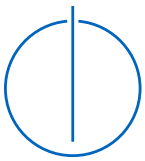


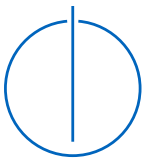
DEPARTMENT OF INFORMATICS

TECHNISCHE UNIVERSITÄT MÜNCHEN

Bachelor's Thesis in Informatics: Game Engineering

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

THE Sofi



DEPARTMENT OF INFORMATICS

TECHNISCHE UNIVERSITÄT MÜNCHEN

Bachelor's Thesis in Informatics: Game Engineering

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

## Titel der Abschlussarbeit

| | |
|---|---|
| Author: | THE Sofi |
| Supervisor: | Supervisor |
| Advisor: | thonk |
| Submission Date: | 15th March 2022 |

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

Munich, 15th March 2022 THE Sofi

# Acknowledgments

# Abstract

# Contents

# 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 TSMC, a chip foundry based in Taiwan, 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. On the one hand, this has lead to heterogeneous computing and increased parallelism. On the other hand, this also leads to data movements becoming increasingly frequent and costly, compared to the operations on said data. [cite] A predictable result is that data transfers, and the interconnects in heterogeneous systems, will soon become the bottleneck for computation speeds. As such, it is increasingly important to gain insight into the data movements in heterogeneous systems to optimize both current and future software.

[TODO: DEEP-SEA project?]

## 1.2 Goals and Aims

The goal of this thesis is to gain insight to 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 requirements, listed below:

- lightweight
- automated

- as few requirements as possible
- low overhead
- no need to modify existing programs
- should gain insight to PCIe link activity

[todo: elaborate on requirements]

## 1.3 Structure and Approach

The introduction should first explain the primary motivation of the thesis, with a brief mention to the DEEP-SEA project being made. Then, the goals and aims of the thesis are to be defined in further detail, along with a set of 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 2 should act as a foundation for the remainder of this thesis and introduce most of the concepts mentioned in future chapters.

[TODO: more here as chapters 3,4,5 are finalized]

- - three approaches / benchmarks
- - bandwidth: for measuring the raw bandwidth capacity of the system
- - nvml: for measuring pcie link activity
- - copy: for measuring pcie link activity
- - further reading and discussion

Chapter 6 contains a comparative and evaluating discussion of the tools developed in chapters 3, 4, and 5. Additionally, similar papers and approaches will be briefly touched upon to both give a brief insight into the state-of-the-art and to serve as a target for evaluation purposes.

Chapter 7 summarizes the contents of this thesis and gives an outlook for further development and research.

# 2 Background

## 2.1 HPC

High-performance computing, abbreviated as HPC, leverages the compute capacity of supercomputers or computer clusters to solve problems that are highly complex in nature [5]. 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 [6]. 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 [6].

[question: does this suffice in 'bridging the gap'?]

## 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 [7]. 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. It should also be noted that the graphics card usually has its own dedicated PCB. [does this need citing? - more or less common knowledge?] There are also integrated GPUs, which can be embedded alongside the CPU. These integrated GPUs are usually less powerful compared to discrete GPUs [7].

### 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 [8]. 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 [7]. On the scientific side, GPUs are frequently used to accelerate workloads that require parallel computing, such as machine learning, fluid dynamics, and data science [9].

### 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. [todo: needs citation, where to find?] As example, current top of the line graphics cards from Nvidia are equipped with GDDR6X memory, which has a theoretical maximum system bandwidth of one terabyte per second. [10], [11] On the other hand, state of the art main memory, currently DDR4, is limited to a bandwidth of about 35 gigabytes per second [12].

[TODO: DDR5 standard, however: numbers hard to find. what to put?]

## 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 [9]. Some sample libraries include: linear algebra, signal processing, and image processing [13]. 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 [14].

### 2.3.2 Memory Management

The memory management functionalities that CUDA provides include allocation, deallocation, and data transfer. CUDA assumes that the CPU and GPU maintain separate memory spaces, and is able to manage both host and device memory [14].

