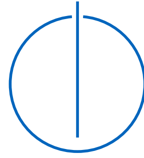


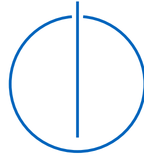
DEPARTMENT OF INFORMATICS

TECHNISCHE UNIVERSITÄT MÜNCHEN

Bachelor's Thesis in Informatics: Games Engineering

Analysis of data movements over the PCIe bus in heterogeneous computer systems

Ruilin Qi



DEPARTMENT OF INFORMATICS

TECHNISCHE UNIVERSITÄT MÜNCHEN

Bachelor's Thesis in Informatics: Games Engineering

Analysis of data movements over the PCIe bus in heterogeneous computer systems

Analyse der Datenbewegungen auf dem PCIe Bus in heterogenen Rechnersystemen

Author:	Ruilin Qi
Supervisor:	Prof. Dr. Martin Schulz
Advisor:	Stepan Vanecek, M.Sc.
Submission Date:	15th April 2022

I confirm that this bachelor's thesis in informatics: games engineering is my own work and I have documented all sources and material used.

Munich, 15th April 2022 Ruilin Qi

Acknowledgments

I would like to thank my advisor, Mr. Stepan Vanecek, for his help and guidance with this project. Furthermore, I'd like to thank my friend Kris, for both the emotional support and help with proofreading. Finally, I'd like to thank everyone who supported me over the last months, especially close friends and family.

Abstract

Data movements, and the interconnects in heterogeneous systems, will soon become the bottleneck of computation speeds due to the shift away from general-purpose processors toward specialized architectures and accelerators.

The aim of this bachelor's thesis is to gain insight into data movements over the PCIe bus in heterogeneous computer systems, and to be more specific, data travel between the CPU and GPU. The thesis describes the development of a set of CUDA-based tools to both assess a PCIe link's capabilities and to monitor other programs' PCIe link activity.

The first tool aims to benchmark a given system's PCIe link capability, namely delay and bandwidth. The benchmark is capable of determining peak bandwidths and the saturation batch size accurately and shows that the transfer duration does not scale linearly with the amount of PCIe packets sent.

The second tool aims to monitor a given program's PCIe activity using NVIDIA's NVML library, which has counters to monitor the PCIe link throughput. The program accurately depicts the PCIe link activity of the program it is monitoring but struggles with short memory transfers due to the counters' update frequency.

The third tool additionally aims to detect shorter memory transfers, improving on the major drawback of the NVML approach. This is done by putting the PCIe link under load by copying small chunks of memory and monitoring the bandwidths, with a lower bandwidth indicating PCIe link activity. This tool can detect shorter memory copy operations at the cost of introducing significant overhead to the program it is monitoring.

Limitations for each tool are listed, although workarounds for these limitations are challenging to find due to the lack of documentation for the libraries used in the development of each tool.

Taken together, the developed tools provide a unique and different approach to the monitoring process of data movements over the PCIe bus in heterogeneous computer systems.

Contents

Acknowledgments **iii**

Abstract **iv**

1 Introduction **1**
1.1 Motivation 1
1.2 Goals and Aims 1
1.3 Structure and Approach 2

2 Background **4**
2.1 HPC 4
2.2 Graphics Processing Units 4
2.2.1 What are GPUs 4
2.2.2 Uses of GPUs 4
2.2.3 GPU Memory 5
2.3 CUDA 5
2.3.1 Kernels and Scalability 5
2.3.2 Memory Management 6
2.3.3 De-allocation 9
2.3.4 NVML 10
2.4 PCI-Express 10
2.4.1 Key Features 11
2.4.2 Functionality 12
2.4.3 Topology and Communication 14
2.4.4 Revisions and Further Specifications 16
2.5 System Specifications 17
2.5.1 P6000 17
2.5.2 Ice1 17
2.6 Kernels 17
2.6.1 Vector Add 17
2.6.2 Matrix Multiplication 18

3 Bandwidth Benchmark **19**
3.1 Concept . . . 19
3.2 Goals . . . 19
3.3 Implementation . . . 20
3.3.1 Bandwidth Measurements . . . 20
3.3.2 Delay . . . 20
3.3.3 Benchmark . . . 20
3.4 Results . . . 21
3.4.1 P6000 . . . 21
3.4.2 Ice1 . . . 23
3.5 Discussion . . . 26
3.5.1 Successes . . . 26
3.5.2 Shortcomings . . . 26

4 NVML Counters **28**
4.1 Concept . . . 28
4.2 Goals . . . 28
4.3 Implementation . . . 29
4.3.1 Parallelism . . . 29
4.3.2 Monitoring . . . 29
4.4 Results . . . 31
4.4.1 Vector Add . . . 31
4.4.2 Matrix Multiplication . . . 32
4.5 Discussion . . . 39
4.5.1 Successes . . . 39
4.5.2 Shortcomings . . . 40

5 Link Saturation **41**
5.1 Concept . . . 41
5.2 Goals . . . 41
5.3 Implementation . . . 42
5.3.1 Parallelism . . . 42
5.3.2 Monitoring . . . 42
5.4 Results . . . 44
5.4.1 Vector Add . . . 44
5.4.2 Matrix Multiplication . . . 49
5.5 Discussion . . . 54
5.5.1 Successes . . . 54
5.5.2 Shortcomings . . . 54

6 Further Reading 55

7 Summary 56
7.1 Successes . . . 56
7.2 Problems . . . 57
7.3 Conclusion . . . 57
7.4 Outlook . . . 58

List of Figures 59

List of Tables 61

Bibliography 62

1 Introduction

1.1 Motivation

Moore's law, based on the paper that Gordon E. Moore wrote in 1965, predicted that the number of transistors in a dense integrated circuit would double every two years. [1] Robert H. Dennard made the observation that a transistor's power use stays in proportion with the size of said transistor. [2] These two observations are the current foundation of the computing ecosystem, where transistor sizes are constantly shrinking and computational capacities are constantly growing. However, this is set to end soon as transistor sizes are approaching atomic scale in the next few years. [3] Even now, this end is fast approaching as Taiwan Semiconductor Manufacturing Company, a chip foundry, has plans to start volume production of its 3nm technology as early as the second half of 2022. [4] As such, there will be, and already has been to some degree, a paradigm shift away from general-purpose processing units and towards more specialized architectures to maintain performance improvements. This paradigm shift has lead to heterogeneous computing and increased parallelism. As result, this also leads to data movements becoming increasingly costly, compared to the operations on said data [5]. Consequently, that data transfers, and the interconnects in heterogeneous systems, will soon become the bottleneck for computation speeds as moving data to the relevant processing units becomes more important. As such, it is increasingly important to gain insight into the data movements in heterogeneous systems to optimize both current and future software.

1.2 Goals and Aims

The goal of this thesis is to gain insight on data movements in a heterogeneous system. To be more specific, the data movements over a PCIe bus in a heterogeneous system. This is done by developing a tool, or a set of tools, to monitor the PCIe link in a heterogeneous system. To be more specific, the tool(s) should meet a set of abstract requirements, listed below, to set it apart from existing tools:

- automated
- as few requirements as possible

- low overhead
- no need to modify existing programs
- should gain insight to PCIe link activity

The majority of the available tools, which will be further touched upon in Chapter 6, have specific hardware requirements and require a significant amount of human input to gather data, necessitating the requirements for the lack of requirements and automation. The requirement for low overhead is present to facilitate an accurate representation of a program's execution times. As the end user is not likely to have access to the source code of a given program, the programs should not need to be modified in order for the tools to work. Due to the different approaches described in this thesis, these requirements will be further discussed and expanded in the relevant chapters.

1.3 Structure and Approach

The introduction starts by explaining the primary motivation of the thesis. Then, the goals and aims of the thesis are to be defined in further detail, along with a set of abstract requirements for the tool being developed. The introduction closes with a brief description of the structure and approach of this thesis, as described in this section.

The next chapter should briefly introduce the concept of High-performance computing (HPC), graphics processing units (GPUs), NVIDIA's CUDA API, and the Peripheral Component Interconnect Express (PCIe) architecture. Chapter 1 should act as a foundation for the remainder of this thesis and introduce most of the concepts mentioned in future chapters. This chapter also introduces the testing environment of the tools developed in the further chapters.

Chapter 3 aims to develop a tool to assess the bandwidth and delay of a PCIe link of a given system, both to validate the specification and to provide a more accurate reading on the real-world performance of the interconnect, which is not only limited by the technical specifications.

Chapter 4 describes monitoring a program's PCIe link activity by utilizing NVIDIA's NVML library and the counters it exposes. The chapter first introduces the basic concept and concrete goals for the tool being developed. After that, the implementation is elaborated upon, going into further detail regarding the monitoring concept and parallelism. Finally, the results and findings are described and discussed in further detail.

Chapter 5 is about utilizing PCIe link saturation to determine PCIe link activity. The structure of this chapter is similar to Chapter 4, leading with the concept and concrete goals, following that with the implementation, results, and discussion.

Chapter 6 touches upon similar papers and approaches to both give a brief insight into the state-of-the-art and to serve as a target for evaluation purposes.

Chapter 7 summarizes and briefly discusses the contents of this thesis and evaluates the developed tools against the state-of-the-art discussed in the last chapter. Finally an outlook for further development and research is given to conclude the thesis.

2 Background

2.1 HPC

High-performance computing (HPC), leverages the compute capacity of supercomputers or computer clusters to solve problems that are highly complex in nature [6]. A computer cluster consists of many different computers (nodes) that are interconnected with high-speed, low-latency interconnects. Each node contains a set of similar components a desktop or laptop PC would contain, such as a CPU, RAM, and storage [7]. It should be noted that modern CPUs, especially ones utilized in HPC applications, usually contain multiple physical cores and multiple hardware threads to enable some amount of parallel processing. [example maybe?] Some nodes, just like some PCs, also have a dedicated graphics processing unit (GPU) to accelerate certain types of workloads [7].

2.2 Graphics Processing Units

2.2.1 What are GPUs

To begin with, it should be noted that the term graphics processing unit (GPU) does not equate to a graphics card. GPUs are specialized processing units primarily designed for parallel processing and accelerating workloads that require parallel processing [8]. A graphics card, on the other hand, is the add-in card that features a PCI-Express link to facilitate communication between CPU and GPU, dedicated memory and power delivery for the GPU, and the GPU itself. There are also integrated GPUs, which can be embedded alongside the CPU. These integrated GPUs are usually less powerful compared to discrete GPUs [8].

2.2.2 Uses of GPUs

GPUs originally began as, as their names suggest, dedicated graphics accelerators optimized for floating-point operations, which are essential to 3D graphics rendering. They were initially developed as a hardware pipeline with fixed functionality, namely to render graphics. Over the years, GPU architecture has evolved from essentially being

an integrated frame buffer into a set of general-purpose, highly parallel, programmable processing cores, enabling more general-purpose computation [9]. Today, a GPU is more of an accelerator for many different use-cases and workloads. Examples for personal use include gaming, video editing, and content creation [8]. On the scientific side, GPUs are frequently used to accelerate workloads that require parallel computing, such as machine learning, fluid dynamics, and data science [10].

2.2.3 GPU Memory

Addressing GPU memory, assuming that the GPU is connected via PCI-Express and has its own dedicated video memory, works in the same way as addressing memory in other PCIe devices, which will be briefly touched upon in Section 2.4.3. However, GPU memory usually has higher throughput bandwidths compared to conventional RAM of a similar period. As example, current top of the line consumer grade graphics cards from NVIDIA are equipped with GDDR6X memory, which has a theoretical maximum system bandwidth of one terabyte per second. [11], [12] On the other hand, state of the art main memory, currently DDR4, is limited to a bandwidth of about 35 gigabytes per second [13].

2.3 CUDA

As GPUs evolved into more general-purpose processing units over the years, there has been a growing need for an application programming interface (API) that enables general-purpose programming on GPUs.

CUDA is a closed source API developed and maintained by NVIDIA for general-purpose GPU computing for their GPUs and graphics cards. It is designed to work with C++ and Fortran and comes with a set of GPU-accelerated libraries, optimization tools, debugging tools, and a C++ compiler [10]. Some sample libraries include: linear algebra, signal processing, and image processing [14]. For this thesis, only the C++ version of CUDA is discussed in further detail.

2.3.1 Kernels and Scalability

CUDA uses kernels, which are an extension to standard C++ functions, scheduled and executed on a GPU. Kernels, when called, are executed N times by N different GPU threads. This enables heterogeneous programming, which allows serial code to run on the host - the CPU - and parallel code, the kernels, to run on the GPU, thereby leveraging the GPU's increased capabilities for parallel computing to accelerate the workload. As modern CPUs are also capable of parallel processing, this is more

of a practice than a rule, and kernels are usually specific workloads that are highly parallel in nature, such as vector and matrix operations. The kernel is executed on a thread, many of which make up a block, many of which, in return, make up a grid, the dimensions of which are defined by the user upon calling the kernel in the source code. Different blocks can be executed in parallel, or in sequence, in any order, on any of the multiprocessors of a GPU, which enables automatic scalability as the compiled program can run irrespective of the amount of multiprocessors present on the GPU [15].

2.3.2 Memory Management

The memory management functionalities that CUDA provides include allocation, de-allocation, and data transfer. CUDA assumes that the CPU and GPU maintain separate memory spaces, and is able to manage both host and device memory [15].

Host Memory

The degree to which CUDA can manage the main memory is limited, and memory allocation is mostly handled by C++ with the function shown in Figure 2.1 [16]. C++, by default, allocates pageable memory, which means that the data can be paged in and out of RAM into a secondary storage device [17].

The function returns a pointer of the type *void* and has *size* as an argument, indicating the size in bytes the function is to allocate [16]. *Size_t* is an unsigned integral type used to represent the size of any object in bytes [18].

CUDA, on the other hand, can allocate page-locked memory, also called 'pinned' memory with the function shown in Figure 2.2. Page-locked memory cannot be paged out of the main memory, which allows for faster access and higher bandwidths when transferring data. It is also worth noting that the GPU cannot access data directly from pageable memory. As such, attempting to copy pageable memory to the GPU will first move it to a pinned memory section before copying the data to the GPU, as shown in Figure 2.3 [19].

The pointer *ptr* points to the location the memory will be allocated at, while *size* is the size of the memory that is to be allocated, in bytes. $flags$ indicates the access permissions for the chunk of memory, defaulting to accessible by any stream from any device. The return type is a *cudaError*, for debugging and error-tracking purposes [20]. It is also worth mentioning that allocating a large amount of page-locked data may compromise system stability and influence system performance, and as such should be done carefully [19].

```
void* malloc(size_t size);
```

Figure 2.1: The C++ function to allocate memory[16]

```
cudaError_t cudaMallocHost(void** ptr, size_t size);
```

Figure 2.2: The CUDA function to allocate pinned memory [20]

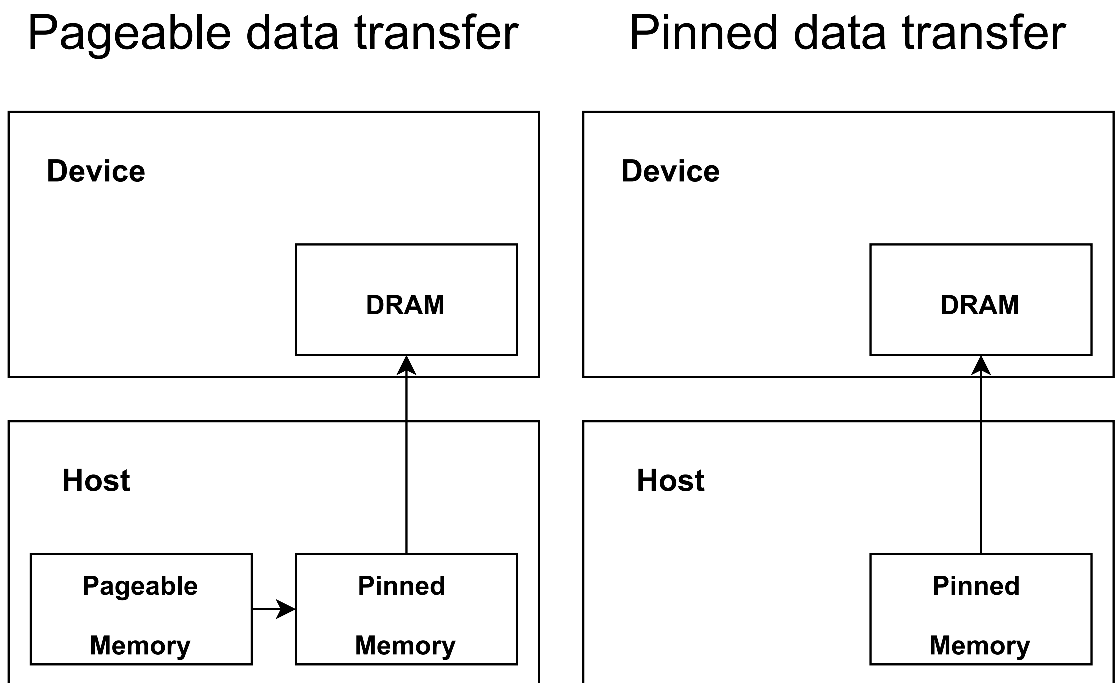


Figure 2.3: The difference between copying pinned and pageable memory [19]

Device Memory

Device memory, in this case, is defined as the RAM found locally on the graphics card, also known as Video RAM (VRAM). It is not accessible to CPU code and data must first be copied into the device memory for the GPU to execute operations on said data. The CUDA API offers the function shown in Figure 2.4 to allocate memory on the device [20].

The function has similar parameters to the CUDA function described in Section 2.3.2, with *devPtr* being the equivalent to *ptr*.

```
cudaError_t cudaMalloc(void** devPtr, size_t size);
```

Figure 2.4: The CUDA function to allocate device memory [20]

Managed Memory

Unified memory, also known as managed memory, is a technique used by CUDA to enable both CPU and GPU to access the same address space. However, it is more of a software feature than hardware allowing the GPU to access CPU memory space or vice-versa, as the data in question will always be moved to the executing processor's address space before operations are performed on said data [17]. Figure 2.5 shows the function required to allocate managed memory [20].

Similar to the other CUDA allocates, an explanation to the parameters *devPtr*, again equivalent to *ptr*, and *size* can be found in Section 2.3.2. The parameter *flags* indicates the access permissions for the chunk of memory, defaulting to accessible by any stream from any device [20].

```
cudaError_t cudaMallocManaged(void** devPtr, size_t size,
        unsigned int flags = cudaMemAttachGlobal);
```

Figure 2.5: The CUDA function to allocate managed memory [20]

Memory Copy

CUDA uses one function to copy memory both to and from the device memory, which can be seen in Figure 2.6. The function copies *count* bytes of memory from the pointer *src* to the location *dst* points to. The enum *cudaMemcpyKind* defines the type of memory transfer it is. The copy types supported are [20]:

- cudaMemcpyHostToHost
- cudaMemcpyHostToDevice
- cudaMemcpyDeviceToHost
- cudaMemcpyDeviceToDevice
- cudaMemCpyDefault: this is the recommended 'kind', as this causes the function to infer the type of transfer from the pointer values [20].

```
cudaError_t cudaMemcpy(void* dst, const void* src,
       size_t count, cudaMemcpyKind kind);
```

Figure 2.6: The CUDA function to copy memory [20]

2.3.3 De-allocation

De-allocating memory in C++ is a simple affair, done by calling the function $free()$, as depicted in Figure 2.7. The variable *ptr* is a pointer to a chunk of memory previously allocated by $malloc()$.

