High-Performance Commodity Data Acquisition Systems for Scientific Applications in the Terascale Era

Zur Erlangung des akademischen Grades eines

Doktors der Ingenieurwissenschaften (Dr.-Ing.)

von der KIT-Fakultät für Elektrotechnik und Informationstechnik
des Karlsruher Instituts für Technologie (KIT)

genehmigte

Dissertation

von

Dipl.-Inf.

Timo Dritschler

aus Ludwigshafen

Tag der mündlichen Prüfung:	06.07.2023
Erster Gutachter:	Prof. Dr. rer. nat. Marc Weber
Zweiter Gutachter:	Prof. Dr.-Ing. Dr. hc Jürgen Becker

Abstract

Advancements in detector technologies and computing systems have enabled observations of physical processes and interactions in unprecedented detail, both in the spatial and temporal domain. This increase in detail leads to a corresponding increase in the data volumes and the rates at which data is being produced by modern scientific experiments. As such, the Data Acquisition (DAQ) systems required to record and distribute the data produced by these experiments have also grown in complexity.
Conventional off-the-shelves (*commodity*) components of Personal Computing systems have mostly been deemed inappropriate for fulfilling the requirements of cutting-edge experiments. This has lead to most experiments putting increased effort into the development of custom DAQ electronics that are tailor-made for their respective needs.

This thesis aims to show that the development of commodity computing components has reached a level at which they can be used to build computing systems adequate for even the highest of DAQ requirements. It puts special focus on the application of standardized and commercially available interconnect technologies, especially with regards to high-performance on-line data processing and fast feedback control loops.

To this end, this thesis will provide an overview of currently available data-transfer and computing technologies. It will identify the technologies that are most useful in the construction of high-performance and low-latency processing system. The respective strengths and weaknesses of chosen technologies is shown through their application in real experimental use-cases. And eventually, these technologies are combined into a prototypical, generalized, scalable high-performance "Software Defined DAQ" and processing system, capable of satisfying the needs of even highly demanding scientific experiments.

Kurzfassung

Fortschritte von Detektortechnologien und Computersystemen ermöglichen die Untersuchung physikalischer Prozesse und Interaktionen in nie dagewesenem Detail, sowohl in der räumlichen, als auch in der zeitlichen Domäne. Diese Steigerung in Details führt ebenso zu einer Steigerung der Datenmenge und der Rate mit der die Messdaten von modernen wissenschaftlichen Experimenten erzeugt werden. Daher sind auch die Datanerfassungssysteme (DAQ), die zur Aufzeichnung und Verarbeitung dieser Experimentdaten dienen, gleichermaßen komplexer geworden.
Dies hat dazu geführt, dass viele Experimente erheblichen Aufwand in die Entwicklung von komplexer Spezialelektronik investieren müssen, um den DAQ Ansprüchen ihrer jeweiligen Messsysteme genügen zu können. Denn konventionelle frei erwerbliche (commodity) Komponenten von standard PC Systemen werden als unzureichend betrachtet, um die Ansprüche fortschrittlicher Experimente erfüllen zu können.
Diese Arbeit zielt darauf ab, aufzuzeigen, dass die Entwicklung von standardisierten PC Komponenten mittlerweile einen Stand erreicht hat, mit dem sich auch mit Standardkomponenten Verarbeitungssysteme erstellen lassen, die selbst den höchsten DAQ-Anforderungen genügen können. Dabei wird besonderer Wert auf die Anwendung moderner Standardkomponenten zur Datenübertragung gelegt, insbesondere im Hinblick auf hochperformante online Datenverarbeitung und Feedback-Loops.

Hierzu gibt diese Arbeit einen Überblick über aktuell verfügbare Technologien zur Datenübertragung und Verarbeitung. Die Technologien, die am besten geeignet zur Erschaffung von hochperformanten Verarbeitungssystemen bei geringsten Latenzen sind, werden aufgezeigt. Und deren jeweiligen Schwächen und Stärken werden anhand von echten Anwendungsbeispielen

dargestellt. Schließlich werden die gezeigten Technologien in ein prototypisches, skalierbares, generelles „Software definiertes DAQ“ Verarbeitungssystem vereint, das selbst den Ansprüchen modernster wissenschaftlicher Experimente genügen kann.

Acknowledgements

This work is dedicated to the remarkable group of people without which this work would have never come to be.

To my grandfather, I say *"Thank you, for igniting the spark of curiosity for the world of electronics in me."*

To Thomas Laun, I say *"Thank you, for introducing me to the world of computer science."*

To my parents and family, I say *"Thank you, for always supporting me and for providing the framework for me to carve out my own path in life."*

To all of my friends, I say *"Thank you, for all of your loyalty even during my darkest times."*

To all of my colleagues and mentors, I say *"Thank you, for teaching me and for all the excitement we had together along the way."*

To all of my supervisors and my professors, I say *"Thank you, for all of the trust you have put in me, even in the face of impending failure."*

To Vivienne, I say *"Thank you, for all of your kindness, and for being my safe haven throughout all of this."*

And finally, to all of those teachers of mine back in school that said I would never reach this goal, I say *"Screw you! I did it anyway!"*

"Fall seven times, stand up eight." —An old Japanese proverb

Contents

1 Introduction

Electronic systems that sample signals from a data source (such as a detector) and make these signals available and accessible to be processed by a computer are called *Data Acquisition Systems* (DAQ). They span the operational range from the moment data is being created, to the moment the data gets handed off to a "permanent" storage, from which they can be accessed, processed and evaluated. In addition, aspects of process control and computing have been woven into the operational range of DAQ, in order to augment and facilitate the operation of increasingly complex instrumentation.

As such, DAQ has taken on an important role in the operation of scientific equipment, and its design and implementation often represents a substantial portion of an experiment's budget, both financially and in development effort. In an ever-evolving scientific and technological landscape, many experiments are faced with having to develop custom electronics, in order to satisfy their DAQ requirements. This comes not only at the cost of slow development cycles, but also has a propensity to "reinvent the wheel", which holds higher risk of reintroducing problems that are commonly associated with the chosen technologies, further increasing the required development efforts, and reducing reliability.

In contrast, standardized equipment and technologies that are commercially available (we will call this *commodity* hardware) offers vastly reduced integration efforts, as they can mostly be used "out of the box" with high performance and reliability, but do so by trading off versatility for specializing on a narrow set of use-cases. In general, commodity hardware is considered to be unsuitable for the requirements of cutting-edge experiments with particularly high DAQ demands, as they were commonly only optimized for data processing, and lacked required connectivity and transfer capabilities.

In this thesis, I will show that these concerns are no longer valid, as the

data transfer characteristics of commodity computing equipment have become powerful enough to be suitable even to the needs of the most demanding experiments. I will demonstrate how scalable high-performance general purpose DAQ can be built from general purpose commodity computing components at minimal development effort, and will propose a software-defined framework to operate such commodity DAQ systems.

1.1 DAQ in the Terascale Era

Advances in detector technologies enable the observation of physical processes and phenomena in unprecedented detail. This increase in detail directly leads to an increase in the amount and the frequency at which data is being produced by such detectors. As an example, we look at the *Compact Muon Solenoid (CMS)* at the *Large Hadron Collider (LHC)* of CERN. It is predicted to produce up to 50Tb (50 Terabits: 50×10^{12} bits) of raw data *per second*, after its "Phase 2" upgrade ([And17]), putting us firmly into what we will call the *Terascale Era* of instrumentation. Other experiments and instrumentations from different scientific domains are certain to reach this milestone in the coming years as well.
These increases in data rates are often a result of an increase not only in the spatial resolution of a detector, but also in its sampling frequency. Experiments that operate at high event rates of MHz or more are a challenge for their respective readout systems, but they also offer potential benefits to the fidelity of their measurement results. When observing dynamically evolving processes at microsecond intervals or even smaller, it can be of benefit to be able to influence process variables of an experiment at pace with its sampling frequency. In some cases, these control decisions are based on an evaluation of data provided by the detector *at runtime*. In this way, an experiment can quickly react to changes in an observed process or phenomenon to increase the quality of measurement results or change the overall system behavior. This is commonly known as a *Feedback Control Loop*. Realizing such feedback loops for experiments with high-frequency sampling electronics requires both, fast data communication channels, as well as computing infrastructure

that is capable of processing measurement data at a sufficiently fast pace. As a result, modern DAQ has evolved to be characterized by low-latency and high-throughput data communication, as well as high-performance computing capabilities.
In the past, commodity computing hardware has mainly focused on computational *bandwidth* rather than latency, as the traditional use-cases for computing have much more relaxed time frames. This has made it challenging to use conventional commodity computing equipment to satisfy the DAQ requirements of highly-demanding experiments. However, in this thesis I argue that deliberately choosing equipment and technologies with inherently low latencies and high computing performance may yet make it possible to construct commodity DAQ systems capable of meeting the demands of experiments in the terascale era. Over the years, key performance metrics of commercially available computing and networking equipment have steadily improved. Arguably, the technological advances of commodity *Personal Computing (PC)* and *High Performance Computing (HPC)* equipment have reached a break-even point, at which it becomes feasible to develop highly performant commodity DAQ, while minimizing the need for custom electronics. This could lead to reduced integration efforts and avoid reintroducing typical problems in the development of custom electronics. Furthermore, such DAQ would profit from the availability of the vast ecosystem of pre-existing software and programming libraries for commodity components, which have already been created over the past. Newer versions of the chosen equipment can potentially serve as "in-place" upgrades with minimal changes to the existing system, once they become available. Overall, this would benefit small and mid-tier experiment in particular, which often do not have the means to finance the development of custom electronics for their DAQ.

In this thesis, I will focus on the development of high-performance DAQ through commodity computing equipment, and aim to answer the question:

> **Guideline Question:** *How can we build DAQ systems from commodity computing components, capable of satisfying the needs of experiments in the Terascale Era?*

My work takes a closer look at the needs and requirements of modern DAQ for a range of different types of experiments. It will identify shortcomings and strengths of such DAQ systems, and aim to alleviate identified problems through the application of commodity computing equipment wherever possible. It will discuss traditionally applied technologies and methods in the field of DAQ systems, as well as recent developments in the personal computing fields, how these recent developments fit into the bigger picture of DAQ and what stands to be gained from the symbiosis of custom electronics with available commodity equipment. It will demonstrate the capabilities of a purely commodity DAQ by implementing a trigger system for one of the most highly demanding experiments to date, both in terms of data rate and latency requirements. It will prove that with careful consideration for components and interconnects, commodity DAQ is a highly attractive prospect, and can definitely keep pace with traditional DAQ technologies.
And finally, it proposes a highly performant, scalable, general-purpose "Software-defined DAQ" setup, built exclusively from commodity computing equipment, that bridges the gap between traditional DAQ and modern *Cloud Computing*.

1.2 Problem statements and research questions

A question as general as the *guideline question* can not trivially be answered directly. Therefore, the research done in this thesis separates the single guideline question into a set of approachable elements, and aims to provide answers to them as follows.

Traditional DAQ and Personal Computing Technologies

Due to the complexity of modern instrumentation, their DAQ systems often have to consider and aggregate data from many different subsystems or

sources in order to operate. This results in complex distributed systems, consisting of many different layers, and a mixture of specialized readout electronics, networking components and computing equipment. Finding suitable and optimal technologies and methods to allow for the effective combination of data transfer and computing is a core challenge of modern DAQ design. Adherence to established standards opens the door to a high level of interoperability, and can often ensure not just *reliability*, but also *flexibility* and *maintainability*. It is in the best interest of any new DAQ design to be aware of available technologies and methods, and to have a solid understanding of their benefits and shortcomings, in order to choose optimal solutions. From this, we can derive the first primary research question:

> **Question 1:** *Which commodity computing technologies can be used to aid the integration of heterogeneous DAQ components?*

Remote Direct Memory Access for Low-Latency Networks

Single computing nodes can not provide sufficient computational performance to control complex instrumentation. A DAQ that needs to take multiple data sources and additional online-processing into account can only meet required performance constraints by means of a distributed system, as they are commonly found in HPC installations. Especially when using commodity computing equipment that is intended for personal computing applications, such systems often turn into multi-layered heterogeneous clusters. The further such a computing network is distributed, the more important it becomes to focus on the design of the network that connects the resources to each other, in order to maintain *scalability*. Additionally, when designing a DAQ that incorporates online-processing with *feedback-loops*, communication latency gains increased importance over just interconnect bandwidth.

Direct Memory Access (DMA) allows devices connected to a common bus inside of a computing system direct bus-level access to attached memories without the need for the CPU to manage the data-transfer. This avoids many of the overheads of data-transfer inside computer systems and frees

up the CPU for more meaningful tasks in the mean time. Remote Direct Memory Access (RDMA) extends this concept onto a network level, in which network devices and protocols allow for the movement of data directly to and from memories attached to their respective system buses. This bypasses the operating system's network stack and significantly reducing transfer latency. There are multiple data-transfer standards available that support (R)DMA, as well as different protocols to facilitate different levels of *reliability*. However, many DAQ systems and networks still heavily rely on Ethernet for their data-transfer needs, which traditionally does not support RDMA. The question arises:

> **Question 2:** *Which RDMA capable network standards are available? How do they differ from each other, and how well are they suited to modern DAQ needs? How can existing Ethernet DAQ infrastructure benefit from RDMA?*

Integration of GPU Computing into Low-Latency RDMA Networks

Traditional computing equipment is commonly optimized for computational throughput, rather than low latencies. Components, such as GPUs, can operate on multiple data at once to reach high computational bandwidth, but data transfer from and to GPUs is notoriously slow, with many overheads and bottlenecks. For DAQ to profit from the high computing performance of GPUs for fast on-line processing, GPUs need to be efficiently integrated into the DAQ network with a low-latency and high-throughput link.
Recently, GPU manufacturers have provided technologies that allow for their GPUs to be the target of DMA operations, which opens up the possibility to perform RDMA operations to and from such GPUs. This provides an ideal opportunity to reduce latencies and turnaround times in distributed computing systems.

> **Question 3:** *How can Remote Direct Memory Access be combined with traditional High-Performance Computing methods, such as GPU computing?*

Application of FPGAs in commodity DAQ Systems

Field Programmable Gate Arrays (FPGAs) are a form of software-programmable, low-power integrated circuit that can be reprogrammed to change configuration, providing full control over their timings and behaviour. Many FPGAs offer a wealth of integrated interconnectivity components, such as fast digital/analogue domain converters, generic high-speed transceivers and a high number of custom programmable General Purpose Input/Output (GPIO) pins. They find broad application in many custom DAQ installations, especially in front-end electronics, where they are used for data-sampling, (pre)processing, distribution and recording. They are also commonly used to implement highly parallel data processing logic and have recently gained increased recognition as general purpose computing accelerators, as well as machine-learning applications. However, most FPGAs can not compete with the computing performance of highly optimized computing equipment, such as GPUs. Integrating them into any system usually requires the development of a highly complex custom circuit board to mount the FPGA onto. Although for many cases there exist pre-fabricated "evaluation boards" that provide access to most of an FPGAs functionality and can be used in a turn-key fashion.
When using FPGAs to their respective strengths and combining them with other components so that they can supplement each other, they are most valuable tools to build modular, flexible and highly performant DAQ systems. To this end, it becomes important to consider suitable strategies of integrating FPGAs into commodity DAQ environments, and we formulate the question:

> **Question 4:** *How can general purpose commodity DAQ systems benefit from the application of FPGAs, and how can FPGAs be integrated into such systems?*

Design and validation of a commodity DAQ processing chain

With the wealth of available components and technologies to realize high-performance commodity DAQ systems, we become interested in a suitable

selection and arrangement that facilitates both, lowest latencies and high-speed computing. To ease integration efforts, such systems should be flexible enough to be compatible with a wide range of DAQ requirements, while remaining scalable to the respective experiment's needs.

> **Question 5:** *What is a suitable commodity DAQ system design that facilitates both, low data communication latencies and high-performance computing? Which standards should be chosen, to aid maintainability and scalability?*

Software for commodity DAQ Systems

After choosing optimal networking and computing equipment, we arrive at a notion of DAQ that consist mostly of commodity computing hardware. It heavily resembles traditional distributed High Performance Computing systems, with an added focus on low-latency data communication. With this, the bulk of the work of the DAQ implementation shifts away from electronic considerations, towards software and firmware design, as almost all commodity computing equipment is software programmable.
There are many programming frameworks and libraries available that may help programmers to implement their DAQ. But due to the novelty of commodity hardware DAQ, no software exists that specifically focuses on such "Software-definded DAQ". A novel concept for a control software specifically for DAQ built from commodity computing and networking equipment needs to be developed. So we ask the question:

> **Question 6:** *How can DAQ benefit from being defined mainly through software, and how could such a* Software-defined DAQ *system look like?*

2 DAQ basics: Data Transfer and Computing

Let us first clarify some basic contexts for this thesis. We understand Data Acquisition Systems (DAQ) as electronic systems that sample data from a source, and make these data available to be processed by a computer. Further, for the scope of this thesis, we assume that the data source is some form of detector, or multiple detectors, that provide their measurement results in the form of analogue electric signals which get sampled and converted into the digital domain. Conventionally, this converted data is then written to some form of storage, from which it can eventually be picked up by any arbitrary computing system that is suitable to the instrumentation's requirements. This computing system processes the recorded data, extracts relevant information and thus provides knowledge to the operators of the instrumentation. Without loss of generality, let us assume that this process is performed in the context of a scientific experiment that uses the aforementioned detector(s) for their investigations.

In most basic cases, the DAQ that samples and converts the data from the detector(s) will eventually store it onto persistent memory, most typically one (or many) computer hard-drive(s). The scheme of waiting for data to be stored onto a hard-drive, and then processing it afterwards, is called *Offline Processing*, or sometimes also *Batch Processing*, in cases where multiple datasets get processed periodically. This paradigm is still feasible for many experiments today. But we can already see the first large-scale scientific experiments emerge, for which the processes of offline- and batch processing would require such prohibitively large computing and storage setups, as well as long data transfer times which make these solutions practically infeasible. For example, with experiments from the field of high-energy physics, such as

the Large Hadron Collider of CERN, even single experiments, like the Compact Muon Solenoid (CMS), can produce multiple Terabytes (10^{12} bytes) of data *per second* ([And17]). To put this number into perspective, the largest computer hard-drive to date, which has only recently (as of 2022) become available[1], provides 20 Terabytes of storage capacity. Meaning, the data from the CMS alone would fill up one of such hard-drives every couple of seconds. With an estimated 7.5 million seconds of operation per year[2], the resulting deluge of data would be—quite frankly—neigh impossible to manage with today's levels of storage technology.

As a direct consequence of those circumstances, the conventional paradigm of offline processing is no longer applicable to such experiments, and forces them into a dilemma. Either the runtime of the experiment needs to be limited to the scale of available storage, severely limiting the experiments efficacy, or they have to find means to reduce the amount of data that needs to be stored, without significant loss of valuable information. Indeed, many methods of achieving data reduction have been developed, ranging from data compression ([Arr01][Pur03]), pre-processing of data with the intent of only storing processing results (sometimes referred to as *Online Knowledge Extraction* [Can15]), as well as so-called *Trigger Systems*, which aim to select measurements that hold relevant information. (Such Trigger Systems will get discussed in more detail in chapter 5 of this thesis.)

Regardless of the solution each experiment might choose, they all have one important thing in common: They require *Computing*. While the traditional notion of DAQ is mostly limited to data recording and transport, it is becoming increasingly more typical for DAQ to also incorporate aspects of computing on its data path from source to storage, in order to facilitate forms of *Online-Processing* in accordance to an experiment's requirements. However, with the aforementioned increase in generated data bandwidth of modern experiments, the computational performance necessary for online processing can

[1] https://www.seagate.com/news/news-archive/seagate-ramps-20tb-hdd-shipments-answering-mass-data-growth-pr-master

[2] http://home.cern/news/news/computing/cern-data-centre-passes-200-petabyte-milestone

not be provided by a single computing node. In most cases, distributed systems with varying types of computing accelerators are required to provide sufficient performance. This introduces yet another challenge into the design of modern DAQ: *Data Transfer* and *Networking*.
Even in the traditional notion of DAQ as a simple data recording mechanism, aspects of data transfer needed to be considered in order to provide interconnects from source to storage that can provide enough bandwidth for the amount of data generated by the source. By today, most detector systems have grown in complexity and consist of a number of different subsystems that operate largely independently of each other. This means that measurement information of such detectors is split into multiple separate data streams, that need to be re-combined into logical *events* in order to get the "full picture". Again, data transfer plays an important role in the design of such *data concentrators*. When aiming to process the collected data in a distributed computing system, interconnect *bandwidth* becomes an important aspect of the *scalability* of those distributed systems. Finally, as detectors not only grow in structural complexity, they also enable measurements at ever higher temporal resolution. Meaning, that any experiment that aims to dynamically control measurement parameters of their detector at pace with its measurement rate, will have to put additional emphasis not only on the bandwidth of its interconnects, but also on their *communication latency*.

In conclusion, the scope of modern DAQ has increased from its traditional notion of simple data-recording system, to sophisticated online-processing and distribution systems which are characterized by both, efficient networking technologies, as well as online high-performance computing capabilities. This "modernized" view of the capabilities and responsibilities of DAQ is visualized in figure 2.1 and will serve as the general "blueprint" for the systems I aim to present during the course of this work.

With this modern notion of DAQ in mind, as well as its two core operational responsibilities (Data Transfer and Computing), we can now set out to begin and choose suitable technologies to realize such a system. According to the mission statement of this work (see Chapter 1) I aim to do this through the use of as much commercially available commodity hardware as possible.

In the following sections of this chapter, I will introduce general concepts and characteristics of the most common forms of data communication and interconnects, as well as basics of data processing and distributed computing. At first, I will introduce networking related technologies and present the interconnects and networking technologies that I have chosen for my work. Then, I will give an introduction to contemporary methods of High Performance Computing (HPC), such as *General Purpose GPU Computing (GPGPU)* and *Field Programmable Gate Arrays (FPGAs)*.

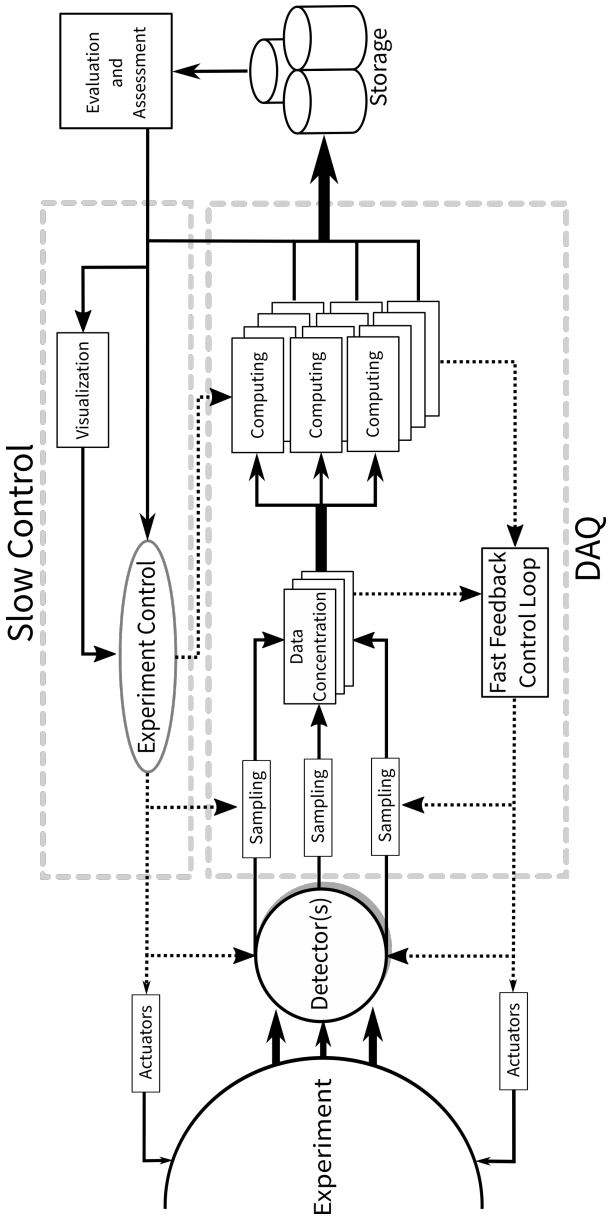

Figure 2.1: The generalized schema of a DAQ chain from data source to storage. Including sampling stages and data concentration, Online-Computing and Fast Feedback Control Loop.

2.1 Data Communication Fundamentals

Electromagnetic signals are a commonly used form of data transmission in contemporary information processing systems. These include electric currents, radio waves and bundles of photons through a range of different media.
Information is encoded on such signals by means of frequency and amplitude. While both of these information domains are continuous by nature, most computing systems have been developed to interpret these signals at discrete amplitude levels and at fixed frequencies. Specifically, most modern computing systems focus solely on two distinct amplitude levels to separate a 'high' and a 'low' state of a signal at well defined transition frequencies, and are therefore called *digital computers*. Computers that interpret the full continuous spectrum of a signal also exist and are respectively called *analogue computers*. They have advantages over their digital counterparts in regards to processing bandwidth and functional range. However, digital systems are superior in regards to flexibility and computing precision ([Rub53]) and have thus become the predominant computing architecture. Therefore, I will focus on the transfer and the processing of digital information, throughout the rest of this thesis.

For computers, the most common form of signal is an electric current through a conductor. However, radio transmission and optical communication by sending light impulses through a fiber, are commonly used as well. It is important to keep in mind that in actual practical applications even digital information is still being transmitted by means of a continuous analogue signal. Suitable mechanisms of encapsulating and extracting the digital information stream onto and from the analogue transmission signal need to be chosen for an interconnect to work efficiently and reliably. The development of reliable interconnects at high transfer rates has spawned whole scientific domains and industries focused solely on this particular aspect of data communication. However, as this thesis focuses on the construction of DAQ with commercially available commodity components, the physical functionality of such

interconnects is mostly out of scope for those considerations, as we are expressly aiming to avoid developing custom data transmission electronics. This is why I will take on the assumption, that we won't have to involve ourselves with the basic functionalities of the interconnects discussed in this thesis, and that we can use and rely upon their functionality basically as a *black-box*.
To be able to make qualitative assessments and comparisons of those black-boxes, we will use three different forms of metrics.

- **Bandwidth:** The bandwidth of an interconnect is a measure of the maximum *potential* amount of data that can be transferred between two endpoints over a fixed unit of time. It is typically denoted in the form of *Bits per Second*, which I will abbreviate as *bps* throughout this thesis. This is not to be confused with an analog signal bandwidth, which is a measure of the difference between lowest and highest attainable frequency of a continuous frequency band, and would be denoted in *Hertz (Hz)* instead.
- **Throughput:** is a measure of the amount of useful data that is *actually* being transferred within a certain window of time. It is typically averaged and normalized to the amounts of *Bits per Second (bps)*, just as the bandwidth is. In practical application, the throughput of an interconnect will always be lower than its maximum bandwidth, due to a variety of factors, such as protocol overheads and retransmissions due to bit-errors.
- **Latency:** The latency of a data transmission describes the systemic amount of time that will pass between sending data and the moment the receiver has fully obtained the data and is ready to process it. It is affected by signal propagation time, path length and various transformation delays when information gets encoded and decoded to and from the carrier signal and the type of transfer protocol that is being used. As it is an amount of time, it is denoted in *seconds*.

> **Note:** Bandwidth, Throughput and Latency are by far not the only metrics that play a role in the design of a complete system. For example, power consumption might be relevant for particularly small-scale systems, as well as heat build-up and the required thermal dissipation equipment. But those aspects have no direct impact on the primary functionality of an interconnect.

Aside of those intrinsic properties of interconnects, it is also of importance to understand the benefits and drawbacks of the types of transmission media an interconnect can utilize. There are a number of properties that need to be understood in order to make an informed choice.

2.1.1 Transmission Media

Whenever information needs to be transmitted, there always needs to be a medium through which the signal that carries the information will travel. The most common forms of transmission media for digital information are in the form of an electrical current through metal wires, packets of photons through an optical fiber and radio waves across the air. These transmission media differ in their properties, which need to be considered when choosing a medium. A summary of those differences and their impact on the design of a data communication system is shown in listing 2.1.1 and will be described in further detail in the following sections.

