tightly coupled with a particular device's communication protocol. Moreover, many necessary OS APIs, such as the ones for pinning memory, are restricted to code running in the kernel address space, so a custom driver is usually needed. [2]

In the case of Windows, there is no default driver loaded for an unknown device, so any meaningful interaction between an application and a device needs an intermediate driver. Microsoft has published multiple driver development frameworks (WDM, KMDF, UMDF) [14]. There are also third-party driver development kits, such as Jungo's WinDriver, that aim to simplify the development process [15].

The way an application interacts with a Windows driver depends on the nature of the device and the development framework used. A driver for a storage device or a sound card would be accessed indirectly through the file system or a sound API, respectively, while a more specialized driver such as for a custom FPGA design could be accessed through a file, in a way analogous to Linux. When using WinDriver, all file handling can be abstracted by using WinDriver APIs [15].

2.8 Previous work

Some research has already been done on the topic of DMA between GPUs and FPGAs. This section gives a brief overview of a selection of the corresponding published papers.

2.8.1 Bitner and Ruf

Bitner and Ruf [1] originally proposed a way to repurpose the existing DMA engines on GPUs to carry out DMA operations to an FPGA. Their implementation is based on two standard CUDA functions: `cudaHostRegister` and `cudaMemcpy`. The `cudaHostRegister` function is intended to pin host memory and simultaneously notify Nvidia drivers that the pinned memory is safe for DMA. Calls to `cudaMemcpy`, which is used to copy memory between the host and a GPU, can then check the internal list of DMA-safe memory regions and program the DMA engines accordingly. If the host memory address provided to `cudaMemcpy` is not registered as safe, Nvidia drivers will use an internal pinned buffer as an intermediary.

Bitner and Ruf's finding is that mapping an FPGA BAR to a process's virtual address space, the resulting address can be passed to `cudaHostRegister` as if it were normal host memory. Even though the FPGA memory passed in this way is not swappable due to its nature as device memory, `cudaHostRegister` does not fail when attempting to pin it, and successfully registers it as safe for DMA. In this way, future `cudaMemcpy` calls can be accelerated by the GPU's DMA engine.

This approach to DMA was tested with a custom FPGA design and a custom Windows driver. Transfers from the GPU to the FPGA saw an increase in throughput between 30% and 80%, depending on transfer size, and settling at 34.6% for the largest transfers. On the other hand, transfers in the opposite direction had increased throughput when transferring less than 16 KiB at a time, but suffered a throughput degradation of up to 52.6% with larger transfer sizes. In both directions, transfer latency was reduced from approximately 60 µs to 40 µs. The cause for the poor performance of FPGA to GPU transfers was not found, but it was suggested to be related to the FPGA design.

2.8.2 Thoma et al.

Thoma et al. [16] aimed to produce an open source framework for GPU-FPGA DMA, called FPGA^2. This framework was based on a Xilinx IP core and an open source GPU driver (noveau) and CUDA implementation (gdev). The open source drivers and runtime allow the framework to pin GPU memory and obtain its bus address in a way similar to GPUDirect RDMA, but requiring an additional copy within GPU memory to make the pinned memory usable by GPU kernels. The cost of this additional transfer is considered insignificant due to the large difference between PCIe bandwidth and internal GPU memory bandwidth.

For small transfers, their DMA implementation achieves throughput about twice as high as that of indirect transfers through the host in either direction. For transfers larger than 256 KiB, there is no clear trend, and the throughput of their DMA seems to be close to that of indirect transfers. It is important to note that their FPGA had a PCIe Gen1 x1 link, while the GPU had a much faster Gen2 x16 link. The performance of host-GPU transfers was scaled by the number of lanes, but not to take into account the different bandwidth offered by the newer PCIe generation.

The use of open source drivers and CUDA runtime make their approach applicable to consumer-grade GPUs, which is not possible with the official GPUDirect RDMA. However, this comes at the cost of performance and features offered by the official drivers.

The FPGA^2 framework was not available at the URL provided in the paper at the time of this dissertation's publication.

2.8.3 Gillert

Gillert's work [17] uses OpenCL to program both the FPGA and the GPU, and GPUDirect RDMA to pin GPU memory. To enable GPUDirect RDMA for GPU buffers allocated by OpenCL instead of CUDA, some CUDA functions had to be reverse engineered by tracking their interaction with Nvidia's GPU driver. Altera's PCIe driver for OpenCL also needed to be extended to properly handle GPU memory addresses.

The FPGA used for testing offered a PCIe Gen3 x8 bus, while the GPU had a Gen2 x16, both of which provide a bandwidth of approximately 8 GB/s. However, the measured bandwidth to or from the CPU is less than 1 GB/s for the FPGA and less than 2 GB/s for the GPU. The cause of these initial inefficiencies is not examined, although it could affect the results obtained for DMA transfers in an unpredictable way.

The peak throughput for transfers from the FPGA to the GPU increased from 580 MB/s going through the host, to 740 MB/s with Gillert's DMA implementation. Transfers in the opposite direction were not implemented due to technical issues which could not be solved during the project.

Gillert also proposes a straightforward optimization to indirect transfers, in which large transfers are split into multiple smaller ones, so that data arriving to host memory can be sent to its destination before the whole initial transfer is complete. In this way, the overall bandwidth should approach the bandwidth of the slowest connection, instead of being smaller than both individual connections'. Latency, however, would not be improved, and the algorithm needs to be tuned with an appropriate size for each individual transfer. An important advantage of this approach is that it could work under any OS and drivers, and is not subject to restrictions like GPUDirect RDMA.

Chapter 3

Common components

A considerable part of the software developed for this project is cross-platform and independent of any DMA implementation. This chapter describes all such cross-platform software components. Section 3.1 covers the FPGA design and how it can be used to master DMA transfers. To access that FPGA from software applications, a "user-mode driver" library was developed, as described in Section 3.2. This library is used by a benchmark application that measures the time taken for different types of transfers, covered in Section 3.3. Finally, Section 3.4 describes the hardware and third-party software used during testing.

3.1 FPGA Design for DMA

As described in Section 2.5, there are two main ways to implement DMA between an FPGA and a GPU. The way supported by Nvidia is to pin a GPU memory region to a BAR, then have the FPGA read and write to it [2]. The way proposed by Bitner and Ruf is the opposite: to map a BAR in the FPGA to the process's virtual address space, then use `cudaMemcpy` to have the GPU read or write to that BAR [1].

For the FPGA-mastered approach, the FPGA design needs to be capable of receiving pairs of source and destination addresses indicating a transfer to perform, then issue the corresponding MRd or MWr TLPs. If the operation is a read initiated by the FPGA, the design also needs to handle the completion TLPs sent back by the GPU.

The GPU-mastered approach is simpler to implement on the FPGA, because all the logic to accept the pairs of addresses and issue the corresponding TLPs is already present in the GPU and its driver. The FPGA only needs to sensibly respond to MRd TLPs and modify its internal state according to MWr TLPs.

The FPGA employed in this project is an Altera Cyclone V GT. Altera provides hard IP cores to access its PCIe functionality at the transaction layer, meaning that the hard IP core provides an interface for custom designs to issue and receive TLPs. This is a relatively low-level interface, because the custom design must still decide when to issue each individual TLP, and with what payload. In particular, the ability to issue a completion TLP after receiving a MRd must be explicitly coded in. [18]

Altera also provides a set of higher level soft IP cores that make use of the hard PCIe core and simplify implementation of common features. The design developed for this dissertation is based

on one of these, called Avalon-MM DMA [19]. This IP core maps an FPGA BAR to external DDR3 memory on the FPGA, and handles any MRd or MWr TLP received on that BAR appropriately. A MWr causes the DDR3 memory to be updated, and a MRd triggers the FPGA to read from the memory and issue the corresponding completion TLPs. This BAR is called the *user memory BAR*.

The Avalon-MM DMA core also includes a DMA engine, so it is also capable of bus mastering [19]. The DMA engine needs to be initially configured through a second BAR, referred to as the *control BAR*. Table 2 describes the layout of the control BAR.

After initial configuration, instructions are then sent to the DMA engine by the host through a *descriptor table*. Each table consists of up to 128 *descriptors*, preceded by as many *status* fields. Each status field is a 32-bit word, of which only the least significant bit is used. This is the *done* bit, and is set by the FPGA when it finishes the operation specified by the corresponding descriptor. The format of each descriptor is summarised in Table 3. There are two descriptor tables, one for read operations and one for writes. [19]

Address offset	**Size (bytes)**	**Register name**	**Description**
0x0000	8	RC_RD_DESCRIPTOR_BASE	Bus address of the read descriptor table
0x0008	8	EP_RD_DESCRIPTOR_BASE	Logical address of the read descriptor FIFO buffer
0x0010	4	RD_DMA_LAST_PTR	Specifies the last descriptor in the read descriptor table that was processed by the FPGA
0x0014	4	RD_TABLE_SIZE	Maximum number of descriptors in the read descriptor table
0x0018	4	RD_CONTROL	Set of control flags to alter the behaviour of the DMA engine.
0x0100	8	RC_WR_DESCRIPTOR_BASE	Bus address of the write descriptor table
0x0108	8	EP_WR_DESCRIPTOR_BASE	Logical address of the write descriptor FIFO buffer
0x0110	4	WR_DMA_LAST_PTR	Specifies the last descriptor in the write descriptor table that was processed by the FPGA
0x0114	4	WR_TABLE_SIZE	Maximum number of descriptors in the write descriptor table
0x0118	4	WR_CONTROL	Set of control flags to alter the behaviour of the DMA engine.

Table 2. Avalon-MM DMA control BAR layout

Address offset	Size (bytes)	Field name	Description
0x00	8	SRC_ADDR	Address from which to transfer data
0x08	8	DEST_ADDR	Address to which to transfer data
0x10	4	CONTROL	Specifies length of the transfer in 32-bit words (bits 0 to 17), and the index of the descriptor within its table (bits 18 to 24)
0x14	12	RESERVED	Unused

Table 3. DMA Descriptor format

Addresses referring to memory in the FPGA are not bus addresses, but addresses from a *logical address space* defined by the Avalon-MM DMA core [19]. A contiguous region starting at address 0 represents the user memory BAR, and other internal control data are given higher addresses [19]. The process for the host to instruct the FPGA to perform a read DMA operation is described in Algorithm 1.

1. Allocate a descriptor table in pinned host memory.
2. Write the number of descriptors to RD_TABLE_SIZE and the bus address of the start of the descriptor table to RC_RD_DESCRIPTOR_BASE in the FPGA's control BAR.
3. Write a new descriptor in the descriptor table, specifying the transfer length, and source and destination addresses.
4. Set the *done* bit corresponding to the new descriptor to 0.
5. Write the index of the new descriptor to RD_DMA_LAST_PTR, instructing the FPGA to read and execute that descriptor.
6. Wait for the FPGA to set the *done* bit to 1.

Algorithm 1. Executing a read DMA operation with Avalon-MM DMA

Steps 1 and 2 are the initial configuration steps, required only once until the FPGA is reset. Subsequent DMA operations require only steps 3 to 6.

When RD_DMA_LAST_PTR is written to in step 4, the FPGA reads all descriptors from indices one plus the previous value of RD_DMA_LAST_PTR until the new value, and copies them to an internal FIFO buffer, from which they will be taken and processed automatically. After processing each descriptor, the FPGA sets its corresponding *done* bit in the descriptor table and fires an interrupt.

After setting RD_DMA_LAST_PTR, the host can either wait for the interrupt to be fired by the FPGA, or simply poll the done bit until it is set. If multiple descriptors are issued at once, they may be completed out of order, so all done bits must be checked. For simplicity and reduced latency, the host software programmed for this dissertation is based on polling instead of interrupts.

To perform a write DMA operation a similar process is followed, but with a different descriptor table and writing to the control BAR addresses containing WR instead of RD.