CUDA de-allocation works in a similar way, but distinctively separates device and host memory. Figure 2.8 shows the function call required to de-allocate device memory, with *devPtr* being the pointer to a chunk of device memory previously allocated by $cudaMalloc()$ or $cudaMallocManaged()$. Similarly, Figure 2.9 shows the function call required to free pinned memory previously allocated by $cudaMallocHost()$.

```
void free (void* ptr);
```

Figure 2.7: The C++ function to free a chunk of memory allocated by $malloc()$ [21]

```
cudaError_t cudaFree(void* devPtr);
```

Figure 2.8: The CUDA function to free a chunk of device memory[20]

```
cudaError_t cudaFreeHost(void* ptr)
```

Figure 2.9: The CUDA function to free a chunk of host memory[20]

2.3.4 NVML

Nvidia Management Library, abbreviated NVML, is a part of the CUDA API that offers, as its name suggests, management and monitoring capabilities. NVML can monitor many aspects of the GPU, such as GPU utilization, currently active processes, the current performance state of the GPU, and more [22]. Delving deeper into the documentation to NVML, it is revealed that the library is also able to monitor some aspects of the PCIe link that connects the GPU to the CPU. Supported metrics are the throughput and basic information about the PCIe link [23]. Figure 2.10 shows the function call required to query the library about the PCIe throughput. The *device* parameter is the identifier of the GPU, and *counter* defines the counter type. Supported counter types are, according to the NVML documentation [23]:

- NVML_PCIE_UTIL_TX_BYTES
- NVML_PCIE_UTIL_RX_BYTES
- NVML_PCIE_UTIL_COUNT

Whilst the documentation does not specify in further detail what each counter means, it can be inferred that TX means transmit and RX means receive, as TX and RX are common abbreviations for transmitter and receiver. Finally, the last parameter *value* points to the location where the counter value will be written to. The function documentation also states that the counters represent the average PCIe throughput per second over the last 20 milliseconds (ms) [23].

```
nvmlReturn_t nvmlDeviceGetPcieThroughput ( nvmlDevice_t device,
       nvmlPcieUtilCounter_t counter, unsigned int* value );
```

Figure 2.10: The NVML function read the PCIe link throughput [20]

2.4 PCI-Express

The last section mentioned the ability to move data from the main memory to the device memory, and vice-versa. This is done, on a lower level, by PCI-Express.

PCIe, or PCI-Express, shorthand for Peripheral Component Interconnect Express, is a "general-purpose serial I/O interconnect" [24]. PCIe, as an interface, allows the CPU to connect with, as the name suggests, peripherals and components. Common components and peripherals include, but are not limited to: Graphics cards, sound cards, video capture cards, WiFi cards, and storage.

PCIe is designed to replace the ageing PCI (Peripheral Component Interconnect), PCI-X (Peripheral Component Interconnect Extended), and AGP (Accelerated Graphics Port) standards [25]. These standards are developed, defined, and maintained by the PCI-SIG group, which is a nonprofit organization with 800+ member companies based in Beaverton, Oregon [26]. This section will briefly introduce the key features and functionality of PCI-Express.

2.4.1 Key Features

PCI-Express is, at its core, a serialized, point-to-point connection that is designed to be processor agnostic, scalable, and backwards compatible with PCI [24], [27], [28]. PCI-Express utilizes a dual-simplex connection to facilitate sending and receiving information concurrently. Additionally, to ensure backwards compatibility with PCI, PCI-Express shares the same memory configuration as PCI, which will be elaborated further upon in Section 2.4.3. Further key features include improvements in error handling and data integrity [25]. To future-proof the standard, current and future generations of PCIe are to be designed to be compatible with current PCIe standards [28]. So far, each generation of PCIe doubled the previous generation's theoretical maximum bandwidth, as seen in Table 2.1. Currently, PCIe 4.0 is gradually being introduced and offers twice the bandwidth of PCIe 3.0, which it is succeeding[29].

Link Width	x1	x2	x4	x8	x16	x32
Gen 1 Bandwidth (GB/s)	0.5	1	2	4	8	16
Gen 2 Bandwidth (GB/s)	1	2	4	8	16	32
Gen 3 Bandwidth (GB/s)	2	4	8	16	32	64
Gen 4 Bandwidth (GB/s)	4	8	16	32	64	128

Table 2.1: PCI Express aggregate bandwidths by generation and link width [30] [29]

2.4.2 Functionality

Packet

PCIe, similar to IPv4 or IPv6, utilizes packets to communicate between the host - the CPU - and the device. As shown in Figure 2.11, the packet consists of a few different elements, which will be further expanded upon below. Additionally, PCIe communication is separated into three primary layers, with each being responsible for a specific set of functions. The Physical layer is responsible for translating the packets to electrical signals and vice-versa. The Data Link Layer is primarily responsible for link error detection and correction. The Transaction layer is primarily responsible for flow control and transaction ordering [27].

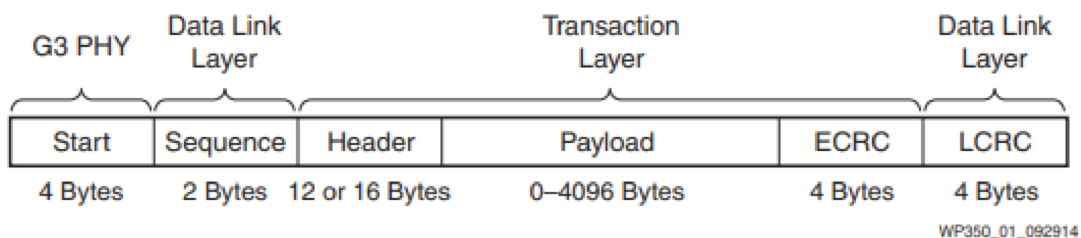


Figure 2.11: An example of a PCI-Express packet [27]

- Start: this is the start component which signals the begin of a packet to the Physical layer.
- Sequence: This two-byte sequence is used by the Data Link Layer to determine the sequence of the packets and to ensure that no packets have gone missing.
- Header: The 12 to 16 Byte header will be discussed in further detail in Subsection 2.4.2. This component belongs to the Transaction layer.
- Payload: The PCIe payload. This is optional, however any memory transferred via memory copy operations will have the memory as payload. This also is a part of the Transaction layer.
- ECRC: a CRC code for error-checking purposes used by the Transaction layer.
- LCRC: a CRC code for error-checking purposes used by the Data Link Layer.

Header

As with IPv4 or IPv6, PCI-Express uses headers to determine the purpose and target of each TLP (Transaction Layer Packet). However, instead of using IP-addresses, stored in the header, to determine the sender and the receiver, PCIe uses the Requester ID to determine the sender. The Address determines the receiver of the intended packet, as the device memory is memory-mapped into the host address domain to enable the processor's native load or store instructions to work with PCIe devices [31]. The header has a semi-fixed format, with some bytes varying with the type of request. The fields of a memory request header, as shown in Figure 2.12, and their uses, are briefly explained below, and a further explanation can be found in the book by Jackson et al. [30].

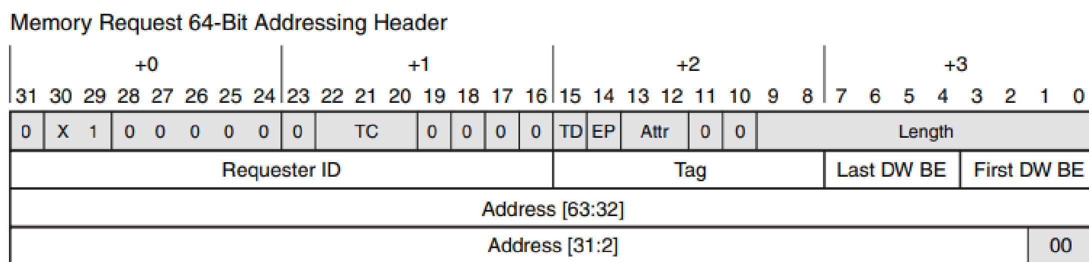


Figure 2.12: An example of a memory request header [27]

- TC: Traffic Class: this denotes the priority of the packet. A larger value represents a higher priority.
- TD: The TLP Digest field. If TD is set to 1, it indicates that there is additional CRC data in the TLP data.
- Length: more or less self-explanatory: length denotes the length of the payload in Double Words (DW), which is 4 bytes or 32 bits in length.
- Requester ID: self-explanatory: the ID of the device that requested or sent the packet.
- Tag: The Tag field has the function of a tracking number, as for read requests, the device must copy this value to its response. All outstanding tags must be unique to ensure data integrity. Some request types, such as write requests, do not utilize tags.
- DW BE fields: DW BE stands for Double-Word Byte Enable. This denotes which of the bytes in the first / last DWs of the payload are valid.

- Address: self-explanatory: The Address to which this packet is addressed, as explained above. Additionally, for read and write requests, this denotes the starting address of the read or write.
- The EP bit indicates whether a payload should be considered valid or not. If EP is set to 1, the payload is considered invalid.
- The Attr field holds two further special attributes for the header which are not of relevance here.

2.4.3 Topology and Communication

Topology

There are four significant components to be mentioned when discussing the topology of a PCI-Express based system. PCIe endpoints, switches, bridges, and a root complex. The communication between CPU cores and memory controllers to the PCIe endpoint is handled by the PCIe root complex. This communication can be routed through (but does not require) PCIe switches. PCIe switches allow for cascading connections, however do not benefit the total bandwidth, which is limited by the PCIe root complex in a CPU [32]. Bridges are used to connect legacy PCI and PCI-X devices with the PCIe root complex [24]. Figure 2.13 shows an example PCIe configuration of an Intel-based processor.

Memory Management

Each PCI-Express device has some built-in storage and registers to facilitate communication between the device and the rest of the system. This memory is, due to compatibility reasons, structured in the same way as the structure found in the older PCI standard. This divides the PCIe device memory into three major parts for addressing and memory access [30]:

- Configuration
- Memory
- IO

The configuration address space enables software to both identify and correctly configure the device, and is defined by its physical bus and device number [31]. It also enables the software to control and check the status of a PCIe device [30].

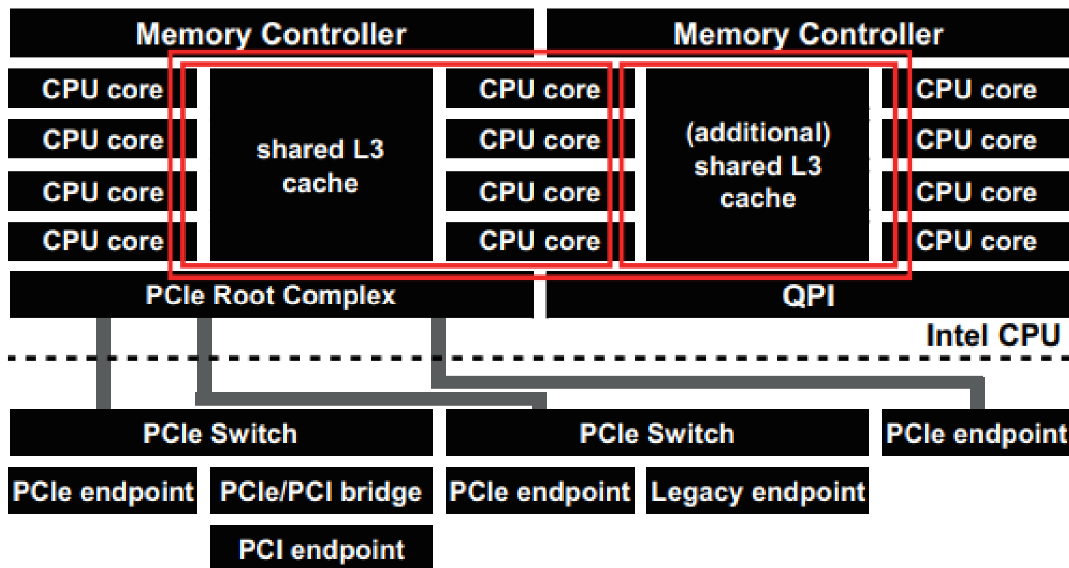


Figure 2.13: PCIe configuration on an Intel-based system [32]

The memory address space is where the internal storage of a PCIe device is mapped [30]. This memory space is also memory-mapped to the CPU's address domain for ease of access by the CPU [31].

The IO address space is a place dedicated to accessing the internal registers / storage of a PCIe or PCI device. However, this is mostly deprecated in PCIe as the internal registers and storage of said devices are simply mapped into the memory address space instead. It is now common practice to map the same set of registers in both memory and IO address space for backwards compatibility purposes. The PCIe specification discourages use of the IO space, which indicates that it remains solely for legacy support purposes [30].

Links and Lanes

A connection between the two PCIe devices is called a link, which is made up of lanes [30]. A PCIe device, in this case, can be the CPU's PCIe root complex, bridges, switches, or a PCIe end point. A lane, on the hardware level, is a set of four copper wires, two for each signal direction [30]. Due to the scalability of PCIe, the amount of lanes in a link is variable, from 1 up to 32, and is represented by a x in front of the lane width, e.g. PCIe x16, which indicates that the PCIe link has 16 lanes. A wider link means higher bandwidths and transmit capabilities, however it also means higher

power consumption, space, and cost [30]. Figure 2.14 illustrates an example PCIe link with several lanes. A normal GPU will usually have a PCIe x16 link.

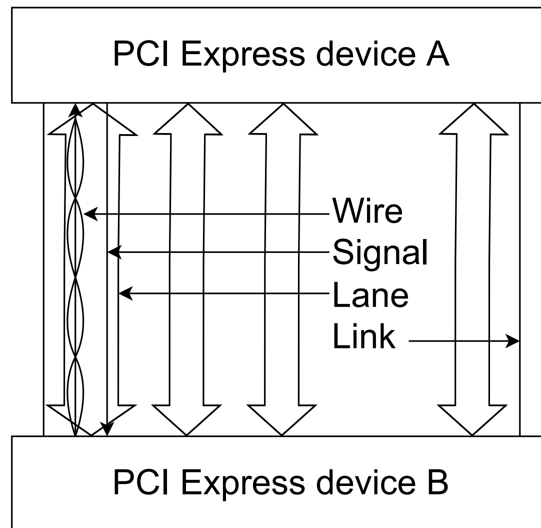


Figure 2.14: An example PCIe link between two devices [33]

2.4.4 Revisions and Further Specifications

PCI-Express was first introduced in 2003, and has received a new revision once every three to four years on average. Whilst most current hardware uses PCIe 3.0 and 4.0, introduced in 2010 and 2017 respectively [34], PCI-SIG has already published their specifications for the PCIe 5.0 and 6.0 standards. These, again, double the bandwidths of the previous generation, enabling theoretical transfer speeds of up to 128GB/sec in both directions on a PCIe 6.0 x16 link [34]. However, it is to be expected that these high-speed interconnect standards will take a few years to become widely available and adopted, as Intel only released their first PCIe 5.0-capable CPUs around the end of 2021 [35].

Additionally, other manufacturers and companies have developed their own protocols and standards to extend the feature-set of PCIe, such as Intel's Thunderbolt, which enables PCIe devices to be connected externally with only a small loss of performance [36]. Another example is the NVMe standard, a PCIe-compatible interface specifically devised and optimized for high-bandwidth, low-latency storage solutions [37].

2.5 System Specifications

Every tool written about in this thesis will be tested on two different systems with different CPU/GPU configurations, which this section will briefly touch upon. One system is part of the TUM CAPS cloud and the other system is a part of the LRZ cluster.

2.5.1 P6000

The first system, named p6000, is equipped with an AMD Ryzen Threadripper 2990WX, featuring 32 cores and 64 hardware threads. The graphics card is a NVIDIA QUADRO P6000, featuring a PCIe 3.0 x16 link and up to 24 gigabytes (GB) of dedicated memory. The system has 64GB of DDR4 main memory and is located on the TUM CAPS cloud.

2.5.2 Ice1

The second system, named Ice1, is equipped with two Intel Xeon Platinum 8630Y CPUs, each featuring 36 cores and 72 hardware threads. The system is also equipped with two NVIDIA Tesla V100 graphics cards, with 32 GB of dedicated memory, connected with a PCIe 3.0 x16 link each. Finally, the system is equipped with 530 GB of main memory. This system is a part of the LRZ cluster.

2.6 Kernels

The tools that monitor a program's PCIe activity will be tested on two different kernels, one which is focused on the memory transfers, and one which is more compute-intensive. Both kernels had timestamps added to facilitate easier correlation between timeline activities and the program's execution order.

2.6.1 Vector Add

The vector add kernel is the parallel elementwise addition of two vectors. Memory for the vectors, 400 megabytes (MB) per vector and 800 MB in total, is allocated and populated in the main memory before being copied into device memory, at which point the GPU will execute the kernel. The vectors are then copied back into main memory, where the values are checked for correctness. After that, the memory is de-allocated and the program terminates.

2.6.2 Matrix Multiplication

The matrix multiplication kernel multiplies two matrices. The program first allocates the memory for a 3200x3200 and a 3200x6400 matrix, about 120MB in total size, and populates the memory with values. Then, the matrices are copied into device memory, where the matrix multiplication kernel is executed a total of 301 times, with the first run being warm-up. This is timed and used as benchmark to assess a GPU's compute capabilities. Finally, the matrices are once again copied back into main memory and checked for errors. This kernel, without the timestamping modifications, can be found on NVIDIA's GitHub repository for CUDA Samples[38].

3 Bandwidth Benchmark

3.1 Concept

The first step to gain insight on data movements in a PCIe link is to assess the basic properties of the PCIe link regarding data traffic, namely bandwidth and delay. Additionally, due to the high bandwidths, it is useful to determine at which batch size the link will become fully saturated to further optimize memory transfers.

The bandwidth is defined as the amount of data that can be transmitted through the PCIe link in a given time, and the delay is determined by the duration of time that one packet takes to traverse the PCIe link. It is worth noting that metric is not the raw PCIe link delay as there is a certain amount of overhead introduced by CUDA function calls regarding memory copies.

Due to the high throughput of the PCIe link, transmitting smaller packet sizes will not fully saturate the link even for a small duration as both CUDA and the PCIe link itself will introduce some overhead to the memory transfer. As such, it is important to assess the batch size at which the PCIe link will be fully saturated.

Finally, it is worth noting that the actual PCIe bandwidth in a system will likely vary from the theoretical maximum due to the fact that it is, as the name implies, in a system. This means that there are several other factors influencing the practical PCIe bandwidth, such as main memory, other PCIe controllers, the CPU's memory controller, and more. As such it is prudent to measure the actual capabilities of a PCIe link by means of a benchmark instead of just relying on a theoretical value.

3.2 Goals

The primary aim of this chapter is to create a program that assesses the PCIe bandwidth of any given system. The primary metrics of interest for this benchmark are the practical PCIe bandwidth and the link saturation batch size.