Radio Transmission

While radio transmission (commonly just classified as “wireless” communication) might appear convenient, as it does not require a wired connection between the communication participants, it is also the least reliable of the three mentioned transmission media, and also the one with the worst average bandwidth and throughput.
While there exist a wealth of different standards for wireless communication that are being widely used for data communication, such as *Bluetooth* [Bis01],

Wireless LAN [Ros04], *Fifth generation mobile communication (5G)* [Jai18], etc., they all exhibit similar limitations in their practical application. Wireless communication is notoriously unreliable as it is prone to interference and suffers from a significant drop in signal-strength/signal-to-noise ratio when transmitting through obstacles like the walls and floors of a building. It is common in many experiment setups that a detector and its respective computing infrastructure are set up in different locations. In the best case, these would be located in the same building. But it is not uncommon for data centers to be located in a different building on the same campus all together. This already makes it exceedingly difficult to transmit a radio signal from data source to the computer infrastructure. In radiation-hardened environments, like they can be found in many experiment setups, it might even be impossible for a radio signal to penetrate the shielded area in the detector cavity to begin with. Let alone any negative impacts the radiation produced by the experiment might have on the radio signal, or even worse, which impact the radio signal might have on the measurements.
Regarding expected bandwidth, typical data rates range from a few megabits per second for Bluetooth, to roughly 10 gigabits per second in 5G (under ideal conditions, including the use of carrier aggregation and non-congested licensed transmission frequencies.) The newest Wireless LAN standards (801.11ax), also called *Wi-Fi 6*, reports to reach a bandwidth of 2200 mbit/s under optimal conditions [Bel16]. These bandwidths are an order of magnitude less than what most common cable-based interconnects are able to achieve and wireless will rarely ever reach their theoretical maximum speeds in practical application.

For those reasons, wireless data transfer only occupies a very small niche in the domain of DAQ and it is typically only used in cases where a wired connection is unfeasible. For the rest of this thesis, I will assume that we will be able to used wired alternatives instead.

Metal Wire

Metal wires or traces on a circuit board are the most common form of transmission medium found today. As virtually all computing circuits nowadays work by means of electric current, metal wires are ideal choices to transport those currents over short or medium distances. Metal wires are robust against mechanical damage and bending. The source electric signals can simply be transmitted through a wire without needing to be adjusted first, at least in cases where current is being transported over short distances only (tens of centimeters). However, when wanting to transport current over medium to long distance (generally beginning at around 1/10 of the signal wavelength), some unwanted side-effects will begin to manifest. At high transmission frequencies, wires begin to act like antennas, emitting radio signals, and introducing cross-talk into neighbouring lines. With increasing cable length, electric impedance, signal reflection and Inter-Symbol-Interference (ISI) will need to be taken into account. These effects limit the maximum possible cable length, as well as the maximum transmission frequency. In addition, other sources of electromagnetic radiation can introduce noise into the transmission lines of the cable, interfering with the transmission. Some of those effects can be mitigated by shielding the cable, and by using *Differential Signalling* instead of single transmission lines, at the expense of increased manufacturing cost.

When using *Differential Signalling*, an electric signal is not sent across a single transmission line, with its amplitude being derived from a comparison against a pre-defined base level. Rather, signals are sent across a pair of lines simultaneously, where both lines transport the signal with an amplitude equal and opposite to each other. The signal is then derived from the difference in amplitude across both lines. This adds reliability to the transmission process, as any external interference usually affects both lines in the same way and therefore has low impact on a differential signal. A visualization of this effect can be seen in figure 2.2. In addition, a differentially transmitted signal tends to reduce the radio emissions of the interconnect, as the electromagnetic fields created by the equal but opposite charges of the transmitted signals cancel

each other out, which significantly reduces cross-talk, and allows for higher transmission frequencies over longer distances.

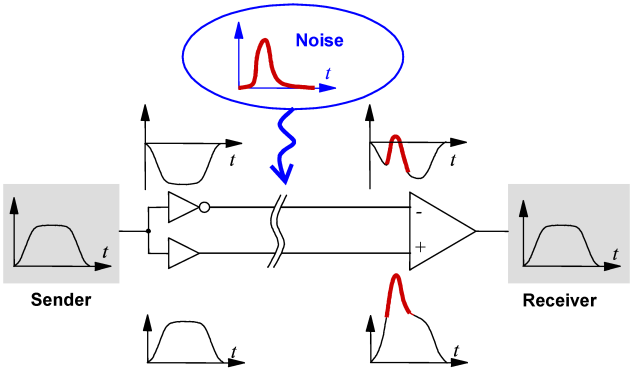

Figure 2.2: An example of noise reduction when using differential signalling [b]

[b] By 'Linear77', Wikimedia Commons, licensed under CC BY 3.0 (https://creativecommons.org/licenses/by/3.0/). The work is being used unaltered.

Employing these strategies, it becomes possible to use metal wires of varying lengths for different types of speed-grades. As an example, we look at the different speed-grades of the commonly used *Ethernet* interconnect. For 1 Gigabit per second, metal cables are able to reach lengths of up to 100 meters. This maximum length will gradually reduce to only 15 meters for up to 40 Gigabits per second. 7 meters when used for 100 Gigabits per second. 3 meters for 200 Gigabits per second. For speed-grades of 400 Gigabits per second and higher, metal cables are no longer foreseen as compatible transmission medium. It is to be noted, though, that Ethernet generally combines multiple pairs of wires to reach the desired throughputs. The mentioned throughputs can only serve as a relative comparison, and do not accurately present the maximum amount of data that can be transferred via a single metal wire or differential pair. For all intents and purposes of this thesis, however, the knowledge that metal cables exist for Ethernet of up to 200 Gigabit per second is sufficient for all further discussions.

Despite the mentioned drawbacks, the simplicity and reliability of metal wires and cables, when being used over short to medium distances only, still make them the most widely used form of transmission medium to date.

Optical Fiber

In fiber-optic-communication, long strands of translucent fibers, made from silica or plastic, are used to transport light from one end of the fiber to the other. The inherent properties of using light as a transmission signal bring a number of benefits. As photons move with close-to-light-speed through the fiber, signal transmission has only very little latency. Simultaneously, fiber-optic data transmission is immune to electromagnetic interference, but they are comparatively fragile and can easily get damaged.
There are two types of optical fibers. They differ by how many *Modes of Light* can travel through the fiber simultaneously and are thus aptly called *Single-Mode* and *Multi-Mode* fibers. The discussion about modes of light is outside the scope of this thesis. The practical implications of single-mode and multi-mode fibers essentially come down to the following understanding:

- Single-Mode fiber installations are generally more expensive, but have higher bandwidth and can reliably transport signals up to 40 kilometers
- Multi-Mode fiber installations are generally cheaper, but are limited in bandwidth and range, and are recommended for distances of less than 400 meters

As of today, it can be approximated as a rule of thumb that optical-fiber installations are roughly double the price of comparable metal-wire installations. This increase in cost comes not only down to the production cost of the fibers themself, but also to the higher complexity of the required networking equipment and connectors. Electric signals first need to be converted into a light pulse, before they can be transmitted through an optical-fiber. They then need to be converted back to an electrical signal at the receiving end. The transmitters and receivers required for this conversion are a driving factor behind

the higher prices and are the main reason that prohibits the proliferation of fiber-optics for data communication over conventional metal wire cables.

Listing 2.1.1: Summary of the qualities of transmission media

- **Wireless Radio Transmission**
  - Supports up to 10 Gbps peak performance, but averages out at below 1 Gbps in practical applications
  - Has a typical range of below 100 meters, but can be extended to multiple kilometers by using directional antennas
  - No mechanical connection/cable required to connect machines, though requires appropriate radio equipment
  - Is notoriously unreliable and has trouble penetrating obstacles, making it difficult to use between installations and in radiation-hardened environments
- **Metal Wire Cables**
  - Up to 200 Gbps in practical applications
  - Supports up to 100 meters of cable length
  - Is comparatively cheap, robust and requires low mechanical complexity
  - Maximum cable length strictly limited by transmission frequency, topping out at a maximum of 3 meters at 200 Gbps applications, and is sensitive to electromagnetic-interference
- **Optical-Fiber**
  - Up to 400 Gbps in commodity applications, but can theoretically be used up to multiple Terabits per second.
  - Up to 40 kilometers for Single-Mode fibers and up to 400 meters for Multi-Mode fibers

  - High signal propagation speed, transmission bandwidth and maximum cable length. Immune to electromagnetic-interference
  - More expensive than regular metal-wire cables and mechanically fragile

From the mentioned properties of transmission media, it can be concluded that fiber-optics-communication is the ideal choice when wanting to transmit signals of very high bandwidth, or whenever distances above 100 meters need to be bridged. While for applications below 100 meters, metal wire cables offer a better trade-off between performance and cost.

2.1.2 Interconnects relevant to this work

There exists a whole ecosystem of different interconnect standards. In this section, I will introduce the interconnects that are most relevant to the objective of this work. But before I will do this, I deem it important to have a quick clarification of the term "interconnect" which is used regularly throughout this thesis, as the term itself is ambiguous. It is used in different capacities whenever talking about aspects of data communication, like the definition of physical interfaces and connectors, as well as whole classes of information exchange systems. To make any further discussions more concise, I will define an *interconnect* as follows:

> An **interconnect** is a data communication standard that consists of a set of *protocols* which describe:
>
> - At least one type of physical interface to a shared transmission medium to which the participants of a communication are connected
> - An encoding of information onto a *carrier signal* which gets transmitted through the shared medium
> - Structures and sequences of information via the carrier signal that make it possible to transmit a generic data *payload* between participants of a communication

In a similar way, the term "protocol" is equally ambiguous. As mentioned in the interconnect definition, there might be protocols for carrier types, modulation and frequencies, permissible signal amplitudes, bit arrangement and interpretation, handshakes, send/receive order, addressing schemes, and many more. Generally, a *protocol* is simply a set of communication rules that both ends of a communication are following in order to exchange information. To better clarify what type of protocol is being referred to during the discussions in this work (wherever necessary) I will also specify the so-called "OSI Layer" of the protocol in question. The *Open Systems Interconnection Model (OSI Model)* is a conceptual model to describe the different "layers" of data communication in order to better describe which aspect of communication any given protocol pertains to. The 7 layers of the OSI model and a short explanation of each layer is presented in table 2.1.

Table 2.1: The 7 layers of the OSI Model

Layer	Name	Explanation
7	Application	Interface definition for the communication of applications
6	Presentation	Interpretation of bit sequences in the payloads
5	Session	Grouping of multiple exchanges into one logical "dialogue"
4	Transport	Sequences of data frames forming one coherent exchange
3	Network	Methodology to chose data path/receiver on a network
2	Data Link	Arrangement of bits to form a consistent "data frame"
1	Physical	Encoding of information (bits) onto the carrier signal

As the goal of this thesis is to discuss how to build data-acquisition system with commodity hardware, I will limit the selection of presented interconnects to those who's primary purpose is to transport generic user-data and are not limited to domain-specific applications. For example, interconnects for graphics devices, such as the *Video Graphics Array (VGA)* [Tho88], *High-Definition Multi-Media Interface (HDMI)* [Eid] and related interconnects in the same domain do find widespread use. They are not intended to transport user-data, and instead are optimized for the connection of displays to a computing device. Many such domain-specific interconnects exist for a range of different

purposes, such as Video, Audio, input devices and more. Those domains and their respective interconnects are not relevant to this thesis.
Similarly, there exists a wealth of point-to-point and bus-based interconnection standards for generic data-transfer, such as *RS-232* [Buc04], *CAN Bus* [HPL02], I^2C [Man14], and similar, which see widespread adaptation in many types of applications worldwide, but are still not relevant for the work presented in this thesis due to their bandwidth being orders of magnitude too small (generally, 5 Mbps or slower) to be feasible for the data rates of modern detectors.
For clarification, I will limit my selection of interconnects to those which can fulfill the following requirements:

- General-Purpose
- Bandwidth of 32 Gbps or greater
- Nominal latency of 1 ms or lower
- Wire-bound (as in: not "wireless")
- Widely used and readily available commodity components for PCs
- No requirement for additional electronics development
- OSI Layer 3 routable (More on this in chapter 6)
- Available Remote Direct Memory Access functionality or extension (See following note)

> **Note:** Direct Memory Access (DMA), as well as its network extension Remote Direct Memory Acces (RDMA), will be introduced in full detail in chapter 3. For this introduction and the overview of chosen standard interconnects and protocols, it is sufficient to understand that (R)DMA is a technology that reduces the latency of a data transfer by eliminating superfluous copy operations on the data path. The resulting decrease in transfer latency thus makes RDMA capable interconnects particularly useful for the high-performance DAQ systems developed in this thesis.

When applying these criteria to the selection of interconnects, the list of candidates out of all available interconnects quickly contracts to only three options. **Ethernet**, **InfiniBand** and **OmniPath**. I will introduce and evaluate these options in the following sections.
However, before I do so, I will start with an exception to this selection. Namely, the *Peripheral Component Interconnect Express (PCIe)* standard used for inter-system communication inside PCs. This communication standard will become important during later discussions, especially for the concepts of *Direct Memory Access (DMA)*.

PCI Express

As of the time of writing this thesis, the *Peripheral Component Interconnect Express (PCIe)* [Bud04] is the driving force behind the inter-system communication inside of PCs. It will provide the necessary communication medium for all of the *Direct Memory Access* applications that will be introduced throughout this thesis. As such, it is important to have at least a conceptional understanding of how this interconnects operates. Even though PCIe can not be used for networking applications.

PCIe is used to connect the internal components of modern PCs, such as memory controllers, storage devices, data drives, graphics adapters, some networking adapters and other components together on a common bus. Its connector specification also provides up to 75 Watts of power to the connected device. At the time of writing, virtually all commodity computer systems use PCIe to connect their internal components together. It is a *serial* data-bus that is based on a point-to-point topology that provides communication lines for devices to connect them to a common *root complex*. It is possible to connect multiple PCIe root-complexes together through a specialized *PCIe switch* in order to expand the amounts of available communication lines. However, this comes at the cost of increased communication overhead when communication between devices has to traverse the switch. All devices that are connected to a root complex share a common bus address space and can freely establish

communication links between one another. Links between devices are organized by combining between 1 and 16 individual communication *lanes* together, with each lane consisting of two pairs of differential signalling lines (4 individual lines in total). One of these pairs is used for sending messages, and the other pair is used to receive messages, ammounting to *full-duplex*. How many lanes are available for each device is dependent on the connector the device is plugged into, and devices will negotiate the amount of lanes they want to utilize with their respective communication partner. The maximum amount of communication lanes a device is able to simultaneously utilized is often denoted in the form of an "x" prefix, followed by that number. e.g.: x1, x4, x16, etc. and there exist individual connectors for each of those lane-counts (Figure 2.3). Connectors differ in size, depending on the amount of PCIe lanes they provide. And they are designed in such a way, that it is possible to plug a device that requires fewer lanes into a connector that provides more lanes than the device needs. The inverse is not possible, however, as devices requiring more lanes than a connector can provide will not physically fit into the connector.

The PCIe communication standard is separated into a *physical layer, data link layer* and a *transaction layer* (Layers 1, 2 and 3 of the OSI Model). These layers describe the electrical properties of the bus connection, the semantics of each individual transmission, as well as the organization of signal transmission on the bus, respectively. PCIe communication is encapsulated into well-defined communication *packages* that are exchanged between a communication "requester" and a communication "responder". It mandates a reliable communication semantic in which a device must store each transmitted data package in a buffer for the purpose of potential retransmission, until the receiver acknowledges their arrival.

PCIe hast evolved through multiple iterations of the standard from version 1.0 in the year 2003, until its current version 5.0 in 2019. Version 6.0 of the standard is expected for 2022. Each version has successively increased the maximum transfer rate and these rates can be seen in table 2.2.

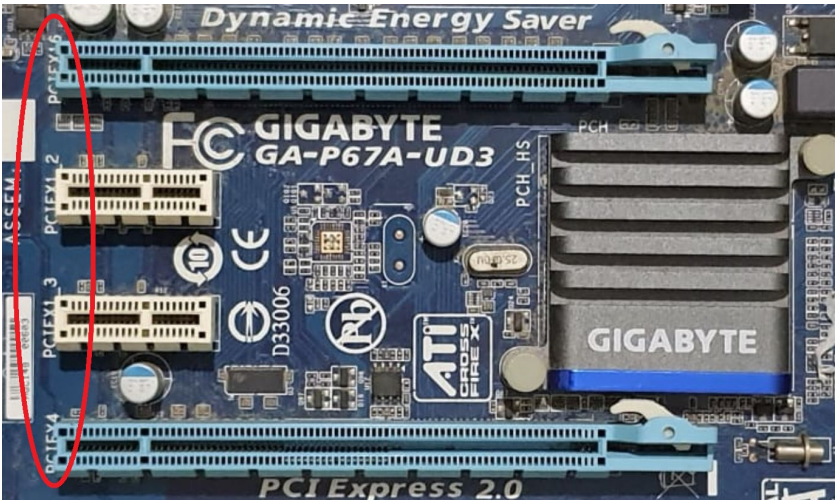

Figure 2.3: Four PCIe connectors of varying sizes on a computer motherboard. An x16 form factor connector with full 16 lanes exposed at the top. Two times x1 in the middle and one x16 form factor connector at the bottom that only exposes 4 lanes. Their respective classifications are printed onto the motherboard for easier identification (red ellipse). [a]

[a] By 'Sayeen', Wikimedia Commons, licensed under CC BY-SA 4.0 (https://creativecommons.org/licenses/by-sa/4.0/). The work is being used unaltered.

Table 2.2: PCIe transfer rates and throughput per version

Version -	Introduced in Year	Transfer Rate Gigatransfers per second	Throughput Gigabytes per second
1.0	2003	2.5 GT/s	0.250 GB/s
2.0	2007	5 GT/s	0.500 GB/s
3.0	2010	8 GT/s	0.985 GB/s
4.0	2017	16 GT/s	1.969 GB/s
5.0	2019	32 GT/s	3.938 GB/s
6.0	2022	64 GT/s	8.000 GB/s

Ethernet

Ethernet[Spu00] is a family of interconnects that offers high bandwidth and error resilient full-duplex communication between any number of computers connected on a network. Its standards and protocols define functionality up to, and including, OSI-Layer 3. It is one of the most ubiquitous interconnects used today, seeing wide adaptation in almost all domains, from industry to consumers and the sciences. It was first developed in 1980 and conceptually uses a single shared differential transmission line to which all participants of a communication are connected. Each participant is assigned a globally unique address (the so-called Media Access Control (MAC) address) and each transmission over an Ethernet network contains both a source MAC address and a destination MAC address. These addresses consist of 48 bits of information, split into 6 individual bytes. They get represented by two hexadecimal digits per byte, separated by a colon character. For Example: 56:FA:A9:08:BD:2F
Conceptually, all participants share a common communication medium, and any transmission can be seen and received by every station connected to that medium. Ethernet networking adapters are expected to ignore all transmissions that are not directly addressed at them. During the early phases of Ethernet, it used a single coaxial cable to which each station on the network was attached to. This causes congestion on the transmission line when multiple stations on the network are trying to transmit at the same time. By today, the most common physical connector standard is the RJ45 plug, seen in picture 2.4, which features 4 differential pairs of transmission lines and allows two ends of an Ethernet connection to communicate in full-duplex. Furthermore, interconnections between different stations are nowadays commonly established through networking hardware called *Switches*, rather than connecting endpoints with each other directly. These switches feature a number of individual Ethernet connectors, commonly called ports. Instead of simply re-broadcasting incoming transmissions from one port onto all other ports, switches implement logic that allows them to forward incoming data packages only to applicable ports, reducing traffic and therefore congestion on unrelated lines. Furthermore, due to their functionality as communication intermediary, they make it possible to translate between differing speed-grades

on separate ports. With the help of switches, it is possible to isolate and connect individual networks together into complex meshes and WANs (Wide Area Networks) to form “networks of networks”.
Ethernet speed grades of up to 400 Gbps are commercially available and speed grades of greater than 1 Tbps are already in development. Single-Transmission latencies of below 1 ms are common, albeit heavily dependent on the protocol embedded into the Ethernet payload, as well as the complexity of the routing through the network.
For speed grades of up to 1 Gbps, the previously mentioned RJ45 connector is the most common standard. Beyond that, starting at 10 Gpbs, Ethernet most typically uses an Small Form-Factor Pluggable (SFP) type connector, shown in image 2.4.

Figure 2.4: Two typical plug standards for network interconnects. RJ45 (left) is most commonly used for Ethernet up to 1 Gbps and SFP (right) is used by 10+ Gbps Ethernet (and also by InfiniBand).

Natively, Ethernet is not capable of RDMA data transfer, as the specification for the networking adapter and protocols of Ethernet do not foresee the necessary DMA engine. However, as we will see in the discussion of communication protocols in section 2.1.3, there exists a protocol extension called RDMA over Converged Ethernet (RoCE) which brings RDMA capabilities to conventional Ethernet connections (but still requires the use of specific RDMA capable network adapters.)
The ubiquity of Ethernet, its high speed grades, decent communication latencies and its protocol extension for RDMA make Ethernet a suitable candidate for the work presented in this thesis.

InfiniBand

InfiniBand[Pfi01] is a specification of the OSI Layer protocol levels 1 through 4, for a wire-bound high-performance interconnect that features high bandwidths at very low latencies. InfiniBand networking adapters of speed-grades up to 400 Gbps are commercially available, with standards for up to 4.6 Tbps currently under development. It features single-transmission latencies of as low as 0.5 μs and also natively supports *Remote Direct Memory Access* capabilities.
It is primarily used in the high-performance-computing domain, with its initial specification released in the year 2000 by the *InfiniBand Trade Association (IBTA)* and was intended to serve as a replacement for the *PCI* interface in PCs at the time. The IBTA had originally even envisioned to base entire computer architectures on top of their InfiniBand fabric, though these plans were never fully realized. By 2014, InfiniBand was the most widely used interconnect for supercomputer installations, according to the *top500.org*[1] list of supercomputers, but has since been overtaken by the Ethernet family of interconnects again. The biggest supplier of InfiniBand hardware nowadays is *Mellanox*, which became a subsidiary of *Nvidia* in 2019.
Unlike Ethernet, InfiniBand does not operate on a shared transmission medium, but rather isolated point-to-point connections over a number of

[1] https:\\www.top500.org

differential transmission lines. It is capable of bundling up to 12 of those pairs together for a single point-to-point connection. The topology of the communication protocols is intended to be *switched*, allowing to relay packets through additional networking substations and thus forming full network meshes. However, this requires to establish local subnet routing tables in order to traverse a network in which sender and receiver do not share a direct point-to-point connection. This requires individual stations on the data-path to properly receive and forward messages that are not intended for them. In an InfiniBand network, this requires the application of specific *subnet managers*, which communicate and exchange information about the individual targets that can be reached from within each subnet and accross those subnet boundaries. High-grade InfiniBand switches often provide a subnet manager as an embedded service. In addition, it is also common to have one of the computers on the network to function as subnet manager through a software service.
In order to separate the concerns of physical transport, connection management, routing and higher level interactions, the InfiniBand standard is sperated into an architecture of multiple communication levels which roughly span from OSI Layer 1 to 4, and can be seen in figure 2.5.

Even though InfiniBand differs in its interconnection strategy from Ethernet, it has still opted to adapt the same type of MAC addressing scheme as is seen with Ethernet networking adapters. This is an intended design decision to facilitate an interoperability with other Ethernet fabrics, so that InfiniBand network adapters can be configured to operate in an "Ethernet Mode". In this mode, a network adapter will send/receive and process Ethernet data frames through the same physical interface, instead of InfiniBand data frames. As 10+ Gbps Ethernet and InfiniBand both opt to use connectors of the SFP family, this makes it particularly convenient to run Ethernet and InfiniBand installations side by side or even interoperate the two standards. Though the required networking infrastructure (e.g.: Switches) can not easily be interchanged and need to be specific to the interconnect standard in use.
Its high speed-grades, very low communication latencies and interoperability

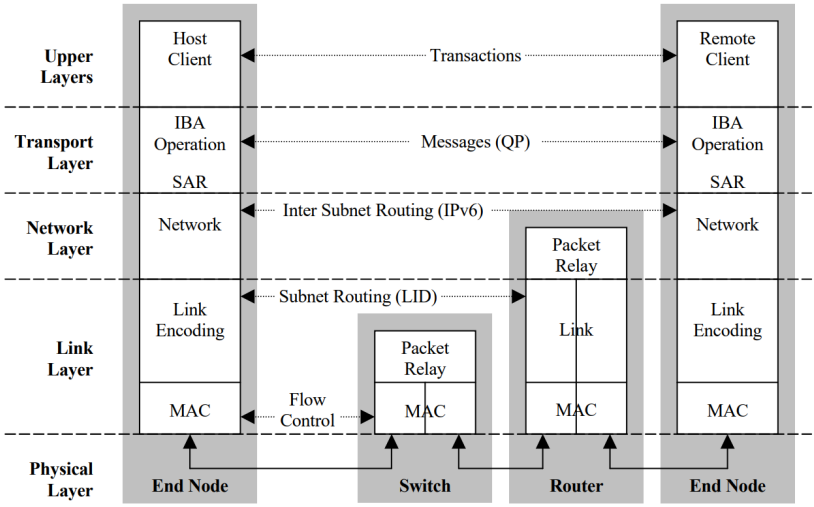

Figure 2.5: The InfiniBand communication standard is split into multiple logical levels which operate mostly independent from each other, and can be considered to span the OSI Layers 1 through 4. [a]

[a] Source: Mellanox Technologies / Nvidia Corporation, "Introduction to InfiniBand" (https://network.nvidia.com/pdf/whitepapers/IB_Intro_WP_190.pdf).

with its strongest competitor (Ethernet) make InfiniBand an excellent choice for the DAQ systems developed in this Thesis.

Omni-Path

Omni-Path is a proprietary high-performance interconnect technology developed by *Intel*, that aims to provide high throughput and low latency data communication for supercomputer installations. First products supporting Omni-Path became available in 2015, which were capable of 100 Gbps communication. By that time, InfiniBand had already developed a significant foothold in supercomputer installations, which positioned Omni-Path and InfiniBand as competitors. Despite Intel's efforts to make Omni-Path hardware a "drop-

in" replacement for InfiniBand by providing driver-level support for InfiniBand commands (verbs) on top of their Omni-Path fabric, they eventually announced in 2019 that they would not continue development of Omni-Path technology beyond 100 Gbps, but would merely continue supporting the older 100 Gbps technology. According to the *top500.org* list of supercomputers, Omni-Path today holds a share of around 8% in the top 500 supercomputers as of 2021. Outside this domain, Omni-Path has not seen any significant adaptation.
Due to its niche specialisation on supercomputers, the proprietary closed-source nature of the interconnect specification and unclear future of the interconnect beyond 100 Gbps data transfer, I do not currently consider Omni-Path a suitable choice for general-purpose DAQ design.

2.1.3 Networking Protocols

Now that the physical layers and bit-level data communication protocols have been established, we can move on to discuss the communication protocols relevant to this thesis. As with the previous section, an OSI Layer or a suitable approximation for each protocol will be provided. Due to their close coupling with the physical-layer interconnects presented in the previous section, the selection of suitable protocols reduces down to two main families of standards. Those protocols are the ones belonging to the **Internet Protocol Suite (IPS)**, like the *Internet Protocol (IP)*[Lei85], *Transmission Control Protocol (TCP)*[For02a] and *User Datagram Protocol (UDP)*[Pos80], as these are almost universally used across almost all Ethernet installations. As well as the protocols related to **InfiniBand**, with additional focus on *iWARP*[Dal05] and *RDMA over Converged Ethernet (RoCE)*[Guo16], which aim to provide RDMA functionality over Ethernet.

> **Note:** Again, Remote Direct Memory Access (RDMA) will get discussed in full detail in chapter 3 of this thesis. For now, it is sufficient to understand that RDMA is a technology that avoids superfluous data copy operations on the data path, thus reducing the effective transmission latency and therefore improving performance.