An important implementation detail is the layout of the CONTROL descriptor field. Since only 18 bits are used to specify the length of each transfer, and length is specified in terms of 32-bit words (4 bytes), the maximum length that can be transferred with a single descriptor is $4\times(2^{18}-1)=$

1,048,572 bytes, or 4 bytes under 1 MiB. This means that any transfer larger than that needs to be split into multiple transfers, each using a different descriptor.

While using the maximum transfer size per descriptor would be natural, this causes problems due to a limitation of the IP core: transfer start addresses must be aligned to host pages. On both Windows and Linux, the default page size is 4 KiB, so transfer start addresses must be multiples of 4096 bytes. If a large transfer needs to be split because it is larger than what one descriptor can specify, and the first descriptor is used to the maximum possible size of 1,048,572 bytes (which is not a multiple of 4096), the start address of the second descriptor would not be a multiple of 4096 bytes either, causing the transfer to fail.

To work around this potential problem, transfer sizes per descriptor are capped to the maximum multiple of 4096 bytes that fits in a descriptor, which is 1,044,480 bytes. This could negatively affect performance, because certain transfers would use more descriptors than necessary. As the simplest example, a transfer of the maximum descriptor size could obviously be done with a single descriptor, but with this cap two descriptors would be necessary.

3.2 User-mode driver

The custom FPGA design described in Section 3.1 does not behave like any of the standard device classes natively supported by operating systems and programming languages, such as storage or input devices. This means that to access the device, low-level and OS-dependent APIs must be used. To isolate and abstract these non-portable API calls, a library was created. This library acts as a "user-mode driver", allowing an application to make use of the device without interacting with the OS directly.

This section describes the functionality offered by this library, disregarding how it is achieved on each OS. OS-dependent implementation details are described in Section 4.2 for Windows, and Section 5.3 for Linux.

3.2.1 Accessing FPGA BARs

The most basic function offered by the library is to map the FPGA's control and user memory BARs to the application's virtual address space as arrays of bytes. To do this, the application must first instantiate the class `AvalonDevice`. On construction of such an instance, the library interacts with the OS-dependent kernel-mode driver to initialize the device and map the two BARs to user space. The application can later access these mappings through the `userMemory` and `controlMemory` functions. This process shown in Listing 1, which uses the user memory mapping to set the first byte to an arbitrary value of 1.

```
AvalonDevice device;
uint8_t * user_memory = device.userMemory();
uint8_t * control_memory = device.controlMemory();
user_memory[0] = 1;
```

Listing 1. Accessing user memory and control BAR mapings

No other action is required from the application to initialise or finalise the library. `AvalonDevice`'s constructor initialises any necessary resources, and the destructor automatically releases them. The nature of these resources is OS-dependent. The constructor assumes that there is only one

AvalonMM FPGA in the system. If this was not the case, the constructor would need to take a parameter to allow the application to specify which FPGA to access.

3.2.2 Accessing default buffers

To execute DMA operations between the FPGA and the host, it is necessary to have pinned buffers in the host with well-known bus addresses. A case of this is sending the DMA descriptor tables to the FPGA: the tables need to reside in a fixed bus address once the FPGA is configured. A second case is to transfer data to host memory with the DMA engine, which needs fixed bus addresses while the DMA transfer is in progress.

While it would be desirable to let the application define the number and size of buffers it needs, it is hard on both Windows and Linux to allocate a large buffer that is contiguous in physical memory. Due to memory fragmentation, the OS may not be able to find a large-enough block of memory to fulfil the requested allocation.

To get around this issue, both operating systems allow multiple ways to allocate memory at boot time, before fragmentation can take place. The drivers developed for both operating systems pre-allocate two buffers in this way, and the `AvalonDevice` class allows applications to access them.

One of the buffers is intended to hold the DMA descriptor tables, and can be accessed through the `dmaTablesBuffer` function. The other buffer is for general purpose transfers defined by the application, and can be accessed by `defaultHostBuffer`.

Both functions return a templated class called `DmaBuffer`, which allows access to the buffers with array semantics through `operator[]`, and to their bus addresses through `physicalAddress`. Given that these buffers are pre-allocated at boot time, they should never be deallocated by the application, so the instances of `DmaBuffer` do not represent ownership of the underlying buffer, acting instead as mere references to it.

3.2.3 Executing FPGA-mastered DMA operations

The final service provided by the library is to start and complete DMA operations. These operations work with similar semantics to MPI sends and receives, although without order guarantees.

To start an operation, the application must first instantiate an `AvalonDevice` as usual, then pass it as a constructor parameter to an `AvalonDmaManager`. The manager can then be used to start a DMA operation through `read` or `write`, then wait for them to be completed with `complete`.

The functions `read` and `write` take as parameters the destination and source bus addresses, as well as the transfer size in bytes. The return type is an opaque handle that represents an in-progress operation. A read is a transfer from the FPGA to the host, and vice versa for a write. However, since the operation is mastered by the FPGA, a read is implemented in terms of MWr TLPs, and a write, in terms of MRd TLPs.

Destination or source addresses referring to host memory should be passed as bus addresses. As described in Section 3.1, these addresses should be aligned to page boundaries due to limitations of the Avalon-MM DMA IP core. On the other hand, addresses referring to FPGA memory refer to the logical address space defined by Avalon-MM IP, which always starts from address 0.

Since the length of a transfer specified by each descriptor is measured in 32-bit words, but the size passed to `read` and `write` is in bytes, this parameter should always be a multiple of 4.

After starting one or more DMA operations, the application can continue executing until it needs the operations to be completed. It can then call the `complete` function once for each operation, passing the corresponding handle as a parameter. The `complete` call will block until the transfer is finished.

Listing 2 illustrates how the library can be used to transfer data in both directions simultaneously. The first 4096 bytes from the FPGA user memory are read to the first 4096 bytes of the default host buffer, and the next 4096 bytes of this buffer are written to the FPGA. While the sizes in the example could be any multiple of 4 bytes, the offsets need to be multiples of 4096 bytes due to the page-alignment requirement.

```
AvalonDevice device;
AvalonDmaManager manager(device);
auto buffer = device.defaultHostBuffer();
size_t read_size = 4096, write_size = 4096;
uint64_t read_offset = 0, write_offset = 4096;
uint64_t read_src_addr = read_offset; // Logical addr, starts at 0
uint64_t read_dst_addr = buffer.physicalAddress() + read_offset;
// Start a read operation
DmaHandle rh = manager.read(read_dst_addr, read_src_addr, read_size);
uint64_t write_src_addr = buffer.physicalAddress() + write_offset;
uint64_t write_dst_addr = write_offset; // Also logical address
// Start a write operation
DmaHandle wh = manager.write(write_dst_addr, write_src_addr, write_size);
// Both operations are running concurrently.
// The application can then wait for them to be completed.
manager.complete(rh);
manager.complete(wh);
```

Listing 2. Executing simultaneous read and write DMA operations

3.2.4 AvalonDmaManager internals

The class `AvalonDmaManager` relies on the abstractions offered by `AvalonDevice`, so it is completely portable across operating systems. Its two main data structures are the DMA descriptor tables and the FPGA's control BAR.

The descriptor tables are pre-allocated by the `AvalonDevice` as described in Section 3.2.2. The DmaManager simply obtains a reference to the pre-allocated buffer, which contains both tables modelled as C-style `structs`. Using `structs` instead of full C++ classes allows the data structures to be accessed by the kernel module in Linux, which is programmed in C due to limitations of the *kbuild* system. Furthermore, since in both operating systems the data structures are pre-allocated by a kernel-mode driver before the library is even loaded, no C++ constructor would be run to initialise them.

A reference to the control BAR is also obtained from the `AvalonDevice` as a byte buffer, then casted to a struct representing the data layout described in Table 2.

Since both the DMA tables and the control BAR structs are duplicated, with one instance for reads and one for writes, these structures are held in arrays of size 2, with index 0 arbitrarily assigned to

reads, and index 1, to writes. This allows using the same code to handle both types of operation, changing only the index to the data structures.

On construction, `AvalonDmaManager` obtains the references to the data structures described above, then initializes them. First, the tables are zeroed, then all done bits set to 1 to indicate they are not in use. Initialisation of each descriptor could be done as it is about to be used, but since the whole table takes only slightly more than 4 KiB, initialising it once at program start-up is practically free, but it can help in debugging. After the tables have been initialized, their addresses can be set on the control BAR. Once these addresses are set, all descriptors in the table must be used before the addresses can be reset. This is verified by checking the `LAST_PTR` field in the control BAR. If this field indicates that a table has been partially used, but the addresses do not match, the FPGA cannot be configured by the library and needs to be reset.

To start a DMA operation, the manager follows the algorithm described in Algorithm 2.

1. Get references to the data structures for the current operation type.
2. Determine necessary number of descriptors based on transfer size.
3. For every necessary descriptor:
   3.1. Obtain reference to next available descriptor.
   3.2. Reset the done bit so the FPGA can set it when the operation is done.
   3.3. Set the descriptor addresses and length.
4. Issue a memory fence.
5. For every descriptor:
   5.1. Set the `LAST_PTR` field in the control BAR to the descriptor's index.
6. Generate a handle, encoding the range of descriptors used and the type of operation.
7. Save the index of the next available descriptor.

Algorithm 2. Starting a DMA operation within AvalonDmaManager

Descriptors must be used in order, starting from index 0 up to the maximum index of 128, then looping back to 0. To be able to determine which descriptor to use in Step 3.1, the library keeps track of the last descriptor sent to the FPGA by reading the `LAST_PTR` control register at initialization time, then updating it after every DMA operation in Step 7.

A descriptor is considered in-use if its *done* bit is 0. After using the last descriptor and looping back, if the next descriptor is still in use, it cannot be reused until the FPGA sets its *done* bit. A production-ready implementation would need to handle this case gracefully, probably either waiting for the descriptor to be freed, or enqueueing the operation so it can be started later. For simplicity, this implementation aborts with a descriptive error message if this situation occurs.

The memory fence in Step 4 might be required to prevent the compiler or processor from reordering instructions in a way that could lead to incorrect behaviour. For example, if the loops in Steps 3 and 5 were fused together and the `LAST_PTR` field in the control BAR was set before the corresponding descriptor was written, the FPGA would read stale or uninitialized data from the descriptor table. However, this was not observed to occur in practice.

To complete a memory operation, the library parses the handle generated in Step 6 to retrieve the range of descriptor indices and the type of operation, then sequentially waits for the *done* bit of each descriptor to be set. This wait could be accomplished either by waiting for an interrupt sent by the FPGA with every completed descriptor, or by simply spinning on the *done* bit until it becomes set. An interrupt-based implementation would let the processor do useful work for other

threads while waiting, but at the cost of higher complexity and latency. For these reasons, this implementation is based on spinning.

Given that the *done* bit is set by the FPGA without involving the code running on the CPU, the `volatile` keyword needs to be applied to the descriptor's *done* bit. This instructs the compiler to issue a memory read every time the bit is accessed for comparison. Without the `volatile` keyword and with compiler optimizations enabled, the `complete` function would stop working, returning before the done bit was set by the FPGA. As a general rule, all accesses to memory either accessed or hosted by the FPGA should be qualified with the `volatile` keyword.

Finally, as a debugging aid, all DMA operations are logged to a file if the `LOG_DMA` pre-processor directive is defined. This is done by default on Windows debug builds, and can be enabled on Linux by passing a `-D` option to `make`.

3.3 A benchmark application

A sample benchmark application was developed to test all the different ways to transfer data between the FPGA and the GPU, and measure their performance, as well as to demonstrate how the user-mode library can be used. This application employs the user-mode library and the CUDA runtime to transfer data from a buffer in the FPGA, GPU or host memory to any of the other two devices. The transfers are performed with sizes varying from 4 bytes to 32 MiB, each one repeated a configurable number of times. The application then reports the average time taken for each transfer size, as well as the resulting throughput.