#### Host Memory

The degree to which CUDA can manage the main memory is limited, and memory allocation is mostly handled by C++ with the method shown in Figure 2.1 [15]. C++, by default, allocates pageable memory, which means that the data can be paged in and out of RAM into a secondary storage device [16].

The method returns a pointer of the type *void* and has *size* as an argument, indicating the size in bytes the method is to allocate [15]. Size_t is an unsigned integral type used to represent the size of any object in bytes [17].

CUDA, on the other hand, can allocate page-locked memory, also called 'pinned' memory with the method 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 [ref: figure] [18].

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 [19]. 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 [18].

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

Figure 2.1: The C++ method to allocate memory[15]

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

Figure 2.2: The CUDA method to allocate pinned memory [19]

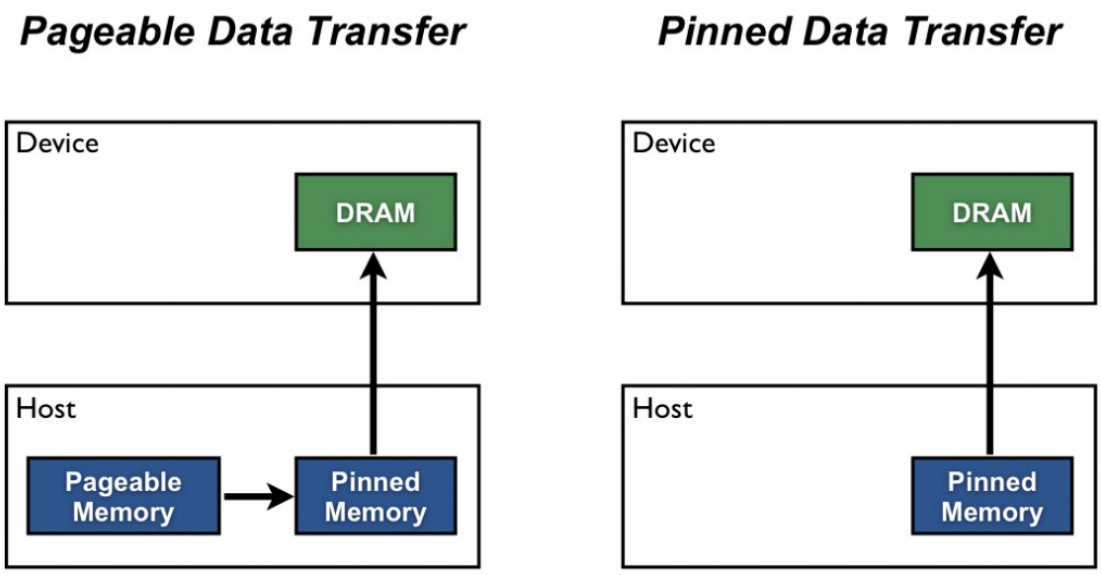



Figure 2.3: The difference between copying pinned and pageable memory [18]

### 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 method shown in Figure 2.4 to allocate memory on the device [19].

The method has similar parameters to the CUDA method 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 method to allocate device memory [19]

### 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 [16]. Figure 2.5 shows the method required to allocate managed memory [19].

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 [19].

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

Figure 2.5: The CUDA method to allocate managed memory [19]

### Memory Copy

CUDA uses one method to copy memory both to and from the device memory, which can be seen in Figure 2.6. The method 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 [19]:

- cudaMemcpyHostToHost
- cudaMemcpyHostToDevice
- cudaMemcpyDeviceToHost
- cudaMemcpyDeviceToDevice
- cudaMemCpyDefault: this is the recommended 'kind', as this causes the method to infer the type of transfer from the pointer values [19].

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

Figure 2.6: The CUDA method to copy memory [19]

### 2.3.3 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 [20]. 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 [21]. Figure 2.7 shows the method 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 [21]:

- 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. [does this need citing? if so cite the book and change wording] Finally, the last parameter *value* points to the location where the counter value will be written to [21].

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

Figure 2.7: The NVML method read the PCIe link throughput [19]

[that should be everything, de-allocation maybe?]