Internet Protocol Suite

The *Internet Protocol Suite (IPS)*[Lei85] is a set of communication protocols closely related to the Ethernet family of interconnects. It provides functionality on how to order, transmit and route information across a "network of networks". Its most notable members are the *Internet Protocol (IP)* which regulates how an information package can be routed across multiple interconnected networks, providing the backbone functionality of its direct namesake, the Internet. Closely related to the IP is the *Transmission Control Protocol (TCP)*[For02a] which provides functionality to establish a persistent connection between two endpoints of a communication and safely exchange information, ensuring correct ordering and retransmissions in case of errors. Because of the close interconnections of the TCP and IP protocols for use in the Internet, the Internet Protocol Suit is often also labeled as *TCP/IP*. The remaining prominent member of the IP is the *User Datagram Protocol (UDP)*, which provides "connectionless" uni-directional datagram communication.
In addition to these three core protocols of the IPS, there are a plethora of other protocols which mostly serve specific use cases, like address resolution, compression, encryption, resource management and more. These protocols might find use in the larger context of computer network installation, but are not meant for general user data transfer and are therefore disregarded for further discussions in this thesis. The only exception to this will be the *QUIC*[Lan17] protocol, which will get showcased in the context of the discussion about the TCP, as QUIC is intended to supersede TCP in the long term.

Internet Protocol

The Internet Protocol is an OSI Layer 3 (Networking Layer) protocol that manages the routing of data through a network of networks (internetworking). To do so, it wraps the payload of a transmission into an additional structure which contains a source- and a destination address, among other header information, such as protocol version, packet length, Time To Live (TTL) and Fragmentation Information. The exact details of these header informations beside the address data field, is outside the scope of this introduction. For further reference, the interested reader may refer to [For02b].
The primary addressing scheme of the IP protocol is Version 4, and is aptly called IPv4. It consists of four groups of one byte each, and it is typically represented by the decimal unsigned integer representation of each of the four bytes, separated by a single period. For Example: 192.168.0.1
There are certain addressing ranges that are reserved for specific applications. As seen in the given example, an address of the type 192.168.X.X, by convention, is used for any non-publicly accessible (private) network. While addresses that hold the value 0 for their fourth octet (X.X.X.0) are usually considered to be broadcasts for the respective network layer.
IP addresses need to be unique on a logical network level, but do not need to be unique globally. But when setting up a service that needs to be accessible from any network anywhere on the planet, it needs to be assigned a globally unique IP. These static assignments have caused the globally unique address space of IPv4 to come close to exhaustion. Which is why the IP protocol has been extended by a Version 6 addressing scheme, and has been declared the new standard for IP based addressing on the internet in 2017.
IPv6 works similar to its predecessor, but uses a different addressing scheme that consists of eight groups of two octets each ($8 \times 2 \times 8$ bits $= 2^{128}$ possible combinations). They are represented as four hexadecimal digits each, separated by a colon character. For Example: 2a00:1450:4016:0808:0000:2003:0000
To save space and make IPv6 addresses easier to read, segments consisting of exclusively zeroes, as well as leading zeroes, are usually omitted. For the given example, that would be: 2a00:1450:4016:808::2003:
IP based networking is the most abundant form of addressing in any computer

networks today, and almost all networking applications and equipment provide support for IP-based networking in some way, shape or form. This makes IP networking the primary choice for the works presented in this thesis.

UDP

The User Datagram Protocol (UDP) is an OSI Layer 4 protocol and a core member of the Internet Protocol Suite. It does not require any previous handshaking or connection negotiation between sender and receiver prior to a transmission and is therefore considered a "connectionless" protocol. UDP is optimized for very little protocol overhead and due to its lack of connection negotiation is well suited for any application that requires low communication latency. However, due to its simplicity, UDP does not provide any form of re-transmission mechanism in case of errors and does not provide protection against packet loss, duplication or incorrect ordering. All of these aspects need to be handled on a higher protocol level. UDP merely provides a single checksum for each package that allows to detect any bit errors during the transmission, as well as a source- and a destination-port to identify the application on the receiver which the package is addressing.

> **Note:** In computer networking, a **Port** is a single 16-bit unsigned integer number and abstractly describes a single communication endpoint to which a software service, running on the computer, can bind to. Through this mechanism it becomes possible to mark specific applications on a computer as the receiver of a transmission, by specifying that application's respective port, despite the computer only providing a single networking interface with a single networking address.

TCP

The Transmission Control Protocol (TCP) is an OSI Layer 4 protocol and another core member of the Internet Protocol Suite. Unlike UDP, the TCP specifies a process through which two endpoints of a communication will establish a logical persistent connection (Handshaking). Once such a connection has been established, all data packages sent through this persistent connection benefit from protocol-level transmission reliability. All packages are guaranteed to be sent and received successfully, completely and in correct order. To achieve this, TCP mandates that each transmission be equipped with a sequence number and the sender hold the transmission in a buffer until the receiver has signified the successful arrival of that package, or requests the transmission to be repeated for the sake of error correction. This makes TCP applicable for applications which require reliable in-order transmissions, but comes at the cost of significantly increased transfer latency, due to protocol overheads, when compared to simpler protocols like UDP.
TCP achieves this protocol-level reliability by adding additional data fields to its protocol header which serve to track the state of the connection over its lifetime. These data fields can be seen in image 2.6. They are the single-bit data fields called CWR, ECE, URG, ACK, PHN, RST, SYN and FIN.

When TCP establishes a connection, the sender will first send a package to a receiver (in listening state) with the SYN flag set, to signify the begin of the connection handshake. If the receiver accepts the connection, it will reply with a package in which both SYN and ACK flags are set, in order to signify the acknowledgement of the opening connection. This is followed by yet another package from the sender, with the ACK flag set, to acknowledge successfully having received the SYN-ACK from the listener. The connection is then considered to be established and similar handshaking steps are taken whenever a sender wishes to transfer data to a receiver. An example of this is visualized in image 2.7.

However, this mechanism has drawbacks when being used over unreliable transmission lines with high signal degradation or error rates. Since TCP guarantees the correct ordering of packages, it might take some time between

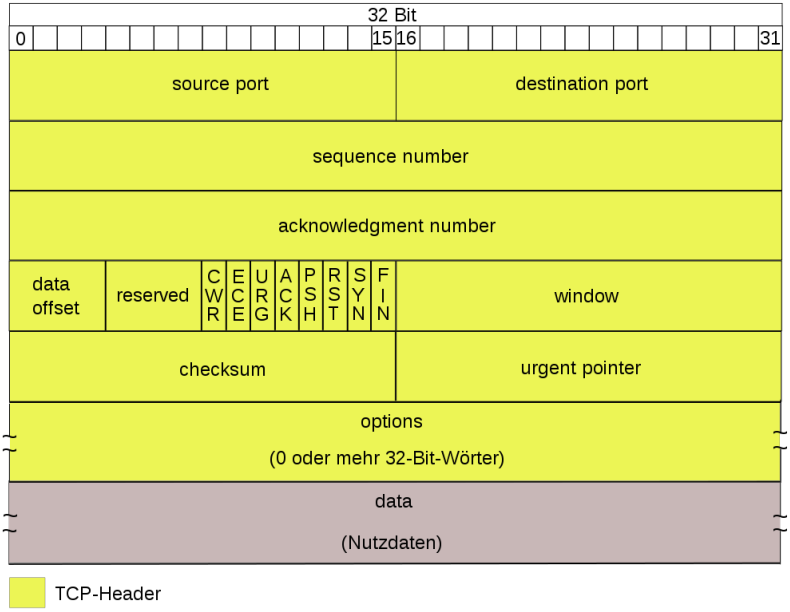

Figure 2.6: Layout of the data fields inside of a TCP Protocol Header [a]

[a] By 'Appaloosa', Wikimedia Commons, licensed under CC BY-SA 3.0 (https://creativecommons.org/licenses/by-sa/3.0/). The work is being used unaltered.

the receiver noticing a missing package, and the sender receiving the corresponding request for re-transmission. All packages that have been sent during this time window are then typically discarded and the transmission sequence is set back to the corresponding missing package. If this happens regularly, a noticeable portion of time will be spent on re-transmitting data and invalidating or blocking all communication across the established TCP connection until the error(s) have been corrected.
To alleviate some of these issues, a proposed successor for TCP, called **QUIC**

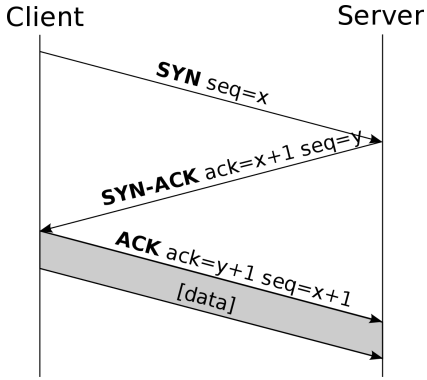

Figure 2.7: The individual steps for a connection handshake specified by the Transmission Control Protocol (TCP). [a]

[a] Derived from a work by 'Snubcube', Wikimedia Commons, licensed under CC BY-SA 3.0 (https://creativecommons.org/licenses/by-sa/3.0/).

has been developed in 2013, which went through multiple standardization iterations and attempts and is now considered to be an official communication standard since 2021. QUIC (Pronounced "Quick". QUIC is not an acronym, but rather the actual name of the protocol) does not expand upon TCP, but rather uses UDP as its base protocol, and aims to establish multiple simultaneous data streams between sender and receiver at the same time. These streams are individually handled by the higher-level implementations of the QUIC protocol, which also provides reliability functions similar to TCP, including retransmissions and sequence checking. The main benefit of this approach is a noticeably reduce in overall connection latency compared to TCP because of the lower protocol overhead of UDP, and in the case of a transmission error, only a single transmission stream will be forced to go into an error recovery state, while the remaining streams can continue operating as normal. Furthermore, QUIC allows to send supplementary user data in its initial stream opening message, which can be used to provide additional information necessary for all following exchanges, like encryption information for Transport Layer Security (TLS)[Die08] encryption, where TCP would first need to fully establish a connection, before such key exchanges can happen

in a second step. Thus saving additional latency over TCP because QUIC can do these "secondary" information exchanges and the stream setup in a single step. Finally, QUIC adds a unique identifier to each data stream, which serves as the primary identifier of an ongoing communication, whereas TCP always uses a combination of source-address and sequence number to identify an ongoing communication. This leads to problems in highly dynamic networks, where one end of a communication might change its local network address (for example due to Network Roaming), which will force a TCP connection to be closed and then re-established and higher-level protocols having to manage the migration of the outdated connection context to the newly established one. Since QUIC does only use the unique stream identifier for such purposes, it is indifferent to an endpoint changing its networking address at any point, so long as the endpoint will continue to use the same unique stream identifier as before.
As QUIC is still a fairly new communication standard, it has not yet been adapted into most data communication applications. This will lead to additional development efforts when establishing a QUIC compatible setup, as QUIC support would have to be added manually, through available software libraries. However, the benefits of QUIC, especially in regards to latency and roaming, are certainly beneficial for latency-sensitive applications and highly dynamic setups like they can be found in modern cloud-computing applications. For these reasons, QUIC will not be considered a suitable choice of protocol for current DAQ designs.

InfiniBand

The InfiniBand standard defines both, a physical interconnect specification, as well as a set of communication protocols. The InfiniBand communication protocol stack reaches up to and including OSI Layer 4. In contrast to Ethernet, InfiniBand is specifically designed around the concept of Remote Direct Memory Access (RDMA).
Any transactions between to endpoint of an InfiniBand connection can be either a self-contained datagram message or an RDMA operation request.

Both of these messaging modes may additionally contain a 4-byte "Immediate" value, which will be transmitted independently of the chosen operation. Additionally, InfiniBand allows for so-called "Atomic" operations, which perform a pre-defined operation of the type "read-compare-write" on a single memory location. The available operations are dependent on the configuration of the respective endpoints, as well as the chosen transport mode between these established pairs.
InfiniBand offers four different transport modes, which fall into two separate categories. Reliable (R) and Unreliable (U). As well as Connected (C) and Datagram (D). (See figure 2.8) In a sense, a Reliable Connected (RC) transport configuration is similar to a connection using TCP. While the Unreliable Datagram (UD) mode is similar to a UDP connection.

Operation	UD	UC	RC	RD
Send (with immediate)	X	X	X	X
Receive	X	X	X	X
RDMA Write (with immediate)		X	X	X
RDMA Read			X	X
Atomic: Fetch and Add/ Cmp and Swap			X	X
Max message size	MTU	1GB	1GB	1GB

Figure 2.8: InfiniBand specifies two groups of transport modes, and influence which transactions are supported. Unreliable (U) and Reliable (R), as well as Connected (C) and Datagram (D). Note that the grey portion of the graph (Reliable Datagram (RD)) is not supported by the current InfiniBand API. [a]

[a] Source: Mellanox Technologies/Nvidia Corporation, "RDMA Aware Networks Programming User Guide", (https://network.nvidia.com/related-docs/prod_software/RDMA_Aware_Programming_user_manual.pdf)

As mentioned in the introduction to the physical interconnect specification for InfiniBand, it is specifically intended to provide easy interoperability between InfiniBand and Ethernet networking installations. For this purpose, a

protocol extension for InfiniBand, called "IP over InfiniBand" exists which allows InfiniBand applications to use the same addressing schemes which are found in native IP based networks, rather than the inconvenient MAC addresses of each individual adapter. Furthermore, there exist two important protocol extension for Ethernet —which does not natively support RDMA — that aim to bring RDMA support to exclusively Ethernet-based networks by emulating InfiniBands protocols and/or wrapping it into an Ethernet-compatible data stream. These protocols are called ***iWARP*** and *RDMA over Converged Ethernet (**RoCE**)*. Since the InfiniBand adapters provided by Mellanox/Nvidia can operate in both, an InfiniBand and an Ethernet mode, using that Ethernet mode in combination with one of the RDMA protocol extension promises to potentially simplify the incorporation of RDMA into existing Ethernet networks, by virtue of being able to retain previous networking equipment, such as switches, and routers. These interoperability features, together with its intrinsic support for RDMA operations, make InfiniBand a primary candidates for the works presented in this thesis.

iWARP

iWARP is a proprietary protocol that encapsulates iWARP-Specific RDMA operation requests inside a TCP/IP data stream, called *Direct Data Placement protocol* or DDP for short, which in turn transports another protocol, called *RDMAP* that specifies the actual DMA operations. iWARP, in contrast to InfiniBand, only supports reliable connection-oriented transport, as this is the only connection mode supported by the underlying TCP. Both, the DDP and RDMAP protocol are proprietary protocols with no public documentation.
In order to gain the actual benefits of RDMA, a networking device that specifically supports iWARP is required. While pre-existing Ethernet networking infrastructure, like switches and routers, can still be used. Due to some quirks in the way TCP handles payload lengths and sequences, iWARP needed to be built around these quirks. In addition, by the way iWARP encapsulates its RDMAP and DDP payloads into the state-reliant connections of TCP, it becomes hard to identify iWARP traffic as such, which prevents most networking equipment, such as switches and routers,

to perform certain traffic-shaping and quality-of-service optimizations. The resulting "best-effort" transport of iWARP as generic TCP traffic causes iWARP performance to fall short of the performance other Ethernet-based RDMA protocols can provide.

RoCE

RoCE follows a different strategy, in which the native InfiniBand protocol gets fully encapsulated into standard UDP packages. While UDP is, by nature, an unreliable protocol, the InfiniBand protocol wrapped inside does support reliable connections (See figure 2.8) and will take care of the required retransmissions and acknowledgements on a higher level. The InfiniBand packages get stripped of their native networking and addressing headers, which get replaced by the UDP and IP addressing headers instead. The payload of the UDP packages then only contain the raw InfiniBand transaction requests and their respective payloads. The generalized layout of such a RoCE package can be seen in figure 2.9.

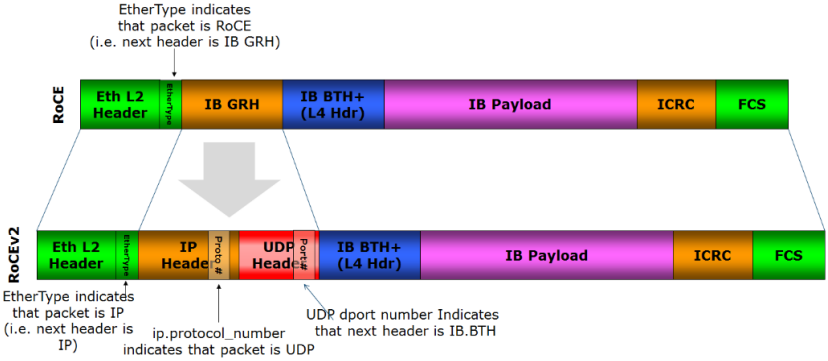

Figure 2.9: The generalized layout of a RoCE data frame for both Version 1 and Version 2 of the Protocol. It is shown how an InfiniBand Header (BTH) and Payload is wrapped in a common UDP package, which is in turn wrapped into an Ethernet data frame. [a]

[a] By 'Ophirmaor', Wikimedia Commons, licensed under CC BY-SA 4.0 (https://creativecommons.org/licenses/by-sa/4.0/). The work is being used unaltered.

This has the strong benefit, that any existing UDP/Ethernet networking equipment can be re-used and doesn't need to be replaced by InfiniBand specific hardware, while still gaining the full benefits of InfiniBand's reliability and RDMA capability. The only requirements are InfiniBand capable networking adapters in the participating computers/server. The remaining Ethernet infrastructure, like switches and routers, however, can remain the same.
Because of this, RoCE will be my primary choice for any Ethernet based application in my DAQ designs, wherever RDMA can be of benefit.

2.2 High Performance Computing Methods

For the development of the computing systems used for the DAQ concepts presented in this thesis, a few key aspects need to be considered. The aspects relevant to this work are the *multi-core paradigm*, *scalable access to shared or slow resources*, the use of parallel *computing accelerators* such as *Graphics Processing Units (GPUs)* and *Field Programmable Gate Arrays (FPGAs)* as typical front-end electronics interface.

2.2.1 The Multi-Core Paradigm

By the early 2000's, linear performance scaling of computing hardware through increased clock-rates alone had reached a practical limit, as leak-current, power consumption and thermal dissipation in highly integrated circuits became significant challenges. With increased integration densities, it became viable to replicate multiple computing structures inside the same processor. Ever since, manufacturers have started to scale processing performance of microprocessors by equipping them with multiple processing cores in parallel. This gave rise to the current era of *multi-core* and *many-core processing*. By today, Central Processing Units (CPUs) with 8 and more logical processing cores are common occurrences, and top-models can offer up to 64 cores at a time. These multiple cores work together and in parallel to complete computing tasks faster than previous single-core systems ever could.

This not only had an effect on the design of processors, but on the architectures of entire computing systems as a whole. While at the end of the 20th century, high-performance computing (HPC) was often performed via single monolithic supercomputers (commonly called *Mainframes*), these systems have largely been replaced by distributed computing clusters consisting of hundreds or even thousands of individual computers, connected by a common interconnect. On a logical level, such distributed systems resemble the multi-core paradigm on a network scale.
The developments towards multi-core and distributed systems introduced a paradigm change in how software for such systems needs to be designed in order to leverage the full computing potential of such systems. Initially, computing systems could be programmed with a single "consumer" of data in mind and programmers needed to pay little attention to concurrent access to data. By now, the locality of data and concurrent access to memories and other shared resources are of high importance, as computing systems become increasingly fragmented, parallel and distributed. Programming libraries, such as the *Message Passing Interface (MPI)*[Gro96] can ease the programming efforts for moving data across system boundaries, but can not generally solve the problems associated with highly distributed computing.

2.2.2 Scalability of data access

In 1994, Wulf and McKee pointed out a limitation in the potential performance increase of computing systems [Wul95]. They realized, that the computing performance of processors was increasing at a faster rate than the data rates of memories. This would eventually lead to a situation in which the execution time of computer programs would be dominated by memory access times, and could no longer scale off an increase in processor performance alone. They dubbed this phenomenon the "Memory Wall".
In distributed and parallel systems, this problem gets further exacerbated when data is stored on a shared resource. If subsystems want to access different data, sequential access to memory will bottleneck overall performance. Already in 1988, John Gustafson proposed a model that describes the

scaling of a system with parallel computing resources, in relation to its purely sequential parts, which has become known as “Gustafson’s Law” [Gus88].

It states:
We define the speedup of a computing process under the application of parallel computing resources as S, with

$$S = p \times N + (1 - p)$$

where p denotes the fraction of a program that can be executed perfectly in parallel, and N is the number of available processors.
This leads to:

$$S = 1 + (N - 1) \times p \tag{2.1}$$

Equation 2.1 shows, that the speedup factor of a program when using multiple processors is proportional to its portion of execution that can be fully parallelized. It is therefore limited by its purely sequential portion. In general, this aspect is known as *scalability*.

From this we derive, that any part of a computing process that is both atomic (meaning, it can not be split into smaller parts) and has a high completion time, is of direct detriment to that process’ scalability. Such types of slow atomic processes are commonly seen when accessing data from external memories, including the transfer of data over an interconnect.

Data Source Speed Grade Hierarchies

In conventional personal computing (and in parts also in HPC), there exists a hierarchy of data sources in regards to their access speeds. Generally, the network can be considered to be the slowest data source. This is followed by permanent storage devices, such as hard-disks, optical drives, USB pen-drives or similar. Yet again followed by multiple layers of volatile memory, such as conventional system Random Access Memory (RAM), and its different layers of caches. The lowest hierarchy layer is made up of register memories, which

are the on-die process memories of the CPU itself. Figure 2.10 illustrates the most typical memory layers inside (super)computers.

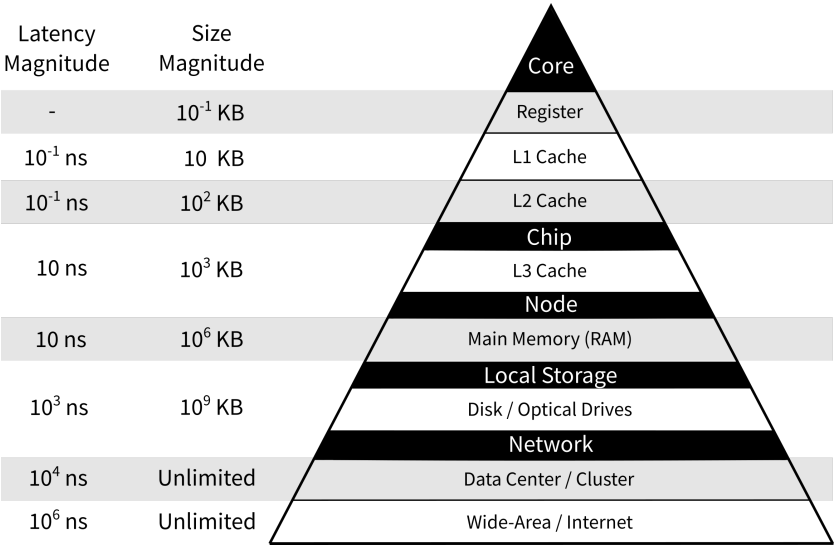

Figure 2.10: Illustration of the hierarchy of different memories of a computer system, as well as their typical access latencies and storage dimensions. [a]

[a] Derived from [Hou17]. Licensed under CC-BY 3.0 (https://creativecommons.org/licenses/by/3.0/)

Data in main memory is often accessed in sequence. For this reason, main system memory has been optimized to be able to continuously return consecutive data elements, following the initial request, at no additional latency cost. This mechanism is called *burst* reading or writing. To lessen the effects of the memory wall, computers have been equipped with memory caches. These caches are designed to offer faster access times than main memory. They can retain data that is frequently reused and can also predict and prefetch additional data before it might get accessed by a processor. This leverages main memory's burst capabilities. When data gets requested by the CPU, the cache can either already have the relevant data stored (hit) or it will need to request data from a lower layer memory system (miss) which incurs additional latency

cost. Is is common for computers to have multiple levels of such caches. In current systems, 3 layers are commonly found, simply denoted as L1, L2 and L3. The lowest layers (L1) are typically placed on silicon, close to the the CPU cores, and offer access speeds of just a few nanoseconds. However, they are limited in size by just a few kilobytes. Consecutive layers of caches are each placed further away from the CPU cores, which allows them to become larger, but at the cost of increased access latencies.
Writing programs so that they are aware of available caches, and optimizing their processing sequences as well as their data structures, so that they optimize the amount of cache hits, can greatly increase a program's runtime.

Below these memory layers and hierarchies lies any form of networked data transfer. Naive data transfer designs for distributed programs can easily make the network become the largest bottleneck. This problem can be approach from hardware, by choosing interconnects with lower latencies and larger throughputs. And it can be approached from software, by optimizing the order and frequency in which data gets transmitted. For example, sending frequent small packets of data is generally undesirable, due to compounding network protocol overheads. It is preferable to collect data and send single large chunks, wherever possible.
Furthermore, it is beneficial for a program to request new data sets as early as feasible, and in parallel to other tasks that are unrelated to the network data transfer. Wherever possible, programs should be designed to already request and receive the N+1 dataset in parallel to the N-th dataset still being processed. Doing this effectively "masks" data transfer behind running computing operations. However, this is not universally possible, depending on the application.

2.2.3 Computing Accelerators and Co-Processors

To assist conventional CPUs in the processing of highly specialized computing tasks, multiple types of co-processors (often also called *Computing Accelerators*) have been developed. Such co-processors often plug directly into a computer system (for example, using the PCIe bus) and provide dedicated

computing architectures that are highly optimized for a specific type of tasks. For example, they can offer dedicated silicon implementations of specific algorithms, such as video encoding or cryptographic applications. They can also offer generally altered computing architectures, that lend themselves better to specific repetitive tasks. Examples of such co-processors can be Artificial-Intelligence Accelerators[Tor22] or in recent times, even Quantum Computing units[Bri17][Ber20]. Also, Field Programmable Gate Arrays (FPGAs) have seen a recent resurgence as dedicated computing accelerator in the form of computing cards with PCIe interfaces. FPGAs will be introduced in detail in section 2.3 of this chapter.
However, the most common type of computing co-processors found today are Graphics Processing Units (GPUs).

GPU Computing

Graphics Processing Units (GPUs) are common *Single-Instruction-Multiple-Data (SIMD)* many-core co-processors found in most computers. Their purpose is to assist the CPU in the processing of raster graphics for display on a computer monitor. By now, GPUs have developed a similar range of capabilities as CPUs do, and are being used to offload more general workloads, besides just graphics processing tasks, in order to accelerate computing-intensive programs. Modern GPUs are equipped with *thousands* of independent computing cores, with top-models reaching into the tens-of-thousands of cores. Figure 2.12 illustrates the differences in scale and number of CPU cores, compared to GPU cores. For example, the *Nvidia RTX 3080 Ti* consumer-grade GPU features 10240 logical processing cores, split accross 80 physical processors (Nvidia calls these: Streaming-Multiprocessors). Their highly parallel computing architectures makes them well suited for repetitive computing tasks that operate on large sets of data.
The two largest GPU manufacturers to date are *Nvidia* and *Advanced Micro Devices (AMD)*.

Figure 2.11: An Nvidia Tesla K40c computing GPU (left) and an AMD Radeon R9 290 consumer-grade GPU (righ)

History of GPUs

In order to reduce the required involvement of the CPU in the processing of raster graphics, manufacturers equipped computers with simple co-processors that had hardware implementations of the most common arithmetic operations required for graphics processing. These co-processors were called *Blitters*. The CPU would program the blitter with a sequence of instructions to perform on the graphics buffer and could then focus on the execution of other tasks, while the blitter would independently execute the programmed instructions, including memory access and data movement.
In time, these blitters grew in complexity and functional range, including the ability to execute instructions on multiple data at once and eventually most computing systems began to offload graphics processing to the blitters all together, relocating the frame buffer from system memory to dedicated memory on the blitter. By today, we know these co-processors as *Graphics Processing Units* or GPUs, for short.