The benchmark takes three parameters. The first one is a 2 or 3-letter code indicating the source, destination and method for the transfer, and its possible values are summarized in Table 4 and further described in this section. The second parameter is the number of iterations to run for each transfer size. More iterations give more consistent results at the cost of longer runtimes. The third parameter determines whether transfers should be verified. A value of 'V' enables verification, while any other value, or omitting the parameter, disables it. Verifying that data was transferred successfully can take longer than transferring the data in the first place, so performance should not be evaluated with verification enabled. Instead, verification mode should be used with a limited number of repetitions to ensure that the implemented algorithms work as intended. Due to the use of MMIO, verification of FPGA memory achieves a maximum throughput of about 16 MB/s on the test system, which is much slower than all tested transfers, but still fast enough for its intended purpose.

The transfer types involving only host memory and either the GPU or the FPGA (FH, FM, GH, HF, HG, HM) are not the goal of this dissertation, but they were useful to have for comparison purposes. Similarly, the indirect transfers between FPGA and GPU memory, passing through host memory (FI[R], GI[R]), were taken as the performance baseline that any DMA implementation should improve. The purpose and method to register memory as pinned will be described in Section 4.1.

The remaining transfer types are the DMA transfers developed for this dissertation. FG[R] and GF[R] are the GPU-mastered method described in 0, while FR and GR are the GPUDirect RDMA method described in Chapter 5.

The general process followed by all benchmark types is outlined in Algorithm 3. The intermediate buffers allocated in step 3 are allocated for FI[R] and GI[R] transfers, in which host memory acts

as an intermediary between the source and destination buffers. Verification buffers from Step 4 are used when verification is enabled and the destination buffer is in GPU memory. In such cases, to verify that data was transferred successfully, it is copied from GPU memory back to host memory so it can be compared to the expected data.

Code	Description
FI	DMA from the FPGA to host memory, then `cudaMemcpy` from host memory to GPU memory.
FIR	As FI, but register host memory as pinned so that `cudaMemcpy` can use DMA.
FG	Use `cudaMemcpy` to transfer directly from FPGA memory to the GPU.
FGR	As FG, but register FPGA memory as pinned so that `cudaMemcpy` can use DMA.
FH	DMA from the FPGA to host memory.
FM	Non-DMA transfer from FPGA memory to host memory, using memory-mapped IO.
FR	DMA transfer from the FPGA to GPU memory using GPUDirect RDMA.
GI	Use `cudaMemcpy` from GPU memory to host memory, then DMA from host memory to the FPGA.
GIR	As GI, but register host memory as pinned so that `cudaMemcpy` can use DMA.
GF	Use `cudaMemcpy` to transfer directly from the GPU to FPGA memory.
GFR	As GF, but register FPGA memory as pinned so that `cudaMemcpy` can use DMA.
GH	Use `cudaMemcpy` to transfer from the GPU to host memory.
GR	DMA transfer from GPU memory to the FPGA using GPUDirect RDMA.
HF	DMA from host memory to the FPGA.
HG	Use `cudaMemcpy` to transfer from host memory to the GPU.
HM	Non-DMA transfer from host memory to FPGA memory, using memory-mapped IO.

Table 4. Benchmark transfer types

The execution times reported by the benchmark in Step 6 are averages per iteration, so that running more iterations should not change the reported average values, but only make them more consistent across runs. Throughput is calculated as the total number of MB of user data transferred per second (i.e. protocol headers are not included in the calculation, which would inflate the reported values by counting data that the user can make no use of. To minimize the impact of timing function calls on the reported values, individual transfers are not timed. Instead, all iterations are measured together, then the average time per iteration is calculated. While the very short times involved could justify this procedure, it prevents analysis of the variability of individual transfer times.

1. Allocate and initialize source buffer with random data.
2. Allocate destination buffer.
3. If necessary, allocate intermediate buffer in host memory.
4. If necessary, allocate verification buffer.
5. Create lambda function that takes a transfer size as a parameter and executes a transfer of that size between the allocated buffers.
6. Pass the lambda function to `measureTransfers`, which executes the function the specified number of times and outputs the average run time.

Algorithm 3. Benchmark process

3.4 Test hardware and software

The performance of the two DMA methods implemented for this dissertation will be analysed in the next two chapters. Table 5 summarises the hardware used to obtain the presented results.

Component	**Description**
CPU	Intel Xeon W3520 @ 2.67GHz (4 cores, hyperthreading disabled)
Motherboard	HP Z400 (FM065UT#ABA)
RAM	6x 1 GiB DDR3 @ 1333 MHz
GPUs	Nvidia Quadro M2000 Nvidia GeForce GTX 950
FPGA	Altera Cyclone V GT FPGA Development Kit
Operating System	Windows 7 Enterprise SP1 (build 7601, 64-bit) Ubuntu 16.04.2 LTS (kernel 4.4.0-87, 64-bit)
C++ compiler	GCC 6.3.0
FPGA Synthesiser	Quartus Prime 16.1.2
GPU Drivers	Nvidia 382.53 (Windows) Nvidia 375.66 (Linux)
CUDA	CUDA 8.0.61

Table 5. Test hardware and software

Unless otherwise noted, GPU test results refer to the Quadro M2000 card. The GTX 950 was used to obtain additional comparison points. Both cards feature PCIe Gen3 x16 connections, but the motherboard does not have any PCIe Gen3 slots, so the cards were connected to PCIe Gen2 x16 slots. PCIe hardware is designed to be backwards compatible, so the cards should work, albeit with half the available bandwidth.

The Altera FPGA features a slower PCIe Gen2 x4 connection. Since the motherboard does not have a slot of those characteristics, the FPGA was inserted into the remaining PCIe Gen2 x16 slot. PCIe allows connecting cards of a narrower link width into a wider slot, with the extra lanes remaining unused and the card running as if it was connected to a slot of the same width. However, as will be covered in Sections 4.4 and 5.5, even slots nominally of the same generation and number of lanes behave significantly differently.

Chapter 4

DMA on Windows

The official API for DMA to Nvidia GPUs is not supported on Windows. However, it is still possible to leverage the GPUs' bus mastering capabilities to access FPGA memory. Section 4.1 gives an overview of how this can be accomplished. Sections 4.2 and 4.3 cover, respectively, how the user-mode driver and the benchmark application were extended to add this DMA method. Section 4.4 analyses the performance obtained, compared to that of indirect transfers.

4.1 Overview

Nvidia GPUs are equipped with DMA engines to accelerate transfers between host memory and GPU memory and allow transfers to happen concurrently with kernel execution [1]. However, there is no official API to program these DMA engines directly. Instead, they are employed at the discretion of the GPU drivers in response to calls to CUDA runtime functions like `cudaMemcpy`. These functions typically take a virtual address in GPU memory and one in host memory, plus a parameter indicating in which direction the transfer occurs. If the host memory is known by the Nvidia driver to be pinned, the memory is accessed directly by the DMA engine. Otherwise, a third buffer in host memory, which is pinned and transparently managed by the Nvidia driver, is used as an intermediary.

To avoid unnecessary copies to the intermediate buffer, two conditions must be true: the host memory buffer must be pinned, and the Nvidia driver must be aware of this fact. The CUDA function `cudaHostAlloc` can be used to obtain a new buffer that fulfils both conditions at once. Alternatively, `cudaHostRegister` can attempt to pin an existing buffer and "register" it as such in the Nvidia driver, making it available for DMA transfers without copies. [20]

In Bitner and Ruf's paper [1], it was found that mapping an FPGA's BAR to the process's virtual address space, then passing the resulting address to `cudaHostRegister` would allow CUDA and the Nvidia driver to employ the DMA engines on the GPU to transfer data directly between FPGA and GPU memory. The `cudaHostRegister` call would not have to pin any memory, since FPGA memory is not swappable in the first place, but it would nonetheless register the memory region as safe for DMA. With this process, transfer bandwidth from the GPU to the FPGA was increased by a factor between 1.4 and 1.7, but in the opposite direction (FPGA to GPU), bandwidth decreased by a factor of up to 0.4. The performance regression was speculated to be caused by traits of the FPGA design used for testing.

For this dissertation, Bitner and Ruf's procedure was reproduced to verify it would still work with current-generation hardware and software, and to test its performance with a different FPGA design. To do this, the platform-dependent part of the user-mode driver library was implemented on Windows, and two new options were added to the benchmark application, one for each transfer direction.

4.2 WinDriver-based AvalonDevice implementation

The cross-platform "user-space driver" library needs a few platform-dependent primitives defined in the AvalonDevice class. To implement these on Windows, Jungo's WinDriver development kit [21] was used.

The required functionality is to map the two FPGA BARs to the user address space, and to allocate two contiguous, pinned buffers: a small one to hold the DMA descriptor tables, and a large one to act as the default application buffer on host memory.

When working with WinDriver, the kernel-space driver is a binary provided by Jungo and loaded by a `.inf` file customized for the device. This kernel-space driver offers a rich API that can be accessed from user-space applications via a library provided by WinDriver.

The `.inf` file can be automatically generated with a Jungo-provided wizard, and its primary function is indicating to Windows that it should load the Jungo kernel-space driver for a device, based on the device's vendor and device identifiers. It can then be manually edited to configure driver features such as interrupt handling and pre-allocated buffers.

The WinDriver library, which provides convenient access to the kernel driver API, must be initialized once during the lifetime of an application to obtain a handle, then closed when no longer needed. To do this in a simple and exception-safe way, a C++ class called `WinDriver` was created. This class uses reference counting to ensure that the library is initialized only once, and then closed when the last reference is destroyed. Any instance of the class can be used to obtain a handle to the library with no overhead.

With the WinDriver class created, the AvalonDevice class constructor was implemented to obtain the four resources required (two BAR mappings and two host buffers), as described in Algorithm 4. Since the resources are obtained during construction, the methods to access those resources are simple, returning copies of instance fields.

1. Create instance of `WinDriver` to initialize WinDriver library if necessary.
2. Call `WD_PciGetCardInfo` to locate the FPGA based on its vendor and device identifiers.
3. Call `WD_CardRegister` to map the FPGA's BARs to the process's virtual address space.
4. Retrieve the relevant BAR mappings from the structure returned by `WD_CardRegister`.
5. Call `WD_DMALock` to obtain a reference to the buffer pre-allocated by the driver.
6. Split the pre-allocated buffer into two regions, one for the DMA descriptor tables and one for the default application buffer, and save the two resulting addresses.

Algorithm 4. AvalonDevice constructor on Windows

To check for errors on any WinDriver call, a macro called `checkWinDriverError` was introduced. When wrapping a call to any WinDriver API with this macro, if the call is not successful, the application aborts with an error message provided by the `Stat2Str` WinDriver function, as well as the file and function name and line number where the error occurred. This was implemented

with a macro so that the pre-processor can automatically fill in the context where the error occurred (function, file and line).

Finally, to prevent WinDriver headers from "leaking" to client applications, no headers in the user-space driver library include WinDriver headers. Where necessary, `structs` are forward declared instead of defined through inclusion of header files. In this way, a client application that includes an Avalon library header does not transitively include any WinDriver header, which could be an issue if compiling separately.

4.3 GPU-mastered DMA benchmark implementation

The final piece to implement DMA on Windows was to create a benchmark option to follow the procedure described in Section 4.1. The benchmark options GF[R] and FG[R] were introduced to this effect. The new options work similarly to the ones to copy data between the host and the GPU, except that the host buffer is replaced by the FPGA's user memory BAR mapping to user space. If the R argument is present (i.e. GFR or FGR instead of GF or FG), this memory mapping is passed to `cudaHostRegister`, also as described in Section 4.1; otherwise, the memory is not registered as safe for DMA.