## 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" [22]. 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 [23]. 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 [24]. 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 [22], [25], [26]. 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 [23]. To future-proof the standard, current and future generations of PCIe are to be designed to be compatible with current PCIe standards [26]. So far, each generation of PCIe doubled the previous generation's theoretical maximum bandwidth, as seen in Table 2.1.

[todo: gen4 bandwidths, mention gen4 as SOTA]

| Link Width | x1 | x2 | x4 | x8 | x12 | x16 | x32 |
|---|---|---|---|---|---|---|---|
| Gen1 Bandwidth (GB /s) | 0.5 | 1 | 2 | 4 | 6 | 8 | 16 |
| Gen2 Bandwidth (GB/s) | 1 | 2 | 4 | 8 | 12 | 16 | 32 |
| Gen3 Bandwidth (GB/s) | 2 | 4 | 8 | 16 | 24 | 32 | 64 |

Table 2.1: PCI Express aggregate bandwidths by generation and link width [27]

### 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 2.8, 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 [25].

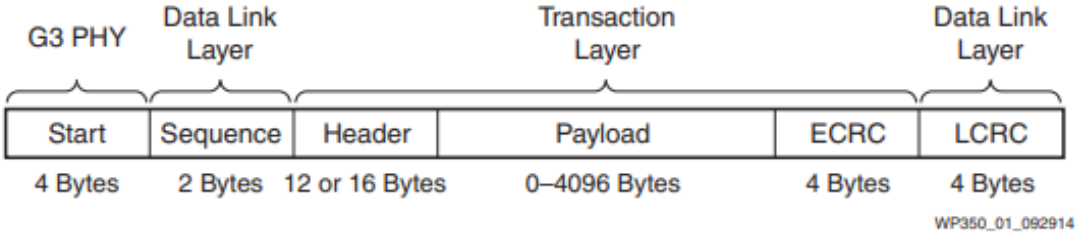



Figure 2.8: An example of a PCI-Express packet [25]

- 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.

[question: do i need a source for each of the bullet points? additionally, the source doesn't outright state that 'this is what this is for', in a degree, but it is inferred knowledge. what should I do about this?]

**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 [28]. As seen in Figure 2.9, the header has a fixed format, similar to an IPv4 or v6 header. The fields and their uses are briefly explained below.

[todo: example IPv4/v6 header]

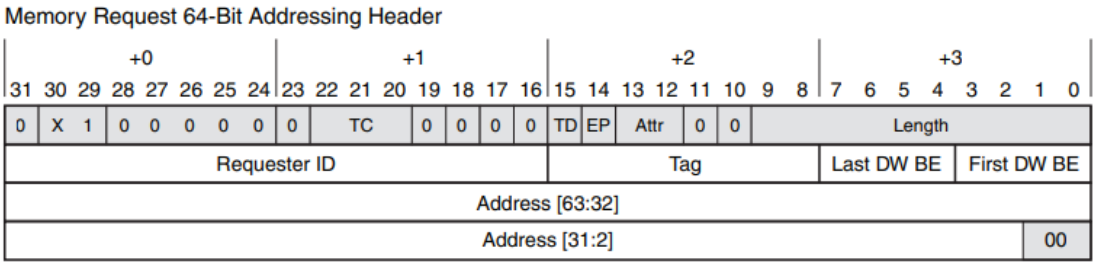



Figure 2.9: An example of a memory request header [25]

- TC: Traffic Class: this denotes the priority of the packet. A larger value represents a higher priority. [27]
- TD: The TLP Digest field. If TD is set to 1, it indicates that there is additional CRC data in the TLP data. [cite xillybus]
- Length: more or less self-explanatory: length denotes the length of the payload in Double Words. [cite xillybus]
- Requester ID: self-explanatory: the ID of the device that requested or sent the packet. [cite xillybus]
- 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. [cite xillybus]
- DW BE fields: DW BE stands for Double-Word Byte Enable. This denotes which of the bytes in the first / last DWs are valid. [cite xillybus]

- 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. [cite xillybus]
- The EP and Attr fields are not further elaborated upon as they are rarely used by PCIe endpoint devices. [cite xillybus]

[todo: update and verify with book]

### 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 [29]. Bridges are used to connect legacy PCI and PCI-X devices with the PCIe root complex [22]. Figure 2.10 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 [27]:

- 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 [28]. It also enables the software to control and check the status of a PCIe device [27].