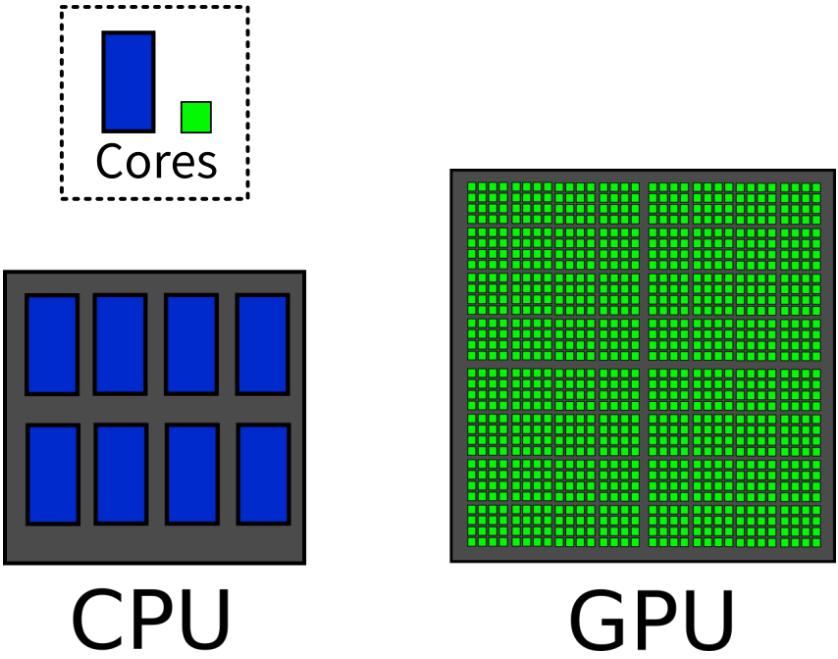

Figure 2.12: A conceptual sketch showing the differences between the number of CPU computing cores and GPU computing cores. The CPU on the left has fewer cores, but each core is highly optimized and features a large functional range. While the GPU on the right features simpler cores with less functional range, but in a greater number.

As the complexity and functional range of most processing systems increased over the years, GPUs eventually also reached a point where their processing units had similar functional range and processing reliability as CPU cores did. At this point, GPUs had become capable of executing general purpose tasks that were no longer limited to graphics processing exclusively. Manufacturers of GPUs realized this potential and provided programming frameworks that allowed to offload computing tasks to the GPU in order to leverage their highly parallel computing architecture. While CPUs are optimized to execute a set of instructions as quickly as possible in sequence, GPUs are optimized to execute individual instructions on a multitude of data at once in parallel. This makes GPU computing particularly useful for any repetitive processing task

in which a short sequence of instructions need to be applied to a large set of data. These types of operations are typically found when working with large vectors of data or in matrix and tensor operations.
By today, General Purpose GPU Computing (GPGPU) has found widespread application in the field of high-performance computing as many supercomputers provide a substantial portion of their overall processing performance through GPGPU Computing.

Furthermore, it has become a common trend for GPU manufacturers to equip their devices with additional specialized dedicated processing units, to extend the functional ranges of GPUs as computing accelerators. For example, Nvidia added so-called "Tensor Cores", which are specialized for the mathematical operations most commonly found in machine-learning applications. In this regard, GPUs share a certain common ground with other dedicated artificial-intelligence accelerators. Processing units for graphics applications, such as ray-tracing, are also common place.

GPU Architecture

Modern GPUs consist of multiple logical processors that consist of a number of sub-components. Nvidia calls these logical processors "Streaming Multiprocessors (SM)" while AMD calls theirs "Scalar Processors (SP)". Both architectures are similar in the types of subsystems they feature, and differ only in detail. They both feature a number of individual compute cores, that are each equipped with a register file to hold processing data. As well as texture processors with specialized texture memory (similar to a cache) that pertain only to the processing of raster-graphics. Additionally, each SM/SP is equipped with an L1 instruction cache, and L2 data and instruction cachees for each computing core, as well as infrastructure modules to access higher-order memory structures.

Just like with conventional CPUs, GPUs implement layered mamory and caches in order to speed up their operations. We distinguish, from slowest to fastest, *Global Memory*, *Shared Memory* and *Registers*.

- **Global Memory:** GPUs come equipped with their own pool of external memory modules that typically reside on the same PCB as the silicon die itself. They are high-capacity SDRAM modules that are highly optimized for memory bandwidth and are commonly labeled as GDDR SDRAM, often simply abbreviated to GDDR. Since these modules reside outside of the GPU silicon die, and need to be externally connected, they suffer from large communication latencies, compared to other on-die memory structures. Global Memory sizes of $8 - 32$ Gigabytes of capacity are not uncommon.
- **Shared Memory:** Each streaming-multiprocessor is equipped with its own L1 and L2 data and instruction cache. As well as pre-configurable static and special purpose caches, such as geometry- and texture-buffers. Addiditonally, most GPUs are also equipped with a small data cache that is individual for each streaming-multiprocessor, called *shared memory*. In some cases, this shared memory can be congruent with the L2 cache. It is part of the integrated circuit silicon die of the GPU and thus can be accessed at lower latencies than global memory. Typical shared memory is between 48 and 96 kilobytes, per processor, with modern GPU designs capable of providing cache access latencies of as low as 50 ns, in cases of a cache hit.
- **Register Memory:** Register memory is the memory structure closest to the arithmetical units of the GPU cores and is highly optimized for access speed. It can usually be accessed with no additional latency. This makes it difficult to fit large amounts of it into a system, as larger register banks would increase critical path lengths and would eventually no longer be accessible with low enough latencies. This is why register memory is severely limited to only a couple hundreds of bytes per computing core. The data that is stored in registers however can only be accessed directly by its respective core.

Accelerator Programming Frameworks

With the highly specialized designs and functionalities of computing accelerators come a number of custom proprietary programming interfaces in order to operate the individual accelerators. The discussion of most of these proprietary interfaces is out of the scope of this thesis. But there exists two particular programming frameworks that are important to know about. Namely, *OpenCL*[Mun09] and *CUDA*[San10].

The Open Computing Language, or **OpenCL**, for short, is a device independent interface definition to perform computing on a wide range of heterogeneous computing architectures, including CPUs, GPUs, FPGAs and some proprietary co-processors. It is developed and maintained as open software, by a non-profit consortium called *Khronos Group*. OpenCL aims to be platform-independent by designing its interfaces in such a way that they target a number of conceptualized "compute devices", which provide a hypothetical set of functionality. Any actual computing hardware can then choose to implement the interface and functional range of any of those compute devices to become OpenCL compliant. Furthermore, OpenCL provides the capability to write custom "extensions" that can be used to target specific non-standardized computing hardware, at the cost of losing the platform independence that OpenCL aims for.
OpenCL offers a language interface similar to C, but with bindings to a wide range of different languages. Due to its broad focus and support for many different computing architectures, OpenCL only features very limited device-specific optimization potential, and is considered by some to be inconvenient to use. However, its universal nature makes it ideal for use on highly heterogeneous computing systems that combine different computing architectures into one logical system.

CUDA is the proprietary GPGPU programming interface of *Nvidia corporation*. It targets exclusively Nvidia's own GPUs and can not be used outside of that scope. Due to this narrow scope, it is highly optimized and provides a number of conveniences specific to Nvidia GPUs. An example that will become relevant in chapter 5 is the flushing of the GPUs global memory cache.

The CUDA language resembles conventional C++ code, but bindings to virtually any contemporary programming language have been developed.

2.3 Field Programmable Gate Arrays

An essential aspect of high-performance DAQ is the link to the detector system. Here, it is common to see a specific type of device being used. The *Field Programmable Gate Array (FPGA)*. Due to how common FPGAs are in many DAQ installations, it is necessary to discuss their integration into commodity DAQ systems.

An FPGA is a form of integrated circuit that provides a number of configurable logic blocks who's functionalities and interconnections between each other can be defined through software in order to implement a large variety of combinatory circuitry. In most contemporary FPGAs, these configurations are not permanent and can be updated and exchanged at any time (hence the term *Field Programmable*). Furthermore, it has become common place for FPGAs to integrate a large variety of pre-fabricated functional blocks that are ready to use, such as clocks, memories, analogue-to-digital and digital-to-analogue converters, transceivers and more. The software interface used to program an FPGA commonly is in the form of a *Hardware Description Language (HDL)*, such as *Verilog* or *VHDL*, which get's compiled into a device-specific block configuration list and the connections between these blocks (a so-called *netlist*) which gets converted into a binary format and is loaded onto the device. Manufacturers of FPGAs commonly provide development environments, including source-code editors and compilers for the HDLs, as well as pre-compiled functional blocks (so-called *Intellectual Property (IP) Cores*) and other tools to ease development efforts. However, as the design process of FPGAs is centered around the abstract description of combinatory logic, the development of FPGA-based applications requires expertise beyond traditional software programming. Furthermore, FPGAs are single chips in an electronic package, and can't be used as stand-alone components. They are intended to be integrated into custom electronics designs, just as any other

common electronic components, which requires yet more development effort. Additionaly, there exist a wealth of pre-fabricated "development boards" for most FPGAs, which provide the means to operate and interact with an FPGA and most of its functional spectrum prior to having to integrate it into custom electronics. For some applications, these development boards can be sufficient.
Currently, the two largest manufactureres of FPGAs are *Xilinx* and *Altera*.

FPGA Architecture

FPGAs are made up of Configurable Logic Blocks (CLBs) that are interconnected in a mesh that makes it possible to arbitrarily connect inputs and outputs of specific CLBs with each other. A single CLB generally consists of three types of logic. Lookup Tables (LUTs), Full-Adders and Flip-Flops. The specific amounts of these components per CLB is dependent on vendor and model of the FPGA. These inner components are interconnected through multiplexers. Control inputs of these multiplexers get defined during device configuration. Through this mechanism, CLBs can be used to realize generic logic functions, arithmetic units and memories. Multiple CLBs are connected together to form larger and more complex logic. The connections between CLBs are also defined during device configuration. An exemplary layout of a CLB is shown in figure 2.13.

In addition to CLBs, most FPGAs feature predefined structures in silicon, such as multipliers, memories, clocks or similar, which can be used in conjunction with the CLBs to extend an FPGAs functional range.

FPGA Programming

An FPGA gets configured by applying a *bitfile* to it. This bitfile contains the configurations of the CLB multiplexers, as well as the interconnection meshes. A bitfile is device-specific and is generated from a higher-level *netlist* that abstractly describes the desired arrangement of logic blocks. The netlist is in turn compiled from a higher level hardware description language, which is

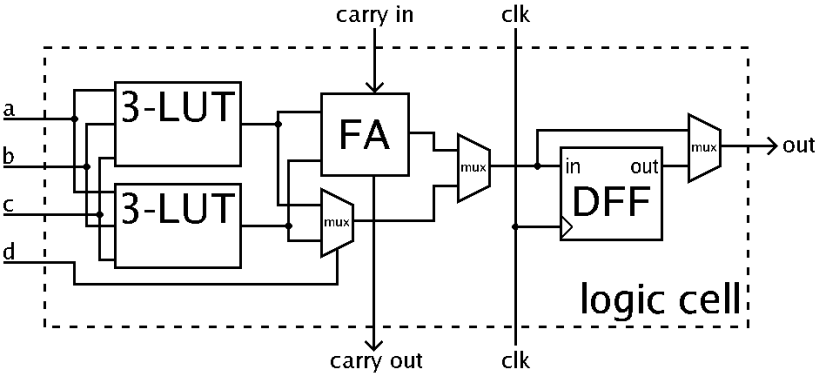

Figure 2.13: An exemplary layout of an FPGA CLB consisting of two Lookup Tables (LUT), a Full-Adder (FA) and a Flip-Flop (DFF). The paths, configurations and interconnects between these components are defined by several multiplexers (MUX). [a]

[a] By "Petter.kallstrom", Wikimedia Commons. No CC.

used to describe the desired functional behaviour of the FPGA. It is also possible to program FPGAs indirectly through other means. For example, FPGAs are also a common target for the previously mentioned OpenCL programming framework. In such cases, the FPGA will be loaded with a pre-defined configuration that is compliant to the OpenCL computing modell. Code written in OpenCL can then be compiled for that infrastructure and transferred to the FPGA to be executed, rather than programming the FPGAs logic cells directly.

Applications

Many FPGAs feature high numbers of general purpose I/O pins that can be used to control external electronics. Together with integrated hard-cores, like signal amplifiers or analogue-to-digital converters, FPGAs are a popular choice for signal processing applications. For these reasons, FPGAs are also very commonly found in DAQ, mostly close the detectors, where they serve as front-end electronics for data sampling and real-time control. They are also often used for data processing, since FPGAs can implement virtually any digital logic. Since it is possible to even synthesize fully functional CPUs

```
531   //#####################################################################
532
533   //#####################################################################
534   //# BEGIN PCIe
535   //#####################################################################
536
537   // Tie off for pipe_mmcm_rst_n
538   assign pipe_mmcm_rst_n = 1'b1;
539
540   // Create a Clock Heartbeat
541   always @(posedge user_clk) begin
542     if(!sys_rst_n_c) begin
543       user_clk_heartbeat <= #dly 26'd0;
544     end else begin
545       user_clk_heartbeat <= #dly user_clk_heartbeat + 1'b1;
546     end
547   end
548
549   pcie3_7x_0_support #
550   (
551     .PL_LINK_CAP_MAX_LINK_WIDTH     ( PL_LINK_CAP_MAX_LINK_WIDTH ),
552     .C_DATA_WIDTH                   ( C_DATA_WIDTH ),
553     .KEEP_WIDTH                     ( KEEP_WIDTH ),
554     .PCIE_REFCLK_FREQ               ( REF_CLK_FREQ ),
555     .PCIE_USERCLK1_FREQ             ( USER_CLK_FREQ ),
556     .PCIE_USERCLK2_FREQ             ( USER_CLK2_FREQ )
557   )
558   pcie3_7x_0_support_i (
559
```

Figure 2.14: An example of FPGA firmware code written in Verilog

inside an FPGA, they can also be used to solve any computational task. FPGAs are particularly useful to implement highly specialized digital logic for specific algorithms. Depending on the device's number of available CLBs and other resources, computational logic can potentially be replicated many times, enabling highly parallel and concurrent processing. Depending on the use case, an FPGA can potentially even outperform conventional software computing on a CPU, even though FPGAs generally feature significantly lower clock rates than common CPUs do. While many commodity CPUs typically provide clock rates of 3+ GHz, the implemented circuits inside of FPGAs can rarely reach beyond 1 GHz. This is mainly due to the large structure sized of

an FPGA's logic blocks, and the interconnections between them, which result in long critical data paths. For these reasons, FPGAs are not ideal to solve general purpose software computing tasks.

To alleviate the problem of slow general purpose computation, a new generation of FPGA devices has emerged over the past couple of years, which combines programmable FPGAs with a fully functional CPU based System-on-Chip (SoC) in a single silicon die. For example, the Xilinx Zynq class of devices combines their FPGAs with a Dual-Core ARM Cortex-A53 conventional CPU, as well as an additional Dual-Core ARM Cortex-R5F real-time CPU. These systems are capable of supporting DDR4 RAM and fast interconnects like Gigabit Ethernet or PCIe. The integrated ARM CPUs make it possible to run conventional computing operating systems, such as Linux, on them, to provide the full flexibility of a software-programmable CPU of decent computing performance, while being able to seamlessly access the synthesized logic of the integrated FPGA. An evaluation and development board of the Zynq family of devices is shown in figure 2.15.

Figure 2.15: Picture of a Zynq Ultra Scale Plus Development and Evaluation board of the ZCU102 family of MPSoCs [a]

[a] Image property of AMD / Xilinx Inc.

More recent developments of FPGAs have focused highly on their application as computing accelerators. Mainly driven by the rise in popularity of machine-learning applications. To this end, FPGA manufacturers have started to develop commodity computing components that integrate special FPGA types that are highly optimized for computational tasks. Similar to the Zynq family of SoCs, the *Alveo* computing accelerator family of devices (fig. 2.16), developed by Xilinx, combine large powerful FPGAs with many-core ARM CPUs. These devices come pre-packaged on a PCIe extender card that can plug into a conventional computer or server. In addition, these card also feature integrated high-speed Gigabit+ Ethernet transceivers. With this, these types of devices are ideally suited to be applied in data-center installations to offload highly specific computing tasks onto them in distributed computing systems.

Figure 2.16: A Xilinx ALVEO computing accelerator card with two SFP slots (left side of card) for Gigabit Ethernet and a PCIe x16 connector (bottom of card) to be plugged into a computer. The card is equipped with an FPGA (covered by the red housing) and internal cooling. [a]

[a] Image property of AMD / Xilinx Inc.

3 Direct Memory Access

Direct Memory Access (DMA) is a method of interacting with memories on a shared interconnect without continuous involvement of the CPU and its memories and caches. Traditionally, whenever data gets moved between devices attached to the system bus, the CPU is the active component that facilitates those data transfers. For example, when a program requests data to be read into main memory from a peripheral device, such as a hard drive, the CPU will be the main active component in that exchange. The CPU will issue the read command to the bus, wait for data to be put onto the bus and then collect the returned data from the bus, temporarily storing the data in its internal registers. It will then take this data from its internal registers and post a write-request to the main memory for that data.
This approach requires constant involvement of the CPU whenever data is being moved to or from memory, preventing the CPU from executing other programs while doing so. It also increases the length of the critical path of data by having to traverse all involved cache layers, further increasing transfer latency. Eventually, a concept has emerged that allows peripheral devices on a common bus to access memories attached to that same bus *directly*, without requiring constant involvement of the CPU. This concept is called *Direct Memory Access*, or *DMA*, for short.
A wide range of computing components that pertain to any form of (temporary) storage nowadays support DMA operations. In fact, DMA has become common place and is widely adapted and used in most contemporary operating systems. For example, most storage devices nowadays exchange data with the systems main memory almost exclusively via DMA.

To be able to perform data copying operations on its own, a device must have a so-called *DMA Engine*. It is a semi-programmable piece of hardware that

can interact with connected memories independently of the CPU. A DMA engine does not receive data directly, but instead receives instructions where to find data in memory and where to copy it to. The data structures used to configure the DMA engine are commonly called *Descriptors*. Descriptors are typically placed in system memory, and the DMA engine receives addresses as to where it can find them. These descriptors contain all necessary information for the DMA engine to perform a transfer operation, as well as some reserved memory for the DMA engine to write process information to. The software running on the CPU that initiated the DMA transfer is expected to periodically monitor these descriptors in order to keep track of the data transfer. In some cases, the system can also be configured to trigger an interrupt upon transfer completion.
The source and destination informations stored in the descriptors can either consist of a matching pair of single start- and stop-addresses of memory for both the source and destination respectively. Or it can contain a list-structure, consisting of multiple such information pairs at once. This is commonly called a *Scatter-Gather-List*. The DMA engine will then read the entries in the descriptors and scatter-gather-list and commit matching read or write requests onto the system bus. This process is repeated for each of the submitted descriptors.
The general schema and comparison of conventional data transfer and DMA enabled data transfer is shown in figure 3.1.

> **Note:** In most cases, the sequence in which descriptors and/or the entries in their scatter-gather-lists get processed is generally considered to be indeterminate. Meaning, the soft- or hardware of the DMA device might decide to process the requested transfers out of sequence for any reason.

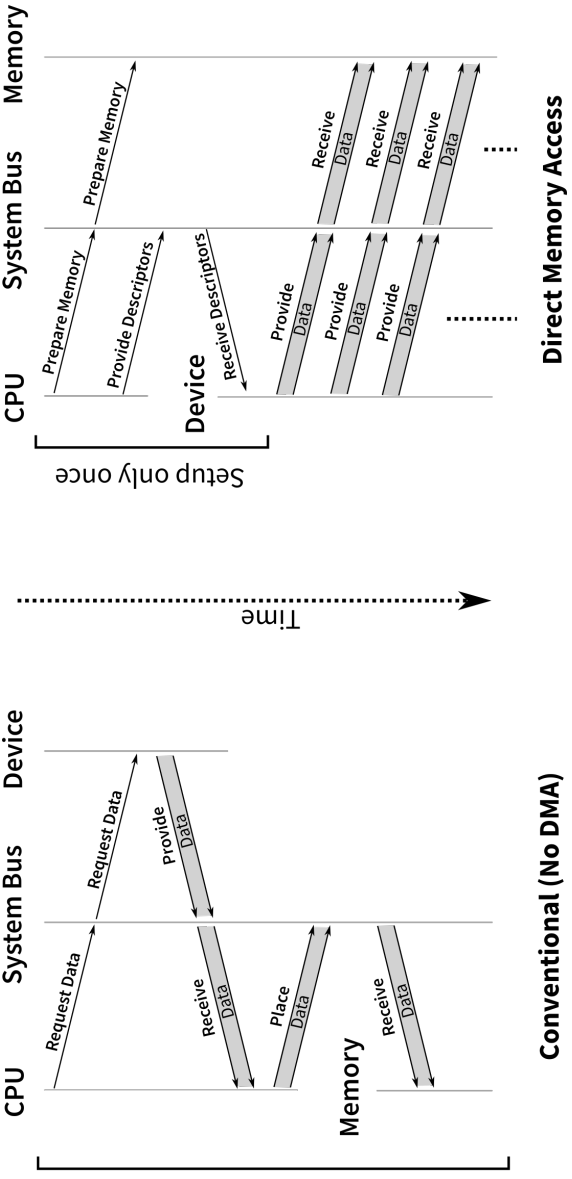

Figure 3.1: Schema of the data transfer schemata and paths from a peripheral device to main system memory. Conventionally (left) the CPU requests and receives data from the peripheral device and forwards the data to main memory. This is repeated for each data. When using DMA (right) the CPU only configures memory and the device. The device can then freely place data directly into the prepared memory without involvement of the CPU.

DMA and Virtual Memory

Before a segment of memory may become the target of a DMA operation, it first needs to be prepared by the system.
When a program requests system memory from the operating system, the operating system keeps track of the memory addresses that have been assigned to the program. Each time the program continues its operation after being woken up by the system scheduler, it can continue its operation where it left off, expecting data to still reside exactly where it was placed previously. However, most operating systems implement a feature called *Memory Swapping*. This feature was implemented in an attempt to better utilize system memory, because memory was a limiting factor in the earlier days of computing. The operating system will identify memory of a program which has not been accessed over a certain period of time. That memory is then written out to a hard drive to free it up. When the corresponding program is woken up again by the scheduler, its data is written back from disk to memory. This works well so long as the operating system is able to write data back into the exact same location. However, if the previous location of data has already been redistributed to other programs, it can not be placed back into the same location. The corresponding program would cause then memory access violations if it tried to access those outdated memory locations.
To solve this issue, most operating systems rely on a concept called *virtual memory*. Virtual memory relies on what is know as a *Page Table*. A page table serves as an indirection to memory access and effectively redirects the requested memory operations of a program to an associated physical location. It works by generating fixed sized blocks in a table (the pages) that each map to a physical location in system memory. This association can be changed and rewritten. Each program that requests memory from the operating system will be assigned an appropriate amount of pages to fulfill that request. And rather than addressing memory directly via physical address, processes will now instead address an offset inside of their assigned pages' memory range when accessing memory. The operating system will then look up the indirection in the page that the program is trying to access and redirects the memory access accordingly. This concept is visualized in figure 3.2.

When a process' memory gets swapped out and eventually is swapped in again, all the operating system needs to do is update the corresponding page table entries. The process can then access its virtual memory as per usual, and the requests will be redirected to their new locations accordingly.
Nowadays, this memory virtualisation is handled by discrete logic blocks that are built into modern CPUs and/or mainboard chipsets. They are called *Memory Management Units* or MMUs, for short. The operating system merely provides these units with the required information. The MMUs will take care of catching memory access requests from progrmas and transparently redirecting them according to their page tables.

> **Note:** Nowadays, swapping has lost part of its significance for common PC system, as system memory is readily available and is no longer as much of a limiting factor. However, memory virtualisation is still abundantly used by many systems as it provides a set of further advantages. Examples include:
>
> - Access permission enforcement and monitoring
> - Sharing of memory between processes by mapping multiple pages to the same physical memory
> - Providing more virtual memory to a process than is physically available, when operating on highly sparse data structures
> - Memory layout randomization, to physically separate critical segments of memory. This makes it harder for an attacker to locate information, should they gain access to the physical memory address space

This virtualisation mechanism provides problems when wanting to use DMA. Once a DMA engine has been configured with the physical addresses of data to access, it will not be aware of that memory changing ownership, should the system swap out that memory region. This could potentially cause the DMA engine to read stale data, or worse, overwrite program data of a completely unrelated process that has been swapped into that location. To prevent this,

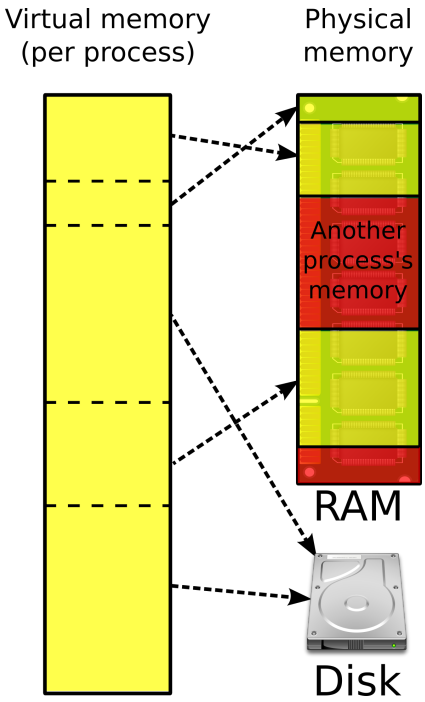

Figure 3.2: Visualization of the concept of Virtual Memory in a computer system. Pages on the left are being mapped onto actual memory location on the right. [b]

[b] By 'Ehamberg', Wikimedia Commons, licensed under CC BY-SA 3.0 (https://creativecommons.org/licenses/by-sa/3.0/). The work is being used unaltered.

any memory region that is to be prepared for DMA access needs to be "pinned" first. Pinning instructs the operating system to not swap a pinned memory segment under any circumstances. Once a process finished DMA operations and wants to free up the memory segment, it simply instructs the operating system to unpin the memory again. Additionally, since a DMA engine will send instructions to physical memory, DMA descriptors need to contain de-virtualised physical addresses. The operating system provides functionality for the translation of virtual addresses into physical ones. In many cases, this is automatically handled by the DMA device driver.

3.1 Remote Direct Memory Access

The concepts of DMA have also been extended onto the network and interconnect level, creating the concept of *Remote Direct Memory Access (RDMA)*. RDMA follows the same basic principals as the basic DMA does but in combination with an attached networking interface.
Conventional networking devices usually contain an internal data buffer where data gets stored after it has been received from the network. The networking device will then notify the operating system about available data, which will then be picked up by the CPU to be stored in main memory. This type of procedure incurs additional effort on the CPU in order to pick up data from the networking device's buffer. In some operating systems, this issue gets further exacerbated by the data processing chain that follows after the data has been received (the Network Stack of the Operating System). The necessary sequences for most data transfer protocols are usually handled by software inside of the operating system. This protocol handling often involves dozens of additional data movement, processing and copying steps. Each of these steps increase the length of the data path and binds up the CPU with the required protocol procedures, increasing the overall transfer latencies.

> **Note:** For the most common communication protocols, such as TCP/IP, most standard networking devices are able to handle a substantial portion of the protocol overhead by themselves, in the form of dedicated hardware on the device. This is known as *TCP/IP Offloading*. While this does reduce CPU involvement, it does not reduce the overall length of the critical data path by much. Received data still needs to traverse the operating system's network stack.

RDMA aims to avoid the overheads associated with this conventional form of networked data transfer. RDMA capable network adapters are equipped with dedicated hardware that handles the entirety of the protocol overhead, as well as a DMA engine. This bypasses the operating system's networking

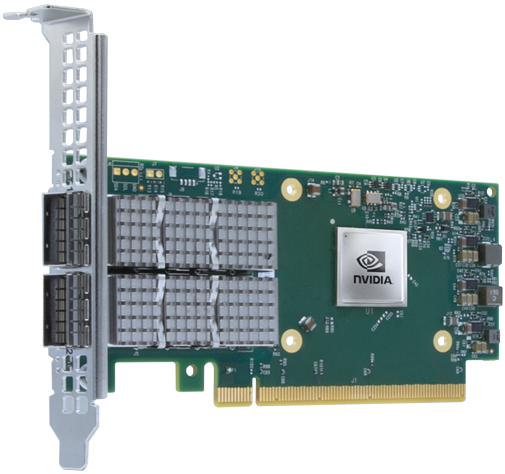

Figure 3.3: An Nvidia Connect X6 InfiniBand network adapter with two QSFP 200 Gbps ports and a PCIe Gen. 3 x16 plug. [a]

[a] Image property of Nvidia corporation.

stack and requires minimal CPU involvement.
The procedure is generally the same as with conventional DMA, where the DMA engines work by processing a set of descriptors, instead of getting payloads passed to them directly. However, in order to send data across the network, sender and receiver first need to exchange access information with each other, in order for the receiver to prepare for the incoming data transfer. This configuration step (Handshake) needs to be managed by software, involving the CPU. Once the connection between sender and receiver has been established, data can be sent freely, without further CPU involvement.
This handshake consists of a set of additional information from the receiving end of the transfer. Specifically, the receiver first needs to pin a region of system memory which will be access via DMA from the network interface. This memory-region will be described in the form of an access token that contains the system-bus-specific addresses of the memory region, as well as a generated *key* in the form of a 64-bit integer number. In order to allow a peer to access this memory-region directly, it has to request an access token from

the host beforehand. The peer will then have to use this token to authenticate against the host before issuing DMA operations, or the host will drop all incoming data packages from that peer. This access token is generated per memory region and will remain valid until revoked by the host.