After creating this buffer and optionally registering it, a GPU buffer is created, then the transfers between the two are measured, and optionally verified if such option is enabled. Enabling verification shows that transfers always succeed if the FPGA buffer is registered, but if not, they fail for transfers between 4KiB and 32 KiB. The cause of this is not clear, but the fact that it happens consistently for some transfer sizes and not for others suggests that it is due to how transfers of different sizes are handled by Nvidia drivers and hardware. Given that these transfers are not reliable, and that their performance is worse in all cases than the alternative with a registered buffer, they will not be discussed further.

4.4 GPU-mastered DMA performance

The benchmark application was run for all available transfer types, with 10 repetitions per transfer and verification enabled to ensure that all transfers were successful. After this, the benchmark was run again with 1000 repetitions per transfer and verification disabled to measure the average time taken for each type of transfer. The results are presented and analysed in this section.

Figure 1 shows the transfer time versus transfer size for DMA transfers from the GPU to the FPGA (GFR) and indirect transfers going through a host buffer (GIR), as well as the individual times for each of the two steps of the indirect transfer (GH for GPU to host, and HF for host to FPGA). A linear fit is shown for each type of transfer as a dotted line of the same colour. Due to the log-log nature of the chart, the linear fits appear as curves with a horizontal asymptote corresponding to the constant term of the linear fit.

Figure 2 shows the same for transfers in the opposite direction. FGR are DMA transfers from the FGPA to the GPU, FIR indirect transfers through host memory, and FH and HG are the two half transfers from FPGA to host and from host to GPU.

All curves follow their linear fits very closely, meaning that transfer time can be accurately estimated as $t = l + b^{-1}\mathrm{s}$, where t is the transfer time, l the latency, b the channel bandwidth, and s the transfer size. Table 6 summarizes the bandwidth and latency of each transfer type. While

bandwidths showed differences of no more than a few MB/s across runs, latencies were less consistent and thus are shown with less precision.

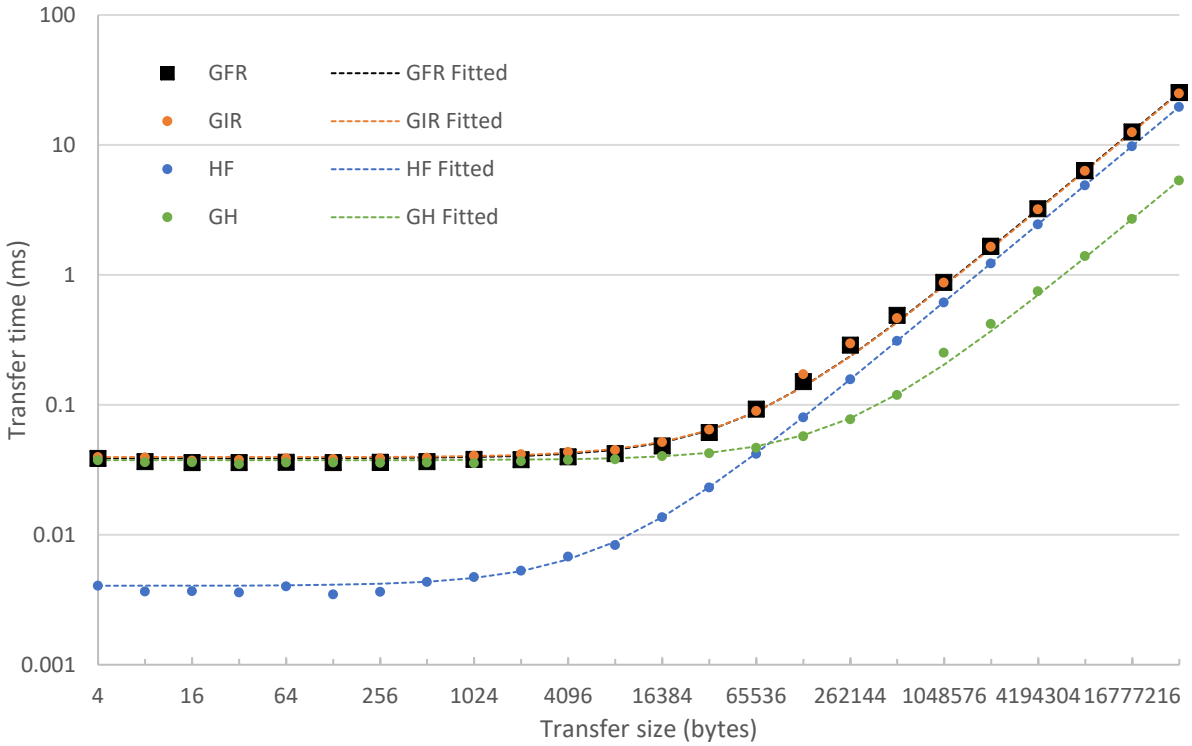

Figure 1. Transfer times from GPU to FPGA on Windows

Since the FPGA uses a PCIe Gen2 x4 connection, it has a maximum theoretical bandwidth of 2000 MB/s, as shown in Table 1. A fraction of that is lost as protocol overhead, depending on max transfer size and the size of headers added by the PCIe protocol stack. The FPGA was configured with a max transfer size of 256 bytes, and since its BARs are configured in 32-bit addresses and transaction-level error correction is not being used, the headers take 20 bytes each [11]. Since every 256-byte packet is accompanied by 20 extra bytes worth of headers, the FPGA can use at most $\frac{256}{256+20} = 92.7\%$ of the bus's theoretical bandwidth, or 1855 MB/s. Transfers from the FPGA to the host manage to make use of 98% of this value, while transfers in the opposite direction only manage 92%. This difference is most likely due to the inherent inefficiencies of PCIe reads compared to writes, as described in Section 2.3.

The GPU is connected to a PCIe Gen2 x16 link, so it has 4 times the available bandwidth (also from Table 1). However, transfers between the host and the GPU only manage to achieve an efficiency between 76% and 78%.

As expected, the transfer time for indirect transfers is the sum of each half transfer, so it can be modelled by the sum of the two linear equations for each half transfer. In this way, the combined latency is the sum of both individual latencies, and the inverse of the combined bandwidth is the sum of the inverses of the two individual bandwidths.

The performance of DMA transfers is in no case better than that of indirect transfers. Direct transfers from the GPU to the FPGA take the same amount of time as indirect ones, while transfers in the opposite direction take significantly longer for transfers larger than 1 KiB.

The close similarity between the performance of GPU to FPGA DMA and indirect transfers suggests that an intermediate buffer is being used at some point. Since neither the application nor the FPGA driver allocate such a buffer, if it exists, it must be managed by Nvidia drivers. It is possible that, even though the call to register the FPGA memory as pinned succeeds, Nvidia drivers still use their internal buffer for some reason. A more in-depth analysis of the TLPs received by the FPGA would reveal whether this is the case.

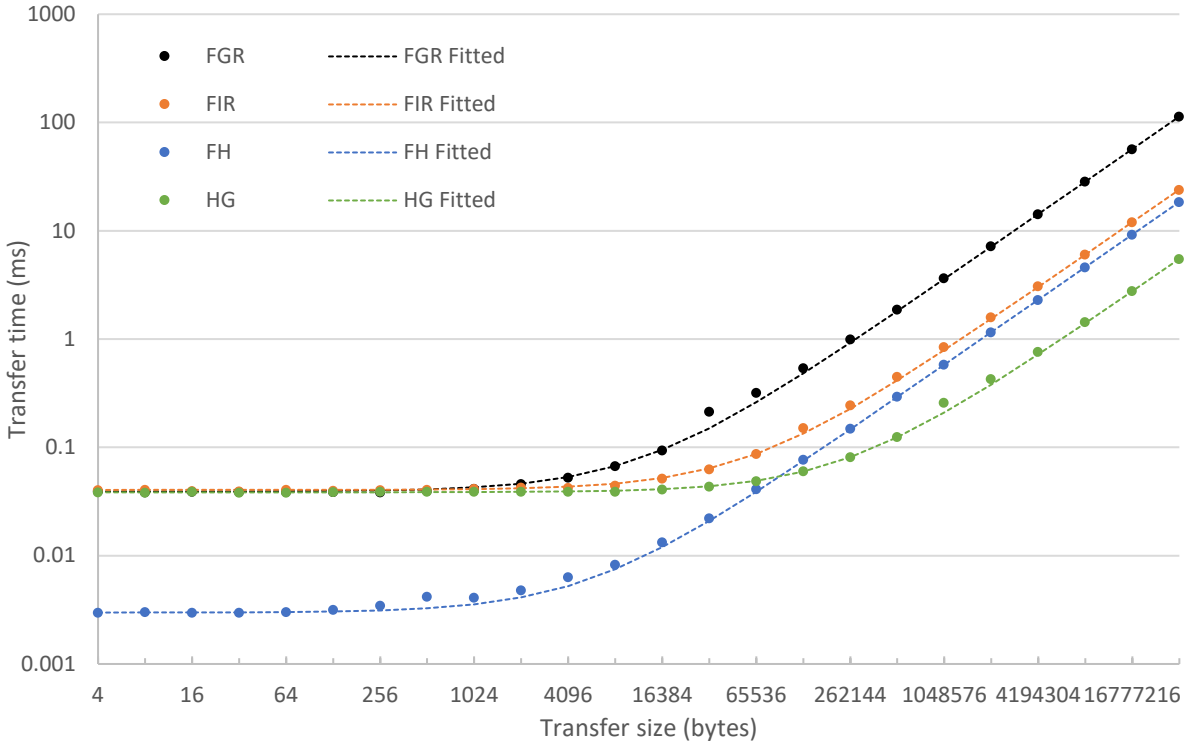

Figure 2. Transfer times from FPGA to GPU on Windows

Transfers from the FPGA to the GPU are carried out as reads, because the GPU is acting as bus master. While reads are inherently inefficient, the performance regression in this case is extreme, with these transfers achieving only 15% bandwidth efficiency at most. In an effort to find the cause of such inefficiency, the FPGA and the GPU were connected to different PCIe slots. This change increased latencies for transfers between the GPU and the host by 3%, but decreased them between the FPGA and the host by 10%. Although the changes were relatively small, they were consistent.

The biggest difference introduced by swapping the cards, however, was in the performance of the problematic DMA transfers from the FPGA to the GPU. The usable bandwidth increased from 295 MB/s to 885 MB/s. While this is still far below the theoretical maximum of 1885 MB/s, the performance gained by simply swapping the cards between different slots is impressive.

The bandwidth of none of the other transfer types was affected, including the equivalent indirect transfer and its two intermediate steps. From inspecting the output of `lspci`, a Linux utility that shows detailed information about PCI devices, there was no difference between the two slot configurations. All slots involved were Gen2 and with the same number of lanes, and no PCI switch was present, so the performance difference is hard to explain without inspecting the TLPs received by the FPGA.

In a second attempt to obtain performance results closer to Bitner and Ruf's, the FPGA was configured to use a maximum payload size of 128 bytes. This had no further effect on FGR transfers, but it reduced the effective bandwidth from FPGA to host by 9%, and by 4% for the opposite direction. This had a small impact on indirect transfers from GPU to FPGA, bringing the effective bandwidth down to 1300 MB/s, but not affecting that of DMA transfers in the same direction, leaving them at a slightly faster 1322 MB/s. This results in an insignificant 1.7% speedup.

Transfer type	Bandwidth (MB/s)	Latency (µs)
FIR	1401	40
FGR	295	39
GIR	1341	40
GFR	1322	39
FH	1817	3
HF	1706	4
GH	6310	36
HG	6135	38

Table 6. Linear model for transfer performance on Windows

Bitner and Ruf do not present numeric performance data for their FPGA to host transfers, but from their graphs it can be seen that their FPGA was less efficient than the one used for this dissertation, managing only about 1400 MB/s for writes and 1300 MB/s for reads. Since the DMA transfers to the GPU are mastered by the GPU instead of the FPGA, Bitner and Ruf's lower initial baselines inflate their reported speedup. In absolute terms, their setup managed 1600 MB/s from the GPU to the FPGA, and 514 MB/s in the opposite direction. In contrast, the setup presented in this dissertation performs at a slower 1322 MB/s in the first direction, but at a faster 885 MB/s in the opposite one, after rearranging the cards.