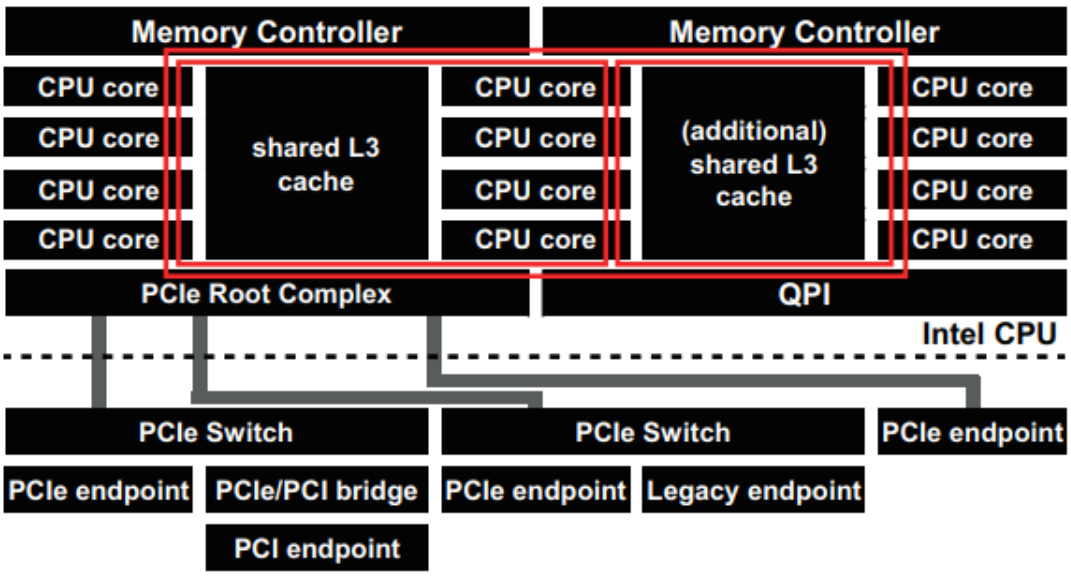



Figure 2.10: PCIe configuration on an Intel-based system [29]

The memory address space is where the internal storage of a PCIe device is mapped [27]. This memory space is also memory-mapped to the CPU's address domain for ease of access by the CPU [28].

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 [27].

### Links and Lanes

A connection between the two PCIe devices is called a link, which is made up of lanes [27]. 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 [27]. 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 [27]. Figure 2.11 illustrates an example PCIe link with several lanes. A normal GPU will usually have a PCIe x16 link.

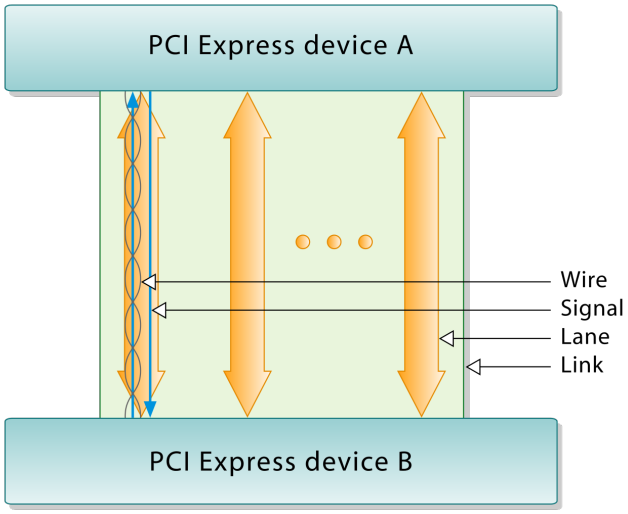



Figure 2.11: An example PCIe link between two devices [30]

### 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 [31], 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 [31]. 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 [32].