Additionally, the tool developed should meet a set of more concrete requirements, based off the abstract ones defined in Chapter 1:

- The tool should not have any requirements on hardware outside of a CUDA-capable GPU

- The tool should support automation by outputting data to a file
- The tool should measure different memory copy sizes and their respective bandwidths to determine a saturation threshold
- The tool should measure both pinned memory and pageable memory transfer speeds

3.3 Implementation

3.3.1 Bandwidth Measurements

Figure 3.1 depicts a flowchart of the bandwidth test procedure, and this section's contents is depicted in the smaller black box. The bandwidth, as defined in section 3.1, is the amount of data that can be transmitted in a given time. This can be measured by measuring the time taken for a $cudaMemcpy()$ call, and dividing the size of the memory copied with the time taken by the method call. The $high_resolution_clock :: now()$ is used to provide accurate timestamps immediately before and after the memory copy. Subtracting the value of the first timestamp from the second yields the amount of time taken. The bandwidth is the result of dividing the memory size with the duration of the method call, calculated the following way:

$$\begin{aligned} bandwidth &= \frac{size}{duration}, \\ duration &= (time_2 - time_1) - delay \end{aligned} \tag{3.1}$$

3.3.2 Delay

A delay measurement, present as $delay$ in Equation 3.1, is measured by the time taken for a memory copy operation with 4 bytes of data from host to device. This time is taken as the combined delay for a call of $cudaMemcpy()$ and the PCIe link, as the duration to transfer 4 bytes of data should be negligible. To ensure consistency, this is done ten times, and the times measured are aggregated and averaged to determine the delay. This is done for both pinned and pageable memory. A more detailed workflow is presented in Figure 3.2.

3.3.3 Benchmark

To both determine the packet size at which the PCIe link is saturated, and to determine the potential maximum bandwidth of the link, sequential memory copies of doubled

buffer sizes are tested with the method described in section 3.3.1, up to a limit of 2 gibibyte (GiB). A further compensation is that a memory copy is executed before measuring the bandwidths, as the first memory transfer is usually marginally slower. The procedure to test the PCIe bandwidth for a specific memory size is depicted in Figure 3.1, and the benchmark consists of such tests for the mentioned memory sizes.

A final note is the PCIe max payload per packet, which is 4 kibibytes (KiB). As such the difference in time between transferring one and two bytes of data would be zero, or close to zero. This leads to either astronomically high or negative bandwidths due to division near zero in the bandwidth calculations. To eliminate any possibility of this disrupting the accuracy of the benchmark, 4 packets, meaning 16 KiB in size, was chosen to be the minimum packet size for the benchmark.

3.4 Results

3.4.1 P6000

Figure 3.3 shows the measured theoretical bandwidth for a PCIe 3.0 x16 link with regards to pinned and pageable memory copies for the P6000 system. The pinned bandwidth stabilizes around 12 GiB per second, whilst the pageable memory is significantly slower, at about 4 GiB per second. This is most likely not the result of the PCIe bandwidth being limited, but rather due to the internal transfer to a chunk of intermediate pinned memory as described in Section 2.3.2 as there is otherwise little discernible difference between the memory transfers. It is also worth noting that the pageable bandwidth declines steadily with increasing chunk size, likely due to overhead from first copying the data into pinned memory, as mentioned in Section 2.3.2. The pinned memory shows highs at around 13 GiB per second at a chunk size of 1MB, before steadying itself at around 12 GiB/sec with larger chunk sizes.

The first two data points for the pageable graph in Figure 3.3, 16 and 32 KiB in size respectively, are not present in the graph due to inconsistencies with the delay measurement. The 16 KiB transfer seems faster than the delay measurements, causing negative values. The 32 KiB transfer is similar, however the time taken is only marginally larger than the delay, causing the bandwidth to grow exponentially. This phenomenon was observed over a sample size of 10 separate runs of the benchmark, with similar resulting values and plots outside of shifts of around 200 MiB/sec at both data points, which can be attributed to per-run variance. The overhead also explains the theoretically impossible measurement for 128 KiB, as it exceeds the 16GB/sec bandwidth that a PCIe 3.0 x16 link is capable of.

Regarding link saturation, the plot in Figure 3.3 shows that the link is saturated at about 64 to 128 KiB for the pinned memory, however the peak bandwidth about

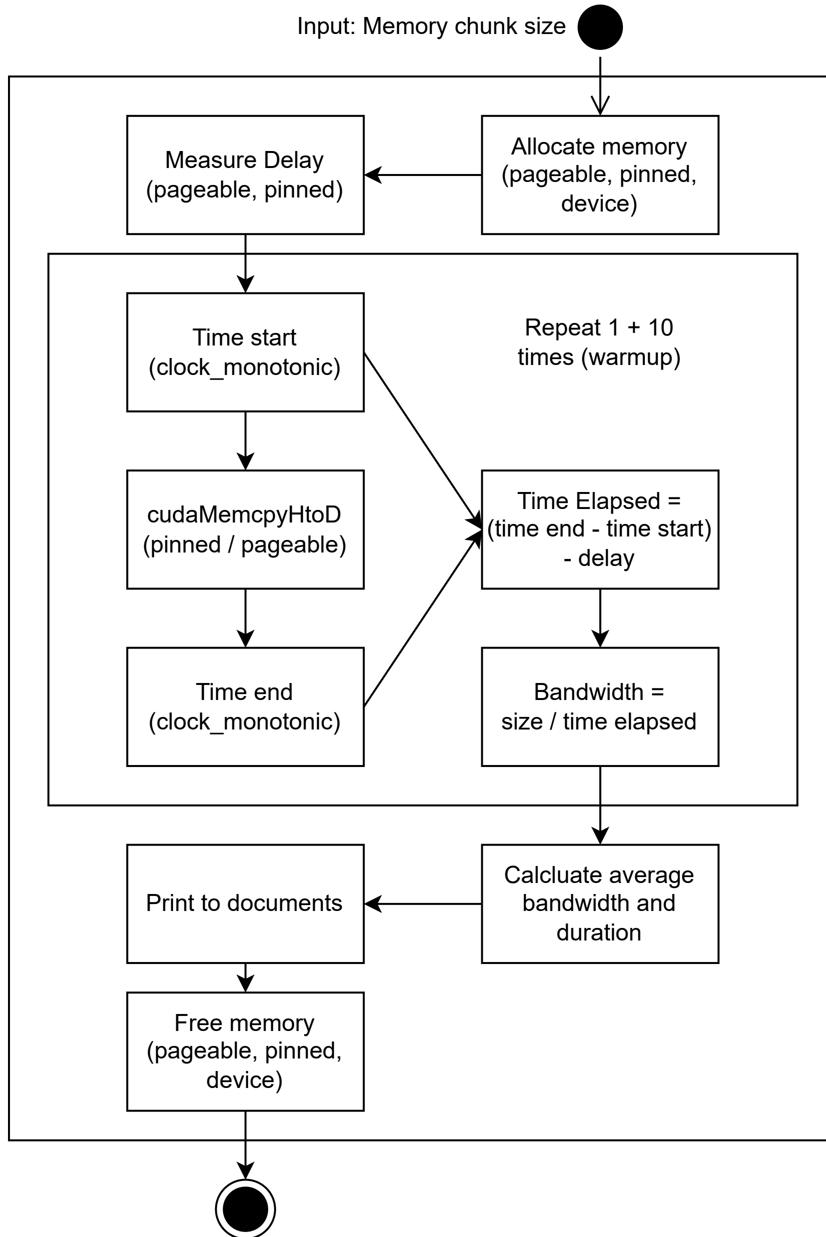


Figure 3.1: Rough Flowchart of the bandwidth measurement method

```
//allocate memory
long delay = 0;
for (int i = 0; i < 10; i++)
{
        auto t1 = std::chrono::high_resolution_clock::now();
        cudaMemcpy(device, host, sizeof(float), cudaMemcpyHostToDevice);
        auto t2 = std::chrono::high_resolution_clock::now();
        auto mys_int = std::chrono::duration_cast
                <std::chrono::nanoseconds>(t2 - t1);
        delay += mys_int.count();
}
//free memory
//calculate and return average duration
```

Figure 3.2: Code snippet measuring the *CudaMemcpy*() delay

1GiB/sec higher than the bandwidth at which the link transfers bigger packets. On the other side, it is difficult to tell at which chunk size the link saturates for the pageable copies, due to the steadily declining bandwidth values as chunks get bigger. However, the biggest jump of bandwidth is between 1 mebibyte (MiB) and 16 MiB, indicating that the intermediate chunk of memory may be limited at about 16 MiB in size.

Figure 3.4 shows the times taken for the memory transfers. It is worth noting that, for the pinned memory, the chunk sizes of 16 and 32 KiB take about the same time to copy, indicating that the duration of a memory copy does not scale linearly with the amount of packets sent over the PCIe link. It is also apparent that the duration, after the link has been fully saturated, grows more or less linearly for both memory types.

3.4.2 Ice1

Figure 3.5 is the measured theoretical bandwidth for the Ice1 system. Technically, the GPU should be connected by a PCIe 3.0 x16 link, just like the P6000. As the links are technically similar in generation and link width, this means that the bandwidths observed should be similar, which is the case for the pinned memory transfer at about 12 GiB/sec. However, the pageable memory transfer has inconsistent bandwidths, fluctuating from 2 to 15 GiB/sec for smaller memory sizes and stabilizing at around 11.5 GiB/sec for transfers above 512 MiB in size. Similar to the P6000 measurements, the first data point was consistently negative in value, and as such, was cut from the graph. However, the bandwidth fluctuated from -16 GiB/sec to around -5 GiB/sec over

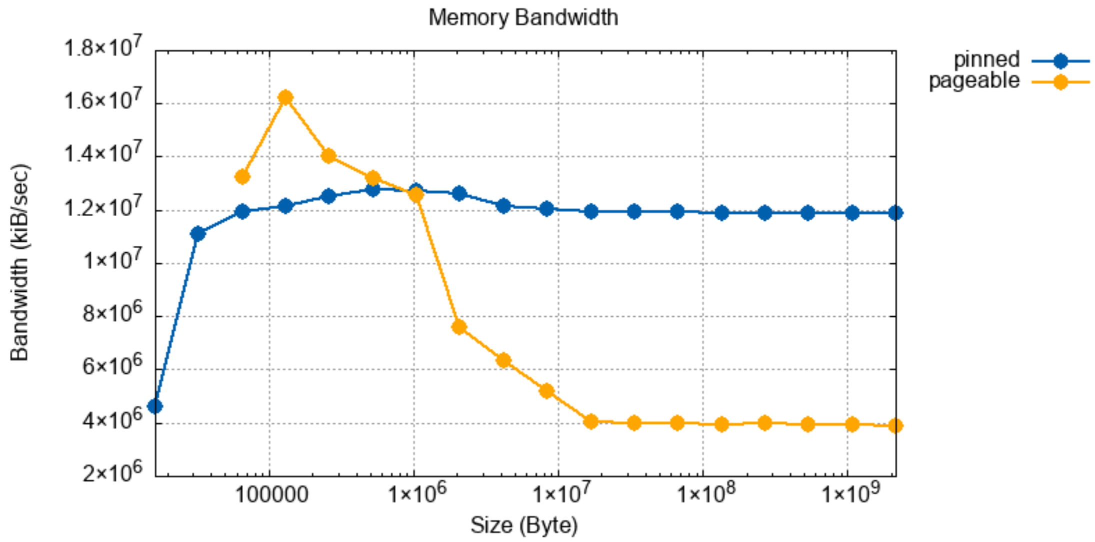


Figure 3.3: Pinned and pageable memory copy bandwidths (P6000)

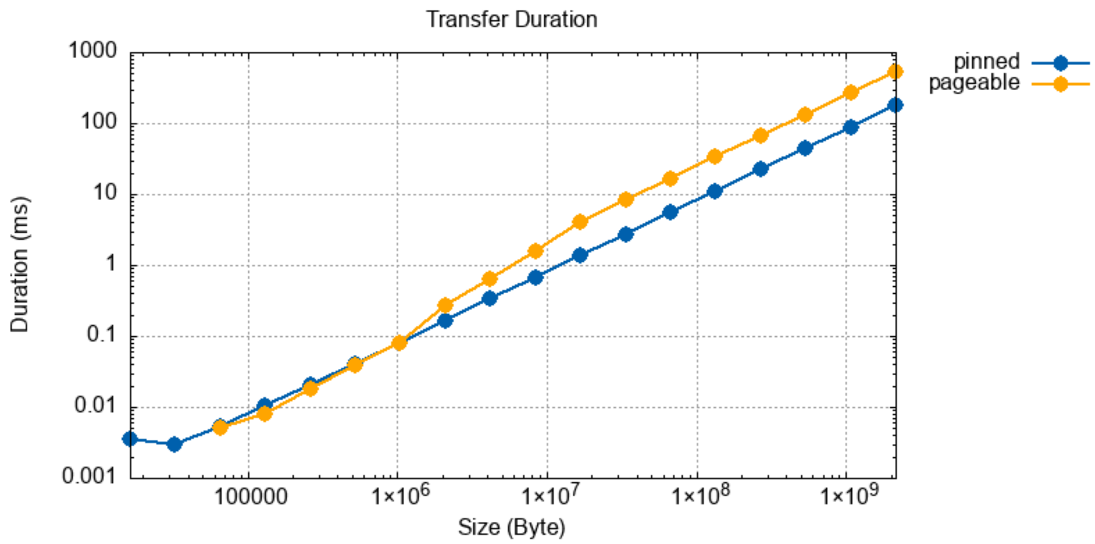


Figure 3.4: Pinned and pageable memory copy duration (P6000)

several measurements.

Regarding link saturation, the plot in Figure 3.5 shows that the link is saturated at about 128 to 512 KiB for the pinned memory. This chunk size is marginally larger compared to the P6000 system, which may be due to a few different factors, such as memory speed, driver / CUDA versions, or CPU/GPU properties. The pageable transfer, however, behaves in a volatile manner and is much faster than the pageable transfers on the P6000 system. This can most likely be attributed to the main memory size of the Ice1 system being several times larger than the main memory size of the P6000 system, as the pinned memory pool scales with total memory size.

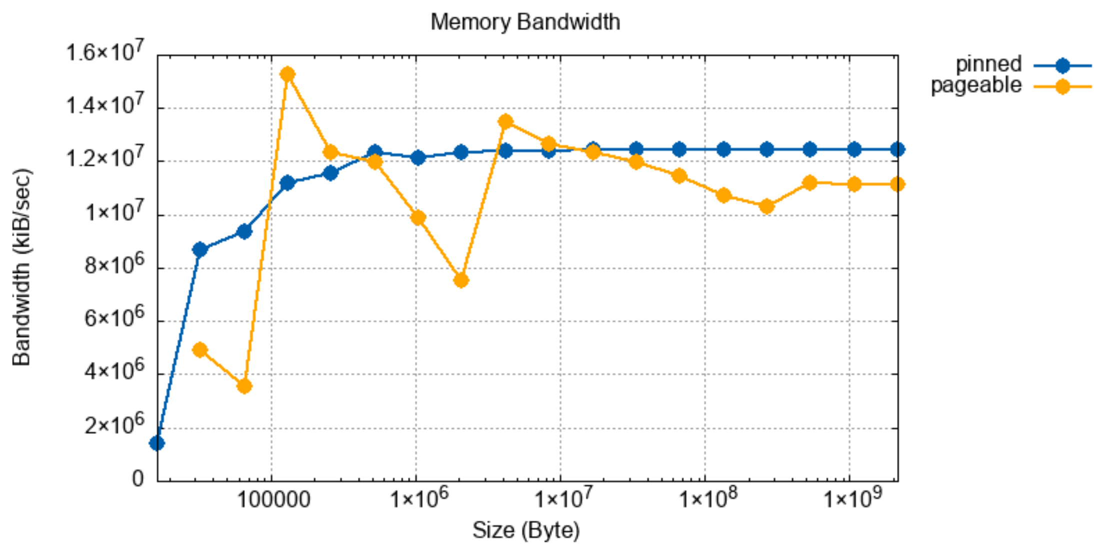


Figure 3.5: Pinned and pageable memory copy bandwidths (Ice1)

Figure 3.6 once again shows the times taken for the memory transfers. The most notable finding for pinned memory is the fact that a 16KiB transfer is actually marginally slower than a 32KiB transfer. This, once again, indicates that the duration of a memory copy does not scale linearly with the amount of packets sent over the PCIe link. On the other hand, just like the bandwidths, the pageable transfer durations are erratic and not linearly scaling. This is not quite as visible in the graph, due to the logarithmic scale of the y-axis.

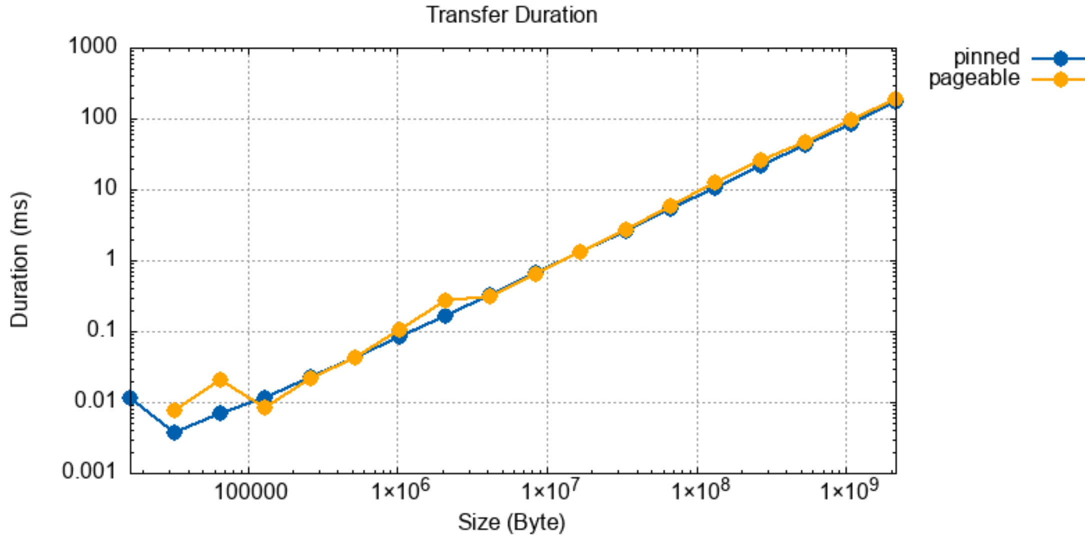


Figure 3.6: Pinned and pageable memory copy duration (Ice1)

3.5 Discussion

3.5.1 Successes

This benchmark gives an accurate reading of both the saturation threshold and the possible bandwidth over a given PCIe link, which is the primary goal of the benchmark. However, it is difficult to verify the measurement accuracy with tools that were not developed by NVIDIA due to the proprietary nature of both the GPU drivers and the CUDA API.

The program also shows that the transfer duration does not scale linearly with the packet count, and that this scaling bears further potential for investigation.

Another success is that the primary requirements for the tool as defined in Section 3.2 are met. Specifically, the developed tool is able to measure memory transfer bandwidths for both pinned and pageable memory transfers across different memory copy sizes, outputting its results to a data file, and required no further hardware besides a CUDA-capable GPU.

3.5.2 Shortcomings

The first issue is the lack of explanation for the first memory copy being slower than the others. This may be due to the CPU's caching and paging algorithms, or due to something else entirely. The exact cause is hard to pinpoint due to CUDA's closed

source nature and the lack of detailed insight into the GPU side of the process.

Additionally, it is worth mentioning that the delay measurement is not representative of the raw PCIe delay, as there is a certain amount of overhead involved regarding the memory copy in CUDA. To more accurately assess the delay of a PCIe link, it is better to time a packet being sent and the packet being received by the GPU. However, due to the closed-source nature of CUDA, the success of this approach is not guaranteed.