As a final test, the Quadro GPU was replaced by an Nvidia GeForce 950 and the test run again. Since both cards were connected to the same PCIe Gen2 x16 bus, there was no appreciable change in bandwidth. Latencies involving the GeForce card were, however, between 10% and 15% higher.

Chapter 5

DMA on Linux

On Linux, Nvidia's official API to directly access GPU memory is supported for some GPU brands. Section 5.1 gives an overview of how this API was used to implement DMA from an FPGA. Since GPUDirect RDMA requires all calls to originate from kernel mode, a custom kernel module was developed as described in Section 5.2. Section 5.3 covers how this module was used to implement the platform-specific functions of the user-mode driver library, and Section 5.4 shows how the benchmark application was extended to add this DMA method. Section 5.5 analyses the performance obtained, compared to that of indirect transfers.

5.1 Overview

On Linux, Nvidia drivers provide an API called GPUDirect RDMA that can be used to obtain the bus address of a buffer in GPU memory. This API is only available for professional-level GPUs from Nvidia, branded Quadro or Tesla. [2]

The main function of this API is `nvidia_p2p_get_pages`, which takes the virtual address of a GPU buffer and a size (among other parameters) and returns a list of bus addresses on which that buffer can be accessed. This function can only be called from the Linux kernel, and not from a user-mode process. It returns more than one address because the buffer is not necessarily contiguous, and can instead be paged in GPU memory in the same way that operating systems handle virtual main memory. After obtaining the physical addresses of the GPU buffer, a device capable of bus mastering, such as an FPGA, can access it directly, without involving host memory or the CPU.

For this dissertation, the benchmark application and user-mode driver library were adapted to take advantage of GPUDirect RDMA, in order to create a reference implementation and measure any obtainable performance benefits that might justify the use of professional-level GPUs instead of more economical alternatives.

Since the GPUDirect RDMA API is available only from kernel mode, a kernel module was created to this effect. The kernel module serves the dual function of acting as a driver for the FPGA, and providing access to GPUDirect RDMA to the benchmark application.

5.2 A kernel module for GPUDirect RDMA

Programming a kernel module is in many ways different from programming user mode applications. First, the module does not have the typical function `main` as an entry point, instead relying on different conventions. Furthermore, a module needs to be compiled using the *kbuild* toolset provided by Linux, which effectively forces the use of the C programming language. Finally, since the module will run as part of the kernel, it has limited access to OS APIs. For example, a module cannot easily access files or the screen, and `malloc` and `free` are not defined.

Since there is no file or console output from the kernel, kernel modules should output their debug output to the kernel log, using `printk` and similar functions. For dynamic allocations, the kernel has many options instead of `malloc`, such as `kmalloc`, `kzmalloc` and `dma_alloc_coherent`.

5.2.1 Module entry points

To define the interface between the entry points to the module and some additional metadata such as module name, version and author, *kbuild* provides a set of macros that must be called from the file scope (i.e. not from inside the scope of any function). The two main macros are `module_init` and `module_exit`, which define the functions that the kernel will call when the module is loaded and unloaded, respectively.

In its function marked as `module_init`, the *AvalonMM* module developed for this dissertation registers a driver for the Avalon-MM DMA-based FPGA design described in Section 3.1. To register a driver, a `struct pci_driver` is filled with the identifiers of the targeted device and a set of call-backs that allow the driver to take control of the device and relinquish it. When the module is unloaded, the driver is unregistered by the function marked as `module_exit`.

5.2.2 Driver entry points

After the driver has been registered and when a compatible device is plugged in, the kernel calls the `probe` function pointer set in the `struct pci_driver`. This function gives the driver an opportunity to take control of the device. The *AvalonMM* module does most of its work in its `probe` function, as summarized in Algorithm 5.

1. Allocate a `DeviceStatus` data structure to track the device's status.
2. Bind the `DeviceStatus` struct to the device so it can be retrieved by later calls to driver functions.
3. Configure the device as a *char device*, meaning that it can respond to standard file operations like open, close, read, write, etc.
4. Enable the device, instructing the kernel to activate its BARs, interrupts, etc.
5. Acquire exclusive access to the device.
6. Enable the device's bus mastering capabilities and inform the kernel of the range of addresses it can access (e.g. 32-bit or 64-bit addresses).
7. Map the device's BARs to the kernel address space.
8. Allocate contiguous buffers for the DMA descriptor tables and as the default application buffer.

Algorithm 5. Handling a probe callback in the *AvalonMM* kernel driver

The driver's `remove` function is called by the kernel when the driver needs to relinquish control of the device, for example during system shutdown. It simply frees all resources acquired by the `probe` function in reverse order.

5.2.3 Driver file operations

After configuring a device as a char device, a file can be created in the file system to represent it. When setting up this configuration, a `struct file_operations` is populated and passed to the kernel. This structure, analogously to the `pci_driver` described in Section 5.2.3, contains a set of function pointers that act as call-backs that are called by the kernel when an application accesses the file representing the device.

The *AvalonMM* module only implements the file call-backs that were necessary for its operation: `open`, `release`, `mmap` and `ioctl`. Notably, this does not include `read` and `write`, because the driver was designed to allow applications to read and write directly to the device through memory mapping, instead of going through kernel calls which would hinder performance.

Command	**Parameter type**	**Function**
`GET_USER_MEMORY_SIZE`	`uint64_t*`	Reads the size of the FPGA's user memory BAR into the provided pointer.
`GET_CONTROL_MEMORY_SIZE`	`uint64_t*`	Reads the size of the FPGA's control memory BAR into the provided pointer.
`GET_DESCRIPTOR_TABLES_PHYSICAL`	`uint64_t*`	Reads the physical address of the descriptor table buffer into the provided pointer.
`GET_DEFAULT_BUFFER_PHYSICAL`	`uint64_t*`	Reads the physical address of the default application buffer into the provided pointer.
`PIN_GPU`	`struct pin_request*`	Pins the GPU memory region indicated by the parameter and returns the number of pages that were pinned.
`GET_PINNED_PAGES`	`uint64_t*`	Reads the physical addresses of the GPU pages pinned by a previous call to `PIN_GPU` into the provided pointer.
`UNPIN_GPU`	(None)	Unpin the last GPU memory region pinned by `PIN_GPU`.

Table 7. *AvalonMM* driver IOCTL codes

The `open` call-back is called when the device file is opened by an application. It obtains the `DeviceStatus` struct associated with the device by the `probe` function and associates it with the

file descriptor so it can be accessed by later file operations. When the application closes the file, the `release` function is called, which does nothing because there are no per-file resources to free.

The Linux function `mmap` allows an application to map a file's contents to its virtual address space, so that it can be accessed without further calls to the kernel. When the file that is mapped in this way is not a normal file in the filesystem, but a device file such as the one defined by this module, the memory region mapped by such a call is defined by the module's `mmap` call-back.

The *AvalonMM* module presents four non-overlapping memory regions that can be mapped by an application. Two of them are contiguous buffers in host memory: one for the device's DMA descriptor tables, and one for the pre-allocated default application buffer. The other two are mapped to memory in the two FPGA BARs (control and user data). The starting address of each of these memory regions is arbitrarily defined by the driver and exposed to applications as a constant in a header file.

The last call-back defined by the driver is `ioctl`, which is intended for applications to communicate with the device driver with a custom protocol. The function takes a numeric command identifier, defined by the driver, and an argument which is formally defined as an unsigned long, but whose actual purpose is determined by the driver based on the command identifier.

The *AvalonMM* module accepts seven different command identifiers, as described in Table 7. Four of them are read commands, intended for applications to get run-time configuration data from the driver. The other three are to interact with Nvidia's GPUDirect API.

5.2.4 Pinning GPU memory

After pinning a region of GPU memory with `nvidia_p2p_get_pages`, there are two ways in which it can be unpinned. The most natural way is with a call to the opposite function, `nvidia_p2p_put_pages`, which must also be called from kernel mode. This is simple to implement, but it requires client applications to explicitly instruct the driver to unpin memory regions that are no longer needed.

The other option is for the memory region to be deallocated by the application that allocated it in the first place, without first notifying the kernel that it should be unpinned. This could occur as part of the normal execution of the application, or when a crashing application's resources are released. To handle this case without leaking resources, `nvidia_p2p_get_pages` takes a call-back as an argument that is called by Nvidia's driver when such a situation occurs. This call-back must ensure that it is safe to unpin the memory region (e.g. that it is not being accessed by the FPGA), then deallocate the associated data structures.

To simplify the implementation of the driver, it only allows one memory region to be pinned at a time. To do so, an application must populate a `struct pin_request` with the virtual address and size of the memory region, then pass it as parameter to a `PIN_GPU` `ioctl` call. The driver will then call `nvidia_p2p_get_pages`, save the returned data structure and set the `page_size` and `page_count` fields of the `pin_request` accordingly. The application must then pass an array of `uint64_t` to a `GET_PINNED_PAGES` `ioctl` call to retrieve the physical addresses. The array's size must be at least `page_count`.

When the memory no longer needs to be pinned, the application should call an `UNPIN_GPU` `ioctl`.

5.2.5 Issues with GPUDirect RDMA

While the procedure described above follows the directions provided by GPUDirect documentation [2] to the best of our ability, some issues were found with the values returned by `nvidia_p2p_get_pages`. The first issue is that the returned page size is always set to 1. This is obviously not the page size in bytes, but Nvidia documentation does not elaborate on the meaning of this value. From empirical data, it was found that GPU pages are of 64 KiB each, so this value was hardcoded into the kernel module.

A second and more important issue is with the physical addresses for the pages pinned by `nvidia_p2p_get_pages`. While the function call does not return an error code, the addresses do not correspond to the memory that was pinned. In all our tests, the second page address is a value low enough to belong to host memory, and if more than a few dozen pages are pinned at once, the address of the last ones is set to 0.

Since all sensible addresses were separated by offsets of 64 KiB, it was assumed that the memory pinned in this way was physically contiguous. Ignoring all addresses except the first one and transferring data under this assumption worked in all cases. After the transfers, reading back the memory from the GPU via `cudaMemcpy` returned the data that was transferred to it. This is by no means what should be done by production code, but it allowed the benchmark application to measure the performance of transfers using this API.

5.2.6 Allocating physically contiguous buffers

As described in Section 3.2.2, allocating a large, physically contiguous buffer can fail due to memory fragmentation (i.e. there may be enough memory to fulfil the allocation request, but not in contiguous addresses). The best way to ensure that this kind of allocation succeeds is to reserve a large enough memory region at boot time. Linux allows multiple ways to do this. One of them is for a module that is statically linked to the kernel to call the `alloc_bootmem` family of functions during boot time, but this requires the whole kernel to be recompiled to link in the new module.

A more user-friendly option is to use the `mem` kernel boot parameter to limit the amount of host memory that the kernel is allowed to use. This makes the kernel use the physical memory region from 0 up to the specified limit, leaving the rest of the physical address space unallocated. The unallocated memory can then be mapped as the required buffer by a dynamically loaded kernel module. The problem with this approach is that the kernel module must be hardcoded to access the unallocated addresses, which are system dependent. Improper configuration could lead to the module accessing memory regions being used by other parts of the kernel or by user processes.

A third alternative, and the one used by the *AvalonMM* module, is the built-in Contiguous Memory Allocator (CMA) kernel module. As the name implies, this module was designed specifically for this purpose. It reserves a configurable amount of memory at boot time, then transparently uses it to fulfil allocations through the `dma_alloc_coherent` function.

This CMA method is, however, not without problems. Since it is a built-in module, it must be enabled when the kernel is compiled. This was not the case for the Ubuntu distribution used for this dissertation (Ubuntu 16.04.2 LTS), so the kernel needed to be recompiled with the CMA enabled. At first, this recompilation was suspected to be the cause of the issues described in Section 5.2.5, but a clean reinstallation of the whole OS and drivers made no difference.