3.2 GPU as Target of RDMA

Over the recent years, General Purpose GPU Computing (GPGPU Computing) has increased in significance for applications with high computational demands. As such, GPGPU computing has also become a valuable asset for scientific applications, be it for simulations, image processing, machine learning or other computing intensive tasks.
GPUs are commonly equipped with large dedicated memories, optimized for their processing architecture. This so-called *Graphics Double Data Rate Random Access Memory (GDDR RAM)* is similar to a computer's normal DRAM Memory. But GDDR is highly optimizes for large throughput at higher transfer frequencies than normal system DRAM. This optimization is done not only through architectural changes in the RAM modules themself, but also by connecting the memory modules directly with the GPU's main processor on the GPU's PCB. Having dedicated memory for the GPU with a direct connection, rather than a generic bus, reduces communication overheads and speeds up data transfer by lowering signal propagation latencies. Since the GDDR is dedicated exclusively to the GPU, it is not connected to the host computer's main data bus and does not share a common address space with the rest of the system memory. It is therefore inaccessible to the CPU directly and will require the assistance of the GPU driver to be accessed. The driver handles the necessary device communication in order to transfer data to and from the GPU.
In systems that utilize multiple GPUs for joint computing tasks, relying on system memory as an intermediary for data transfer between GPUs can slow down performance of the whole system. To tackle this problem, GPU manufacturers have started to expand their GPUs to be DMA capable. To achieve

this, the GPU registers a segment of the system bus address space during device enumeration (the so called *User Bank Register (BAR)*). The rest of the system can then address data transfers to this BAR and the GPU will pick up data from the bus as if the GPU were conventional memory. The GDDR is not actually exposed to the system bus, but the hardware on the GPU that has access to the bus will instead *redirect* incoming data to the GDDR. In most cases, the GPU's driver still needs to be involved in order to do necessary configuration for the GPU to be aware of the incoming DMA. Once this configuration has concluded, the acquired BAR address from the GPU can be addressed like conventional memory for the purpose of DMA operations.

With this new functionality, GPUs can also become the targets of RDMA operations over a network. In the case of a distributed computing environment that uses GPU computing resources across a network, the data path from data source to GDDR (and potentially back) can becomes very long. Data needs to arrive on the network device first, traverse system main memory and then be transferred to the GPU by its driver. This introduces considerable overhead into the network communication with a GPU, diminishing its performance and suitability for tasks that require fast response times. Letting the RDMA network adapter access GDDR "directly", without a detour through system main memory, can therefore reduce critical data path lengths and lead to shorter systemic data transfer latencies. This is visualized in figure 3.4. The two largest GPU manufacturers, Nvidia and AMD, have registered different trademarks for their respective implementations of these GPU RDMA features. Nvidia is calling their implementation *GPUDirect*, while AMD calls their implementation *DirectGMA*. The principles behind their respective technologies are the same, however.

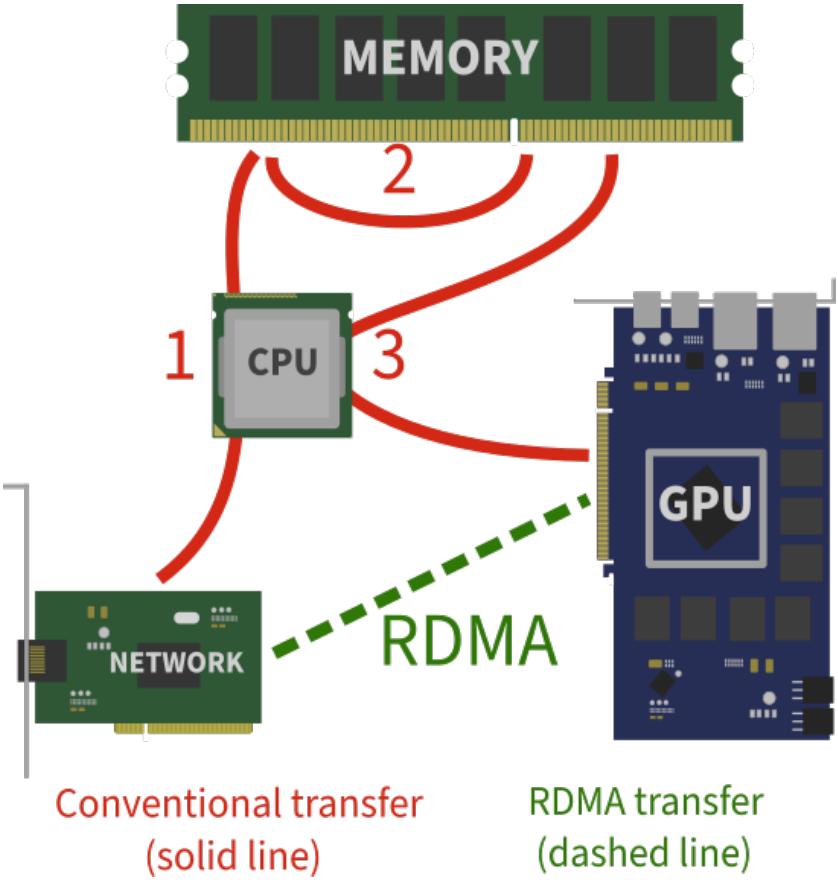

Figure 3.4: Visualization of the different data paths for data transfer from the network into a GPU. The solid line represents the conventional data path without DMA of any form. The dashed lines represent the data path when utilizing GPU RDMA.

3.3 GPU DMA using an FPGA

FPGAs can provide a number of benefits to high-speed DAQ. The article [Rot15] introduces a DMA capable FPGA data acquisition platform that we call "High-Flexibility FPGA DAQ Platform", or simply *High-Flex* for short (fig. 3.5). It implements DMA data transfer functionality over a PCIe bus as a DMA master device. The platform is based on a Xilinx Virtex 7 FPGA and offers external connectivity via two high-density FMC connectors and one PCIe x16 plug. This PCIe plug allows for the device to be plugged into the PCIe bus of a conventional computer system. A Linux driver for the device is available, through which the device can be configured and operated.
In [Rot16] and [Cas17] we have successfully demonstrated the feasibility of using High-Flex to transfer data into a GPU's memory, using DMA. In an x8 PCIe configuration, the system was demonstrated to reach transfer throughputs of up to 52 gbps, as well as transfer latencies of 1–2 μs.

Performance Measurements

To get a better understanding of the transfer latency behavior of the High-Flex platform in a GPU computing context, we analyzed its transmission latencies under a number of different scenarios.

We used an ASUS X99-E WS/USB3.1 motherboard, equipped with an Intel Xeon E5-1650 CPU. This system features seven PCIe Gen. 3 slots that are split across two separate PCIe bridges (think: Internal Switches) which are connected to the same PCIe Root Complex. One of these slots is statically set up for x16 operation. The remaining slots are each paired together with a so-called "Quick Switch (QSW)" which can be configured to either operate both of its connected PCIe ports in an x8 configuration, or operate only one of the two ports in an x16 configuration. The layout of the system is shown in figure 3.6

We are interested in the transfer latencies of the system under different configurations and scenarios. We tested the system using the High-Flex board as

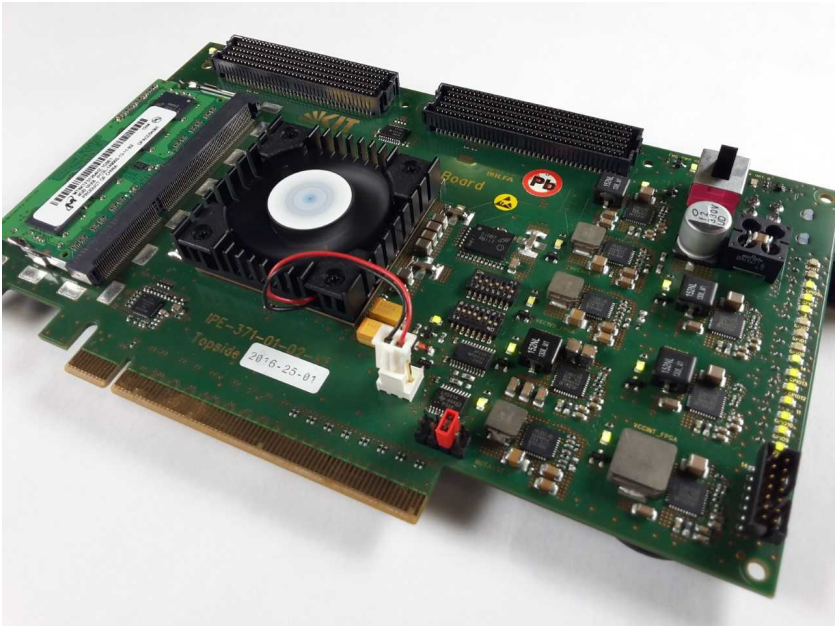

Figure 3.5: Picture of the KIT high-flexibility FPGA DAQ board (High-Flex). The FPGA in the center of the board is covered by a heat sink and fan. The board is designed to be plugged into an x16 form-factor PCIe port as an extension card. Among other interconnects, the board also features two FMC connectors, shown on the top of the PCB.

data generator and two different GPUs (one Nvidia Tesla K40 and one Nvidia Tesla P100) as target devices. The system is assembled in such a way that all three devices are connected to a PCIe port that is configured for x16 operation. Each GPU can either be connected to the same PCIe bridge as the High-Flex board, or it can reside on the other bridge. In the first case, the High-Flex board can directly communicate with the GPU on its bridge. In the latter case, the PCIe communication needs to be switched across the Root Complex and the second bridge, in order to establish communication.
A software was developed that configures the communication between each of the devices, then starts and monitors data transfer. The data used for the

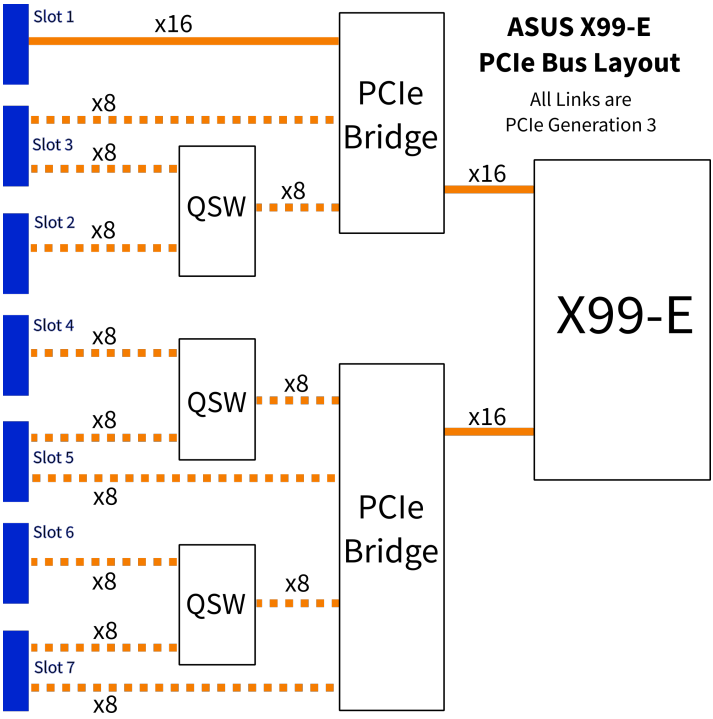

Figure 3.6: Schematic representation of the ASUS X99-E PCIe bus layout. One Root Complex is split into two x16 lanes which are each connected to a PCIe Bridge. PCIe slot 1 is fixed to x16 operation at all times. While all other remaining ports are arranged in pairs of x8 links each. Each pair is attached to a Quick-Switch (QSW) which can be used to reassign the switche's x8 link to one of the connected slots. This forms either one x16 slot and leaves the other slot in the pair without function, or forms two independent x8 slots.

transfer is generated internally by the High-Flex FPGA. It is a simple increasing counter. Target memory on the GPUs is configured and pinned by the software. It then creates DMA descriptors for those target memories and transfers them to the High-Flex board. The board is equipped with 4 GB of DDR memory that is used both as data buffer, as well as internal operational memory for the DMA engine. Parts of this memory is mapped to the device's PCIe BAR which can be accessed directly by the CPU. The CPU loads the descriptor into

this memory and instructs the DMA engine to start transfer. The DMA engine on the High-Flex device writes transfer progress information into the descriptors previously loaded onto the device. The CPU continuously monitors these descriptors to measure the start and end of a transfer. A total of 4096 bytes are transferred at once. Each transmission is repeated 500 times, and we observe the distribution of the transfers.

Three different scenarios were tested to compare the impact of data paths in the inter-system data transfer performance.

- **Conventional Transfer:** Data is transferred from the FPGA to GPU memory in two steps. Data is first transferred from the FPGA to system main memory via DMA, and then further transferred to the GPU. The software program running on the CPU observes the completion of data transfer from the FPGA to system main memory and then uses the CUDA programming interface call *cudaMemcpy* to transfer that same data to the GPU. Results are summarized in table 3.1 and are detailed in figure 3.7.
- **Transfer on same PCIe Root-Complex:** In this scenario, both the target GPU and the High-Flex FPGA were connected directly to the same PCIe root-complex on the bus. Data was transferred from FPGA to GPU using DMA with no involvement of the CPU or system main memory. Results are summarized in table 3.2 and are detailed in figure 3.8.
- **Transfer over PCIe Switche:** In this scenario, both the target GPU and the High-Flex FPGA were connected via a PCIe switch to the same root-complex. Data was transferred from FPGA to GPU using DMA with no involvement of the CPU or system main memory. Results are summarized in table 3.3 and are detailed in figure 3.9.

Conventional Transfer

Table 3.1: Latency measurements (in μs) of data transfer from High-Flex FPGA to RAM and then further to GPU (Nvidia K40 and Nvidia P100). Sampled over 500 iterations of 4096 bytes.

GPU	Average	σ	min	max
Nvidia K40	19.48	±4.24	8.19	55.25
Nvidia P100	20.08	±3.66	8.94	41.44

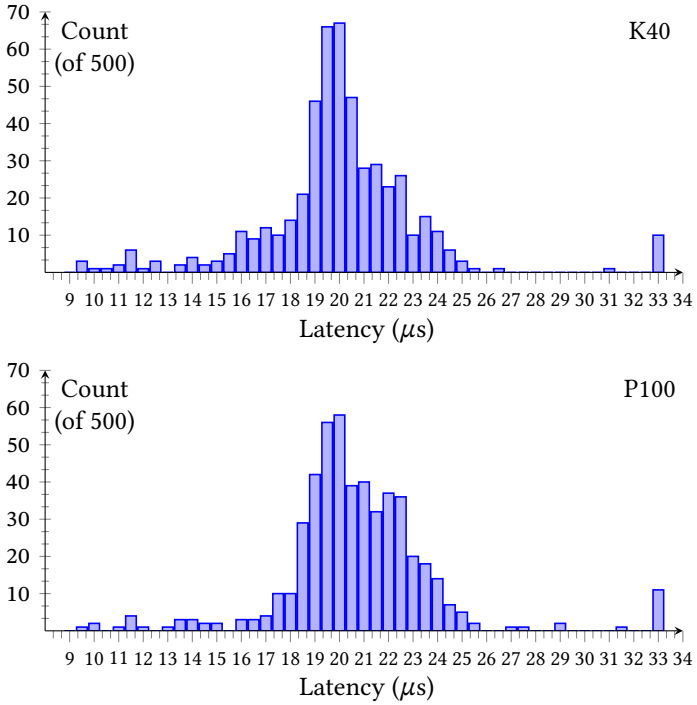

Figure 3.7: Distribution of latency measurements (in μs) of data transfer from High-Flex FPGA to RAM and then further to GPU (Nvidia K40 (**top**) and Nvidia P100 (**bottom**)). Sample sizes were 500 iterations of 4096 bytes.

Transfer on same PCIe Root-Complex

Table 3.2: Latency measurements (in μs) of data transfer from High-Flex FPGA to GPU (Nvidia K40 and Nvidia P100) connected to the same PCIe switch, using DMA. Sampled over 500 iterations of 4096 bytes.

GPU	Average	σ	min	max
Nvidia K40	1.67	$\pm$0.055	0.884	1.816
Nvidia P100	1.73	$\pm$0.46	0.804	11.88

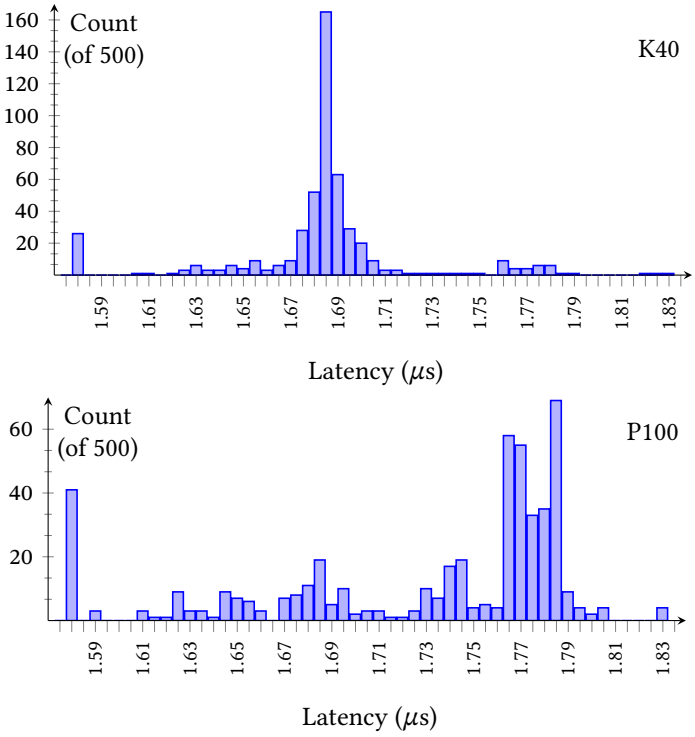

Figure 3.8: Distribution of latency measurements (in μs) of data transfer from High-Flex FPGA to GPU (Nvidia K40 (**top**) and Nvidia P100 (**bottom**)) on the same PCIe root-complex, using DMA. Sample sizes were 500 iterations of 4096 bytes.

Transfer via PCIe Switch

Table 3.3: Latency measurements (in μs) of data transfer from High-Flex FPGA to GPU (Nvidia K40 and Nvidia P100) connected to separate PCIe switches on the same root-complex, using DMA. Sampled over 500 iterations of 4096 bytes.

GPU	Average	σ	min	max
Nvidia K40	1.70	±0.063	0.848	1.796
Nvidia P100	1.74	±0.078	1.572	1.844

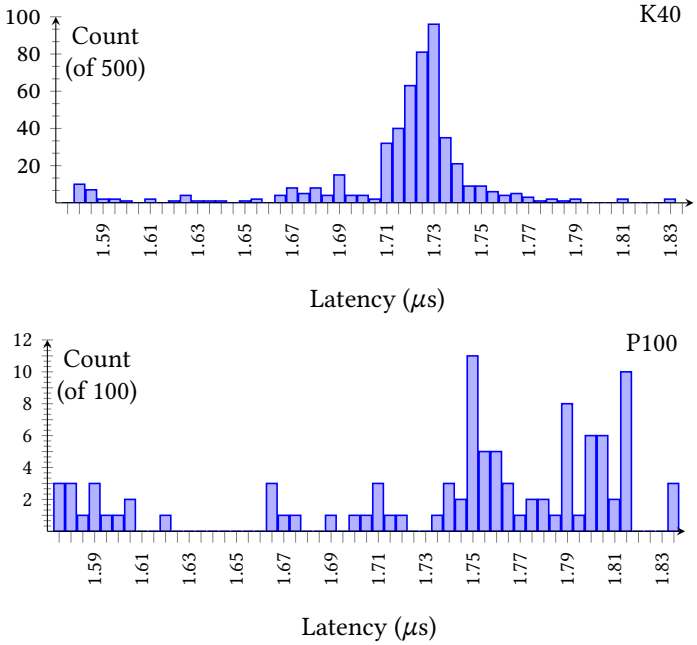

Figure 3.9: Distribution of latency measurements (in μs) of data transfer from High-Flex FPGA to GPU (Nvidia K40 (**top**) and Nvidia P100 (**bottom**)) on different PCIe switches on the same root-complex, using DMA. Sample sizes were 500 iterations of 4096 bytes for the *top* (K40). Due to a misconfiguration in the measurement script for the *bottom* (P100), only 100 iterations of 4096 bytes were performed. The general shape of the distribution is still representative, and aligns in its overall shape to the distribution of the P100 in the "same root-complex" scenario in figure 3.8.

From the presented data it can be understood that DMA data transfer into GPU memory directly, without detour through system main memory, reduces transfer times by one order of magnitude, from 20 μs to 2 μs. Furthermore, standard deviations of transfer times were decreased by almost two orders of magnitude, from 4 μs to 0.05 μs, which significantly increases real-time applicability of DMA data transfer over conventional methods. The "professional grade" GPU (K40) performed generally better than the "consumer grade" GPU (P100), showing much more consistent transfer times overall.
The overhead introduced by the PCIe switch increased the average transfer times from 1.67 to 1.70 while simultaneously negatively affecting the standard deviation, increasing it from 0.055 to 0.063.
In conclusion, this shows that DMA data transfer significantly increases transfer performance and reduces transfer time deviations, which is of great importance for real-time applications. For such applications, a configuration with both sender and receiver of a data transfer on the same PCIe root-complex is preferable.

4 KIRO: An RDMA programming library

RDMA has been identified as a key component for data communication at lowest latencies. To explore the necessary software requirements for RDMA communication, I created **KIRO**. The *KIT Infiniband Remote Object* RDMA networking library. It is intended to allow a host/server to provide unrestricted RDMA-Read access to the memory contents of any single abstract "data object" (think: a single contiguous region of memory). Hence the name "remote object". KIRO is geared towards minimalism, to ease integration complexity into other software projects. The establishment and management of a barebones KIRO connection heavily resembles that of conventional UNIX sockets. KIRO's primary usecase is a server-and-client based unidirectional RDMA data transfer. In addition, it also offers higher-level convenience functionality, such a self-synchronizing ring buffer and a many-to-many message passing module.

KIRO is open source and available under an LGPL 2.1 license. It can be found on Github under https://github.com/ufo-kit/kiro

4.1 KIRO Software Design and Components

An outline of the general software stack KIRO is built upon can be seen in figure 4.1. KIRO was developed with Mellanox-brand InfiniBand network adapters in mind. It is however compatible with any RDMA network adapter that is supported by the *RDMA Communication Manager (rdma_cma)* and *IB_VERBS* programming libraries.

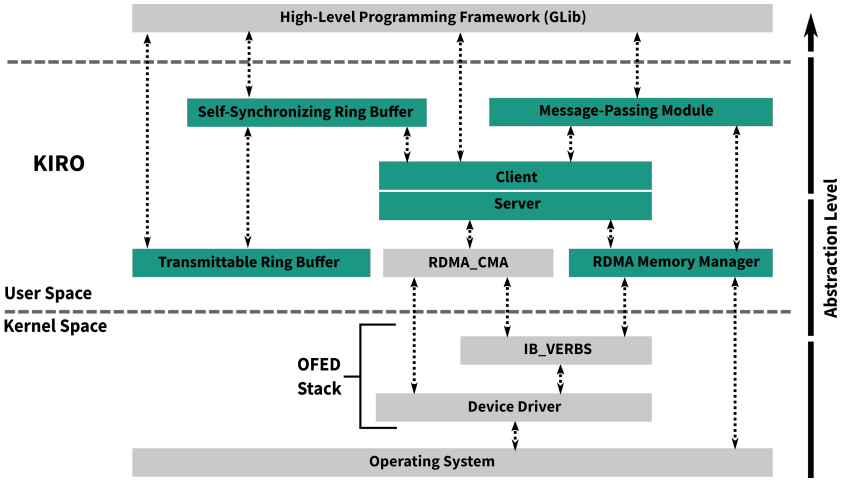

Figure 4.1: A visualization of the software stack KIRO is based upon, as well as the functional units that KIRO is comprised of, ordered by abstraction from highest (top) to lowest (bottom). Pre-existing software is marked in grey. KIRO-specific software is marked in teal.

IB_VERBS is an open software provided by the *OpenFabrics Alliance (OFA)*. Members of the OFA provide their specific device drivers, as well as related software, in the context of the *OpenFabrics Enterprise Distribution (OFED)*. This software package provides a multitude of drivers and software to facilitate the adoption and development of RDMA data transfer communication and storage technologies. As part of this OFED software stack, Mellanox distributes a basic programming interface for its InfiniBand protocol, the aforementioned *InfiniBand Verbs (IB_VERBS)*. It provides low-level application calls to their network adapters. On top of this low-level interface sits the *RDMA Communication Manager (rdma_cma)*. The rdma_cma abstracts the device-specific interface into a generalized RDMA-centric one that is agnostic of the actual RDMA hardware being used. The rdma_cma interface provides some convenient high-level functions that bundle some of the low-level ib_verbs calls into logical blocks. However, it still requires the user to execute the full RDMA connection setup sequence manually. KIRO sits on top of this rdma_cma interface and aims to provide a high level of abstraction, similar to the behaviour

of standard Unix Sockets. KIRO is written using the *GLib* development framework, which aims to emulate object oriented programming patterns in a pure C environment. This framework has been chosen to stay consistent with other frameworks that have previously been developed by our working group. GLib further provides some additional convenience features, such as simplified creation of function bindings for other programming languages, and a wealth of pre-defined function blocks. KIRO uses some of these blocks, such as mutexes, signals, timers and some data structures like dynamic arrays, lists and hashes.

> **Note:** KIRO relies upon the assumption that the InfiniBand network it is connected to uses the *IP-over-InfiniBand* extension in order to address and identify clients on the network through a conventional IPv4/IPv6 address. This extension is exclusively used for the initial connection setup between two machines. The actual data transfer between two peers on the network is still performed via native InfiniBand.

The KIRO software library itself is comprised of a set of functional units:

- RDMA Memory Manager
- Server
- Client
- Transmittable Ring Buffer
- Self-Synchronizing Buffer
- Messenger

RDMA Memory Manager

Most user-facing modules of KIRO rely on a shared back-end component which manages the creation, commission and cleanup of RDMA capable memory. It is not meant to be interacted with by the user directly, but rather is an internal interface for other KIRO components.
Whenever a KIRO component wants to create a memory segment for RDMA,

or make a third-party memory segment RDMA capable, it will use the interfacec provided by this RDMA Memory Manager. This includes the memories for the InfiniBand-specific protocol state abstractions (the so-called *Queues*) and all necessary steps to create, attach, configure and destroy these Queues of a logical InfiniBand connection.

To leverage the benefits of GPU-RDMA, specifically the Nvidia variation GPUDirect, a flag can be passed to the memory manager to instruct it to place payload memory on a GPU. Queue-Pairs and connection context memories, as well as descriptor memory will still reside in conventional system memory. When the flag is passed in, the memory management module will use the Nvidia GPU programming API (CUDA) to enumerate all available and compatible GPUs in the system, pick the first one in that list, and allocate a memory segment on that GPU. Once this allocation succeeds, the memory will then be run through the appropriate preparation steps, required to make that memory RDMA accessible. In the case of Nvidia, this consists of an RDMA specific CUDA API call, instructing the driver to enable the GPU to process Bus-Level memory access calls to its registered address range.
Once this memory segment has been created, the rest of the memory management chain can continue as normal. Resolving the Bus-Level address of the GPU memory segment is no different than how conventional system memory addresses are being resolved. This is thanks to the Nvidia GPUDirect Kernel Module, which injects RDMA enabled GPU memory into the systems page table, allowing to re-use the exact same system calls as for conventional system memory.
Additional information about the GPUDirect feature can be found in the Diploma Thesis that implemented and verified the technology [Rie15].