Finally, it is worth noting that not every system guarantees a max PCIe payload size of 4096 bytes. The maximum size is usually negotiated between the different endpoints.

4 NVML Counters

4.1 Concept

The benchmark in Chapter 3 does not really provide more detailed information on when a memory transfer could be happening, and at what bandwidths said transfers happen. The goal of this chapter is to utilize the hardware counters exposed by the NVML library, as discussed in Section 2.3.4, to assess PCIe link activity. As the counters are separated by transmit and receive, monitoring of both input and output data transfers should be possible.

4.2 Goals

The primary goal of this chapter is to develop a tool that monitors the PCIe link activity of another given program with the help of NVIDIA's NVML library, primarily to measure when and at what bandwidths the data is being moved to and from the GPU memory.

Just like the Chapter 3, the tool developed should meet a set of more concrete requirements, extending the abstract goals defined in Chapter 1:

- The tool should not have any requirements on hardware outside of a CUDA-capable GPU
- The tool should support automation by outputting data to a file
- The tool should measure the PCIe link activity of another program
- The tool should measure both host to device and device to host data transfer
- The tool should induce as little overhead as possible while monitoring
- The tool should not require modifications to the program it monitors

4.3 Implementation

4.3.1 Parallelism

As the primary goal of this tool is to monitor the PCIe link activity of another process, including but not limited to memory operations, there is a requirement for the tool to run in parallel with the programs it is intended to monitor. This is facilitated by creating a 'wrapper' that starts both threads, the monitor and the program to be monitored, simultaneously. Figure 4.1 shows the code snippet responsible for creating and starting the different threads. The two *pthread_create*() calls are responsible for starting the monitoring threads, which should be running before the monitored process executes. The variable *child* is the process to be monitored and was forked beforehand, and put on hold immediately, which is given as command line argument. The *ptrace*() call signals *child* to continue. A do-while loop monitoring *status* allows tracking of the monitored process to terminate the monitor when said program has concluded. The monitoring threads are then stopped by calling the *stop_monitoring*() method, which sets a global variable called *running* to false.

```
pthread_t thread_id;
pthread_create(&thread_id, NULL, start_monitoring_rx, (void *)0);
pthread_t thread_id_2;
pthread_create(&thread_id_2, NULL, start_monitoring_tx, (void *)0);
ptrace(PTRACE_CONT, child, 0, 0);
do
{
        wait(&status);
} while (!WIFEXITED(status));
stop_monitoring();
```

Figure 4.1: Code snippet showing thread behavior

4.3.2 Monitoring

As the NVML method call takes about 20 milliseconds (ms) to execute because the counters are only updated in a 20 ms interval, it seems prudent to monitor both transmit and receive in parallel. To monitor the counters in sequence would lead to a loss of granularity and a potential loss of data as essentially half the data for each counter is lost, as the counters are updated once every 20 ms. Thus, there is a monitoring thread for each direction, transmit and receive. As the counters are only updated once every

20 ms, there is no benefit in including additional monitoring threads in either direction as they will all wait for the same counter to update at the same time.

To go into further detail, the monitoring thread, a rough flowchart of which can be seen in Figure 4.2, first initializes the necessary environment for the measurement, such as the output file, the NVML library, and the start timestamp. After that, it repeatedly loops a call of $nvmlDeviceGetPcieThroughput()$, which was further explained in Section 2.3.4, a timestamp after the call has concluded, the elapsed time calculation, and printing both the timestamp and the counter readout to the file. The condition for the thread to terminate is tied to the global variable *running*, as mentioned in Section 4.3.1. After the while loop terminates, cleanup is done by shutting down the NVML library and file streams.

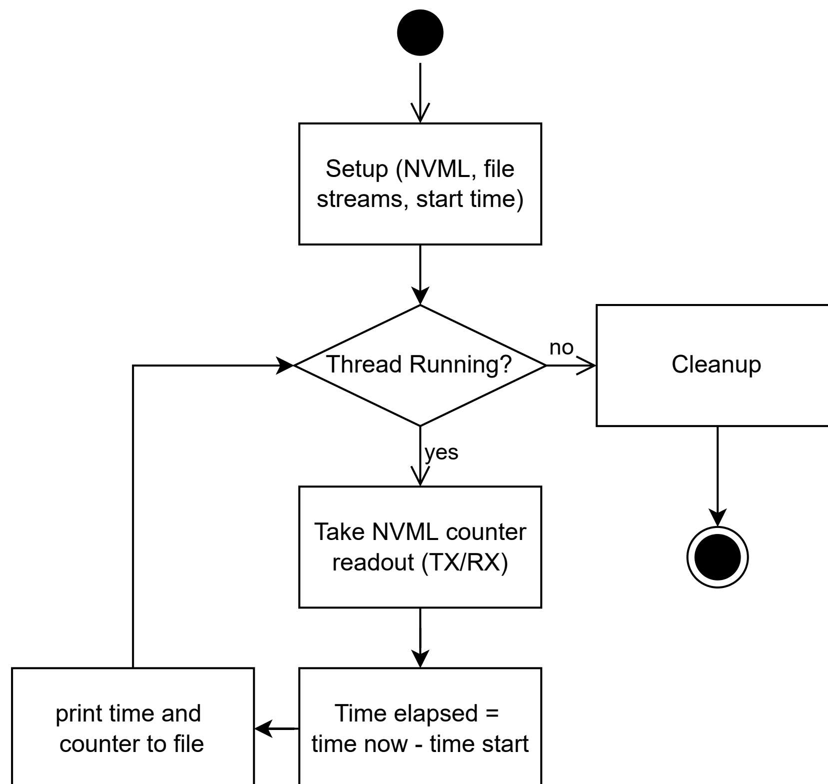


Figure 4.2: Rough Flowchart of the bandwidth measurement method

To summarize: The wrapper starts the two monitoring threads, one for transmit and one for receive, before starting the program that the threads are to monitor. As the program is being executed, the monitoring threads continuously print each readout and the corresponding timestamp into a dedicated file. After the program terminates,

both monitoring threads are terminated as well. The files the threads printed to can then be used to plot a timeline of the PCIe link activity while the monitored program was running.

4.4 Results

4.4.1 Vector Add

P6000

As Figure 4.3 shows, the memory copies in the vector add program described in Section 2.6.1 are clearly visible as spikes of activity on the P6000 system. However, there are some bigger gaps between data points, indicating that there may be an issue with either repeatedly calling the method or the update frequency of the NVML counters, as both TX and RX graphs show gaps in the graph. This is especially apparent as there seems to be a lack of data entirely for a time after the two memory transfers has concluded.

Figures 4.4 and 4.5 show similar graphs as Figure 4.3, with varying vector sizes. The smaller the vector size, the harder it is to pin down the exact time of the memory transfer, just as it is harder to appropriately measure the bandwidth of the memory transfer. This is especially visible in the 25MB and 50MB graphs, as the memory transfers are portrayed to be happening in parallel, which is clearly not the case, as the memory copies are coded in sequence. The sequential nature becomes more visible with bigger vector sizes.

It is also worth noting that there is no significant increase in computation time, no matter the vector size. The execution time of the vector add program varies to some extent from run to run, and wrapping the NVML monitor around the program seems to have no impact on the time taken.

Ice1

The results form the Ice1 system, a graph of which is shown in Figure 4.6, show the data transfers of the monitored program. However, the receive (RX) bandwidth, as in host to device bandwidth, is measured at 45 GB/sec. This is technically impossible for a PCIe 3.0 x16 link, which is limited to 16 GB/sec in bandwidth. As the memory copy completed without issue, it indicates that the NVML library did not reset the value of the counter for a period of time, resulting in the aggregation of the counter's value. This is also supported by the lack of data on the RX graph for the last 60ms, three counter cycles, before the spike occurred.

Additionally, whilst the receive counter does not show any visible inconsistencies in the graph, the transmit thread shows bigger gaps in data points over several times

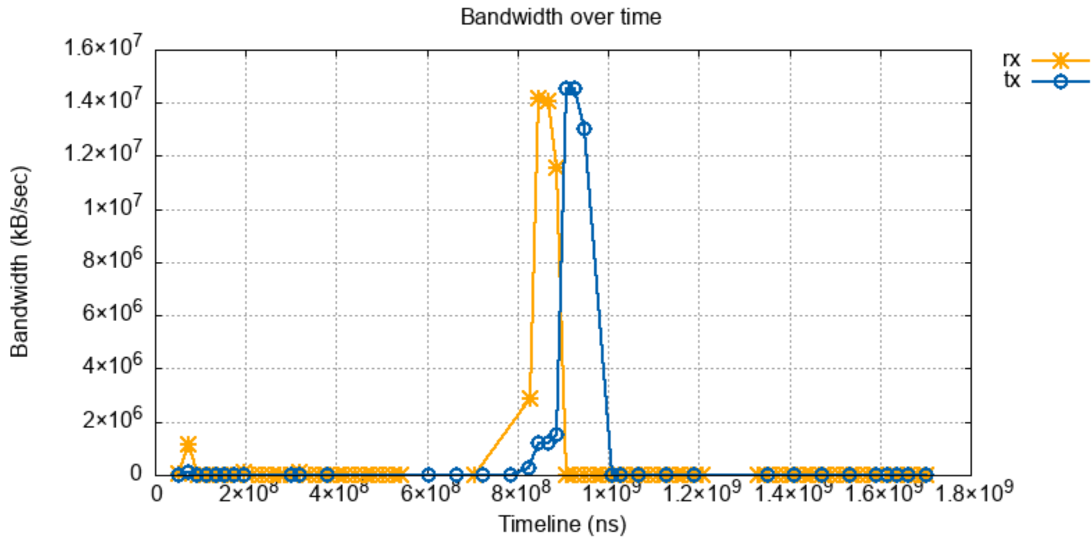


Figure 4.3: NVML monitoring of vector addition (400MB vector, P6000)

in the graph. This issue was present in both directions on the P6000. As the issue seems to not persist when switching hardware, at least in one direction, it seems to be a software-based issue. And as the RX thread is working as intended, whilst the TX thread is not, and both are implemented in the exact same way, it indicates that it most likely is an issue with the NVML library.

Figures 4.7 and 4.8 further demonstrate the impact of changing vector sizes on the monitoring results. The result is, as expected, similar to the ones from the P6000, with smaller vector sizes being much harder to accurately portray. It is also worth noting that the receive (RX) bandwidth value goes beyond what is technically possible for a PCIe link for vector sizes of 200MB and above, similar to the behavior shown in Figure 4.7, indicating that it is not just an issue with that specific vector size. Additionally, the TX graph seems to display the data gaps in several of the graphs, missing a few data points here and there, indicating that it is not limited to bigger vector sizes.

Similarly to the results from the P6000, there is no significant overhead or increase in computation time compared to running the program without the monitor.

4.4.2 Matrix Multiplication

P6000

Figure 4.9 depicts the results of using the NVML wrapper to check the matrix multiplication program's PCIe link activity. The graph shows the two memory transfers

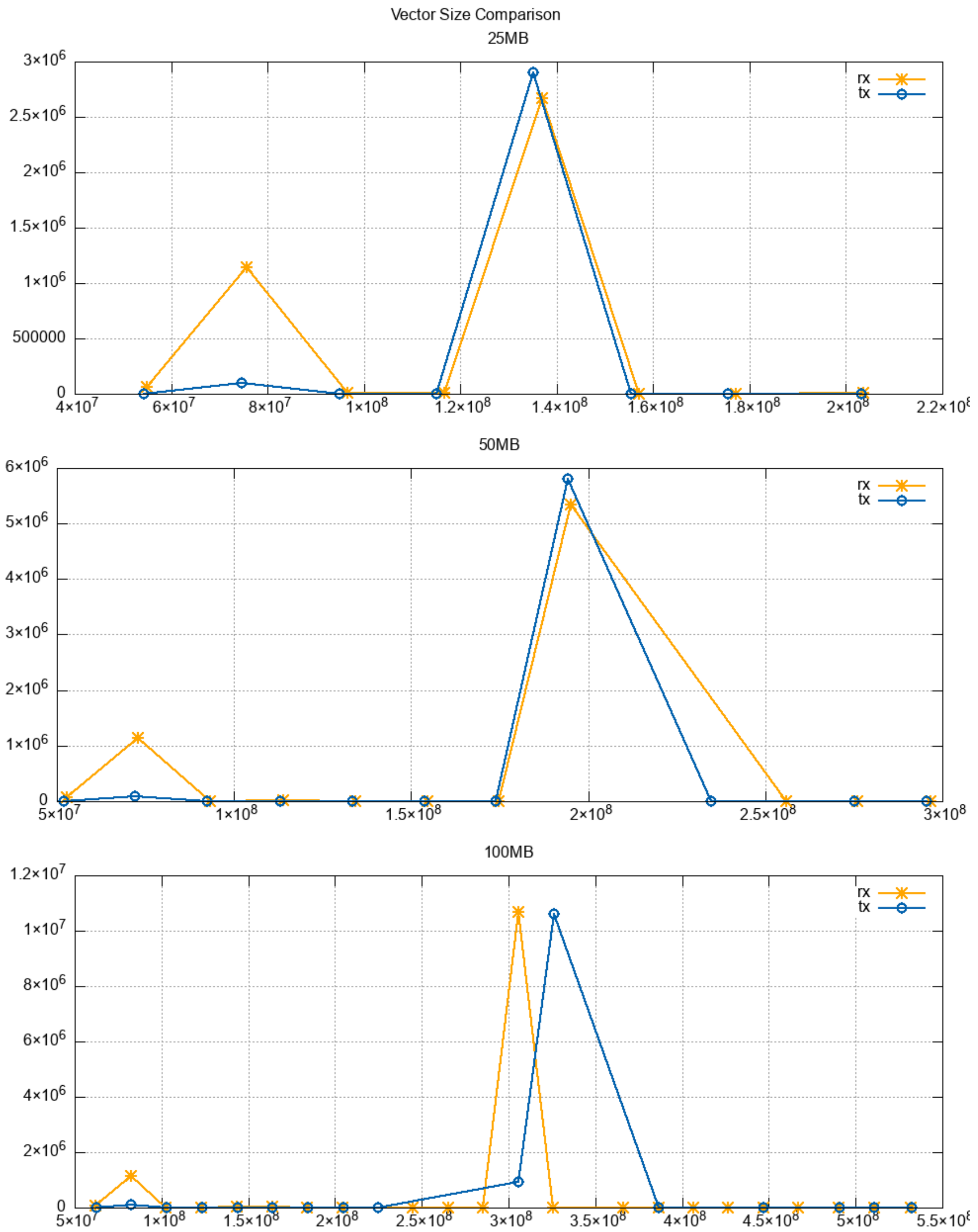


Figure 4.4: NVML: Monitoring vectors of different sizes (P6000, part 1)

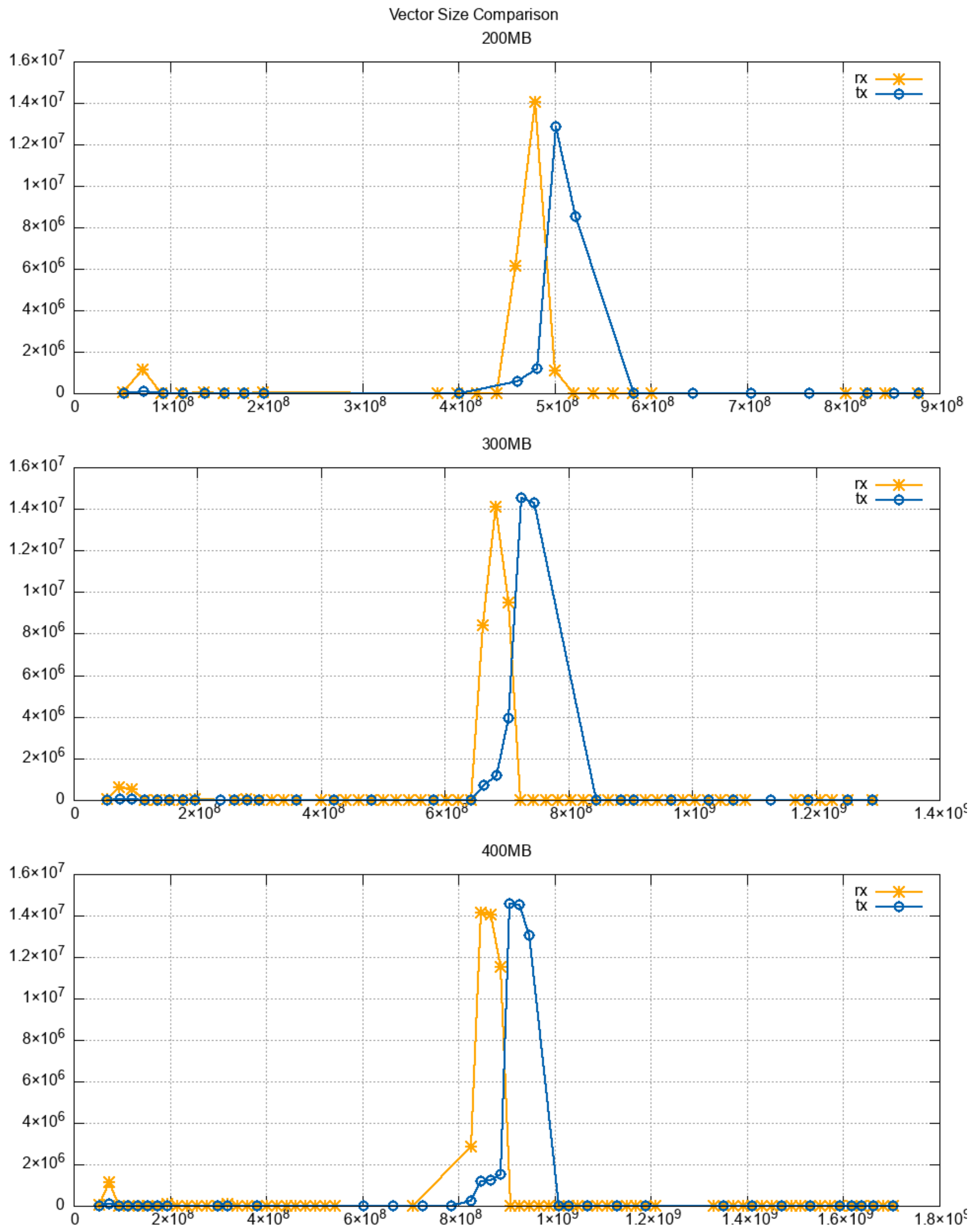


Figure 4.5: NVML: Monitoring vectors of different sizes (P6000, part 2)

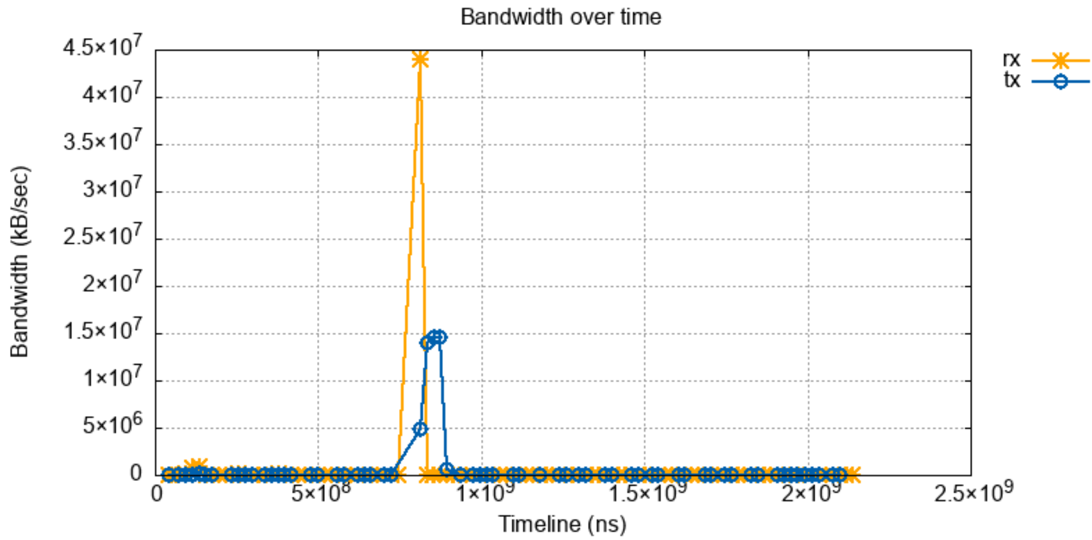


Figure 4.6: NVML monitoring of vector addition (400MB vector, Ice1)

at the very beginning and at the end of the program. However, the bandwidths don't quite match up with the results shown in Figure 5.4, as technically the 120MB data transfer should be showing peak bandwidths of around 10 GB/sec. Instead, the peak bandwidth, both for transmit and receive, lies at around 4.5 GB/sec. It is worth noting that the RX counters show that there was traffic over two counter readouts, indicating that the bandwidth was either not fully saturated or the counter resetting during the memory copy. The TX graph shows a lack of counters immediately after the peak, which implies that the memory copy was not completely shown on that graph either. Furthermore, this indicates that the issue with missing datapoints is not solely related to the measured program.