5.3 AvalonDevice implementation for Linux

Having developed the kernel module described in Section 5.2, implementing the platform-dependent functions of the *AvalonMM* user-mode driver encapsulated by the `AvalonDevice` class was as simple as it was under Windows with WinDriver. All necessary resources are acquired in the constructor, then the functions to access them by client applications simply return the existing references.

The resources that need to be acquired by the constructor are the mappings to the user address space of the two FPGA BARs and the two pre-allocated buffers, as well as the bus address of the latter so they can be set on the FPGA. To obtain the physical addresses and the sizes of the BARs, the library first opens the file representing the FPGA, then calls the first four `ioctl` codes described in Table 7. Having obtained these values, the library uses the standard `mmap` Linux system call to map to its address space the two pre-allocated buffers and the two BARs, represented by the arbitrary memory regions defined by the kernel module's header file.

Since the `ioctl` calls are simple read operations and do not acquire resources, `AvalonDevice`'s destructor only needs to unmap the four addresses, then close the file representing the FPGA.

The functionality described so far would be enough to implement GPU-mastered DMA transfers as was done for Windows. However, to expose to client applications the GPU pinning capability developed for the kernel module, a new function called `pinGpuBuffer` was added to the class `AvalonDevice`. This function takes a pointer to a GPU buffer and its size, and returns a structure representing the GPU pages where the buffer was pinned, including the page addresses and their count and size.

To pin the provided GPU buffer, the library uses the three remaining `ioctl` codes from Table 7. First the memory is pinned with `PIN_GPU`, which returns the number of pages that were pinned. The library then allocates an array large enough to hold that many page addresses, and uses `GET_PINNED_PAGES` to retrieve them. When the device is destroyed, the memory is automatically unpinned.

This GPU pinning functionality is not available on Windows, but to keep a consistent interface, the `pinGpuBuffer` function was also defined on that OS, but it simply throws an exception with an error message indicating the lack of support.

5.4 FPGA-mastered DMA benchmark implementation

To test the new driver options and the kernel module described in previous sections, and GPUDirect RDMA in general, a new pair of benchmark types was added to the benchmark application. The new benchmark type GR allocates and pins a buffer in the GPU, then instructs the FPGA to read data from it via the `AvalonDmaManager` class. The other new benchmark type, FR, does the same, except it instructs the FPGA to write to the pinned GPU buffer instead of reading from it.

An initial implementation used all page addresses obtained when pinning the GPU buffer, issuing a concurrent DMA transfer to each page, then waiting for all of them to be completed, as outlined in Listing 2. When testing this, the issues described in Section 5.2.5 were encountered. After failing to find the root cause of these issues but finding that treating the buffer as physically contiguous

gave the right results, the benchmark was modified under this assumption. After these changes, running the new benchmark options with verification enabled was always successful.

5.5 FPGA-mastered DMA performance

As with the Windows implementation, the benchmark was run for all available transfer types for 10 iterations each with verification enabled to ensure that all transfers were successful. At this point, it was found that some transfer types would not run at all.

On this Linux system, the CUDA function `cudaHostRegister` that tries to pin host memory fails with an "invalid argument" error code if the memory is already pinned. This means that the benchmark types FIR and GIR, which transfer data between the FPGA and the GPU indirectly through a pinned and registered host memory buffer, fail at the buffer initialization step on Linux because the buffer is already pinned. The equivalent benchmark types that do not call `cudaHostRegister`, FI and GI, run without issues.

The other two benchmark types that call `cudaHostRegister` are GFR and FGR, the two that carry out GPU-mastered DMA on Windows. The `cudaHostRegister` call issued by these benchmark types also fails with the same error code, meaning that the DMA approach implemented for Windows does not work on this Linux system. Since attempting to register pinned host memory also fails, it seems that, contrary to the case on Windows, Nvidia drivers need whatever Linux function they call to attempt to pin memory to succeed before they can register the memory. Further research is required to find out why this could be the case.

The benchmarks that were found to work correctly were run again with verification disabled for 1000 iterations each to obtain a stable average of their run time. The results are presented and analysed in the rest of this section.

Figure 3 shows the average time taken to transfer data from the FPGA to the GPU, both with DMA (FR) and without it (FI), as well as the two intermediate transfers when not using DMA (FH from the FPGA to host memory, and HG from there to the GPU). A linear fit for each type of transfer is presented as a dotted line of the same colour. Due to the log-log nature of the graph, the linear fits appear as curves with a horizontal asymptote corresponding to the constant term of the linear fit.

Figure 4 shows the same for transfers in the opposite direction. GR are DMA transfers from the FGPA to the GPU, GI indirect transfers through host memory, and GH and HF are the two half transfers from FPGA to host and from host to GPU.

As was the case for the Windows implementation, most curves follow their linear fits very closely, meaning that transfer time can be accurately estimated by the same linear model as $t = l + b^{-1}\mathrm{s}$, where t is the transfer time, l the latency, b the channel bandwidth, and s the transfer size. Table 6 summarizes the bandwidth and latency of each transfer type. As was the case on Windows, while bandwidths showed differences of no more than a few MB/s across runs, latencies were less consistent and thus are shown with less precision.

Since the limiting factor for large transfers is the bus bandwidth, it was expected that the effective bandwidth achieved by the benchmarks between host memory and either the FPGA or the GPU would be the same as on Windows. This was indeed the case for the FPGA, where measured effective bandwidth was the same on both platforms, up to sub-megabyte per second precision.

For transfers between GPU and host memory, the measured bandwidth was also very similar, although slightly faster on Linux by an insignificant margin.

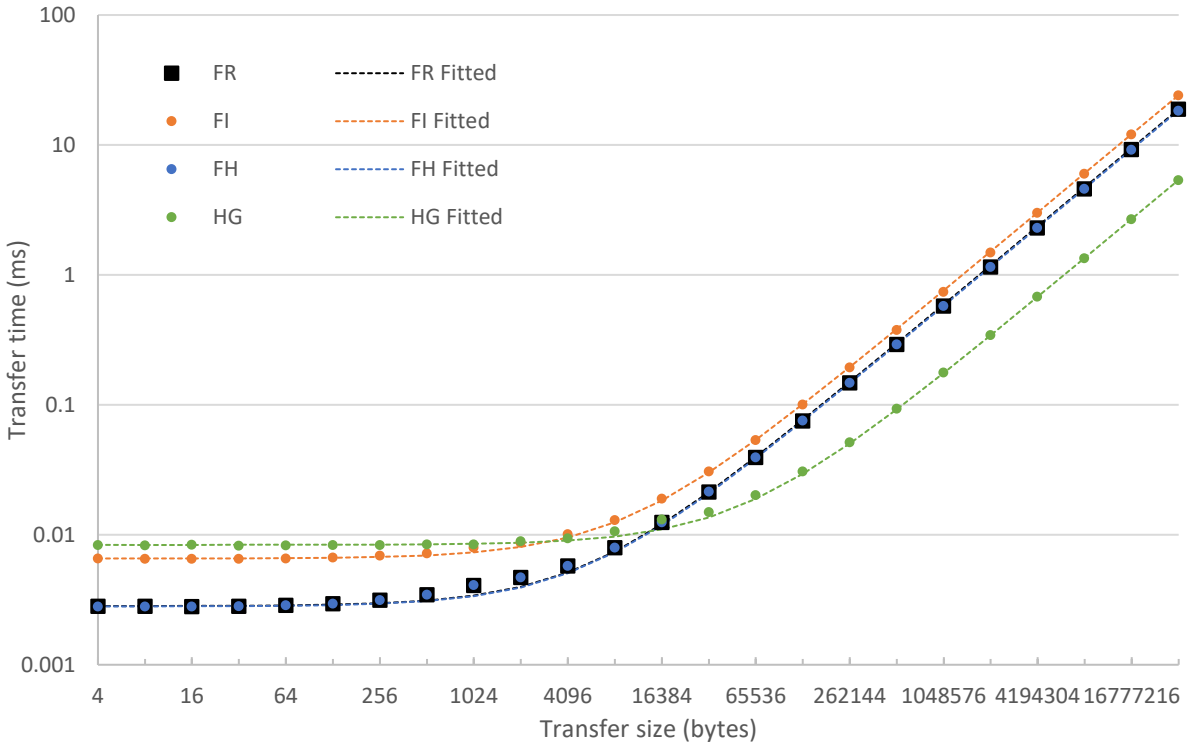

Figure 3. Transfer times from FPGA to GPU on Linux

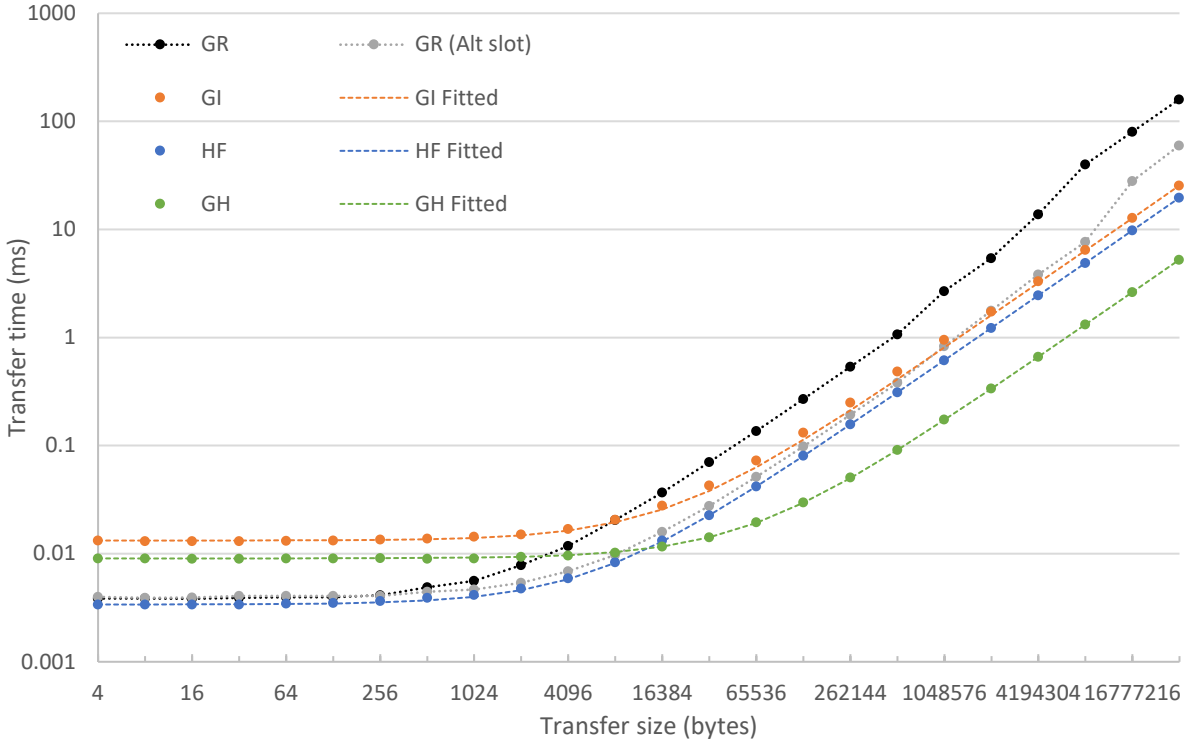

Figure 4. Transfer times from GPU to FPGA on Linux

Transfer type	Bandwidth (MB/s)	Latency (µs)
FI	1389	7
FR	1800	3
GI	1314	13
GR	N/A	4
FH	1817	3
HF	1706	3
GH	6389	10
HG	6243	8

Table 8. Linear model for transfer performance on Linux

The latencies of transfers between the FPGA and host memory were also the same as on Windows, in the range of 3-4 µs. Since all such transfers are handled by user-mode software and the hardware, it is natural that the OS would have little impact on their performance. GPU transfers, however, were found to have a latency of 8-10 µs on Linux compared to 38-40 µs on Windows. The reason for this difference could be in the implementation of Nvidia drivers, or in the cost of switching contexts between user mode and kernel mode on each OS.