KIRO Server

The KIRO Server is the *passive* part of the software, and is run on the host computer that will function as a data *source*. An already existing memory region must be passed along to the server upon creation, together with appropriate size information, to register that memory for RDMA operations. The server

will then open a "listening" connection, and wait for any clients to connect to it. Once a connection request has been received, it will interchange all necessary information to access the reserved memory region with the newly connected client. The client is then free to perform RDMA operations on that memory region, without further involvement of the KIRO server. The listening connection will stay open for any further clients that wish to connect to the server. The KIRO server can support an arbitrary number of connected clients. It will monitor the connection status of each connected client, and inform each client about changes in the reserved memory region, should the user chose to relocate or resize the memory. The maximum available bandwidth for RDMA operations is dependent on the number of connected clients. Clients that concurrently want to perform RDMA operations on the server's memory will have to wait their turn, effectively reducing the throughput of the connection by a factor of the number of concurrent requests.

KIRO Client

The KIRO Client is the *active* part of the software, and is run on the *destination* computer. Its job is to connect to a KIRO Server on the network and RDMA-Read the data that is provided by that server into its own local system memory. To do so, it will first send a connection request to a KIRO Server with an address provided by the user. It will then handle the necessary handshake and information exchange with the server, including receiving the RDMA access information for the memory region provided by the server. It tries to reserve a memory region of identical size in the local system memory as a destination to copy the data from the server to. If that connection setup succeeds, and a sufficiently sized memory region could be reserved, the client can then be used to *fetch* data from the server, overwriting the local reserved memory region. It provides interfaces to either copy the entire remote memory content, or only specific parts of it. In case of the server changing any information about the provided memory region, the client will automatically re-negotiate the necessary RDMA information and tries to reserve a new suitable memory region in local memory. Once the user calls the fetch function the next time, the client will automatically switch to that newly reserved memory region

and free up the old region. It is the users responsibility to either copy the received data into a separate memory region for further processing, or refrain from fetching new data from the server until all operations are complete.

Transmittable Ring Buffer

This module is an implementation of a ring buffer, that holds self-containing information about its current iteration status within its own memory. It is specifically designed so that the raw data which represents the contents of the ring buffer can be transmitted across a network and re-introduced (we call this "adopting") into a local instance of a ring buffer without loss of coherence. It is intended to be instantiated on a KIRO Server and used to ensure that no incoherent data that is in the middle of being recorded is being read by the connected clients. Analogously, a KIRO client is supposed to instantiate a local transmittable ring buffer as well to "adopt" the data it fetches from the server into, in order to restore coherence. The user can decide the number of cells in the ring buffer, together with the size of those cells. The size of all cells is identical and may not change during normal operation. The ring buffer provides a user interface to retrieve data from the ring buffer, or to add a new element to the ring buffer. It is the users responsibility to ensure that only appropriately sized data is added to the ring buffer. Once the buffer is full (e.g.: all cells have been filled) it will automatically wrap back to the beginning of the buffer and start to overwrite the oldest cell available upon receiving new data.

Self-Synchronizing Buffer

A KIRO Self-Synchronizing buffer (KSB) is a wrapper around the previously mentioned Server, Client and Transmittable Ring Buffer classes. Its purpose is to keep a region of memory on a local machine consistently and automatically updated with data from a remote KSB. Whenever the remote KSB updates its data, the local KSB will automatically fetch and incorporate that data.
In order to accomplish this, a KSB will create internal instances of KIRO

Server, Client and Transmittable Ring Buffer, depending on its operation mode, and will constantly monitor the *header* information in the remote Ring Buffer of the KSB it is connected to, automatically fetching the most recently added element of the remote Ring Buffer into its own memory.

From a user-perspective, a KSB manages only a single memory block. (e.g.: memory for one individual frame of a camera.) The KSB provides an interface to manage the data of this local element. In reality, as KSB will create a Transmittable Ring Buffer of appropriate size in order to provide *Triple-Buffering* for the managed memory region. This is chosen to ensure that a connected KSB will always be able to read a fully coherent element from the Transmittable Ring Buffer, without running the risk of fetching an element that is either not finished being written or has already partially been overwritten. The software interface of the KSB is designed to suggest that only a single memory block is automatically being synchronized from the remote server.

A KSB can be configured to be either "serving" or "cloning". When a KSB is *serving*, it will create an internal KIRO server and then register the memory of its internal Ring Buffer for RDMA access via that server and listen for incoming connections.
When a KSB is *Cloning* it will create an internal KIRO client and tries to connect to a remote KSB with an address provided by the user. After successful connection, it will create a clone of the Transmittable Ring Buffer provided by the remote KSB. If this connection setup is successful, the KSB will then continuously fetch specifically only the data from the remote KSB that is required to identify whether a new element has been added to its internal ring buffer. Upon detection of a new element, it will only fetch that specific element from the remote ring buffer, embedding it into its own local buffer. A *cloning* KSB provides interfaces to register function callbacks for whenever a new update has occurred, and provides functionality to temporarily stop automatic updating (freezing) or resume automatic updating (thawing). It is a users responsibility to either copy the data of the most current element into a safe memory location for further processing, or to freeze the KSB while data is being processed, in order to prevent the current element from automatically being overwritten.

KIRO Messenger

The KIRO Messenger is an extension and combination of both KIRO Server and Client. Its purpose is to RDMA-transfer an arbitrarily sized block of memory in any direction, between two connected messenger. In contrast, the conventional KIRO Server and Client combination will always transfer a fixed-sized memory block from Server to Client. Messengers can transfer dynamic blocks of memory of any size in both directions. It is not possible to connect to a KIRO messenger by using a normal KIRO client.
Once a KIRO messenger has been instantiated, it will automatically open a listening connection and wait for any peers that want to connect to it. Upon receiving a connection request, the two peers will allocate small memory regions for each other (postboxes) and exchange RDMA information for these regions with each other, in order to send unannounced control flow messages to each other. Each connected peer will be assigned a unique ID that is only valid in the context of the local KIRO messenger.
The client can then provide the messenger with a pre-existing memory region and the ID of a peer that the memory is to be transferred to. The messenger will first prepare the memory for RDMA access. It will then notify that peer about its intent to transfer data to it by sending the RDMA access and size information to the peer. The peer is then expected to allocate an appropriately sized memory region and use the received RDMA information to read the remote data into the newly create local memory. Afterwards, a request completion message will be sent to the peer, completing the transfer.
Like the Self-Synchronizing Buffer, a KIRO messenger provides interfaces to register function callbacks in order to be notified about received messages and transfer completion, as well as interfaces to perform “blocking” send operations (meaning, the function call will not return until the send operation has finished).

Code Examples

Listing 4.1: Code example for setting up a KIRO Server

```
#include <stdio.h>
#include <kiro-server.h>

int main (int argc, int[] argv) {
    //Create KIRO Server
    KiroServer *server = kiro_server_new();

    //Create memory segment
    int sizeBytes = 1024 * 1024;
    void *mem = malloc(sizeBytes);
    if (mem == null) {
        printf("Failed to allocate memory for the server!\n
            ");
        goto end;
    }

    //Start the server
    if (0 > kiro_server_start(server, "192.168.11.61", "
        60010", mem, sizeBytes) {
        printf("Failed to launch server!\n");
        goto end;
    }

    /*  <Program Main Loop Here>  */

end:
    kiro_server_free(server);
    return 0;
}
```

Listing 4.2: Code example for setting up and connecting a KIRO Client

```
#include <stdio.h>
#include <kiro-client.h>

int main (int argc, int[] argv) {
    //Create KIRO Client
    KiroClient *client = kiro_client_new();

    //Connect client to server
    if (0 > kiro_client_connect(cient, "192.168.11.61", "
        60010") {
        printf("Failed to connect to server!\n");
        goto end;
    }

    //Get memory segment from server
    kiro_client_sync(client);

    //Access the data
    int bytesData = kiro_client_get_memory_size(client);
    void *mem = kiro_client_get_memory(client);


    /*  <Program Main Loop Here>  */

end:
    kiro_client_free(client);
    return 0;
}
```

4.2 KIRO Performance Measurements

To illustrate the performance of KIRO, a series of measurements were performed. These measurements all used the same Nvidia Mellanox Connect-X3 QFP network interface adapter in native InfiniBand protocol mode on both KIRO server and client. The software was running on 2 separate PCs, connected directly to each other without a network switch.
Three measurements were taken: *Throughput per payload size*, *KIRO protocol overhead latency* and *RDMA invocation latency*.

Throughput per payload size

Throughput was measured for varying payload sizes. Payloads were scaled from 1 byte to 1 Gigabyte in steps of × 10 each. The presented values are averaged over 1000 individual measurements per payload size. The results are shown in figure 4.2.

KIRO Protocol Overhead

When KIRO establishes a connection or when other control-flow information are being exchanged (such as Ping requests), no RDMA operations take place. Instead, KIRO server and client exchange common *datagrams* that embed the control-flow information as "immediate" values into the transmission. As these messages have to traverse the operating system network stack to be processed by the KIRO software layer, they lose the latency benefit of RDMA and have generally higher processing latencies. To illustrate these latencies, KIRO has an integrated *Ping* functionality. A Ping sends a message from a KIRO client to a server, to which the server will immediately send an appropriate reply back. The client that initiated the Ping measures the elapsed time between sending the request and receiving the reply. This measurement has been repeated 40.000 times and the distribution of turnaround times is shown in figure 4.3

KIRO RDMA invocation latency

Once a persistent connection between KIRO client and server have been established, the client is free to perform RDMA Read operations on the memory provided by the server. This does not rely on the datagram communication that KIRO uses its control-flow (such as the Ping functionality in the previous measurement). Since pure RDMA operations do not need to traverse the operating system network stack, they have much lower completion latencies. However, there is still a systemic delay between invoking the software function that initiates an RDMA operation, and the moment the KIRO software has received a completion notification from the device driver. To illustrate the systemic latencies involved in reading memory from a KIRO server, a single RDMA read operation was measured. Times measure the difference between the invocation of the software function call to the KIRO Sync() function and the moment the function call returns (completes). The resulting latency distribution for 40.000 iterations is shown in figure 4.4.

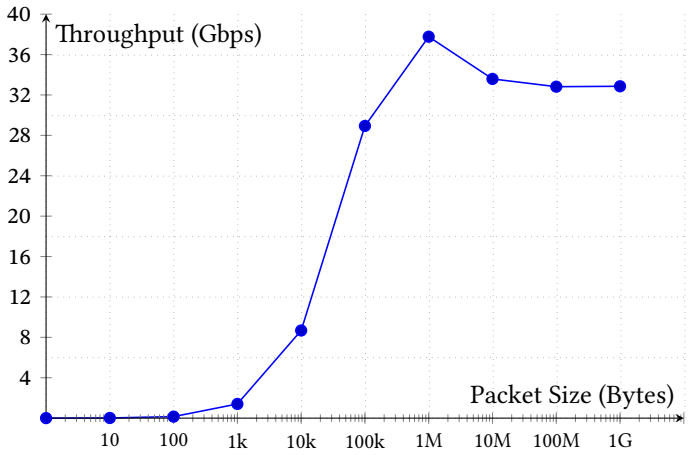

Figure 4.2: KIRO throughput performance in Gbps vs. payload size. The maximum bandwidth is 40 Gbps, based on the QFP ConnectX-3 NIC that has been used. Measurements were averaged over 1000 individual transmissions each.

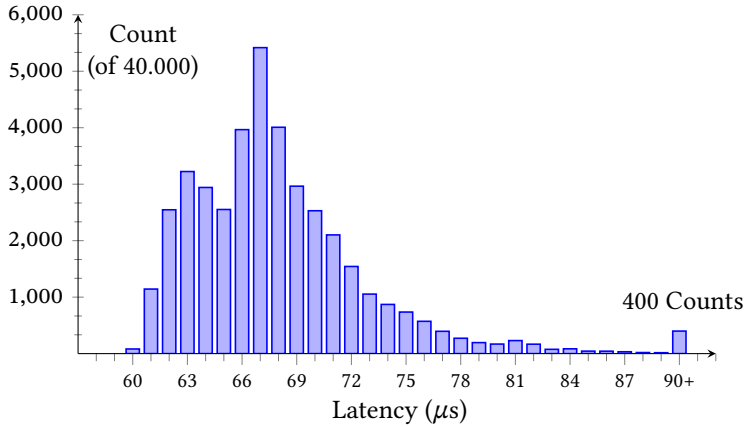

Figure 4.3: Latency histogram of 40.000 round-trip messages, measured using the KIRO "Ping" functionality. When using Ping, no RDMA operations are being conducted. KIRO Client and Server only send control-flow messages. This gives a representation of the KIRO protocol overhead latency. Measured using an Nvidia Mellanox QFP ConnectX-3 NIC.

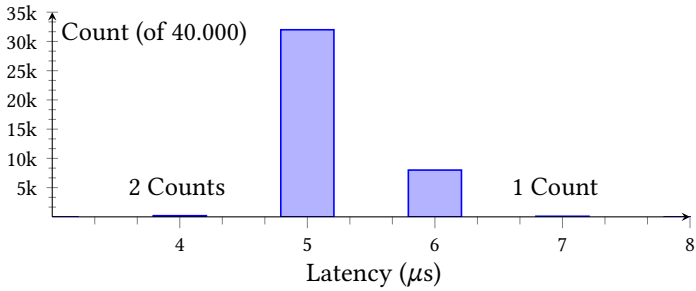

Figure 4.4: Latency histogram of 40.000 single-byte RDMA read operations, after a persistent KIRO connection has been established. Times include one RDMA Read operation, as well as the software-layer invocation latency of the KIRO Sync() function call. Measured using an Nvidia Mellanox QFP ConnectX-3 NIC.

4.3 Case Study: High-speed RDMA side-channel for control systems

To manage the complexity of modern scientific instrumentation, experiments usually deploy software systems that perform high-level management tasks for controlling and monitoring the involved pieces of equipment. In general, such systems are commonly described as *Control Systems*. There exist multiple control systems that have found broad adaptation in many installations, such as EPICS (Experimental Physics and Industrial Control System) [Dal91], or TANGO[Göt03]. The latter will get explained in greater detail later in this chapter. By design, most control systems are highly distributed and are intended to control multiple pieces of decentralized equipment through a common network. To this end, almost all control systems natively support some combination of Ethernet, UDP or TCP communication.

One typical application of such control systems is known as *Slow-Control*. Slow-control systems concern themselves with controlling and monitoring tasks that are not time-critical. In cases where timing is of particular importance, other proprietary systems are often developed on a case-by-case basis (e.g.: SODA, the time distribution system for the PANDA experiment [Kon09]).
In most cases, running such systems on top of a conventional Gigabit Ethernet network is sufficiently performant for their purposes. However, with the increase in data rates of modern scientific equipment, corner cases emerge in which even slow-control systems need to operate at higher data-rates and lower communication latencies than their native network implementations can provide.
In this section, one of such cases will be illustrated, at the example of an X-Ray Tomography installation at the KARA synchrotron light-source. I will show how KIRO was used to implement a high-performance side-channel for the TANGO control system.

4.3.1 X-Ray Imaging at the IMAGE beamline of KARA

X-Rays, or *Röntgen radiation*, as it is also known in many different languages, are an energetic form of electromagnetic radiation, ranging from wavelengths of 10 nm to 10 pm and energies of up to 124 keV (See figure 4.5). Due to their high energies, X-rays can penetrate object and materials whereas visible light can not. Depending on the atomic composition of a material, X-rays passing through the material may be partially absorbed or deflected, allowing for the discrimination of those materials based on the measurements of remaining radiation after passing the object. Ever since their discovery at the end of the 19th century, X-rays have been used in medical applications for the visualization of internal features of objects, such as bones in the human body. Nowadays, X-rays are not only used for medical applications, but for a wide variety of non-destructive investigation of the internal features of objects and materials, from material science, to engineering or security applications at airports.

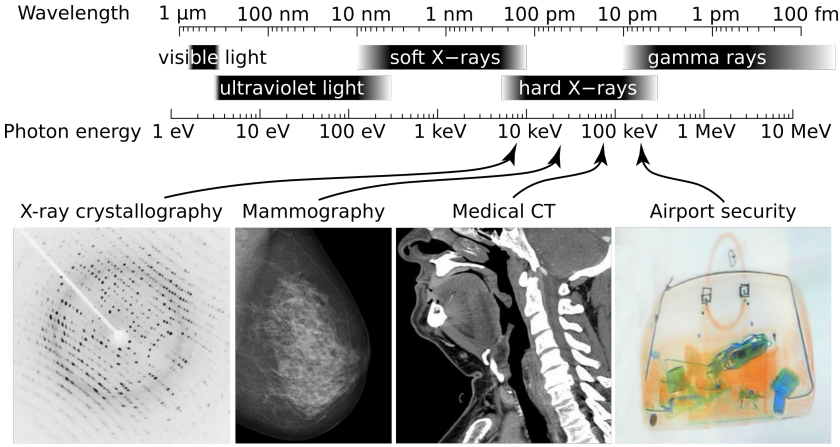

Figure 4.5: Visualization of some different wavelengths and energies of the electromagnetic spectrum, and some typical applications of X-rays at their respective wavelengths. [b]

[b] By 'Ulflund', Wikimedia Commons, licensed under CC BY-SA 3.0 (https://creativecommons.org/licenses/by-sa/3.0/). The work is being used unaltered.

KIT operates its own synchrotron light source: the *KArlsruhe Research Accelerator (KARA)*. KARA is a circular electron accelerator that uses a *cyclotron* to accelerate electron bunches, which are then injected into a booster and storage ring. The circulating electrons emit different forms of electromagnetic radiation (simply speaking: Light) of differing energies and wavelengths. The light is generated via *Bremsstrahlung* and emitted into individual sections and outlets in the KARA facility (so-called *beamlines*) that are arranged tangentially around the ring. KARA is capable of emitting light from the visible (infrared) spectrum, all the way up to X-rays, and the provided beams are used for a multitude of different scientific applications.
One of these applications uses the coherent X-Rays at KARA's *IMAGE and TOPOTOMO* beamlines for the investigation of the physiology of small insects ([San14],[Kam11]). The investigation is performed through X-ray *tomography* and/or *laminography*. In these methods, a sample is placed in the center of the X-ray beam on a spinning platform. X-rays are sent through the sample either perpendicularly, in the case of tomography, or at an angle, in the case of laminography. The attenuated light after passing through the sample is visualized by a scintillator crystal before being recorded by a camera. The sample is spun around one axis during the recording in order to generate projections from a number of different angles. These projections are then used to mathematically reconstruct a volumetric 3D tomogram of the sample[Bro75].

The UFO Ecosystem

In the conventional process of X-Ray imaging at the KARA facility, the recording and reconstruction of a single tomogram could take up to 1.5 hours. This is in large parts due to inefficient CPU-based processing of the recorded datasets and slow data links between camera station and processing system. Furthermore, most commercial high-speed cameras will not transfer recorded frames directly, but rather buffer frames in an internal memory. These frames need to be retrieved from the camera in a separate step, making the whole system real-time incapable. In cases when recording parameters have been chosen incorrectly or errors in the setup have been made, it will only become apparent hours after the recording has already finished. With the scarcity

of available operation time at a beamline, such mistakes can be very costly. In order to streamline the recording process and increase turnaround time of tomographic reconstruction, a framework for Ultra fast X-ray tomography with Feedback control loops and Online reconstruction (UFO) was proposed[Kop16]. This framework aims to bring a substantial speed-up to the conventional workflow of tomographic imaging by adding high-speed cameras with continuous streaming capabilities, parallel GPU computing, fast interconnects and a modular control system into the workflow. One of the core novelties of the UFO system is the concept of feeding the image results from a fast tomographic reconstruction on GPUs to the control system in soft real-time. This forms a feedback control loop with which recording parameters of the imaging process can be adjusted dynamically at runtime. A high-level abstraction of the UFO processing chain is shown in figure 4.6.

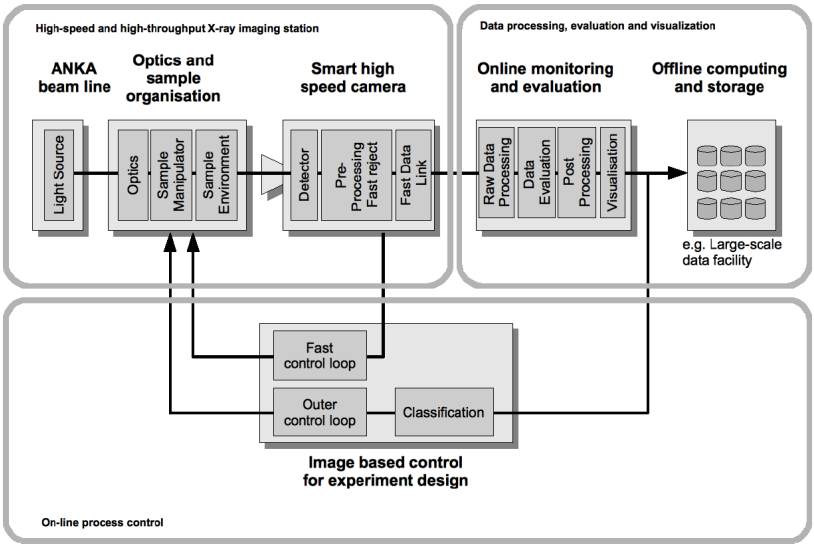

Figure 4.6: Visualization of the UFO Processing Chain. Shown are the individual processing steps from Light-Source all the way to final storage, including the on-line processing parts that form a feedback control loop. [a]

[a] Graphic courtesy of Max Riechelmann

When comparing the intent of the UFO Framework with the proposed blueprint of modern high-performance DAQ as shown in chapter 2, figure 2.1, we can already begin to see similarities.
The detector used for the data chain is a custom built camera with high temporal and spacial resolution of up to 20 megapixels and up to 5000 frames per second, producing data rates of up to 8 Gigabytes per second [Ste18]. Actuators to manipulate the experiment are present in the form of the experimental setup, holding the measurement sample, rotating it along one axis to produce the required projections. The data recorded by the high-speed camera is buffered on the beam-line-computer, to which the camera is attached to, serving as a form of data concentrator, before data is being forwarded to a computing stage for reconstruction. The UFO research group had already identified distributed and parallel GPU computing as their desired computing architecture, and developed a software system which manages the distribution, processing and collection of data onto such a distributed system [Vog12]. The information generated by this processing stage was expressly intended to be used in a feedback control loop, as indicated in the main title of the project "[...] with Feedback control loops [...]". Further more, fast data links from the processing stage to the storage cluster had already been provided in the form of an InfiniBand fabric. However, this InfiniBand fabric was configured to run in Ethernet mode, with TCP as underlying transport in order to utilize a distributed storage communication framework called iSER (iSCSI Extension for RDMA), which in turn relies on the iWARP protocol introduced in chapter 2.1. The entirety of the setup was controlled by two software control systems: *Concert* and *TANGO*. *Concert* [Vog13], which orchestrates the high-level procedures of the experiment and interfaces with *TANGO* [Göt03], which is used to control electronic equipment at the beamline. TANGO is one of the major control systems used at the KARA facility (and in the synchrotron community in general). The software system stack of UFO is shown in figure 4.7.

One of the remaining bottlenecks of the system was the data link between the acquisition PC of the high-speed camera and the control system and processing stage. The up-to 8 Gigabytes per second of data produced by the camera could not be transported by the available 1 Gbps Ethernet fabric at the facility.

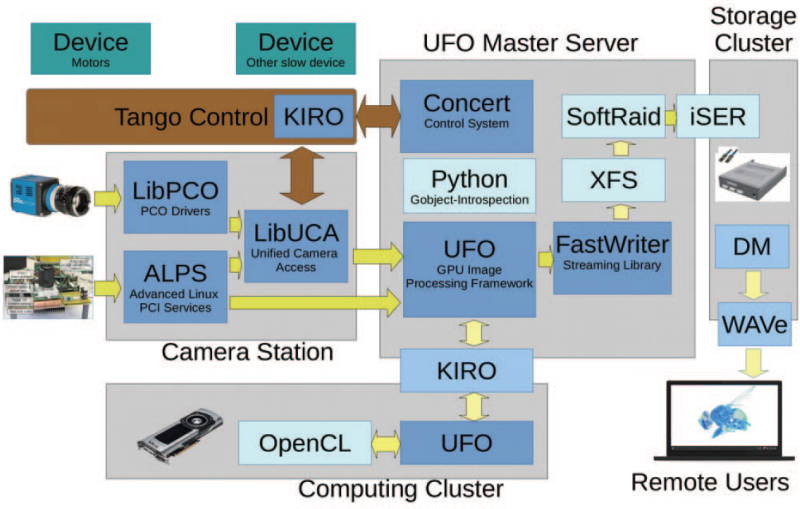

Figure 4.7: The individual components and intercommunications of the UFO software stack. The brown box at the top, containing TANGO and KIRO, are the software elements that will be introduced and discussed during this section of the thesis. (As seen in: [Kop16])

However, InfiniBand network adapters were available and already in use for connecting the processing stage with the storage cluster. Because of this, it was decided to implement another InfiniBand link between the camera PC and the computing stage. Due to the reliance of the project on the TANGO control system to control data flow and processing, a solution had to be found as to how to implement an InfiniBand data-channel into the TANGO environment.

TANGO, the control system

TANGO[Göt03] is a control system which is aimed to provide network access and control to remote hardware and instrumentation. Its primary applications are large scientific light-source installations, such as synchrotrons and lasers, and it is actively applied at multiple facilities, mainly in Europe, but also worldwide.

TANGO uses a *remote procedure call* paradigm to model remote devices. This is achieved through the adaptation of the Common Object Request Broker Architecuter (CORBA) standard. Specifically, by integrating the *omniORB* implementation of the CORBA standard.
Since TANGO builds on top of the CORBA paradigm, it works in much the same way. It defines the notions of "Devices" which encapsulate the local hardware specific implementation, or the remote user interface implementation respectively. In addition, TANGO also introduces a database component, which is used to manage the distribution of device implementations, as well as providing persistence for device parameters and event logging.

The CORBA standard

The CORBA standard defines a software architecture design which provides functionality to create and interact with platform independent abstract device descriptions over a network. It achieves this by demanding the interface to any hardware which should be controlled through CORBA must use a strict interface definition language (IDL) to abstractly describe the functions and properties of the hardware. This IDL code will then be translated by the CORBA tool chain into an "intermediary" software layer, which binds the provided abstract interface descriptions to a CORBA compatible implementation on one end, and a "skeleton" implementation of the same interfaces in any programming language of the users choice on the other end. An outline of this scheme can be seen in figure 4.8.

CORBA supports a large set of target programming languages, such as C, C++, Java, Python and many more. Once this intermediary layer has been compiled, the provider of the hardware device can then extend the skeleton implementation of the device interface with the actual hardware specific implementation. The other end of the intermediary layer serves as a "plug-in" to a so-called Object Request Broker (ORB) whose functionality is to manage the object live cycle of the device representation and manage the translation of the CORBA specific interface implementation to a network interface. One or many of such ORBs can then be exposed to a network.

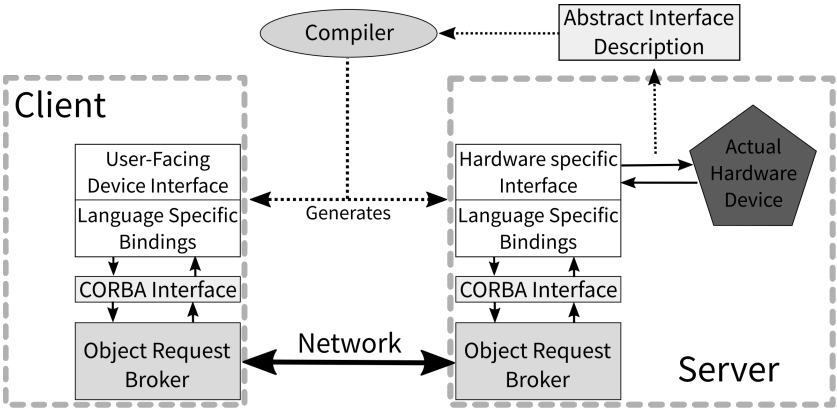

Figure 4.8: The CORBA software paradigm. The interface to an actual hardware device gets transcribed in the form of an interface description language and then compiled into a local and a remote representation. These binding layers then attach to the CORBA transport implementation and communicate with each other through an Object Request Broker.