Additionally, the counters showed minimal but steady PCIe traffic, around 5 KB/sec on the TX graph and 16 KB/sec on the RX graph, when the kernel was running the 300 iterations. This is an indication that the commands the CPU issues to the GPU, and the GPU's responses, are both visible on the GPU's NVML counter, indicating that it captures all traffic over the PCIe link, and not just memory copy operations.

Finally, the computation time seems to remain static, regardless of whether the monitor is attached or not. This once again indicates that the wrapper introduces little to no overhead to the program it is to monitor.

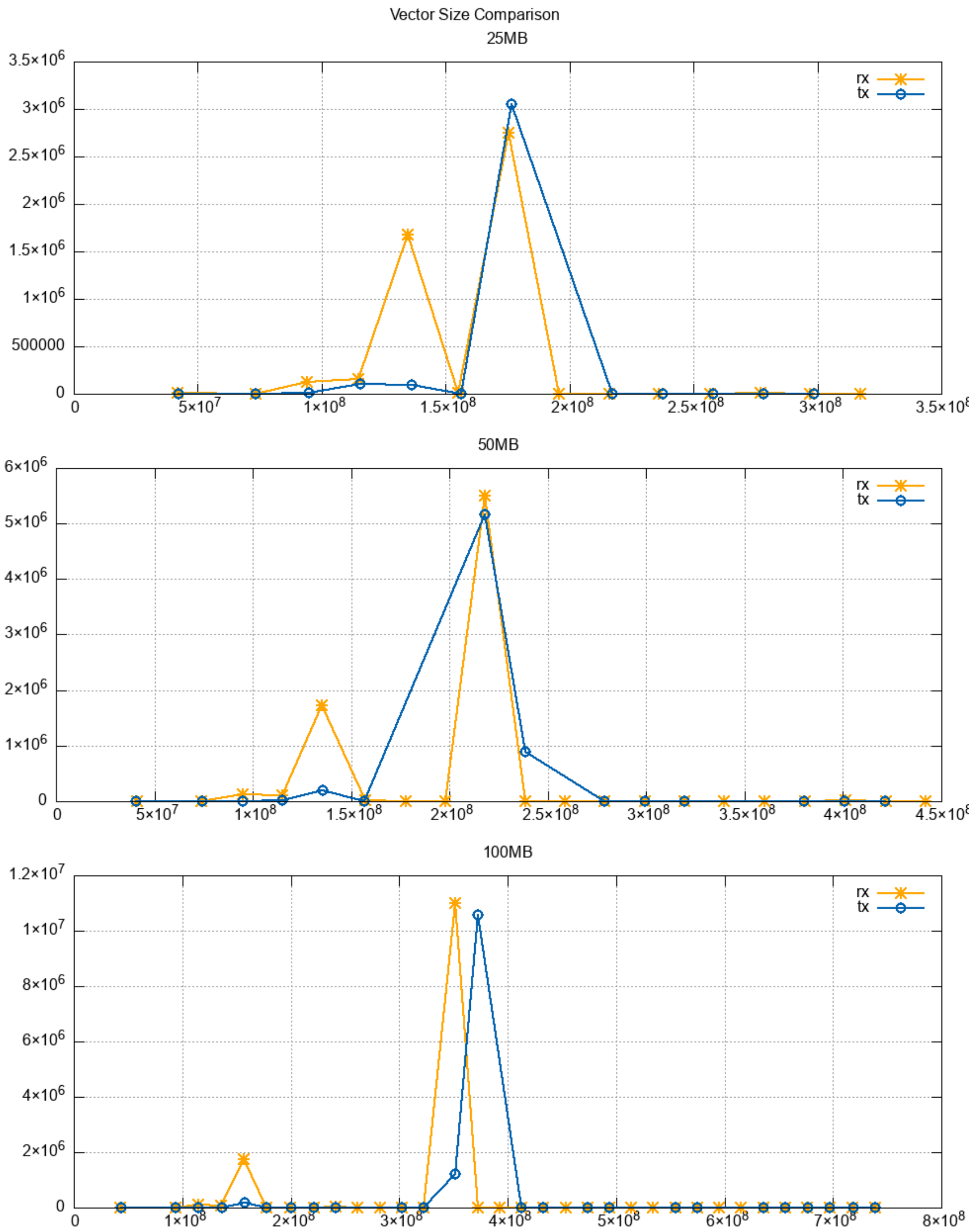


Figure 4.7: NVML: Monitoring vectors of different sizes (Ice1, part 1)

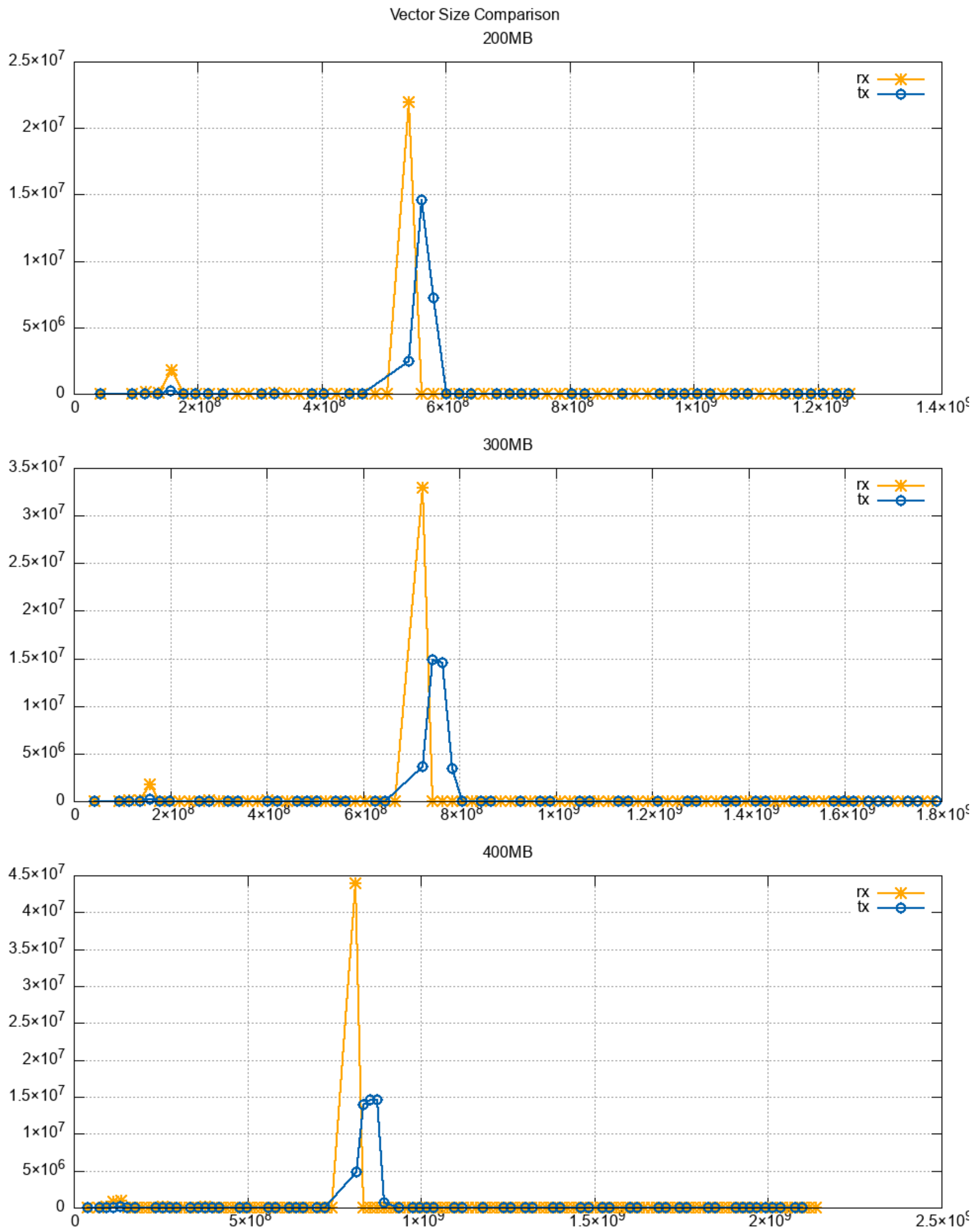


Figure 4.8: NVML: Monitoring vectors of different sizes (Ice1, part 2)

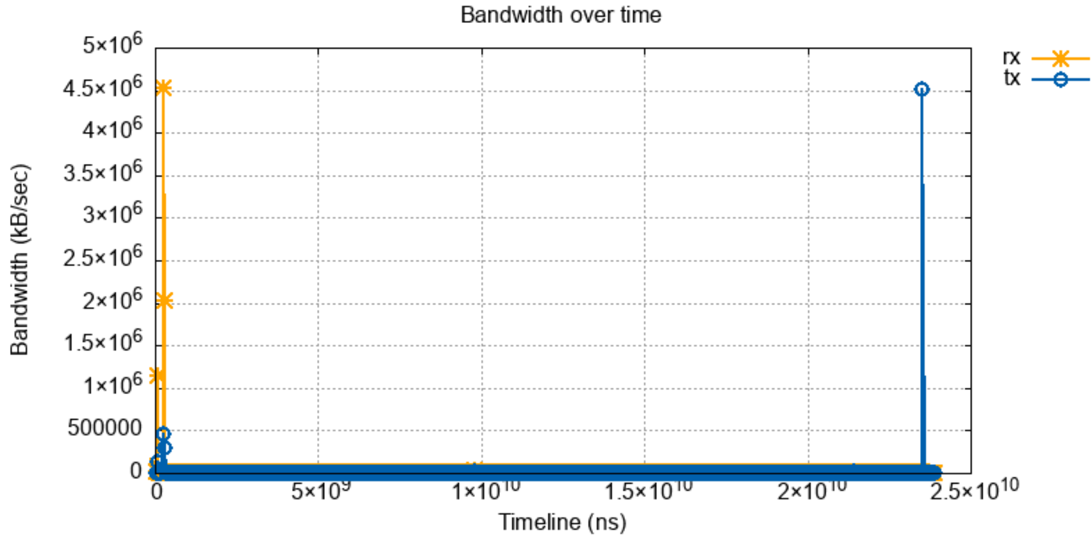


Figure 4.9: NVML monitoring of matrix multiplication (P6000)

Ice1

Figure 4.10 once again clearly shows the two memory transfers at the very beginning and at the end of the program. Similar to the results in the P6000, the bandwidth results for a 120MB transfer do not quite match up with the observations from Figure 5.7, capping out at 7GB/sec for the RX graph and 4.5GB/sec for the TX graph, instead of the expected 10 or 11 GB/sec. Furthermore, the TX counter once again stutters immediately after the TX graph peaks, similar to the results observed from the P6000 system. The RX graph, on the other hand, does not seem to miss any datapoints.

It is also worth mentioning that the memory copies from Figure 5.10 do not correspond to the timestamps that were output into the console, indicating that the code may not be executing in sequence due to optimizations when compiling. Another possible explanation for this occurrence is the non-blocking nature of $cudaMemcpyAsync()$, which was used in this kernel.

Additionally, the counters once again show minimal but steady PCIe traffic, around 6 KB/sec on the TX graph and 19 KB/sec on the RX graph, during the compute section. This once more indicates that the communication between CPU and GPU, is visible on the GPU's NVML counter. This further indicates that the NVML counters capture all traffic instead of only memory copy operations.

Finally, the computation time once again seems to be within margin of error, deviating by around 1 or 2% in either direction, regardless of whether the monitor is attached

or not, which can be attributed to per-run variance. This once again indicates that the wrapper introduces little to no overhead to the program it is to monitor.

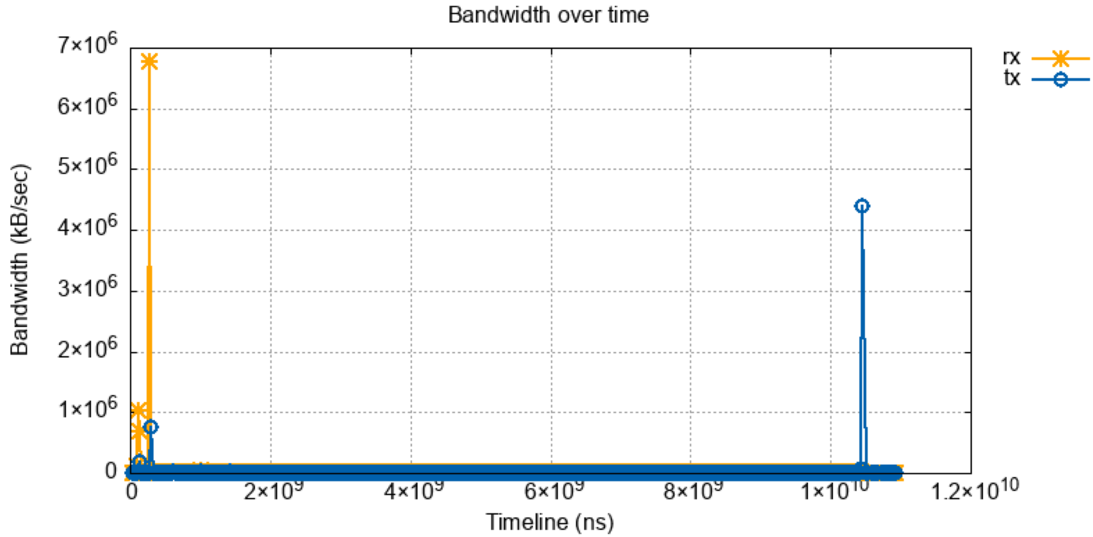


Figure 4.10: NVML monitoring of matrix multiplication (Ice1)

4.5 Discussion

4.5.1 Successes

The primary success of the tool is that it is not only able to pinpoint when a memory transfer happens, but also able to measure the bandwidth at which it happens. This gains some insight into an application's utilization of the PCIe link. Furthermore, the tool is able to capture most of the PCIe traffic, including instructions and responses from the CPU and GPU respectively.

Another point to the tool is that it introduces little overhead despite being executed in parallel with the thread it is measuring, as the results show. Granted, if the software is capable of fully leveraging all hardware threads in a system, this overhead would likely not be negligible.

Additionally, there is no requirement in terms of what it can or cannot measure, as long as the program to be measured can be started as a process.

Finally, the primary requirements defined in Section 4.2 were all fulfilled. The program does not have any requirements except for a CUDA-capable GPU, it does not require any modifications to the software it monitors to accurately read the PCIe link

bandwidth in both directions, and both the execution and plotting of the data can be automated.

4.5.2 Shortcomings

The primary shortcoming of this approach is the limitation of the NVML method call. As it is limited to a 20ms interval, data transfers that are shorter than that are not accurately represented, as seen in the results. Furthermore, the NVML library seems inconsistent in terms of monitoring and result accuracy, as shown by the inconsistent counter readout frequencies. This is also likely tied to the driver and CUDA API versions, with more recent drivers showing fewer inconsistencies.

Additionally, there is nothing that helps in identifying what method or software thread called the memory transfer, except for adding timestamps into the wrapped source code, which would fail the requirement of not needing modifications to existing software to work.

Furthermore, the method seems to stutter every once in a while, leading to periods of time where there is no data at all.

A further issue is that the tool is not able to isolate the transfers of the software it is wrapped around, but rather measures the throughput of the entire PCIe link. This means that other software running on the system may cause inaccuracies in the measurements.

A final issue is that the closed-source nature of CUDA is not transparent about how these counters are aggregated, and that the documentation about these counters is murky at best. As such, it is a concern that the readings may be inaccurate to some extent. This also means that the exact cause for the data leaks found in Section 4.4 is difficult to pinpoint, and probably just as hard to fix.

5 Link Saturation

5.1 Concept

The primary flaw in the NVML approach described in Chapter 4 is data granularity, which is directly tied to the counters' update frequency. The 20ms update interval of the counters does not allow for correct profiling of shorter memory transfers. To address this issue, a measurement method that is faster in nature needs to be devised. The primary concept of this tool is to saturate both directions of the PCIe link by repeatedly copying small chunks of memory to and from the device, and monitoring the transfer bandwidths to determine PCIe link activity. In theory, if a memory transfer happens while the link is already fully saturated, both memory transfers should technically slow down. Due to the dual-simplex nature of PCIe, the transmit and receive links must both be saturated and monitored separately.

5.2 Goals

?? The primary goal of this chapter is, just like Chapter 4, to develop a tool that monitors the PCIe link activity of another given program. However, this chapter aims to detect shorter memory transfers more accurately, to improve on the biggest issue of the NVML approach.

Just like the last two chapters, the tool developed should also meet a set of more concrete requirements, based off the abstract ones defined in Chapter 1 and similar to those described in the last chapter, with the addition of the ability to detect shorter memory transfers:

- The tool should be able to detect shorter memory transfers accurately
- The tool should not have any requirements on hardware outside of a CUDA-capable GPU
- The tool should support automation by outputting data to a file
- The tool should measure the PCIe link activity of another program

- The tool should measure both host to device and device to host data transfer
- The tool should induce as little overhead as possible while monitoring
- The tool should not require modifications to the program it monitors

5.3 Implementation

5.3.1 Parallelism

Similar to the approach described in Section 4.3.1, this tool needs to be run in parallel as the program it is supposed to monitor. As such, a similar 'wrapper' approach is used, a flowchart of which can be seen in Figure 5.1. A further addition over the NVML wrapper is a command line argument to define the memory chunk size the monitor is supposed to copy.