As was also the case on Windows, indirect transfers between the two devices through host memory took exactly as long as the sum of the two half transfers, and their performance can be modelled accordingly my summing the latencies and the inverses of the bandwidths.

Getting to the performance of DMA transfers, the ones from the FPGA to the GPU took barely longer than those from FPGA to host memory, as demonstrated by the overlapping FR and FH curves. The measured latencies were the same, and the bandwidths faster by 1% in favour of transfers to host memory. Due to the tree topology of PCIe, there is no direct connection between the FPGA and the GPU, meaning that all data originating from the FPGA will always be limited by the slower FPGA-host link. This in turn means that matching the performance of transfers to the host is as good a result as can be expected.

DMA transfers in the opposite direction, from GPU to FPGA, had a completely different behaviour. Not only were they much slower than indirect transfers for transfer sizes above 8 KiB, they also became less efficient as transfer size increased. The peak throughput was achieved with transfers between 64 KiB and 512 KiB, at almost 500 MB/s. While this is slow compared to the available channel bandwidth of about 1855 MB/s, increasing transfer size dropped throughput to less than 209 MB/s at 8 MB per transfer or more. Due to this behaviour, transfer times for these operations cannot be accurately predicted by a simple linear model, so there is no single bandwidth value that can describe the performance and no fitted curve is shown in Figure 4. The latency can still be approximated as the transfer time for the smallest transfer.

Given that transfers from the FPGA to the host do not exhibit this kind of behaviour, and that at this point, the host is not involved at all, this lack of performance is likely to be caused by the GPU, or by the PCIe bus itself. The GPU has been proven to be able to handle much larger transfers

without losing efficiency when transferring to the host, but such transfers between the host and the GPU are never reads for the GPU. Since those transfers are mastered by the GPU, a transfer to the host consists of MWr TLPs from the GPU to the host, while transfers in the opposite direction consist of MRd TLPs, also from the GPU to the host. When the FPGA is mastering a transfer from the GPU to the FPGA, it issues MRd TLPs to the GPU. Since these never occur for the common case of transfers between the GPU and the host, it is likely that their handling by the GPU PCIe hardware is not as well optimized as other types of TLPs, leading to the encountered bottleneck.

As was done on Windows, the transfers were measured again with the cards connected to different slots. None of the transfers were affected, except the DMA from GPU to FPGA that was originally slow. Its new performance is presented in Figure 4, labelled as “GR (Alt slot)”. While the same bottleneck was found for large transfer sizes, overall throughput was increased by a factor between 2 and 3 for all transfer sizes larger than 4096 bytes. This was enough to improve the performance of +--some DMA transfers over indirect ones, up to the point where the bottleneck discussed above starts degrading performance for transfers larger than 512 KiB.

The benchmark was also run with a max payload size reduced to 128 bytes. As was the case on Windows, this caused a bandwidth reduction of 10% from the FPGA to the host and 5% in the opposite direction. DMA transfers to the GPU were affected in the same proportion as to the host, while indirect transfers were affected less because the FPGA max payload size has no impact on host-GPU transfers.

Finally, it was attempted to run the benchmark with the GeForce 950 card, but as stated in Nvidia documentation, GPUDirect RDMA is not available for GeForce cards [2], so no DMA was possible.

Chapter 6

Conclusions

In this dissertation, two possible ways to implement DMA between FPGAs and GPUs were described and analysed. Sections 6.1 and 6.2 summarise the findings obtained for each of them, and Section 6.3 gives recommendations on when each could be applied in production code. Finally, Section 6.4 outlines further research that could be carried out to address questions raised by this dissertation.

6.1 GPU-mastered DMA

The first DMA implementation presented relies on undocumented functionality of Nvidia's CUDA runtime and drivers, and only works on Windows. Although implementing it was simple, the throughput obtained was on par with indirect transfers in the direction from GPU to FPGA, and lower than them by up to 37% at the largest transfer size in the opposite direction. The performance of the latter direction was also found to be very dependent to the way in which the cards are physically connected to the motherboard. Slot configurations that were the same in terms of PCIe generation and number of lanes nonetheless changed the throughput from 295 MB/s to 885 MB/s.

The reason for the lack of throughput of transfers from GPU to FPGA was not found conclusively. The similarity between the transfer time curves for DMA and indirect transfers suggest that perhaps the supposedly DMA transfers are being internally buffered by Nvidia drivers. Inspection of the PCIe TLPs received by the FPGA could reveal whether this is the case. For the same direction, the same method was applied more successfully by Bitner and Ruf with hardware from previous generations, obtaining a throughput increase of up to 34.6%, depending on transfer size.

DMA transfers in the opposite direction, from FPGA to GPU, were always slower than indirect ones, both in the results presented in this dissertation and by Bitner and Ruf. As described in Section 2.3, PCIe memory read operations are inherently slower than writes. This is the case even for transfers from the root complex to endpoint (host-to-GPU and host-to-FPGA), where reads are between 3% to 7% slower than the equivalent writes, but much more so in endpoint to endpoint transfers. While with this implementation read operations can only be executed to transfer from the FPGA to the GPU, the other presented DMA implementation presented in this dissertation performs reads in the opposite direction with equally poor results. This suggests that the factor limiting performance is not related to the FPGA or the GPU, but to the PCIe bus itself.

Another important observation about this DMA implementation is that it did not work on Linux. The `cudaHostRegister` call that enables Nvidia drivers to directly access FPGA memory fails on

Linux with an “invalid argument” error message. The same happens when attempting to register host memory that has already been pinned. Pinning memory is a platform-dependent operation, so it’s possible that Windows reports a success while Linux reports a failure when attempting to pin non-swappable memory, or that Nvidia drivers are simply handling such failures differently on the two operating systems.

6.2 FPGA-mastered DMA

The second DMA implementation makes use of Nvidia’s GPUDirect RDMA API to obtain the bus address of a buffer in GPU memory, and have the FPGA access it directly. Since this API is only available for Linux and Nvidia Quadro or Tesla GPUs, this implementation is only applicable for that OS and hardware combination. This limitation notwithstanding, the throughput obtained for transfers from the FPGA to the GPU was the same as that from the FPGA to the host, which is as good a result as can be expected, as mentioned in Section 5.5.

Transfers in the opposite direction, however, exhibited mixed results and a behaviour similar to that of reads from the FPGA to the GPU with the first DMA method. The throughput obtained varied significantly depending on which PCIe slots the devices were connected to, even though the slots were the same generation and had the same number of lanes. The throughput obtained with one of the configurations was higher than that of the other configuration for all transfer sizes greater than 4096 bytes by a factor between 2 and 3.

A notable difference with the read performance of the GPU-driven approach was that for transfers up to 512 KiB, the GPUDirect RDMA-based DMA provided higher throughput than indirect transfers. Above that size, however, these DMA transfers started losing efficiency, with throughput at the largest transfer sizes falling to less than half of its peak value at 512 KiB. This anomaly cannot be explained simply by the inefficiency of two-step PCIe reads instead of single-step writes, because that alone would not cause throughput to decrease as transfer size increases. Perhaps a bottleneck is introduced by the tracking of in-flight read transactions on the FPGA, or by the mapping of GPU physical memory to bus addresses in the GPU.

It is important to note that a bug was encountered when pinning GPU memory, wherein only the first GPU page address would be correct. The cause of this would need to be identified before using GPUDirect RDMA in production code, but since this API is officially supported by Nvidia, this should not be a significant problem.

6.3 Recommendations

The two presented DMA implementations introduce complexity, limitations and unpredictability. For this reason, a decision about employing these DMA techniques is not straightforward, and should carefully balance the potential performance benefits with the increased development and support costs.

The best case be made for applications where professional-grade GPUs are already being used under Linux. If the transfers are mostly from the FPGA to the GPU, DMA based on GPUDirect RDMA would provide a performance increase at any transfer size, and with support from the hardware vendor. If the transfers are primarily in the opposite direction, testing would be required to ensure that performance is not affected by the bottleneck encountered at larger transfer sizes.

If the application is not based on professional-grade GPUs, economic analysis would be required to determine whether the performance offered by DMA can justify the cost difference between consumer and professional GPUs.

If the application is based on Windows, GPUDirect RDMA is not applicable. The alternative, with GPU-mastered DMA, was not found to provide an increase in throughput for any transfer size and direction. If Bitner and Ruf's results cannot be reproduced, this DMA implementation is hard to recommend for any application due to its lack of throughput improvements and its brittleness caused by relying on undocumented functionality.

In any case, the maximum possible benefit obtained from DMA depends on the relative speed of the connection of each of the two devices. In the best case, when both the FPGA and the GPU are connected to buses of the same generation and link width, DMA transfers could potentially take half as long as indirect ones. However, for asymmetric connections such as the test system used for this project, where the GPU's bus has a theoretical bandwidth 4 times higher than the FPGA's, transfers can never become faster than allowed by the slowest link. If it takes 4 ms to transfer from FPGA to host, and 1 ms from host to GPU, DMA will probably not improve transfer times by more than 1 ms out of the total 5 ms. In such cases, improving the bus speed of the slowest device should probably take priority over implementing DMA.

For applications where latency is not as significant as throughput, executing multiple transfers concurrently could achieve a level of bus utilization perhaps even higher than that of DMA. This would be an extension to the technique used to hide transfer times in CUDA programming by employing asynchronous command streams. If the application does not offer sufficient parallelism to allow this, but each transfer is sufficiently large, transfers could be split into smaller ones and treated as proposed by Gillert.

Finally, for applications where latency is significant, Linux should be the platform of choice due to its much lower latencies overall. Using GPUDirect RDMA could further reduce latencies by about half.

6.4 Future work

In this dissertation, some questions were raised which would need further research to answer conclusively. One of the most important would be to determine why the GPU-mastered DMA does not improve throughput over indirect transfers. Inspection of the PCIe traffic, perhaps by recording all received TLPs on the FPGA, could verify whether these transfers are indeed working as DMA, or using an intermediate buffer in host memory.

Also related to GPU-driven DMA, it was not determined why it does not work on Linux while it works on Windows. This would probably require reverse engineering some CUDA functions to analyse how they achieve memory pinning and registration.

Another important open question is the poor throughput of DMA read operations from device to device. Whether the transfers are mastered by the GPU or the FPGA, reads are not only much less efficient than the equivalent writes, but also slower than reads from the host. Since this issue seems to be caused by the bus instead of the devices connected to it, PCIe hardware inspection tools could provide the necessary data.

On the matter of PCIe hardware, a significant performance difference was encountered by connecting devices to different PCIe slots rated for the same speeds. Finding out the cause of this

difference would allow predicting configurations that will yield poor performance, simplifying hardware acquisition and configuration.

On the topic of DMA performance, other avenues to obtain GPU memory bus addresses instead of GPUDirect RDMA could be explored. While Thoma et al.'s research with noveau and gdev did not yield conclusively good performance results for large transfers, different open source alternatives or newer versions of the same could make an improvement.

Finally, the software developed for this project would need to be improved in at least two ways to be used for production purposes. First, the FPGA needs to be reset by software when there is an error. The current implementation requires it to be reprogrammed and the system rebooted, which is not acceptable in a production environment. Second, the user-mode driver library needs to gracefully handle DMA descriptor exhaustion. If a client application attempts to start more concurrent DMA operations than there are available descriptors, the library should either enqueue the requests and process them as descriptor become available, or wait for a descriptor to become available before returning control to the client application.