Simultaneously, the IDL complier also generates a remote representation of the defined interfaces. The binding to an ORB works in much the same way as the hardware specific local implementation, but rather than providing a skeleton implementation on the other end, it instead generates a functional user-facing software interface. When the user interacts with this interface, the interactions with the interface get transcribed onto the CORBA interface through the binding layer and are relayed by the ORB to the respective remote device on the network. There, its ORB will receive the relayed interactions and translate them to the hardware specific interface through its respective binding layer. Any output generated by the hardware specific implementation will then be passed back through the same software chain in reverse.

4.3.2 State of the System and Implementation Considerations

The high-speed camera used for the UFO project can provide up to 5000 frames per second at 512x512 pixels of resolution at 8 bit color depth. This means

a single frame must be acquired and transmitted to the control system and computing station in less than *200 μs* to prevent frames from piling up. This serves as the lower latency bound of any proposed solution. Additionally, we are aiming for a data rate of 8 Gigabytes per second, or *64 Gbps*, in order to be able to operate the camera at its maximum performance.
To determine current state of the available network at the facility, we ran preliminary test measurements with data blocks of 265 kilobyte each (data size of one 512x512 pixels frame at 8 bit color depth), totalling at 1 Gigabyte per transmission.
The 1 Gbps Ethernet network at the facility is insufficient for either the required 64 Gbps throughput, but also failed to meet the required 200 μs latency constraint by an entire order of magnitude. Depending on the amount of clients that are simultaneously connected to the camera server, Ethernet performance would peak out at 750 Mbps for an individual client, scaling down as more clients connect. While latencies stayed comparatively consistently within a range of 84 to 86 ms. These results can be seen in the diagonally striped bars and dotted line in figure 4.9.

Furthermore, we evaluated the performance of operating the InfiniBand connection of the available 32 Gbps adapter with the TCP-over-InfiniBand (TCPoIB) protocol extension, to remain compatible to TANGO's native network implementation.
Measurements showed that the TCPoIB performance was considerably lower than we would have expected, in comparison to the native InfiniBand performance of KIRO had demonstrated. Using the 32 Gbps specified InfiniBand network adapter for testing, the single-client performance topped out at 7.4 Gbps, lowering slightly with each additional concurrent client connection. These connections showed large latency variations of 23 to 200 ms. This is most likely due to the additional load the TCPoIB extension puts onto a systems CPU, as it will have to handle the entirety of the TCP protocol overhead due to a lack of TCP offloading with InfiniBand networking adapters. Additionally, the comparatively small individual package sizes of just 265 kilobytes might have added excessive protocol overhead and could potentially be optimized by collecting multiple images into larger transmission blocks.

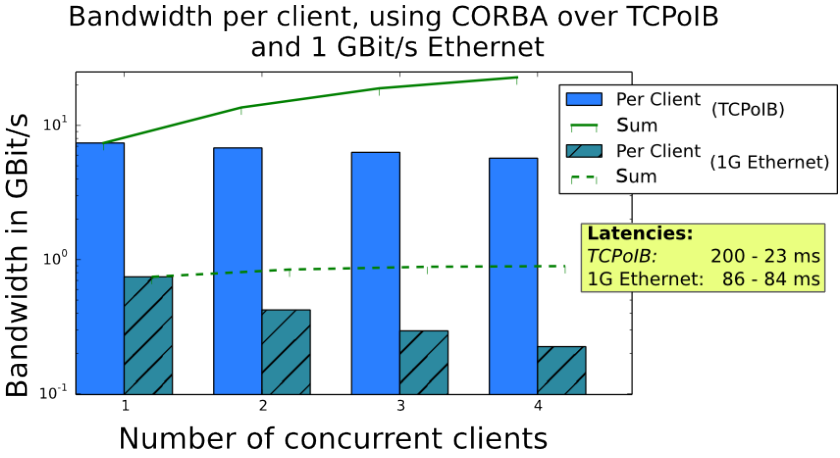

Figure 4.9: Shown are the measurement results of the throughput and latency ranges for 1 Gbps Ethernet and a 32 Gbps InfiniBand adapter, transporting TCP via the TCPoIB (TCP over InfiniBand) protocol extension. Transmitted datasets were 1 Gigabyte in size, consisting of individual 512x512 pixel images at 8 bit color depth, resulting in 265 Kilobyte blocks each.[a]

[a] Source: Own Publication [Dri14]

The results of the 32 Gbps TCPoIB InfiniBand adapter measurements are shown in the solid bars and line in figure 4.9.

In comparison, previous measurements of the performances of a native InfiniBand connection, using the KIRO library, strongly indicated that such a channel should be able to meet the required latency constraints. (Though the available 32 Gbps InfiniBand adapter would not be able to provide the required 64 Gbps throughput even under ideal circumstances. This problem could only be solved by a hardware upgrade.) In order to implement such a native InfiniBand channel into the TANGO ecosystem, we first had to identify on which layer of the CORBA/OmniORB stack the implementation should be made. As seen in figure 4.8, there are three potential layers to which InfiniBand support could be added.

Implementation Considerations

When implementing the InfiniBand/KIRO side-channel, we aimed to be as lightweight as possible with the implementation. It would have been, in principle, possible to add full InfiniBand support to either TANGO or its underlying CORBA/OmniORB implementation. However, this would have forced us to operate a separate personal code-base to the official TANGO project that we would have needed to continuously maintain and merge with any official upstream changes made by the TANGO consortium. We could also have contributed the implementation to the official TANGO main-line source tree, but the effort was deemed inappropriately large for the scope of the UFO project at that time.
However, there was another option of a piece of software that was already completely custom built for the UFO project: the software used to control the UFO camera through the TANGO control system.

To control the high-speed cameras used by the UFO project, a software had been created that aims to unify the separate driver interfaces of the cameras into a single high-level interface. The software is called *libUCA* (Unified Camera Control). It provides a plugin mechanism through which multiple differing driver standards for the separate cameras can be loaded and libUCA wraps these plugins into a unified interface to the rest of the UFO framework.
To control the cameras through the TANGO control system, libUCA and an appropriate OmniORB implementation had been created. The camera-facing part of this pattern (UCA Device) is run on the camera-PC and receives instructions from the TANGO network. The user-facing part (UCA Camera) takes instructions from the user and relays them through the TANGO network to the UCA Device, returning resulting data and camera frames. This schema is illustrated in figure 4.10.

As explained earlier in this section, we want to extend the pre-existing software infrastructure with a native InfiniBand channel to transport camera frames. To this end, we extended both UCA implementations for the camera-PC and the user-facing side with a KIRO server and a KIRO client module respectively. When the libUCA layer of the camera-PC UCA Device

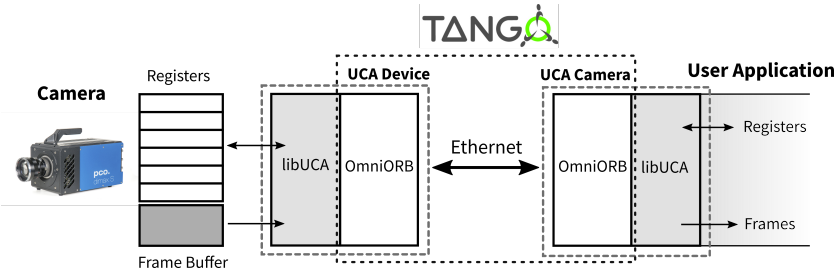

Figure 4.10: Schematic of the software stack for remote control of an UFO camera, using libUCA for both camera control and remote interface abstraction. TANGO is used to transport all of the camera's data and frames.

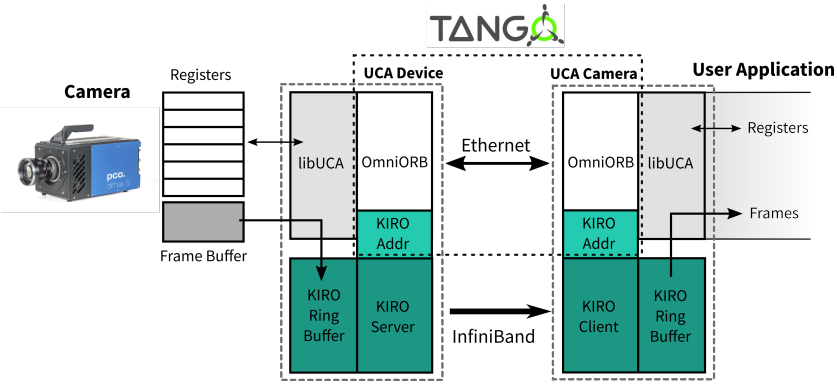

Figure 4.11: Schematic of the software stack for remote control of an UFO camera, using libUCA for both camera control and remote interface abstraction. TANGO is used to transport all of the camera's registers. While a KIRO Server and Client component has been added to the respective sides of the control to provide a native InfiniBand side channel. All recorded camera frames are transferred through the side channel and are transparently fed back into libUCA on the user-facing UCA Camera layer.

implementation collects frames from the camera, it no longer stores them in a separate internal buffer, but instead pushes them into a newly added KIRO Transmittable Ring Buffer. This ring buffer is getting served to a connected InfiniBand network by the KIRO Server module added to the UCA Device.
On the user-facing side, the UCA Camera implementation adds its own KIRO

Transmittable Ring Buffer. When libUCA wants to serve a camera frame to the user, it no longer requests the frame through the TANGO network, but instead takes the newest available frame from the newly added ring buffer. To keep the two ring buffers synchronized, the user-facing UCA Camera software uses a KIRO Client module to connect to the respective KIRO Server of the UCA Device on the camera-PC. The buffer is then continuously synchronizes from the server to the client using native InfiniBand RDMA data transfer. To connect the KIRO Server and KIRO Client of the UCA Device and UCA Camera with each other, both module's OmniORB implementations were extended with a new data field that stores the required KIRO addresses to setup the connections. This extended schema is illustrated in figure 4.11.

4.3.3 Side-Channel Benchmarks and Results

In this chappter, I demonstrated the possibility to augment a traditional slow-control environment, based on Ethernet, with an additional native high-performance InfiniBand data link, using the KIRO programming library. The approach was deliberately chosen in order to keep the necessary software changes at a minimum, while still providing the full benefit of the high-speed channel for camera data transfer. The conventional control schemes from the previous control system were kept fully in tact.
The resulting architecture shown in figure 4.11. In this way, it is possible to control a remote UCA camera through a TANGO network as if it was a locally attached camera. While the time-critical transfer of camera frames runs through an InfiniBand side channel that is transparent to the users. This approach is not limited to any specific form of detector and can be extended and exchanged for other types of readout hardware. Furthermore, as long as readout data from the detector is first stored in any form of PC main system memory, a similar architecture can be developed which picks up the data from system memory independently of the readout mechanisms and software libraries. Data can be integrated back into the rest of the data chain down the line by a symmetrical piece of software on the user-facing end of the chain.
In the demonstrated case, we were able to achieve a "close-to-native" performance of the InfiniBand channel, with 30 Gbps throughput out of an indicated

adapter bandwidth of 32 Gbps. Latencies ranged between 5 and 8 μs for each 265 kilobyte package. This package size was deliberately chosen, as it represents the expected data size of a single camera image, when operating in full 5000 frames per second mode, with image sizes of 512x512 pixels at 8 bit color depth. The native channel could be shown to clearly outperform a network based on InfiniBand, running a TCPoIB protocol extension. Measured latencies were sufficient to fulfill the 200 μs constraint for 5000 frames per second operation. However, the maximum data throughput of the camera, at 64 Gbps, was not able to be accommodated. Results are shown in figure 4.12.

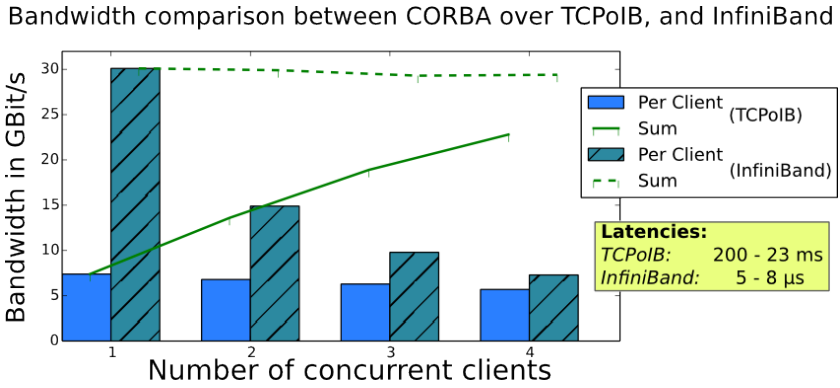

Figure 4.12: Shown are the measurement results of the throughput and latency ranges for 32 Gbps native InfiniBand link and a 32 Gbps InfiniBand link, transporting TCP via the TCPoIB (TCP over InfiniBand) protocol extension. Transmitted datasets were 1 Gigabyte in size, consisting of individual 512x512 pixel images at 8 bit color depth, resulting in 265 Kilobyte blocks each.[a]

[a] Source: Own Publication [Dri14]

The available network adapter would not have been able to provide the required 64 Gbps throughput even under ideal circumstances. It was expected that technology extrapolation and the implications of Moore's Law would eventually lead to fast enough single-component commodity adapter that will be able to provide the required throughput. Indeed, at the time of writing, InfiniBand adapters with 100 Gbps bandwidth are readily available.

It remains unclear if these advancements would have been able to properly outpace any potential increase in data rate from further improved detector equipment.
However, the fact that newer generations of InfiniBand equipment with higher key performance metrics did become available, and could potentially be used as full drop-in replacements of the old adapters with no additional requirements for software updates, provides support to the validity of the initial claim of this work. As discussed in chapter 1 of this thesis, the quality of commodity hardware to eventually become available with increased performance, while retaining full backwards compatibility, is one of the core concepts driving the decision to build DAQ specifically from such commodity components.

The results of this work have been published in the proceedings of the *Personal Computing and Particle Accelerator Conference (PCaPAC)* 2014 and can be found in [Dri14].

5 Low-Latency GPU computing system for commodity DAQ

In the previous chapters we selected and introduced key technologies for building high-performance commodity DAQ systems. With the help of those technologies, we now want to create a system that combines them into a high-performance computing node with lowest possible systemic latencies.
To provide a reference point for the performance of the resulting system, we are going to implement part of the new *Level-1 Track Trigger* concepts for the Compact Muon Solenoid (CMS) detector at the Large Hadron Collider (LHC). Specifically, we will implement the track-finding algorithm, based on a Hough-Transformation, using the GPGPU computing capabilities provided by our system.

5.1 System Design

To illustrate the performance potential of the commodity DAQ concept, we aim to build a demonstrator system that operates at lowest possible systemic latencies while providing high general purpose computing performance.
In accordance to the results presented in earlier chapters of this thesis, the chosen key technologies for this goal are (R)DMA capable interconnects and networks, multi- and many-core computing (specifically GPU computing) and FPGAs. From these, we aim to assemble a general purpose system that fits the DAQ blueprint shown in chapter 2, figure 2.1. This requires the following components:

- A **sampling** stage that reads data from the source and converts the data into the digital domain
- A **data concentrator** that receives, arranges and distributes the data from the sampling stage onto the available computing stages
- A **computing system** that receives data from the concentrator and processes the data according to the chosen algorithms
- A **feedback loop** that allows the computing stage or the concentrator to communicate directly with previous stages in order to influence data generation
- At least one type of **interconnect** in order to connect the aforementioned components to each other

First, we require a **sampling** stage. The task of this stage is to sample signals from the source and convert them into the digital domain. Unfortunately, this can not easily be provided as a generalized commodity component. Especially for highly advanced experiments, the front-end electronics and sensors for detectors vary drastically in design and purpose. This makes it difficult or even impossible to generalize their respective requirements. However, we can at least stipulate that whatever solutions may get chosen for the sampling stage should provide their data output through a commodity interconnect system, such as Ethernet, InfiniBand or PCIe. This guarantees that readout data can be ingested into the remainder of the commodity DAQ chain as early as possible.

Next, we are interested in the **data concentrator**. The purpose of this system will be to receive the sampled datasets and distribute them across the following DAQ stages. Depending on the detector and general experimental setup, a concentrator stage might need to aggregate readout data over time and from multiple subsystems. The layout of a concentrator is therefore heavily dependent on the measurement methodology and the data rates of the experiment. For example, system such as the CMS feature a large number of separate subsystems with thousands of individual readout channels. In such cases, a concentrator stage will need to be designed in the form of a mesh of multiple devices, in order to provide and operate an adequate amount of data transceivers.

Assuming the task of most concentrators is primarily to order and re-arrange data, an FPGA based solution is preferred, when aiming to operate at lowest possible latencies. PC systems arrange data in regularly sized chunks (today's system use mostly 64 bit words) and rely on external memory modules (RAM) to hold data, which are heavily optimized for sequential access. While FPGAs can be programmed and optimized to operate on arbitrarily sized chunks of data entirely in internal logic pipelines, removing the overheads associated with external memory module communications. Furthermore, FPGAs are well-suited for the integration of multiple data transceiver standards, both for inbound and outbound data streams and can thus easily serve as "translators" between proprietary readout data formats and commodity standard data communication interfaces.

One of the core components of the commodity DAQ concept are high-performance low-latency **computing systems**. To this end, we have chosen to use GPGPU computing. As shown in chapter 3, using DMA data transfer in conjunction with GPU computing opens up the possibility of providing the high computing performance of GPUs to a distributed system at lowest systemic communication latencies. This introduces a requirement for at least one PC host system into our DAQ design, featuring a PCIe bus, to which the required GPUs can be attached. The CPU of that PC is tasked with the configuration and control of its GPUs. As well as setting up the required (R)DMA communication channels prior to operation. The resulting PC system can be considered as a *general purpose* computing node. Data can be provided to such a node through a common network, such a InfiniBand, or directly from the concentrator stage through PCIe integration.
Depending on the required computing performance for a DAQ system, such a computing node could be replicated multiple times onto the same network. However, keep in mind that this will again introduce general *scalability* concerns to such a distributed system, as they have been mentioned in chapter 2.2.2 "Scalability of data access". Generally speaking, any distributed system design should always aim to keep the amount of intermediary memories on any given data path to a minimum, while simultaneously choosing the fastest available interconnect technologies with lowest systemic transfer latencies.

Furthermore, our system should also be capable of implementing **feedback loops**. Generally speaking, a feedback loop is formed whenever any stage of the DAQ system possesses the capability to communicate with any of the previous stages of the processing chain. In many cases, such a feedback loop benefits from the same considerations as those of general data communication of the DAQ as a whole. Assuming data communication between DAQ stages is built on top of a general purpose network, and no stages communicate with each other in any capacity other than that network, feedback loops can be formed freely, so long as the chosen type of communication on the network is bi-directional. This leads directly to the considerations of the chose **interconnects**. For intra-system communication, we did already establish PCIe as the chosen standard, as it is almost universally used by GPUs and most network adapters. For network communication, we rely on the results of chapter 2.1.2 and suggest an RDMA capable standard, such as InfiniBand or Ethernet with RoCE support.

5.2 GPGPU Specific latency optimization

Outside of the general design considerations, we have applied further optimizations to the overall systemic latencies of the GPU computing nodes, based on intrinsic properties of the GPGPU concept.

Spinning Kernel and CUDA Framework

GPU computing execution is organized into small executables that are loaded onto the GPU and get executed on it's processors. These executables are commonly called *Kernels*. The pre-loading and transfer of a kernel to the GPU is managed by it's driver, and therefore is running as software on the system's CPU. Loading and launching a kernel to GPU incurs overheads in the range of 20–50 μs. To reduce these overheads, we design our kernel in such a way, that it can be pre-loaded and executed on the GPU and will stay in continuous operation. It runs in a tight loop, in which it polls GPU memory for a "Start Flag", before beginning operation, and then returns to this polling stage, once

the dataset has been processed, waiting for new data to arrive. By doing this, we push the kernel launch delay out into a pre-preparation step, that can be performed before measurements even start. However, we identified a shortcoming of this method.
GPU memory is also equipped with multiple layers of caches. When accessing the same region of memory in a loop in order to observe the start flag, the GPU will continuously read "stale" data from the memory cache, unless the cache somehow gets flagged to be flushed and renewed. In most computing systems, there exist specific instructions in order to flush a cache. Unfortunately, in the case of GPU computing, there exists no such instruction in the *OpenCL* computing framework. However, it is possible to instruct Nvidia GPUs to flush their caches, using their vendor-specific *CUDA* framework. This limits us to only being able to use Nvidia-brand GPUs and their respective CUDA software framework.

GPU Process Control and Data Exchange

Once the spinning kernel has been launched on the GPU, we need to implement some mechanism that is capable of detecting when the GPU has finished processing of the current dataset, and when it is ready to receive new data to process. The same is true for the process of retrieving computation results from GPU memory.
In conventional DMA communication, most of these operations are performed by software running on the CPU. Such software could, for example, continuously poll and monitor appropriate flags in GPU memory to manage data exchange or rely on communication primitives provided by the GPU programming framework. However, doing so would inevitably introduce multiple additional memory layers to the data path, as the data being processed by the software being executed by the CPU needs to traverse CPU memory caches and system main memory before it can be accessed. To prevent this, we aim to control the signalling and exchange of data exclusively from within the active GPU kernel.
This introduces the challenge of needing to exchange DMA descriptors between communication peers. Into which memory location on a peer these

descriptors need to be written and how these descriptors need to be structured is highly dependent on the specific device and is normally handled by the device driver, running on the CPU. As we want to avoid communication between the GPU kernel and software running on CPU as much as possible, this means the GPU has no means of knowing when it is safe to communicate with its peer. Furthermore, it would require us to re-implement at least parts of the device driver logic in GPU.
To bypass these issues, we have instead opted to use a "High-Flex" FPGA DAQ platform, as introduced in chapter 3 of this thesis, as our means to exchange data. Since we have full control over the functionality of the FPGA, we can know its exact communication specifications and use those to program the GPU spinning kernel to communicate directly with the FPGA, using DMA on the FPGA's exposed *user bank registers*. This allowed us to keep device specific "driver" implementation inside the GPU to a minimum, since we were able to implement simple control patterns on the FPGA as we needed them.
Memory on the GPU gets reserved and pinned during the process initialization and the resulting descriptors are passed to the High-Flex board, so it can write data directly to GPU memory. In the same way, the address of the BAR of the High-Flex board is passed to the GPU kernel, so the GPU can write process control information to the boards registers. After this setup, the High-Flex board and the GPU can then communicate with each other directly, using DMA, without any further required assistance by software running on the CPU. This cuts out all of the CPU-side memory hierarchies from the system's critical data path.

The resulting system is shown in picture 5.1

GPU Memory Optimization

As discussed in chapter 2 of this thesis, the memory hierarchies of computing systems play an important role in the overall processing latencies of a system. In the case of GPGPU computing, the most important memory layers

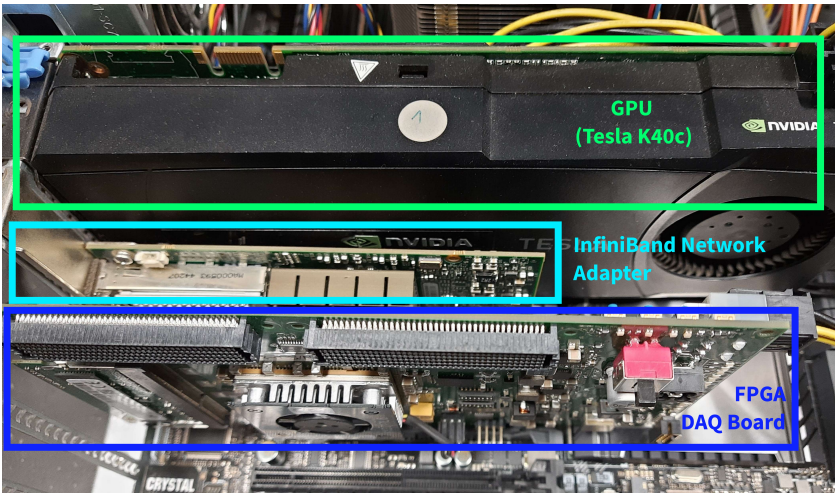

Figure 5.1: A PC setup featuring all the required components to realize a computing node as described in chapter 6.1.1.
The system features a GPU, a High-Flex FPGA board and an InfiniBand network adapter. All components communicate exclusively via DMA on the PCIe bus.

are *register memory, shared memory* and *global memory*. When designing any GPGPU algorithm, these criteria should be kept in mind:

- **Global Memory:** While global memory is the most abundant one, it is conventionally also the one with the slowest access times. However, when using GPU RDMA to transfer data into a GPU, there is no other choice for initial data transfer than to target global memory. Once data arrives in global memory, it needs to traverse GPU memory caches and can then be loaded into shared or register memory for further processing. The same is true for the opposite direction. Register and shared memory must first be written back into global memory, before then can be accessed via (R)DMA.
  There is a benefit to global memory, though, due to its large size. In many cases, it can be possible to transfer multiple datasets into main memory at once or in parallel to any ongoing computing process. When done ideally, this may enable a process to “mask” parts of, or

even the entire data transfer behind ongoing computing, in cases where multiple datasets get processed in sequence. This concept is commonly known as *Pipelining*.

- **Shared Memory:** Each of the Streaming Multiprocessors (SM) of a GPU is equipped with fast local memory, called *shared memory*. In some cases, this memory coincides with one of the cache layers of global memory. Any data structures that get accessed frequently should ideally be placed into shared memory to reduce access times. Due to its limited size, it is sometimes necessary to swap shared memory contents with other data from global memory.
- **Register Memory:** Registers are essentially the working-memory of any processor's Arithmetic Logic Unit (ALU). Almost all operations take operands from, and store results into registers. Also, data that gets fetched from higher memory layers must traverse register memory at some point, as the processor is the intermediary of most memory transfers. Especially for loops, these circumstances can provide optimization potential when arrays and operands can be pre-fetched into register memory. However, register memory is severely limited to just a few hundred datawords on most GPUs, making it difficult to fully utilize.

Note: Optimizing algorithms for memory locality is a challenging task that requires intimate knowledge of processor systems and computing systems. These concerns are not exclusive to GPUs, but apply to most computing systems. The presented hierarchy optimizations and considerations only scratch the surface of available techniques. Further detailed considerations are, however, out of the scope of this work. A detailed review of GPU architectures and related optimizations can be found in [Chi20].

5.3 Case Study: Low Latency Trigger

To investigate and illustrate the performance of the described DAQ systems, we aim to implement part of a track-trigger for the Compact Muon Solenoid (CMS) experiment. The DAQ requirements of the CMS after its Phase-2 upgrade ([And17]) are assumed to be among the most challenging DAQ requirements to date. Comparing the performance of the described commodity DAQ system against these requirement provides a suitable quantitative measure for the feasibility of the commodity DAQ concept as a whole.

5.3.1 The CMS and its L1-Track Trigger

The Large Hadron Collider (LHC) is a large particle collider at the European Organization for Nuclear Research (CERN). It is designed to accelerate and collide proton and heavy ion beams, currently reaching energies of up to 6.5 TeV (13 TeV total collision energy). The resulting collisions are observed by seven independent detectors, arranged around four beam crossing points, with each detector aiming to observe different phenomena. Their aim is to experimentally test and verify predictions from the field of particle physics, as well as potentially discovering new physics. The experiments conducted at the LHC are currently best know for the first experimental verification of the existence of a particle consistent with a prediction made from the standard model of physics: the Higgs Boson [ATL12][CMS12]

One of the detectors at the LHC is the Compact Muon Solenoid (CMS), shown in figure 5.2. It is a cylindrical, multi-layered general-purpose detector, which is equipped with a strong electromagnet (a solenoid) that produces a 3.8 Tesla magnetic field. The magnet is used to bend the trajectory of the particles created upon collision, in order to derive momentum and energy information from observing the curvature of their trajectories. The higher the energy of a particle, the less its trajectory will be bent by the magnetic field, and inversely, the lower the energy of a particle, the more its trajectory will be bent.