5.3.2 Monitoring

The primary difference to the implementation of the NVML approach described in Section 4.3.2 is that running transmit and receive threads separately seems to negatively impact the performance of both threads. This could be due to an overload of the memory controller by repeatedly copying small chunks of data in both directions, or due to some other, unknown reason. As such, the transmit and receive copy operations are executed in one thread, in sequence, to bypass the performance penalty.

A code snippet depicting this behavior can be seen in Figure 5.2, however it is worth noting that only the host to device transfer is depicted in the snippet due to the identical nature of both transfers. First, the timestamp for the memory transfer is calculated by comparing the current time to the start time. Then, the PCIe link latency is measured once, and saved to the variable *latency*. Finally, the bandwidth is measured similar to the concept described in 3.3.1, using the $high_resolution_clock :: now()$ call to time the memory transfer, and derive the bandwidth. It is worth noting that there is no delay compensation when timing the repeated memory copies, as the delay may affect bandwidths in a similar way to how they did in Section 3.4, leading to negative values and unsuitable data.

As the default memory chunk size being copied is only large enough to briefly saturate the PCIe link, 200 KiB in size, the granularity of the measurements shouldn't be impacted. At full bandwidth, a memory transfer of 128KiB, which is the saturation threshold determined in Section 3.4, takes about 0.01 ms, which should be enough to pick up most memory transfers longer than a millisecond in duration.

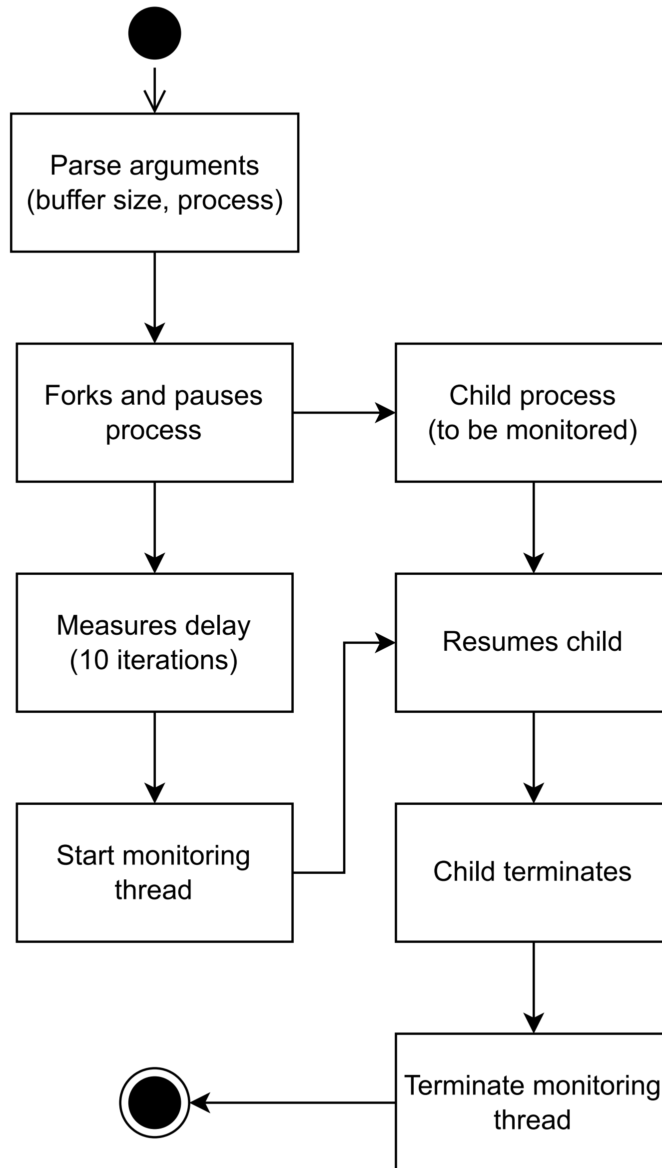


Figure 5.1: Flowchart of the copy wrapper

Another addition is the monitoring of the delay of the PCIe link, to see if there is any visible correlation between PCIe link load and PCIe delay. To do this, a memory copy of 4 bytes is sent to the device and the method call timed, similar to how it was done in Section 3.3.2. However, as everything is being executed in sequence, this delay measurement is only run once per cycle. This is represented by the $measure_delay(1)$ call in the code snippet. The delay is also printed to the text file to allow easy plotting and data gathering.

```
while (running)
{
        //timestamping
        auto time_point = std::chrono::high_resolution_clock::now();
        auto ns_int = std::chrono::duration_cast<std::chrono::nanoseconds>
                (time_point - time_start);
        outdata << std::fixed;
        outdata << ns_int.count() << "\t";
        long latency = measure_delay(1);
        auto t1 = std::chrono::high_resolution_clock::now();
        cudaMemcpy(device, host_pinned, bytes, cudaMemcpyHostToDevice);
        auto t2 = std::chrono::high_resolution_clock::now();
        auto mys_int = std::chrono::duration_cast<std::chrono::nanoseconds>
                (t2 - t1);
        outdata << (bufsz / (mys_int.count() / 1e9)) << "\t";
        outdata << latency << std::endl;
        \\repeat, but device to host
}
```

Figure 5.2: Code snippet depicting monitoring thread measurements

5.4 Results

5.4.1 Vector Add

P6000

Figure 5.3 clearly shows the two memory copies that happen in sequence, with the Host to Device (HtoD) and Device to Host (DtoH) graphs dropping to a near zero bandwidth as the memory copies happen during the vector addition. It is also worth

noting, however, that the other respective link was also only able to copy at reduced speeds, indicating that the full dual simplex bandwidth may be impossible to maintain. This may be due to limitations either in the memory controllers or in the software, however the exact cause is difficult to pinpoint due to the closed-source nature of CUDA and the lack of insight into the process in general. The graph also shows that there is a lack of correlation between the delay measured over a PCIe link and the load said PCIe link is under, as the delay does not increase noticeably when the link is fully saturated by the memory copy operations of the vector add program. However, the delay can be found spiking as the kernel of said program is running, which may indicate that delays may be related to the GPU load, instead of the PCIe load.

A further irregularity shown in Figure 5.3 is the inconsistency of bandwidths before the first memory copy. The speeds seem to fluctuate heavily between 5 and 7 GiB per second, and then increasing in value to around 7 to 8 GiB per second. Only after the memory copies have concluded does the bandwidth settle just below 9GiB per second. This indicates that the allocates performed before the memory copies may have some impact on the PCIe link bandwidths.

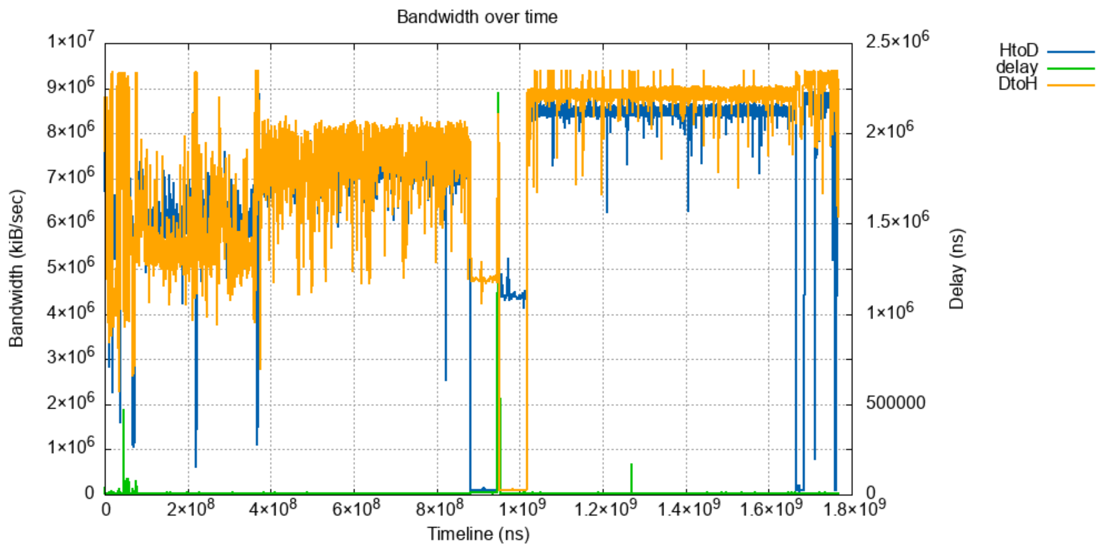


Figure 5.3: Bandwidth saturation monitoring of vector addition (400MB vector, P6000)

Figures 5.4 and 5.5 on the other hand show that, with smaller chunk sizes, the wrapper is still able to monitor the copy operations accurately, to some degree, despite not fully saturating the link. The plots also show that the speed reductions found in Figure 5.3 are constantly scaling with the respective maximum bandwidth as well, dropping to about half of their original values when the other link direction is fully

saturated.

It is also worth noting that only a chunk size of 200KB and above was able to utilize the link at more or less full bandwidth, around 10 GiB/s. However, this was expected as the saturation threshold was pinpointed at somewhere around 128KiB, as mentioned in Section 3.4. The accuracy drops at the smallest chunk sizes of a few packets (5KB), as the drop in host to device bandwidth doesn't drop to near zero as expected.

Finally, there seems to be no significant increase in computation time on the P6000, similar to the approach devised in Chapter 4. Once again, the execution times, with and without attaching the monitor, are within margin of error.

Ice1

Figure 5.6 shows the same test as Figure 5.3, with a vector size of 400MB and a chunk size of 200KB. The graph also shows that there is less inconsistency in terms of bandwidth fluctuation before the memory transfer. This difference can either be attributed to the A100's superior compute capacity or to the more modern drivers installed on the system. Another major finding is that there seems to be a lack of data when the first memory copy occurs. This is not quite visible in the graph, but there is no data from 0.86 sec to 0.92 sec (just before $1 * 10^9$ in the graph) in the data files used to generate the plot. This time frame also corresponds to the duration of the memory copy from the console output timestamps. This is caused by the low bandwidth of the measured transfer, combined with the sequential nature of the monitoring thread, resulting in a single datapoint taking up the entirety of the duration.

Another visible difference is that the bandwidth of the link which is not under load by the program is not as affected on the Ice1. During the DtoH transfer, the HtoD is at around 7 GiB/sec, compared to the 4.5 GiB/sec on the P6000. This, once again, can either be attributed to the higher performance of the A100 or to the updated driver versions. Furthermore, the graph shows a clear fluctuation of the bandwidth in the timeline after the memory copy operations have concluded. As the program should only be running the error checking and freeing the memory at this time, it is difficult to say what exactly causes the memory transfer speeds to fluctuate, as none of those operations should have any PCIe link activity.

Figure 5.7 and 5.8 show that the behavior of the wrapper, when varying the chunk sizes used to monitor the link with, behave similarly to the default chunk size of 200KB. It is also worth noting that the same lack of data applies for the same duration over the first memory copy on the graph, across all different chunk sizes, once again caused by the sequential execution and low measured bandwidths. Furthermore, the fluctuations in bandwidth after the memory copies have concluded are also present, but in different lengths and magnitudes across the various buffer sizes.

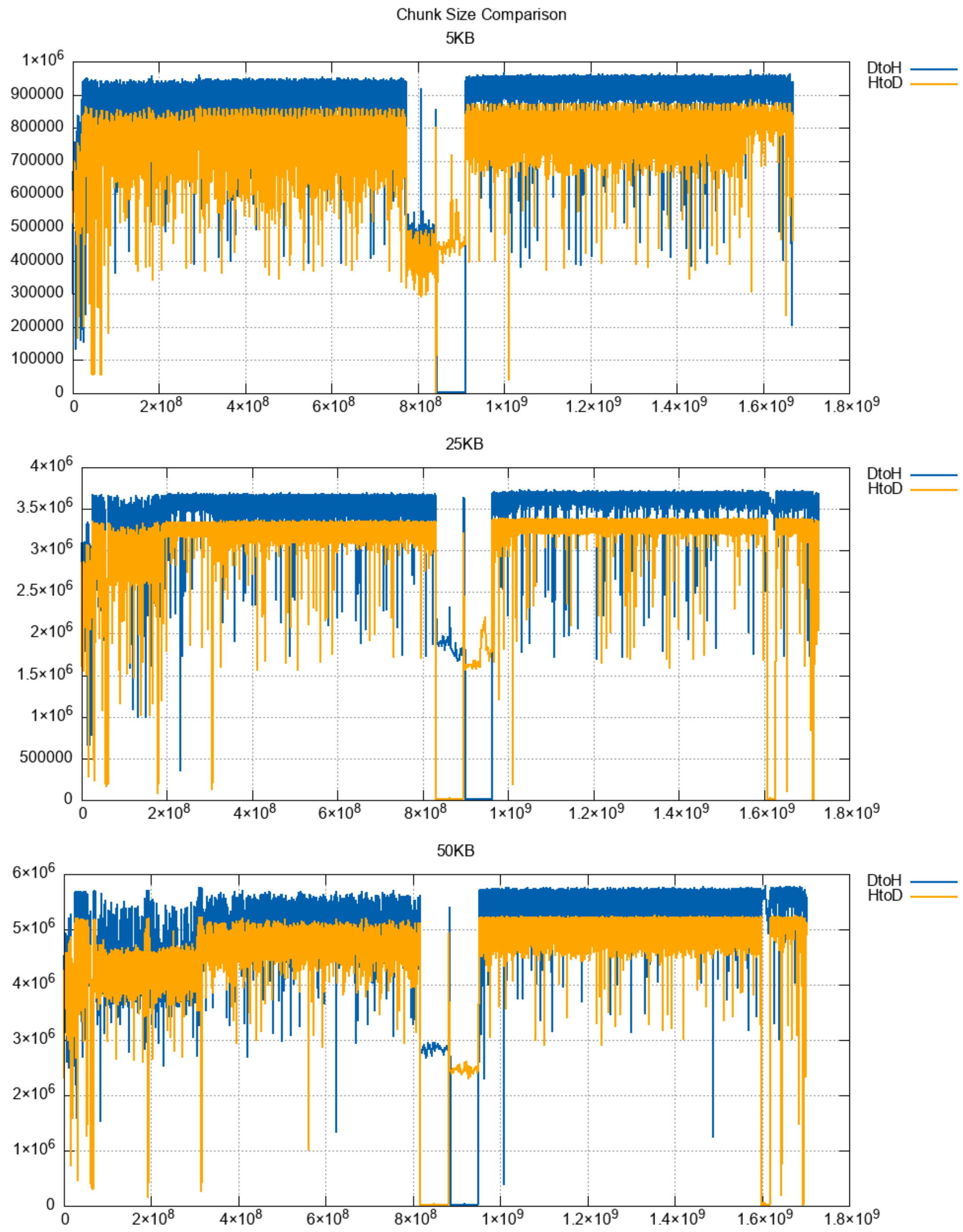


Figure 5.4: Chunk size variation of bandwidth saturation (400MB vector, P6000, part 1)

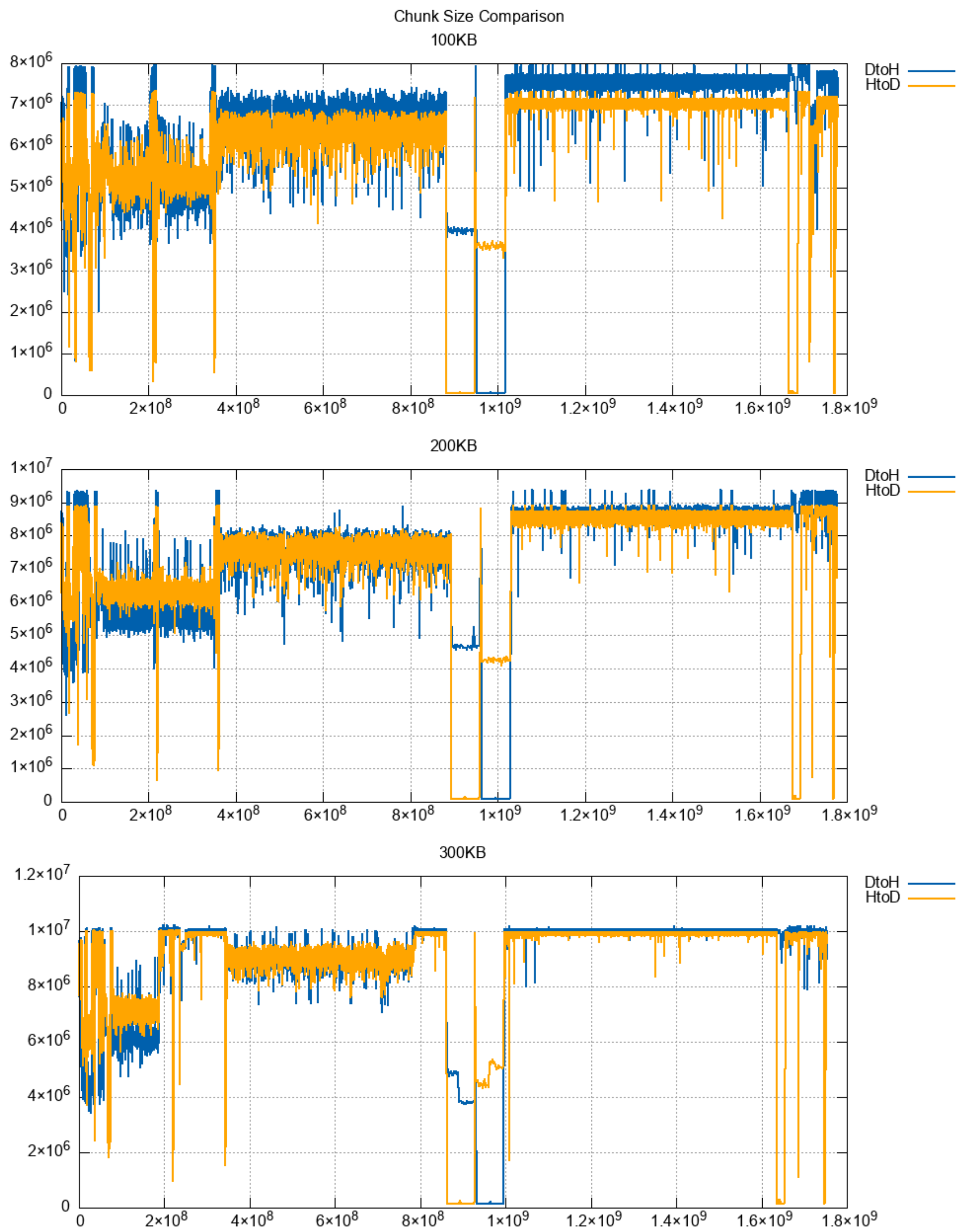


Figure 5.5: Chunk size variation of bandwidth saturation (400MB vector, P6000, part 2)

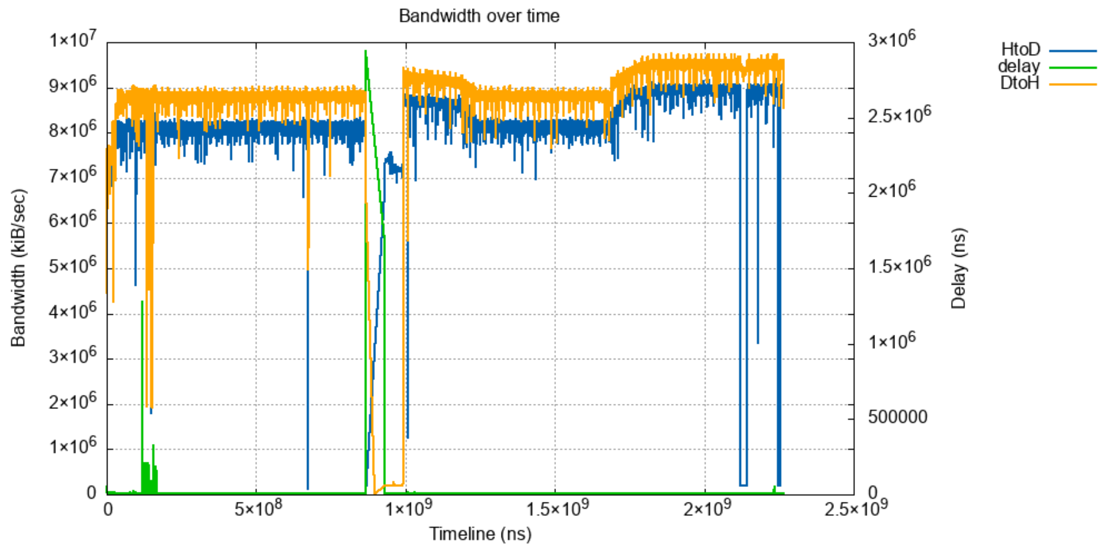


Figure 5.6: Bandwidth saturation monitoring of vector addition (400MB vector, Ice1)

The 300KB graph shows a similar decrease in performance around the beginning of the graph as Figure 5.3, indicating that this fluctuation may be more prominent with larger buffer sizes. However, this is not in line with the observations from Figure 5.4, as some fluctuations were present even on the smaller buffer sizes on the P6000.