References

[1] R. Bittner and E. Ruf, "Direct GPU-FPGA Communication Via PCI Express," 17 5 2013. [Online]. Available: https://www.microsoft.com/en-us/research/wp-content/uploads/2016/02/2012062520UCAA2012_Bittner_Ruf_Final-1.pdf. [Accessed 19 3 2017].

[2] Nvidia Corporation, "Developing a Linux Kernel Module using GPUDirect RDMA," 2017 1 2017. [Online]. Available: http://docs.nvidia.com/cuda/gpudirect-rdma. [Accessed 19 3 2017].

[3] Nvidia Corporation, "Nvidia NVLink High Speed Interconnect," February 2017. [Online]. Available: http://www.nvidia.com/object/nvlink.html. [Accessed 13 August 2017].

[4] Nvidia Corporation, "CUDA C Best Practices Guide," 12 1 2017. [Online]. Available: http://docs.nvidia.com/cuda/cuda-c-best-practices-guide. [Accessed 19 3 2017].

[5] National Instruments, "Introduction to FPGA Technology: Top 5 Benefits," 16 April 2012. [Online]. Available: http://www.ni.com/white-paper/6984/en/. [Accessed 13 August 2017].

[6] Embedded Micro, "How Does an FPGA Work?," [Online]. Available: https://embeddedmicro.com/tutorials/mojo-fpga-beginners-guide/how-does-an-fpga-work. [Accessed 13 August 2017].

[7] C. Grozea, Z. Bankovic and P. Laskov, "FPGA vs. Multi-Core CPUs vs. GPUs: Hands-on Experience with a Sorting Application," Fraunhofer Institute FIRS, Berlin, Germany, 2010.

[8] IEEE, 1800-2012 - IEEE Standard for SystemVerilog--Unified Hardware Design, Specification, and Verification Language, IEEE, 2013.

[9] Intel Corporation, "Introduction to Intel FPGA IP Cores," 8 May 2017. [Online]. Available: https://www.altera.com/documentation/mwh1409960636914.html. [Accessed 13 August 2017].

[10] Xilinx, "Intellectual Property," [Online]. Available: https://www.xilinx.com/products/intellectual-property.html. [Accessed 13 August 2017].

[11] M. Jackson and R. Budruk, PCI Express Technology: Comprehensive Guide to Generations 1.x, 2.x and 3.0, MindShare, Inc., 2012.

[12] J. Corbet, A. Rubini and G. Kroah-Hartman, Linux Device Drivers, 3rd Edition, O'Reilly Media, 2009.

[13] H. G. Cragon, Computer Architecture and Implementation, Cambridge University Press, 2000.

[14] Microsoft, "Choosing a driver model," 4 May 2017. [Online]. Available: https://docs.microsoft.com/en-us/windows-hardware/drivers/gettingstarted/choosing-a-driver-model. [Accessed 13 August 2017].

[15] Jungo Connectivity Ltd, "WinDriver PCI/ISA Quick-Start Guide," 2017. [Online]. Available: https://www.jungo.com/st/support/documentation/windriver/wdpci_qsg.pdf. [Accessed 13 August 2017].

[16] Y. Thoma, A. Dassatti and D. Molla, "FPGA²: An Open Source Framework for FPGA-GPU PCIe Communication," in *2012 International Conference on Reconfigurable Computing and FPGAs, ReConFig 2013*, Cancun, Mexico, 2013.

[17] A. Gillert, "Direct GPU-FPGA Communication," Technische Universität München, Munich, Germany, 2015.

[18] Altera, "PCI Express Compiler User Guide," 2017. [Online]. Available: https://www.altera.com/content/dam/altera-www/global/en_US/pdfs/literature/ug/ug_pci_express.pdf. [Accessed 13 August 2017].

[19] Altera, "V-Series Avalon-MM DMA Interface for PCIe Solutions User Guide," 2017. [Online]. Available: https://www.altera.com/content/dam/altera-www/global/en_US/pdfs/literature/ug/ug_pcie_avmm_dma.pdf. [Accessed 13 August 2017].

[20] Nvidia Corporation, "CUDA Runtime API," 23 June 2017. [Online]. Available: http://docs.nvidia.com/cuda/cuda-runtime-api. [Accessed 13 August 2017].

[21] Jungo Connectivity Ltd., "WinDriver PCI/ISA User's Manual," 2017. [Online]. Available: https://www.jungo.com/st/support/documentation/windriver/12.4.0/wdpci_manual.pdf. [Accessed 13 August 2017].

Appendix A

Building and running the code

The following instructions are reproduced from the Readme.txt file included with the code that supports this project.

Project modules

The code developed during this project consists of 4 modules, each located in a different subdirectory within the project's root directory.

- "AvalonMM" FPGA design ($PROJECT_ROOT/avalonmm)
    - Configures the FPGA to respond to memory read and write operations through PCIe, as well as to give it a DMA engine to act as bus master. This design is based on Altera's "Avalon-MM DMA" IP core.
    - All files in this module were generated through Quartus's GUI utilities, so although most of them are text-based, they are not necessarily well-formatted.
- Kernel module ($PROJECT_ROOT/avalonmm-kernel-module)
    - Provides a kernel-mode device driver for the FPGA design. Only applicable for Linux.
    - The provided makefile builds the module through Linux's 'kbuild' framework. The scripts 'load' and 'unload' can be used to load and unload the module once built. The 'install' script unloads the previous build of the module, if any, then rebuilds the module and loads it. Loading and unloading modules requires root privileges, so the 'install' script should be run as a superuser.
- User-mode driver library ($PROJECT_ROOT/avalonmm-driver)
    - This C++-2017 library provides a high-level API to access features of the FPGA design.
    - It can be built on its own with the makefile in its directory, or together with the benchmark application with the makefile in the project's root directory.
- Benchmark application ($PROJECT_ROOT/benchmark)
    - A C++-2017 application that uses the user-mode driver library and CUDA to implement multiple data transfer benchmarks.

- The makefile to build it requires a path to the user-mode driver library, so it is more convenient to invoke it from the higher-level makefile in the project's root directory.

Building prerequisites

On Windows:

- Quartus Prime 16.1.2
- Visual Studio 2017
- CUDA 8.0
- WinDriver 12.4.0
- Environment variable: WD_BASEDIR pointing to WinDriver's installation directory

On Linux

- Quartus Prime 16.1.2
- GCC 6
- CUDA 8.0
- Environment variable: NVIDIA_KERNEL_SRC pointing to Nvidia's kernel module source files, obtained during CUDA installation

Building procedure

To build the project, the following steps must be completed in order:

1- Compile the FPGA design.

Open the Quartus project file $PROJECT_ROOT/avalonmm/avalonmm.qpf with Quartus and click on the compile button.

This will generate a binary file in $PROJECT_ROOT/avalonmm/output_files/avalonmm.sof.

2- Program the FPGA with the compiled design.

From Quartus, open the Programmer utility.

Select the FPGA and the binary file generated in Step 1, then click start.

3- Load the FPGA's driver.

On Windows: From the device manager, right click on the FPGA, select "Update driver".

Select the provided $PROJECT_ROOT avalonmm-driver/AvalonMM.inf driver file.

On Linux: Change directory to $PROJECT_ROOT/avalonmm-kernel-module.

Run `sudo ./install`. This will compile and load the kernel module.

4- Build the user-mode library and benchmark application.

On Windows: Open the avalonmm.sln solution file with Visual Studio and compile it.

On Linux: Switch to the project's root directory and execute `make`.

On both operating systems, the binary files will be placed in the $PROJECT_ROOT/bin directory.

Enabling DMA logging

All transactions executed by the AvalonDmaManager class can be logged to the dma.log file in the current directory from which the benchmark is run. This is controlled by the `LOG_DMA` preprocessor macro.

On Windows, the LOG_DMA macro is defined by default for the Debug configuration, but not for the Release one. This can be safely altered by editing the "Preprocessor Definitions" field in the 'avalonmm-driver' project properties pane.

On Linux, the macro is not defined by default. It can be defined by adding `'CXX_FLAGS_ADD=-DLOG_DMA'` to make's command line.

Running the benchmark

After building the application, the benchmark can be run from the $PROJECT_ROOT/bin directory. It takes 3 parameters. The first parameter determines the transfer type to measure, the second one the number of iterations to execute, and the third one whether transfers should be verified.

Possible values for the transfer type are as follows:

Code	Description
FI	DMA from the FPGA to host memory, then `cudaMemcpy` from host memory to GPU memory.
FIR	As FI, but register host memory as pinned so that `cudaMemcpy` can use DMA.
FG	Use `cudaMemcpy` to transfer directly from FPGA memory to the GPU.
FGR	As FG, but register FPGA memory as pinned so that `cudaMemcpy` can use DMA.
FH	DMA from the FPGA to host memory.
FM	Non-DMA transfer from FPGA memory to host memory, using memory-mapped IO.
FR	DMA transfer from the FPGA to GPU memory using GPUDirect RDMA.
GI	Use `cudaMemcpy` from GPU memory to host memory, then DMA from host memory to the FPGA.
GIR	As GI, but register host memory as pinned so that `cudaMemcpy` can use DMA.
GF	Use `cudaMemcpy` to transfer directly from the GPU to FPGA memory.
GFR	As GF, but register FPGA memory as pinned so that `cudaMemcpy` can use DMA.
GH	Use `cudaMemcpy` to transfer from the GPU to host memory.
GR	DMA transfer from GPU memory to the FPGA using GPUDirect RDMA.
HF	DMA from host memory to the FPGA.
HG	Use `cudaMemcpy` to transfer from host memory to the GPU.
HM	Non-DMA transfer from host memory to FPGA memory, using memory-mapped IO.

The number of iterations defaults to 1000, but can be increased to obtain more consistent results, or decreased for shorter run times.

Verification can be enabled by passing 'V' as the third parameter. Each transfer is verified by copying the data back from the destination buffer to a buffer in host memory and compared with the original data. If the data does not match, an error is printed to stderr. This verification is meant as a debugging tool and is not at all optimised (e.g. it uses MMIO to copy from the FPGA, and a simple `cudaMemcpy` call without pinning memory to copy from the GPU), because simplicity and robustness were desired over performance. Transfer times should therefore be benchmarked with verification disabled.

While running, the benchmark application will output the average time taken for transfers of each size, and calculate the corresponding throughput by dividing the transfer size by the elapsed time.

Appendix B

Project review

During the Project Preparation course, a detailed plan was formulated for this project. Actual execution, however, diverged significantly from the plan in two main ways. First, the original plan was to implement the GPU-mastered DMA operations, then apply them to an Optos application to measure real-world performance gains. After encountering the disappointing results obtained with this DMA implementation, applying it to a real application would have been pointless.

Nevertheless, one of the main risk factors identified during Project Preparation was the possibility of not being able to reproduce Bitner and Ruf's results. The proposed mitigation strategy was to procure the necessary hardware to test GPU Direct RDMA instead, so by the time the performance issues were encountered, it was simple enough to alter the project's goals. However, this brought an alteration to the project's schedule large enough that there was no time left to try to adapt a real world application, so all results presented were based on a synthetic benchmark.

The second important deviation from the project was in how the writing of the dissertation was handled. It was planned to be done concurrently with the development of the code, but the uncertainty added by the first issue above would have meant that many sections would need to be rewritten or comprehensively revised. Instead, the writing was left for the last weeks of the project, once the scope and contents of the dissertation were more clearly defined.

As for the outcome of the project, its main success was producing reference implementations for two different DMA techniques. While some performance and correctness issues still remain unanswered, these reference implementations could prove useful when Optos applications start requiring more bandwidth than they currently have available.

Finally, as a learning opportunity, the project can be counted as a resounding success, although not in the fields originally imagined. Before the first formal meetings, the project was chosen because of the possibility of working with GPGPU, but instead there was little of that, and the majority of the research time was spent on FPGAs and PCIe. Due to the more niche nature of FPGAs, having the opportunity to work with them was probably more valuable than a GPGPU optimization project would have been.