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 [33]. Another example is the NVMe standard, a PCIe-compatible interface specifically devised and optimized for high-bandwidth, low-latency storage solutions [34].

# 3 Bandwidth Benchmark

## 3.1 Concept

- - Measures raw theoretical maximum bandwidth of pcie link by measuring the duration of memory copies of various chunk sizes
- - Checks at which chunk sizes the bandwidth of the link is fully saturated
- - goal: insight into very basics of pcie data transfer

## 3.2 Implementation

- - Compensates for delay / overhead - 1 packet with 4B measured as delay / overhead
- - Pageable and pinned memory benchmarks measured
- - Warmup-feature: first transfer usually has some sort of longer delay, compensates for that (windows-finding, verify on p6000)

## 3.3 Results

- - Transfer durations don't really increase until 8kb
- reason: the nature of the PCI-E packet having a max payload of 4kb
- - First transfer usually has a bit longer delay (warmup?)
- - On windows: not executable that calls functions, but rather nvcuda64.dll - requires compiling on windows and then using a profiling tool like AMDUprof to look at the program
- - TODO: graphs

## 3.4 Discussion

### 3.4.1 successes

- - gives accurate reading of pcie bandwidths compared to theoretical maximum (gen3)
- - non-linear scaling of packet transfer durations (2 packets does not equal double the duration of one packet) -> overhead per transfer

### 3.4.2 shortcomings

- - does not really compensate for other bottlenecks, as seen on time-x with gen4 link bandwidth speeds
- - gives little to no insight on 'what is being transferred', and more along the lines of 'how much is theoretical max bandwidth'
- - why warmup? not fully explained, further research needed
-

# 4 NVML Counters

## 4.1 Concept

- - nvidia has hardware counters, accessible via nvml library
- - counters measure average bandwidth over the last 20ms, in kb/sec
- - Transmit and Receive have seperate counters

## 4.2 Implementation

- - method call to read counters takes about 20ms
- - to increase data granularity and measurement consistency, measuring of TX and RX was done in parallel

## 4.3 Results

- graphs and discussing the graphs

## 4.4 Discussion

### 4.4.1 successes

- - measures bandwidths accurately to some degree
- - introduces little overhead (probably, still to be measured)
- - does not require any modifications to software

### 4.4.2 shortcomings

- - granularity of method calls prevent more accurate readings -> short memory transfers may be not detected

- - black box approach of nvidia's source code doesn't allow for proper sanity-checking
- - for more accurate readings and measurements: timestamping of original program -> modification

# 5 Link Saturation

## 5.1 Concept

- - if bandwidth is saturated, copy operations should slow down
- - full duplex, so HtoD and DtoH both need measuring

## 5.2 Implementation

- - Started as a thread that just continuously monitored the counters for set duration of time and printed output into console
- - Added wrapper and file output in subsequent versions to simplify data-gathering
- - Added chunk size options for the buffer copy chunks to get as little overhead as possible while getting most consistent data gathering
- - Delays are measured, no clear correlation between delay and bandwidth
- - Overhead yet to be properly assessed, however, introduces somewhat significant overhead. TODO: ASSESS OVERHEAD
- - Measure transmit and receive in the same thread, sequentially

## 5.3 Results

- graphs
- descriptors of graphs

## 5.4 Discussion

### 5.4.1 successes

- - gives somewhat detailed going-on about PCIe link activities
- - more accurate than nvml counters -> global discussion?

### 5.4.2 shortcomings

- - introduces significant overhead due to occupying PCIe link
- - Delay compensation sometimes leads to negative values due to delay inconsistencies

# 6 Summary

## Outlook

# List of Figures

# List of Tables

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

[6] 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).

[7] 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).

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

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

[10] 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).

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

[12] 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).

[13] 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).

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

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

[16] 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).

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

[18] 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).

[19] 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).

[20] 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).

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

[22] 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).

[23] 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.

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

[25] J. Lawley, "Understanding performance of PCI express systems," p. 16, 2014.

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

[27] 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.

[28] 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).

[29] 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.

[30] 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).

[31] 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.

[32] 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).

[33] 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).

[34] 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).