The time taken to execute the vector add program is actually shorter when using the wrapper, compared to running it standalone. Execution times are around 2.2 seconds when the monitor is attached, and range anywhere from 3 to 4.5 seconds when the wrapper is not attached. The majority of the increased runtime is located before the first memory copy, indicating that the CPU is slower in either allocating or populating the vectors when the program is running standalone. However, the data gathered on another occasion indicates that the runtime is around 1.5 seconds, with no difference regarding the wrapper's presence.

5.4.2 Matrix Multiplication

P6000

Figure 5.9 shows the PCIe link activity of the matrix multiplication kernel, or lack thereof, when monitored through the link saturation approach. The bandwidths are at a constant 9.5 GiB/sec, with some fluctuations throughout the runtime, similar to the results shown in Figure 5.3. The memory copies are not very visible on the graph, but, similar to the results observed on the P6000, can be found in the raw data. Technically,

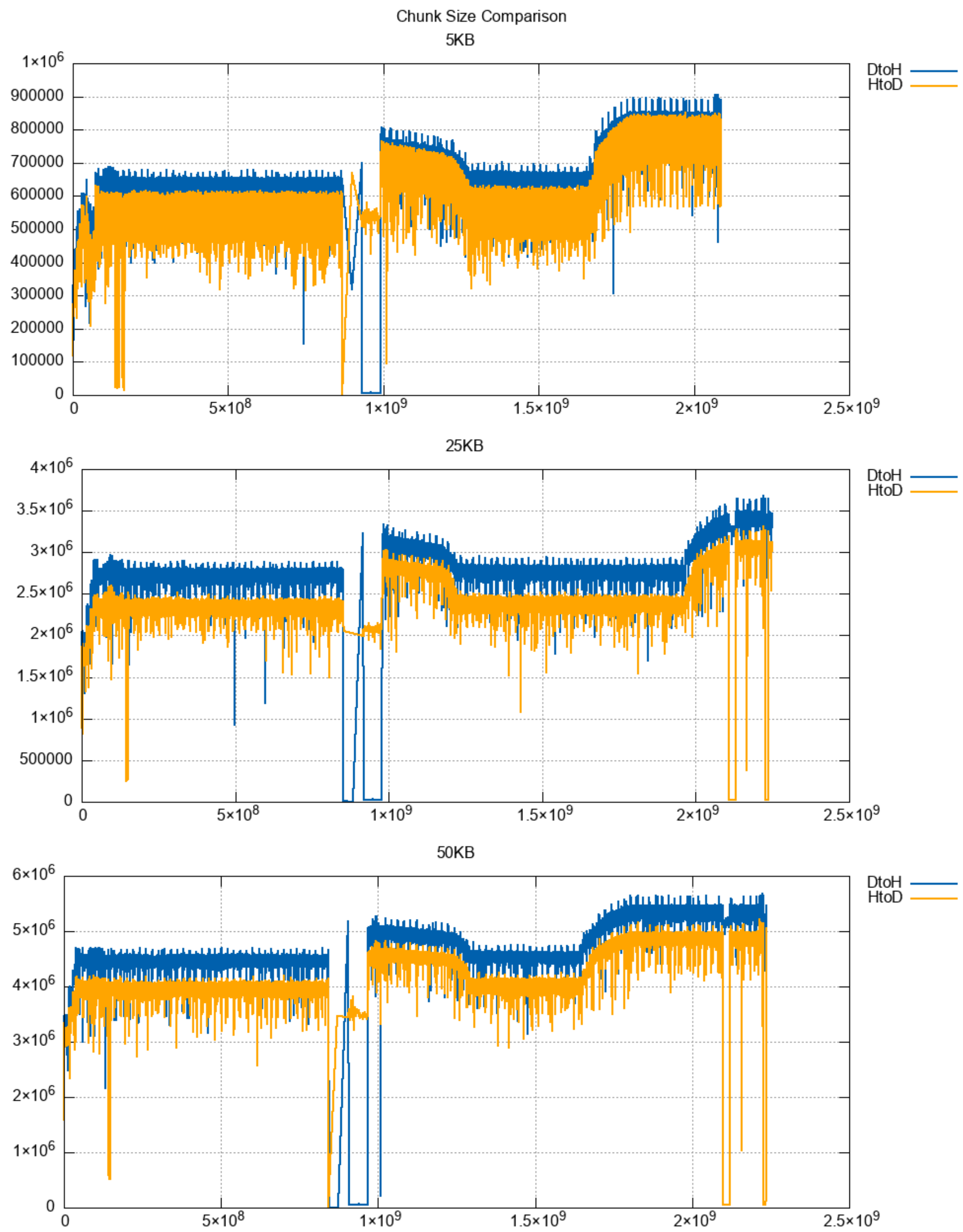


Figure 5.7: Chunk size variation of bandwidth saturation (400MB vector, Ice1, part 1)

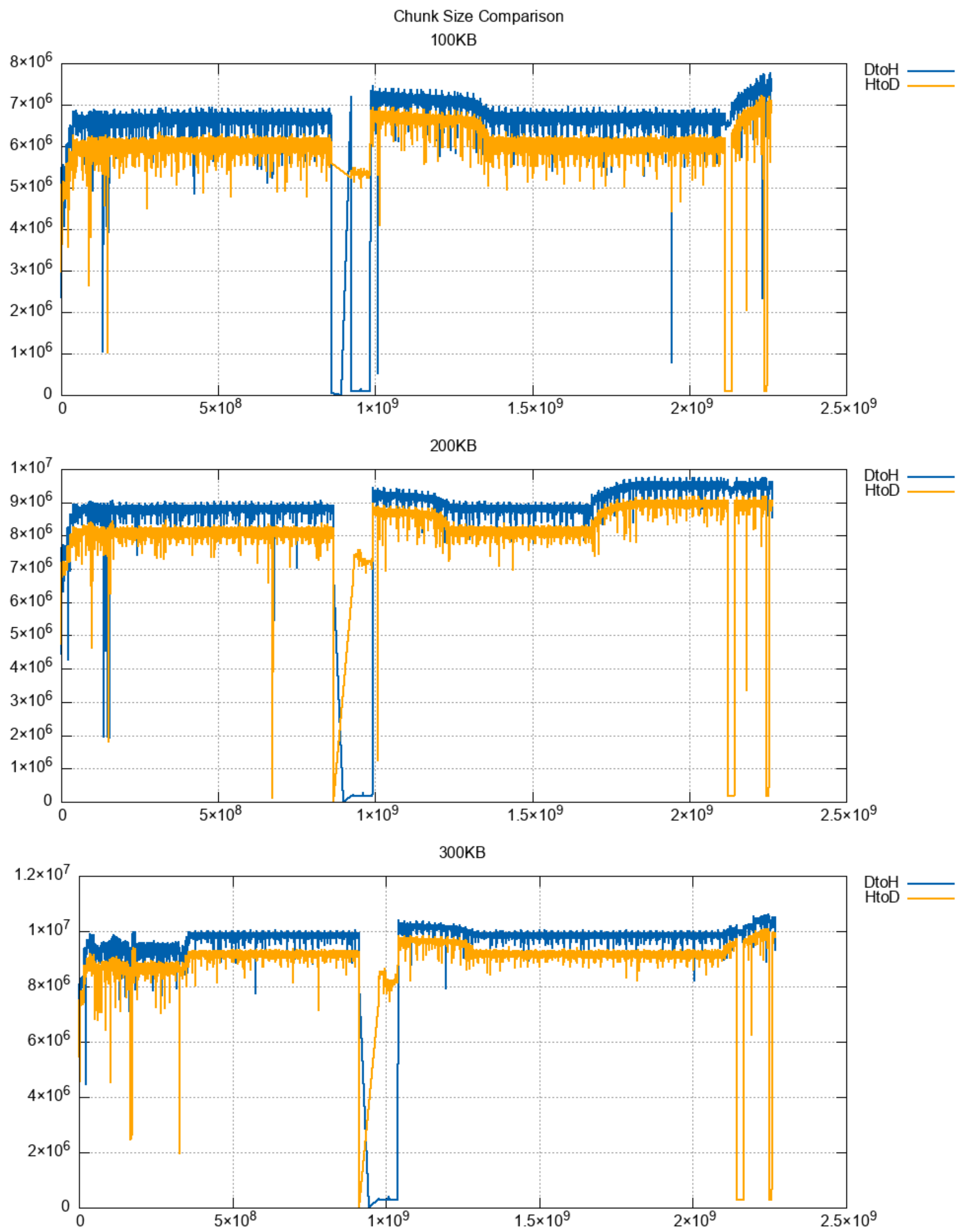


Figure 5.8: Chunk size variation of bandwidth saturation (400MB vector, Ice1, part 2)

the dip of the graphs in the very beginning (HtoD) and the very end (DtoH) are the memory transfers, which are not very visible.

An interesting finding is that there is a consistent delay of 2.2 milliseconds that is shown in the graph, which corresponds to the time the GPU is under load as indicated by the NVML graph. This shows a clear correlation between GPU load and PCIe link delay, indicating that memory transfers are ideally done when the PCIe link is not under load.

Finally, there is a significant overhead involved when it comes to using this monitor, as runtimes of the program increased by as much as 10 to 15% when the monitor was attached, over an average of about 10 runs. The average runtime of the standalone program is about 24 seconds, and attaching the monitor increases that runtime to about 26.5 seconds on average.

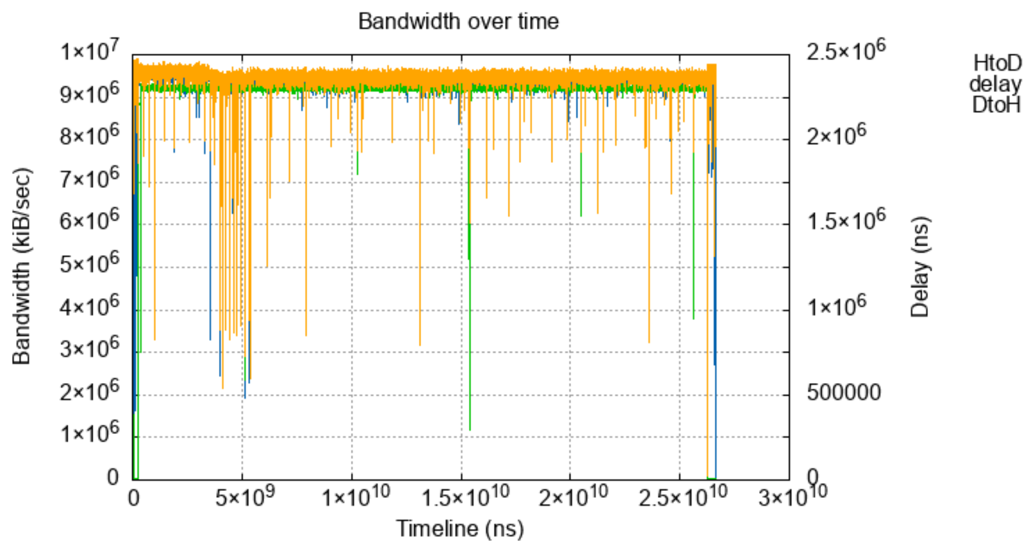


Figure 5.9: Bandwidth saturation monitoring of matrix multiplication (P6000)

Ice1

Figure 5.10 shows the link saturation monitoring of the matrix multiplication program on the Ice1 system. The memory copies are not very visible on the graph due to the large amount of datapoints that make up the graph, but, similar to the results observed on the P6000, can be found in the raw data.

Once again, the delay of about 2.2 milliseconds is observed when the GPU is performing the matrix multiplications, once again indicating that there is a clear correlation

between PCIe delay and GPU load.

Furthermore, the timestamps do not match with the PCIe load and delay measurements, similar to the graphs shown in Section 4.4.2. The delay starts roughly at the same time as the memory transfer is shown in Figure 4.10, indicating that the kernel is started, and the GPU put under load, around that time. This once again indicates that some sections of the code are not executed in sequence due to optimizations made by the compiler.

The graph also shows a peak bandwidth of 10 GiB/sec, which is lower than the 12 GiB/sec measured in Chapter 3. This bandwidth is also affected by the PCIe link delay and is likely to be higher if the delay was compensated for in the calculation.

Finally, this test once again shows that there is a clear overhead when using this wrapper, as execution times have gone up by about 1 to 2 seconds with the wrapper attached, from 11 up to 12 or 13 seconds. This observation was made by running the program 10 times with and without the monitor respectively, and taking the timestamps from the console output as execution time, and the percentage overhead of about 15% is in-line with the observations from the P6000.

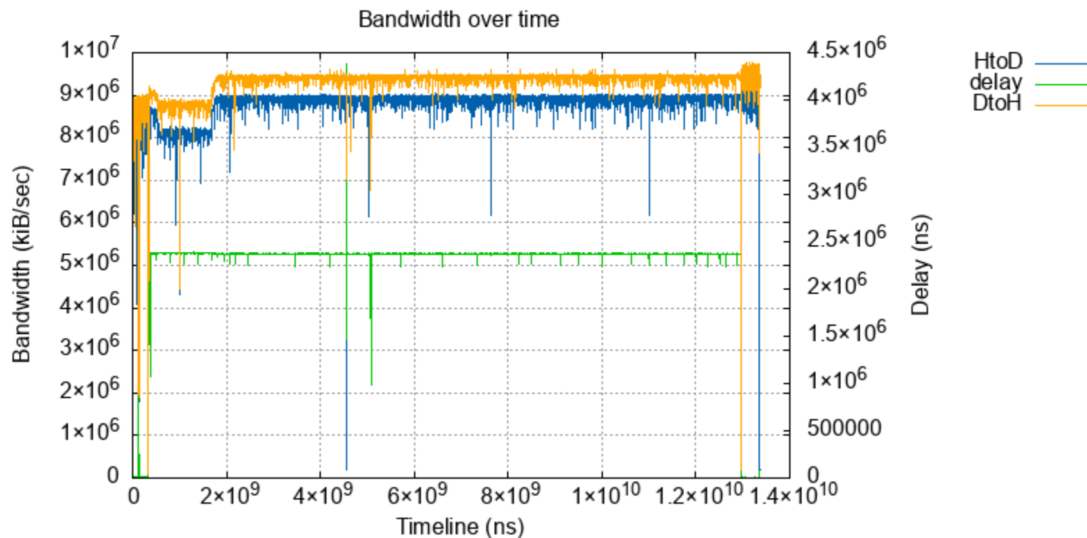


Figure 5.10: Bandwidth saturation monitoring of matrix multiplication (Ice1)

5.5 Discussion

5.5.1 Successes

The primary success of the link saturation approach is the ability to measure shorter memory transfers, improving on the primary shortcoming of the NVML-based monitor. This is especially apparent regarding the memory transfers in the matrix multiplication program, which is about a seventh of the 800MB total found in the vector add program.

A further finding is the clear correlation between GPU load and PCIe link delay, as shown in the matrix multiplication results. Whilst the delay does not impact bandwidth in a major way, it does mean that any data transferred will take a bit longer to arrive.

Furthermore, the monitor is able to detect data travel over the PCIe link despite not fully saturating the link, as even small packets show the data transfers with full bandwidth. However, it is worth noting that the lower the bandwidth of the transfer, the less likely it is to be detected with smaller chunk sizes that do not fully saturate a PCIe link. It is also worth noting that smaller chunk sizes will lead to marginally less overhead.

Finally, the tool fulfills most of the goals described in Section **??**. It is able to detect shorter data transfers, doesn't require any hardware outside of a CUDA-capable GPU, outputs the data to a file, and monitors both host to device and device to host traffic of a given program, with no requirements to modify said program.

5.5.2 Shortcomings

The major drawback of this approach is the amount of overhead it introduces to the program it is monitoring, with an runtime increase of up to 15%, which is in direct opposition of a goals defined in **??**. Furthermore, the granularity of the measurements is dependent on the chunk size, with smaller chunk sizes equating to more datapoints and a more detailed measurement. This also directly opposes the initial set goal of 'low overhead'.

Another issue is the sequential implementation of the monitoring thread, as a single blocking call of *cudaMemcpy*() can block the entire measurement. This includes the delay measurement as well, further compounding the issue, especially when the GPU is under compute load, which increases delays significantly.

Finally, the wrapper does not give a comprehensive insight into the PCIe traffic properties, but rather a timeline of PCIe link utilization, with lower values indicating a higher link utilization. This, however, is not entirely accurate, as seen by the results described in Section 5.4.1, as the bandwidth fluctuates without the PCIe link experiencing traffic.

6 Further Reading

A paper by Zhu et al. discusses intercepting PCIe traffic and dissecting packet payloads to reconstruct DNN models, using the PCIe link as attack vector[39]. This is accomplished by using a hardware-based bus snooping device to dissect PCIe packets and payloads. A further finding noted in that paper is that PCIe payload structures are not documented, along with the fact that there is significant noise involved when dissecting CUDA data traffic. The paper also dissected the payload structure of CUDA packets in further detail.

Additionally, there are several different vendors for hardware-based PCIe traffic analyzers and interposers. The paper presented by Nakamura et al. used a PCIe interposer by Teledyne LeCroy to dissect PCIe data in further detail [32]. These interposers are usually expensive, ones made by Teledyne LeCroy are priced on quote [40]. Software by Teledyne is also closed-source, however the solutions provide a detailed insight into PCIe packets and traffic.

Furthermore, there is a set of software-based tools available to monitor hardware counters on both the CPU and GPU side. These counters vary by CPU architecture to some degree, meaning that counters are not consistent across different generations of CPUs.

Likwid, one of these software-based tools, is able to monitor counters for most Intel and AMD CPU architectures with $likwid - perfctr$. The program either be used to monitor specific regions of code by using a header file and linking a library, or be used in a wrapper-style format similar to the tools developed in Chapters 4 and 5 [41]. Furthermore, Likwid has a NVIDIA GPU backend, which is currently deprecated for modern Nvidia GPUs [41].

Similar tools include Intel's Processor Counter Monitor (PCM), which is only compatible with Intel CPUs, perf, which is a performance counter monitor built into the Linux kernel, and PAPI, a tool developed by the University of Tennessee's innovative computing laboratory [42], [43], [44].

Another approach to monitoring PCIe traffic is the NVIDIA Visual Profiler (NVVP), which, as the name suggests, is capable of profiling CUDA applications. AMD has a similar tool, called AMD uProf, which is also able to monitor CPU events, along with AMD GPUs [45], [46].

7 Summary

This thesis documented the development of a set of tools with the goal to gain insight on PCIe data movements. The process first defines the basic concept of the tool and assesses the primary goals and a set of requirements the final tool should fulfill. After that, the implementation of the program is described in further detail, and data is gathered. The findings from the data and the tool's properties are then discussed.

The bandwidth benchmark is mostly a way to assess a system's PCIe link capabilities, which is done by timing memory copy operations of increasing chunk sizes to determine the link bandwidth. The benchmark measures both pinned and pageable memory transfers, device to host, and compensates for the delay. However, that delay compensation sometimes causes negative values for pageable memory transfers.

The NVML monitor relies on the NVML library's exposed hardware counters to measure another program's PCIe link activity. This is done by creating a wrapper-style program that starts both the monitor and the program that is to be monitored. The monitoring thread continuously queries the counters and prints the results into a file to facilitate plotting the graph. The results can be plotted into a graph that shows the PCIe link activity over the course of the program's runtime.

The final tool utilizes PCIe link saturation to measure both PCIe link traffic and GPU processing load by repeatedly copying small chunks of memory to and from the GPU. The metrics measured are the link delay, which indicates GPU load, and the bandwidth, which indicates data traffic. This approach shares the wrapper-style implementation with the NVML monitor, and monitors the PCIe link traffic over the duration of a program's runtime.

7.1 Successes

The tools developed in this thesis give a basic insight into the properties of PCIe data movements, such as the bandwidth, and link activity profiles.

Further insights worth mentioning is the correlation between PCIe delay and link activity, and the lack of correlation between PCIe delays and PCIe link load found by the link saturation approach described in Chapter 5.

Another success is that there is no external and expensive hardware required to use the tools developed, unlike the approaches discussed in Chapter 6. Further, all

the requirements defined in the respective approaches' chapters were met, with the exception of the 'low overhead' requirement in Chapter 5. This was expected to some degree, due to the concept of putting the PCIe link under load.

It is also worth mentioning that the tools developed are compatible with both Intel and AMD's CPUs. The software-based approaches described in Chapter 6 are limited in compatibility, with Intel's PCM only being compatible with Intel processors, and Likwid only being compatible with a given set of processor architectures.

Finally, it should be noted that these tools can easily be translated to be compatible with AMD's HIP API due to the similarities HIP and CUDA share, for compatibility with AMD-based GPUs. This can even be automated, to a degree, with the HIPify tool providing an automated source-to-source transformation from CUDA to HIP [47], . It is worth noting, however, that HIP does have an equivalent to the NVML library (ROCm-smi), which however lacks an equivalent to the counters that NVML uses, and as such, translating the wrapper from Chapter 4 is likely an impossibility [48] [49].

7.2 Problems

The first major failing of this thesis is the inability to delve into lower-level data traffic, unlike the hardware-based solutions presented in Chapter 6.

Additionally, it is impossible to leverage both the NVML and link saturation measurements at the same time, because NVML will record the activity of the repeated copy operations as well. As such, measurements on the same program need to be done over at least two separate runs, subjecting the results to some level of variance and uncertainty. This issue is further compounded by the overhead the link saturation approach introduces, making the task of comparing timelines difficult.

Finally, it is worth noting that both wrappers, by themselves, have multiple significant drawbacks and are not universally applicable in most use cases. Both approaches will struggle to pick up memory transfers shorter than a few hundred kilobytes in size, as example. Additionally, workarounds to these drawbacks will be challenging to develop, primarily due to the closed-source nature of CUDA.

7.3 Conclusion

To conclude, it seems that there is a hardware requirement for a PCIe interposer in order to gain deeper insight into PCIe data traffic. Additionally, there is a large amount of research to be done towards understanding PCIe payloads and packets, as seen in the closed-source approach of NVIDIA and their CUDA API.

The benchmark developed in Chapter 3 of this thesis is a baseline tool to determine the capabilities of a given PCIe link, such as bandwidth and saturation threshold.

Additionally, the two wrappers give a basic insight into the activity of the PCIe link caused by a given program. The primary drawback is that any non-related PCIe traffic to the target device will also be recorded, and that this may lead to measurement inaccuracies if there is other software running.

Finally, the tools developed offer a unique blend of universal compatibility, automation, and insight into the data movements between CPU and GPU.

7.4 Outlook

A further direction to research may be the manual writing of data headers and payloads, attempting to manually communicate with a PCIe device. Some information regarding this approach can be found in the Linux kernel documentation [50].

Furthermore, there are more complex tests possible for the monitors implemented in Chapters 4 and 5, to further gain insight on how they work and their functionality, as the two programs used were simple in nature to facilitate data-gathering and a proof of concept. One such example may be a series of concurrent memory copies on the Ice1 system, to gain insight on how the GPU schedules memory copy operations. Another example would be a program that simultaneously puts the GPU under load and copies data to or from the GPU, to verify the monitors' behaviors.

Additionally, it bears merit to test the tools developed in different hardware configurations, as they have not been tested or verified for PCIe 4.0 traffic. It also bears merit to translate the CUDA code to HIP, where applicable, to gain further insight into AMD GPUs.

Finally, the automation could be improved upon, as there is still some amount of human input needed to both gather data and to plot the data.

List of Figures

2.1 The C++ function to allocate memory[16] 7
2.2 The CUDA function to allocate pinned memory [20] 7
2.3 The difference between copying pinned and pageable memory [19] . . . 7
2.4 The CUDA function to allocate device memory [20] 8
2.5 The CUDA function to allocate managed memory [20] 8
2.6 The CUDA function to copy memory [20] 9
2.7 The C++ function to free a chunk of memory allocated by $malloc()$ [21] 9
2.8 The CUDA function to free a chunk of device memory[20] 9
2.9 The CUDA function to free a chunk of host memory[20] 10
2.10 The NVML function read the PCIe link throughput [20] 10
2.11 An example of a PCI-Express packet [27] 12
2.12 An example of a memory request header [27] 13
2.13 PCIe configuration on an Intel-based system [32] 15
2.14 An example PCIe link between two devices [33] 16

3.1 Rough Flowchart of the bandwidth measurement method 22
3.2 Code snippet measuring the $CudaMemcpy()$ delay 23
3.3 Pinned and pageable memory copy bandwidths (P6000) 24
3.4 Pinned and pageable memory copy duration (P6000) 24
3.5 Pinned and pageable memory copy bandwidths (Ice1) 25
3.6 Pinned and pageable memory copy duration (Ice1) 26

4.1 Code snippet showing thread behavior 29
4.2 Rough Flowchart of the bandwidth measurement method 30
4.3 NVML monitoring of vector addition (400MB vector, P6000) 32
4.4 NVML: Monitoring vectors of different sizes (P6000, part 1) 33
4.5 NVML: Monitoring vectors of different sizes (P6000, part 2) 34
4.6 NVML monitoring of vector addition (400MB vector, Ice1) 35
4.7 NVML: Monitoring vectors of different sizes (Ice1, part 1) 36
4.8 NVML: Monitoring vectors of different sizes (Ice1, part 2) 37
4.9 NVML monitoring of matrix multiplication (P6000) 38
4.10 NVML monitoring of matrix multiplication (Ice1) 39

5.1 Flowchart of the copy wrapper 43
5.2 Code snippet depicting monitoring thread measurements 44
5.3 Bandwidth saturation monitoring of vector addition (400MB vector, P6000) 45
5.4 Chunk size variation of bandwidth saturation (400MB vector, P6000, part 1) 47
5.5 Chunk size variation of bandwidth saturation (400MB vector, P6000, part 2) 48
5.6 Bandwidth saturation monitoring of vector addition (400MB vector, Ice1) 49
5.7 Chunk size variation of bandwidth saturation (400MB vector, Ice1, part 1) 50
5.8 Chunk size variation of bandwidth saturation (400MB vector, Ice1, part 2) 51
5.9 Bandwidth saturation monitoring of matrix multiplication (P6000) . . . 52
5.10 Bandwidth saturation monitoring of matrix multiplication (Ice1) 53

List of Tables

2.1 PCI Express aggregate bandwidths by generation and link width [30] [29] 11

Bibliography

[1] G. E. Moore, "Cramming more components onto integrated circuits," vol. 38, no. 8, p. 6, 1965.

[2] R. Dennard, F. Gaensslen, H.-N. Yu, V. Rideout, E. Bassous, and A. LeBlanc, "Design of ion-implanted MOSFET's with very small physical dimensions," *IEEE Journal of Solid-State Circuits*, vol. 9, no. 5, pp. 256–268, Oct. 1974, ISSN: 0018-9200, 1558-173X. DOI: 10.1109/JSSC.1974.1050511.

[3] J. M. Shalf and R. Leland, "Computing beyond moore's law," *Computer*, vol. 48, no. 12, pp. 14–23, Dec. 2015, ISSN: 0018-9162. DOI: 10.1109/MC.2015.374.

[4] TSMC. "5nm technology - taiwan semiconductor manufacturing company limited." (2022), [Online]. Available: https://www.tsmc.com/english/dedicatedFoundry/technology/logic/l_5nm (visited on 02/25/2022).

[5] J. Shalf, "HPC interconnects at the end of moore's law," in *2019 Optical Fiber Communications Conference and Exhibition (OFC)*, Mar. 2019, pp. 1–3.

[6] IBM. "What is HPC? introduction to high-performance computing | IBM." (2022), [Online]. Available: https://www.ibm.com/topics/hpc (visited on 02/20/2022).

[7] I. S. University. "What is an HPC cluster | high performance computing." (2020), [Online]. Available: https://www.hpc.iastate.edu/guides/introduction-to-hpc-clusters/what-is-an-hpc-cluster (visited on 02/20/2022).

[8] Intel. "What is a GPU? graphics processing units defined," Intel. (2022), [Online]. Available: https://www.intel.com/content/www/us/en/products/docs/processors/what-is-a-gpu.html (visited on 02/15/2022).

[9] C. McClanahan, "History and evolution of GPU architecture," 2010, p. 7.

[10] Nvidia. "CUDA zone," NVIDIA Developer. (Jul. 18, 2017), [Online]. Available: https://developer.nvidia.com/cuda-zone (visited on 02/19/2022).

[11] Nvidia. "NVIDIA GeForce RTX 3090 graphics card." (2020), [Online]. Available: https://www.nvidia.com/en-us/geforce/graphics-cards/30-series/rtx-3090/ (visited on 02/19/2022).

[12] I. Micron Technology. "GDDR6x." (2022), [Online]. Available: https://www.micron.com/products/ultra-bandwidth-solutions/gddr6x (visited on 02/26/2022).

[13] I. Micron Technology. "RAM memory speeds & compatibility | crucial.com," Crucial. (2022), [Online]. Available: https://www.crucial.com/support/memory-speeds-compatability (visited on 02/19/2022).

[14] Nvidia and R. Pramod. "CUDA 11 features revealed," NVIDIA Developer Blog. (May 14, 2020), [Online]. Available: https://developer.nvidia.com/blog/cuda-11-features-revealed/ (visited on 02/15/2022).

[15] Nvidia. "CUDA c++ programming guide." (Feb. 3, 2022), [Online]. Available: https://docs.nvidia.com/cuda/cuda-c-programming-guide/index.html (visited on 02/22/2022).

[16] "Malloc - c++ reference." (2021), [Online]. Available: https://www.cplusplus.com/reference/cstdlib/malloc/ (visited on 02/27/2022).

[17] Nvidia and M. Harris. "Unified memory for CUDA beginners," NVIDIA Technical Blog. (Jun. 20, 2017), [Online]. Available: https://developer.nvidia.com/blog/unified-memory-cuda-beginners/ (visited on 02/25/2022).

[18] "Size_t - c++ reference." (2021), [Online]. Available: https://www.cplusplus.com/reference/cstdlib/size_t/ (visited on 02/27/2022).

[19] Nvidia and M. Harris. "How to optimize data transfers in CUDA c/c++," NVIDIA Technical Blog. (Dec. 5, 2012), [Online]. Available: https://developer.nvidia.com/blog/how-optimize-data-transfers-cuda-cc/ (visited on 02/25/2022).

[20] Nvidia. "CUDA toolkit documentation: Memory management." (Feb. 22, 2022), [Online]. Available: https://docs.nvidia.com/cuda/cuda-runtime-api/group__CUDART__MEMORY.html (visited on 02/27/2022).

[21] "Free - c++ reference." (2021), [Online]. Available: https://www.cplusplus.com/reference/cstdlib/free/ (visited on 04/11/2022).

[22] Nvidia. "NVIDIA management library (NVML)," NVIDIA Developer. (Oct. 21, 2011), [Online]. Available: https://developer.nvidia.com/nvidia-management-library-nvml (visited on 02/25/2022).

[23] Nvidia. "NVML device queries." (Jul. 29, 2021), [Online]. Available: http://docs.nvidia.com/deploy/nvml-api/index.html (visited on 02/27/2022).

[24] PCI-SIG. "PCI express architecture frequently asked questions." (Sep. 17, 2011), [Online]. Available: https://web.archive.org/web/20110917001426/http://www.pcisig.com/news_room/faqs/faq_express/pciexpress_faq.pdf (visited on 02/17/2022).

[25] A. Verma and P. Dahiya, "PCIe BUS: A state-of-the-art-review," *IOSR Journal of VLSI and Signal Processing (IOSR-JVSP)*, vol. 7, pp. 24–28, Jul. 12, 2017. DOI: 10.9790/4200-0704012428.

[26] PCI-SIG. “Contact us | PCI-SIG.” (2022), [Online]. Available: https://pcisig.com/membership/contact-us (visited on 02/14/2022).

[27] J. Lawley, “Understanding performance of PCI express systems,” p. 16, 2014.

[28] PCI-SIG. “Membership | PCI-SIG.” (2022), [Online]. Available: https://pcisig.com/membership (visited on 02/14/2022).

[29] PCI-SIG. “Frequently asked questions | PCI-SIG.” (2022), [Online]. Available: https://pcisig.com/faq?field_category_value%5B%5D=pci_express_4.0&keys= (visited on 04/08/2022).

[30] M. Jackson, R. Budruk, J. Winkles, and D. Anderson, *PCI Express technology: comprehensive guide to generations 1.x, 2.x, 3.0* (MindShare technology series), 1st ed. Monument, Colo.: MindShare, 2012, 986 pp., OCLC: ocn824814290, ISBN: 978-0-9770878-6-0.

[31] I. Oracle. “PCI address domain - oracle documentation.” (2010), [Online]. Available: https://docs.oracle.com/cd/E19253-01/816-4854/hwovr-25/index.html (visited on 02/14/2022).

[32] H. Nakamura, H. Takayama, Y. Yamaguchi, and T. Boku, “Thorough analysis of PCIe gen3 communication,” in *2017 International Conference on ReConFigurable Computing and FPGAs (ReConFig)*, Dec. 2017, pp. 1–6. DOI: 10.1109/RECONFIG.2017.8279824.

[33] R. Budruk. “PCI express basics.” (Jul. 15, 2014), [Online]. Available: https://web.archive.org/web/20140715120034/http://www.pcisig.com/developers/main/training_materials/get_document?doc_id=4e00a39acaa5c5a8ee44ebb07baba982e5972c67 (visited on 02/19/2022).

[34] D. D. Sharma, “PCI express® 6.0 specification at 64.0 GT/s with PAM-4 signaling: A low latency, high bandwidth, high reliability and cost-effective interconnect,” in *2020 IEEE Symposium on High-Performance Interconnects (HOTI)*, ISSN: 2332-5569, Aug. 2020, pp. 1–8. DOI: 10.1109/HOTI51249.2020.00016.

[35] Intel. “Product specifications - i9 12900k.” (2022), [Online]. Available: https://www.intel.com/content/www/us/en/products/sku/134599/intel-core-i912900k-processor-30m-cache-up-to-5-20-ghz.html (visited on 02/20/2022).

[36] Intel. “What is thunderbolt 4?” Intel. (2022), [Online]. Available: https://www.intel.com/content/www/us/en/gaming/resources/upgrade-gaming-accessories-thunderbolt-4.html (visited on 02/20/2022).

[37] Kingston. "Understanding SSD technology: NVMe, SATA, m.2 - kingston technology," Kingston Technology Company. (Feb. 2017), [Online]. Available: https://www.kingston.com/germany/en/community/articledetail/articleid/48543 (visited on 02/20/2022).

[38] *CUDA samples*, original-date: 2018-03-27T17:36:24Z, Apr. 11, 2022.

[39] Y. Zhu, Y. Cheng, and H. Zhou, "Hermes attack: Steal DNN models with lossless inference accuracy," p. 17,

[40] T. LeCroy. "Teledyne LeCroy - protocol analyzer - PCI express." (2022), [Online]. Available: https://teledynelecroy.com/protocolanalyzer/pci-express (visited on 04/11/2022).

[41] "Likwid perfctr · RRZE-HPC/likwid wiki," GitHub. (Mar. 2, 2022), [Online]. Available: https://github.com/RRZE-HPC/likwid (visited on 04/11/2022).

[42] opcm, *Opcm/pcm*, original-date: 2016-10-31T10:24:58Z, Apr. 11, 2022.

[43] "Perf wiki." (Jun. 12, 2020), [Online]. Available: https://perf.wiki.kernel.org/index.php/Main_Page (visited on 04/11/2022).

[44] M. S. Müller, Ed., *Tools for high performance computing 2009: proceedings of the 3rd International Workshop on Parallel Tools for High Performance Computing, September 2009, ZIH, Dresden*, Meeting Name: International Workshop on Parallel Tools for High Performance Computing OCLC: ocn506253045, Heidelberg ; New York: Springer, 2010, 184 pp., ISBN: 978-3-642-11260-7.

[45] NVIDIA. "NVIDIA visual profiler," NVIDIA Developer. (Feb. 6, 2012), [Online]. Available: https://developer.nvidia.com/nvidia-visual-profiler (visited on 04/11/2022).

[46] AMD. "AMD uprof," AMD. (Jan. 17, 2022), [Online]. Available: https://developer.amd.com/amd-uprof/ (visited on 04/11/2022).

[47] AMD. "HIP programming guide v4.5 — ROCm 4.5.0 documentation." (2022), [Online]. Available: https://rocmdocs.amd.com/en/latest/Programming_Guides/HIP-GUIDE.html (visited on 04/11/2022).

[48] AMD. "ROCm command line interface — ROCm 4.5.0 documentation." (2022), [Online]. Available: https://rocmdocs.amd.com/en/latest/ROCm_System_Managment/ROCm-SMI-CLI.html (visited on 04/11/2022).

[49] *HIPIFY*, original-date: 2020-03-02T22:10:40Z, Apr. 8, 2022.

[50] "1. how to write linux PCI drivers — the linux kernel documentation." (2022), [Online]. Available: https://docs.kernel.org/PCI/pci.html (visited on 03/05/2022).