The innermost layer of the CMS, Layer 1, is the so-called *tracker*. It is itself comprised of multiple layeres of silicon sensors which detect the positions of

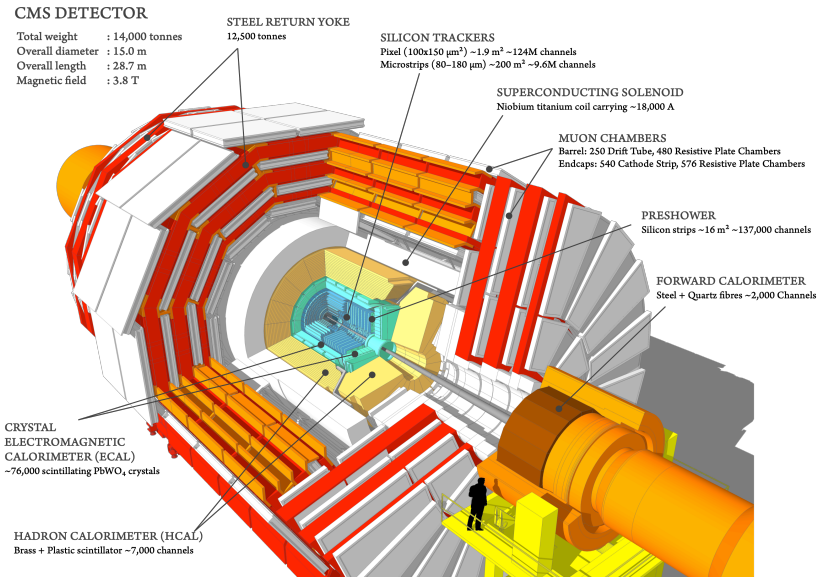

Figure 5.2: A cutaway diagram and scale model of the CMS. Illustrated and labeled are the different detector layers around the beam crossing point in the center of the detector. [b]

[b] By Tai Sakuma, CERN, licensed under CC BY-SA 4.0 (https://creativecommons.org/licenses/by-sa/4.0/). The work is being used unaltered.

electrons, muons and hadrons with high precision. There are 14 tracker layers in total, with the endcaps being equipped with an additional 15th layer. The silicon sensors on the four tracker layers closest to the impact point are designed as high-precision pixel sensors, while the remaining layers are designed as microstrip sensors. By measuring the trajectory of a particle across the individual tracker layers, it is possible to derive its momentum, when taking the magnetic field of the solenoid into account. The information provided by the tracker will be used extensively for the functionality of the track trigger demonstrated in this chapter.

The individual detector layers and modules of the CMS are read out through millions of individual data channels that get aggregated into separate events

of roughly 1 Megabyte in size each. Due to the rarity of the events the CMS aims to observe, the experiment needs to be operated at a very high event rate of 40 MHz. At 2 Megabytes raw data size per event this results in a raw data rate of approximately 80 Terabits per second. As explained in the introduction of this thesis, storing a data stream of that size under continuous operation is technically unfeasible. Because of this, the CMS experiment implements a *trigger* system ([CMS17]) which aims to discard any dataset which does not contain any events of interest before full readout and storage occurs. With the application of this trigger, the acceptance rate of the experiment is reduced down from 40 MHz to roughly 100 kHz, resulting in approximately 200 Gigabits per second output data rate.
The full trigger system is separated into two distinct steps. A *Level 1 Trigger (L1T)*, and a *High Level Trigger (HLT)*. The L1T makes a decision to discard or keep the date from a colission event, based on partial information provided from the calorimeters and muon chambers of the detector. Due to the length of the readout pipeline of the entire CMS, the L1T needs to provide its trigger decision within 3.5 μs of the collision occuring. Due to this very tight constraint, the current L1T systems and algorithms are developed entirely in dedicated hardware, such as FPGAs and ASICs. [Jei14]

After the Phase-2 upgrade, increased collision rates would lead to an increase in acceptance rate of the trigger. To keep the readout data rate at a manageable level without loss of efficiency of the observation, the trigger system needs to be upgraded in order to handle the increased pile-ups (PU). Multiple strategies have been proposed. Among these strategies is the introduction of an *L1* ***Track*** *Trigger*, which would process, for the first time, data from the silicon tracker subsystem of the CMS.
The strategy follows the idea of using tracker information to calculate "Track Candidates". These informations can then further be used to increase the effectiveness of the trigger decisions of the Muon and Calorimeter triggers, by adding additional discriminatory information.
In order to accommodate the additional latency introduced by this system, the CMS project aims to increase the pipeline, and therefore the timing constraint, to 12.5 μs. Of which 5 μs are intended for the track trigger (1 μs for data transfer and 4 μs for computing). Together with the 3.5 μs for the Muon

and Calorimeter trigger latencies, and 1 μs to relay the global trigger information to the readout electronics of the CMS, this sums up to 9.5 μs. Onto this, roughly 30% of safety margin are applied, to arrive at the aforementioned 12.5 μs total trigger latency.

Therefore, the timing constraint that we will compare our commodity DAQ system against will be the ***5 μs allotted to the track trigger stage.*** As these times include data transfer, the goal for out system will be to transfer data from the concentrator stage to the computing stage and perform one iteration of the chosen track trigger algorithm on GPU within the given 5 μs.

The function of the L1-Track-Trigger

The task of the L1-Track-Trigger is to analyze the data received from the CMS tracker and try to ascertain the presence of particles above a certain threshold energ. This is done by reconstructing particle trajectories through the tracker (we call these *tracks*) and derive information about the particle's energies from the track's curvature.
As mentioned in the introduction to CMS, one of the deciding features of the experiment is a strong magnetic field that influences the trajectories of the particles generated upon impact. A particle's trajectory through the detector will be affected by the magnetic field and become curved in accordance to its velocity away from the beam-crossing and the strength of the magnetic field. In the context of the CMS experiment, this velocity is described as the "Transverse Momentum" of a particle, denoted as p_T. In particular, the CMS experiment is interested in finding particles above a specific p_T threshold of 2–3 GeV/c (configurable). The data provided by the tracker to the track trigger is a list of the coordinates of detected particle hits, in reference to the detector geometry.

The tracker will be upgraded with new modules that can perform transverse momentum pre-filtering. This is done by putting two layers of pixel and/or strips in close proximity to each other and comparing the hit locations on both layers. When a hit is registered on the lower layer of the module, it generates an expected "window" for a subsequent hit on the layer above. The

size of this window controls the maximum permissible curvature of a particle track that is considered to be high-p_T. If another hit is registered within that window, a "stub" is created and will be passed on to the track trigger for processing. Essentially, a stub holds the approximate location information of the hits (limited by the sensor resolution) as well as the estimated p_T of the hit. This process is shown in figure 5.3

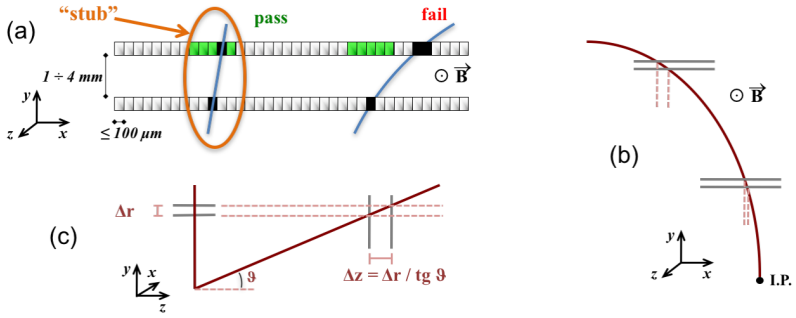

Figure 5.3: The upgraded CMS silicon tracker modules consist of double layers of either pixel-strip or strip-strip modules in close proximity to each other. Informations from neighboring layers are used in the formation of "stubs", based on permissible trajectory curvature. **(a)** shows the general concept of stub-building. The size of the neighboring acceptance window regulates the lower bound for permissible transverse momentum. Based on projection of tracks into further detector layers, the size of the acceptance window needs to be adjusted accordingly for barrel layers **(b)** and endcaps **(c)**. [a]

[a] Source: [Agg17]. Licensed under CC BY 3.0 (https://creativecommons.org/licenses/by/3.0/). The work is being used unaltered

This form of pre-filtering reduces the amount of data that needs to be processed by the track trigger, but it is insufficient on its own when wanting to derive an accurate curvature of a track. This is because the short track segments between two individual module layers will resemble something close to a straight line. In order to derive the appropriate curvature radius R of a track, it is necessary to observer its trajectory over multiple detector layers in time. However, since the tracker can only provide individual interaction points with

an estimated p_T (stubs), a challenge arises from how those individual stubs can be assembled and assigned to corresponding tracks. A visualization of the process is shown in figure 5.4

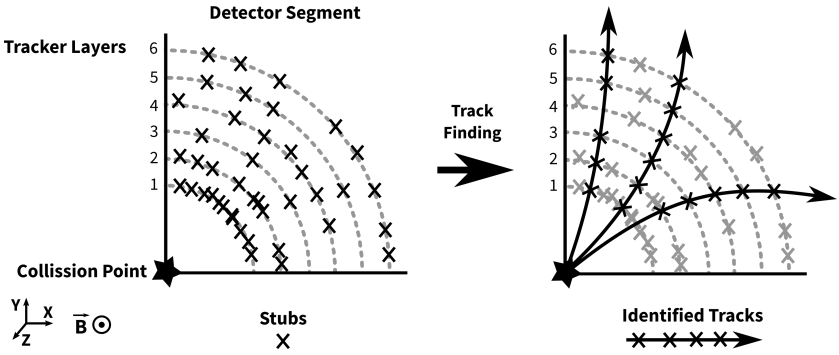

Figure 5.4: The process of identifying tracks from stubs generated by the silicon tracker. Shown is only a single segment of the detector. On the left, only stubs are known. After the track finding process, tracks and their corresponding stubs have been identified, shown on the right. At least 5 out of 6 layers need to participate to a track in order to be considered.

In order to solve this track-finding challenge, multiple approaches have been investigated. One approach matches stubs against a pattern data-bank formed from associative memory ([Ams17]). Another approach extrapolates potential follow-up stubs in outer tracker layers from the approximated p_T of a stub to form "Tracklets" ([Bar17]). These approaches leverage the separation into individual tracker layers and assemble an appropriate computing architecture. They operate on both, the predictions made by the previous layer, as well as their resepctive layer's hit information from the detector.

The proposed solution that we have selected as case study for the commodity DAQ system does not require the separation into individual layers. Instead it evaluates the complete set of stubs from an individual event as a whole (albeit split into multiple individual detector segments). This approach leverages a *Hough Transformation*, which transforms the angular detector coordinate

space of the stubs into a coordinate space in which certain geometric dependencies are easier to recognize ([Ams16]).

5.3.2 Track-Finding based on a Hough-Transformation

A *Hough Transformation* is a well-understood method of feature extraction in image processing systems. The method works by transforming conventional image features into a specific parameter-space in which certain features are more easily identified and distinguished. In particular, the transformation chosen for the L1 Track Trigger is interested in identifying circular features, as the tracks formed by particles in the detectors magnetic field describe helix paths, which project into circular paths when viewing them along the beam axis. The exact transformation is described in [Ams16]. The principal operation of the algorithm is centered around the concept, that the parameters of the helix paths can be sufficiently described by just two parameters. These two parameters are the production angle at the origin/the beam crossing (Φ) and a radius of the curvature of the path (R) which correlates to a transverse momentum (p_T) of the respective particle. The algorithm processes two points: an origin and the location of a stub in normalized detector coordinates. The coordinate normalization transforms absolute detector coordinates into coordinates relative to a specific *pivot* point of each detector segment. This makes it so the value ranges of all coordinates of each segment are identical and therefore simplifies algorithm design. From these two coordinates (pivot point and stub location) an infinite amount of potential circle parameters emerge that cross both of these points. All potential circle parameters are noted down in a so-called "Hough-Map" that is parametrized by the two values Φ and p_T, in which the given circle parameters will form straight lines. Due to limitations in the precision of stub locations (caused by the pitch of the pixels and strips) the Hough-Map is inherently separated into discrete bins. The size of these bins is actually an important parameter for the algorithm as it has direct impacts on the memory footprint of the map in device-memory, its runtime and most importantly, on the precision of the derived information. A simplified visualization of the process is shown in figure 5.5.

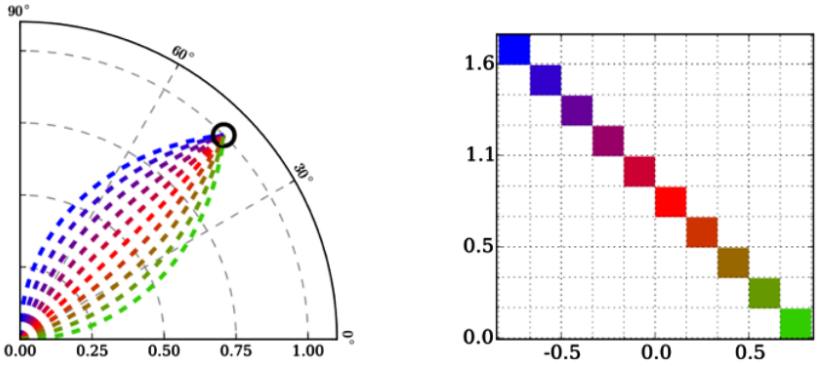

Figure 5.5: Left: Two points in regular space (origin and stub location) can be described by an infinite number of possible circles that will cross both points. **Right:** All of the possible circle parameters form a straight line in Hough parameter space, describing the angle of incident at the origin and the curvature of the circle. Note, that this graph is not drawn to scale. Axes have no meaningful units, as this graph was made purely to visualize the conceptual process of the transformation. [a]

[a] Graph courtesy of Hannes Mohr.

After all stubs have been processed, the corresponding lines in Hough parameter space will form intersections. These intersections abstractly describe sets of circular path parameters that are compatible with multiple individual stubs. In places where there are intersection clusters that are formed from stubs from at least 5 out of the 6 separate tracker layers, the corresponding set of circular path parameters is considered to be a *track candidate*. The algorithm gathers these track candidates and passes them on to the next trigger stage.
So the task of the algorithm is to find sets of stubs that most likely belong to one consistent track through the tracker layers, derive their momentum information and pass on a list of all identified tracks with their respective stubs and parameters to the global trigger system. From there, the derived information will be used to increase the fidelity of the trigger decisions made by the Calorimeter- and Muon-Trigger systems.

5.3.3 Hexagonal Hough Space

A common side effect of the way the chosen Hough Transformation is targeting a rectangular parameter-space, due to its evenly spaced binning in both dimensions, is the creation of so-called *Fake Track Candidates*. Due to the binning, the lines generated by the Hough Transformation will effectively cause "aliasing" when the transformed line is binned/rasterized into the hough map. This aliasing causes some neighboring cells to be voted into, even though a line only barely intersects with the cell. As a result, especially in areas where many lines intersect, clusters will often "bleed" into neighboring cells, giving them enough votes to be considered viable candidates, when in reality, they are merely an artifact of the parameter uncertainties of the Hough Space binning. Figure 5.6 shows an example dataset rasterized into a conventional rectangular Hough parameter space. It can clearly be seen how lines of similar slope close together will form many fake intersections, due to the aforementioned cell bleeding.

In order to alleviate some of these shortcomings, we developed a modified version of the Hough Transformation that leverages the floating-point computing capabilities of GPUs by transforming stubs not into a regularly spaced rectangular parameter space, but instead into a more complex *hexagonal paramter space* (See [Moh17]).
In a hexagonal parameter space, the resulting Hough-transformed lines can more accurately be represented. Furthermore, neighboring cells in the hexagonal space are equidistant to their respective centers, unlike in rectangular space, where their distances vary based on arrangement. At the same time, each hexagonal cell only has 6 distinct neighbors, compared to 8 neighbors in a rectangular grid, which reduces the worst-case cluster sizes. This leads to less cell bleeding while rasterizing a line into the hexagonal parameter space and thus forms less fake track candidates. The difference is shown in figure 5.7.

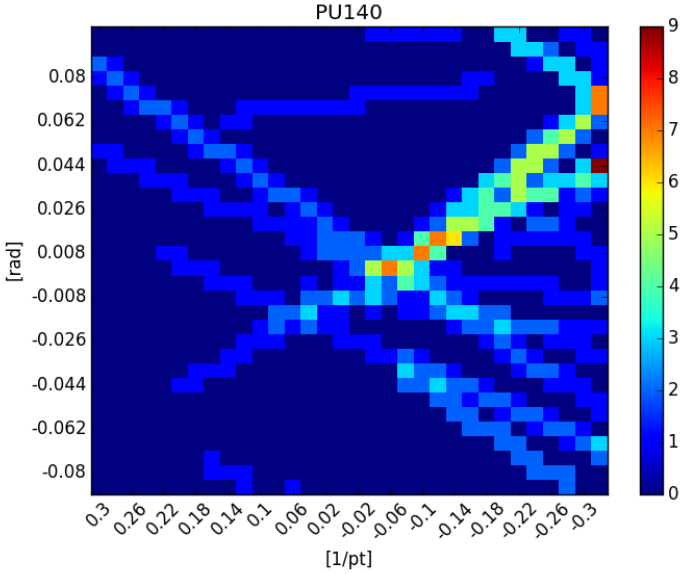

Figure 5.6: Shown is a Hough Transformation of the stub information of a single detector segment, from an exemplary event with a pileup of 140. In this case, some lines of similar slope rendered close to each other, causing many fake clusters from the aliasing caused by rasterizing the lines into the rectangular Hough parameter space. [a]

[a] Graph courtesy of Hannes Mohr.

5.3.4 Implementation details of the Track Trigger

As the primary active component of this test system, we will be using a 2013 *Nvidia Tesla K40c* GPU. It is a "professional grade" GPU that is primarily intended for computing tasks, rather than video processing, and was one of the first generations of Nvidia GPUs that fully supported GPUDirect GPU RDMA. It features 2880 computing cores, arranged over 15 *Streaming Multiprocessors (SMX)* operated at a 745 MHz clock rate, and is being promoted by Nvidia as reaching peak performances of roughly 5 TFlops/s at 32 bit Floating Point precision. It is equipped with 12 GB of GDDR5 memory.

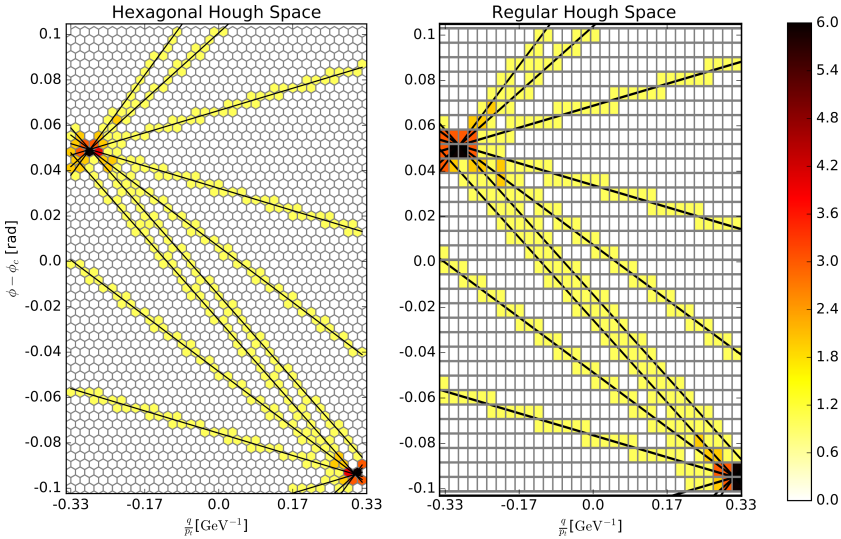

Figure 5.7: Comparison of hexagonal (left) and rectangular (right) parameter space of a Hough transformation. The hexagonal parameter space more accurately represents lines and leads to formation of tighter clusters, which reduces "bleed over" of clusters into neighboring cells. [a]

[a] Graph courtesy of Hannes Mohr.

In our specific case, we have a High-Flex FPGA platform connected on the same PCIe bus as the GPU. As described earlier in this chapter, we chose this option over using a conventional network adapter to optimize system complexity in regards to process control. Because we have this FPGA platform available, we imagine it could also double as part of the data concentrator subsystem. Regardless whether the FPGA will be part of the concentrator subsystem or serve merely as communication device with the rest of the network, it will in both cases be the entry point into the computing node that provides fully assembled datasets to the GPU. For the demonstrator setup, we have manually selected a number of test datasets, exported from the CMS simulation software suite ([Ban12]) and pre-loaded those datasets into the FPGAs internal memory. For our tests there is no conceptual difference between data

arriving from an external source or being already stored on the FPGA. This step is merely meant to reduce the complexity of the test system.

Overall, this results in a hypothetical DAQ system that conceptually sees the CMS software as the data source, the manual selection of test-datasets as sampling, the FPGA as concentrator, and the GPU as computing node. An overview over this system is shown in figure 5.9

GPU memory requirements and considerations

As mentioned in the discussion about GPGPU computing in chapter 2.2 of this thesis, a major aspect towards reducing computational runtime is the use of fastest available memories. For this, we need to be aware of the amount of storage our processing task will require.
In the case of our Hough transformation, we needed to consider two main aspects. The storage for the received stub information, as well as the storage for the Hough-map. We assumed an estimate of 160 stubs being generated per sector of the CMS detector, per event. Stubs are transferred in a compressed data format of 48 bit per stub. As the GPU we are using can only operate on data-words of 32 or 64 bit alignment, we decompress stub information into two 32 bit words. Giving us 160 x 2 x 32 bit = 10240 bits = 1280 bytes of storage for the input data of a single event.
We structure the storage for the resulting Hough-map as fallows: The largest size of map we support is 32 by 32 bins. Each bin needs to keep track of the individual stubs that might contribute into the cell after transformation. We arbitrarily assume a cell to never have more than 30 individual stubs contribute to it. To keep track of contributing stubs, we provide an array of 30 individual 32 bit words per cell. Furthermore, we add an additional 32 bit word to each cell to keep track of participating detector layers. One additional 32 bit word is added as a counter for the amount of participating stubs. This is so we can keep consistent runtime and do not need to count the resulting list of individual contributing stubs per each cell afterwards. Instead, we have the stub count readily available after processing. As a result, our Hough-map requires (2 + 30) × 32 × 32 × 32 bit = 1048576 bit = 131072 bytes of memory.

Neither the data for the stubs, nor the Hough-map can fit into register memory. Unfortunately, as the Tesla k40c that we used only offers 48 kilobytes of shared memory per thread block, the Hough-map does not fit into share memory either. The remaining memory would be main GPU memory. However, latency tests that we performed indicated main GPU memory access times to be in the magnitude of 300 clock cycles. At the GPUs clock frequency of 745 MHz, this worked out to be around 0.4 *µs per non-sequential access*. We expect multiple memory accesses *per stub*, which disqualifies GPU main memory due to its excessive access latencies.
We are therefore forced to separate the map into its individual 32 rows or columns and distribute the processing across 32 individual thread blocks. This is possible due to the "embarrassingly parallel" nature of the chosen Hough-transformation that is entirely independent on any outside parameters other than the raw stub input data. However, since the k40c only offers a total of 2880 threads, this means we can only process $2880 \div 32 = 90$ stubs per GPU per event in parallel. On a more positive side, this separation only requires us to use $(2 + 30) \times 32 \times 32$ bit = 32768 bit = 4096 bytes of shared memory per thread block. Meaning, it is possible to copy the entirety of the 160 stub input event data into shared memory alongside the individual Hough-map section, per thread block. This allows us to work almost exclusively in shared memory, at the cost of the significantly reduced throughput of the limited number of available threads per block.

Masking Data Transfer by Pipelining

Since preliminary testing results showed that the computing time exceeds the average data transfer time by about a factor of 2, and enough main graphics memory is available to hold multiple input datasets at once, we can implement a pipeline schema for our work execution. In a pipeline, we leverage the fact that data transfer into GPU memory can be performed in parallel to execution of a GPU kernel. This means, in cases where data can be provided fast enough by the sampling electronics, we can already start transferring the $n+1$ dataset into GPU memory, while dataset n is still being processed. In this case, we can perform data transfer and computation in parallel, rather than having these

steps be in sequence. Once the processing of the current dataset is complete, the GPU can assume that data for the next dataset can already be found in main memory. Therefore, the actual data transfer times only need to take into consideration the time it needs for the GPU kernel to confirm that a new dataset has indeed already arrived (polling).

The general execution order of our implementation, as well as a comparison between the sequential and pipelined approaches is shown in figure 5.8.

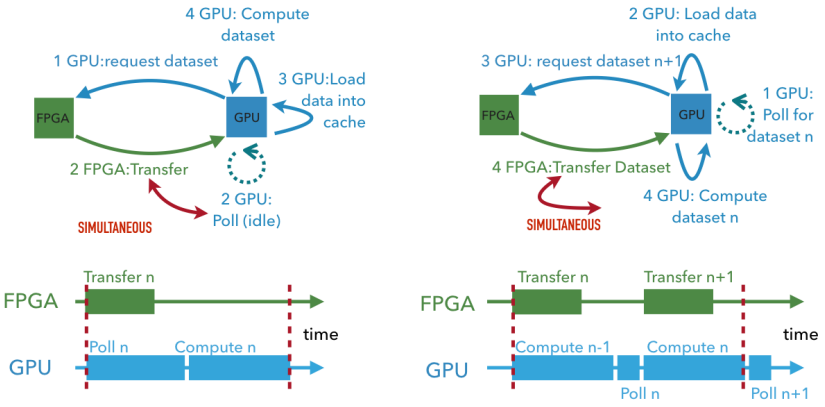

Figure 5.8: The schema of execution-order for the commodity GPU-based track trigger. **Left:** The conventional, linear execution order, in which data and computation are strictly sequential. **Right:** A pipelined approach, in which data transfer for the next dataset is "hidden" behind the computation time of the previous dataset and performed in parallel to ongoing computation. [a]

[a] Taken from own work: [Moh17]

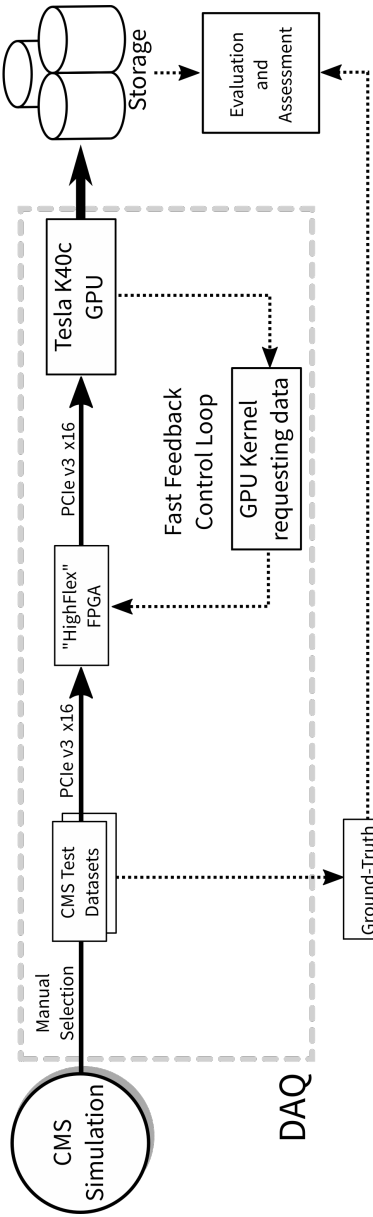

Figure 5.9: A schema of the commodity hardware CMS L1 Track Trigger test system. Experiment data is generated by a simulation software and datasets are manually exported and pre-loaded onto the FPGA, which effectively serves as data concentrator. The computing is done a by a Tesla K40c GPU which sends requests to the FPGA via DMA to receive the next dataset to process. Afterwards, results are dumped onto the harddrive and compared to the expected ground-truths of the test dataset.

5.4 Results and Discussion

The results of the demonstration system show that computing and transfer times of a few microseconds are, in fact, achievable for GPGPU systems, when using appropriate technologies and interconnects. Measured results are given in table 5.1

Table 5.1: Execution times for the commodity trigger implementation. Shown are the results for standard rectangular Hough Space implementation, both sequential and pipelined. As well as the sequential hexagonal Hough Space implementation. All times given in μs.

Rectangular (Sequential)	mean	σ	max
Polling and Transfer	1.96	$\pm$0.02	2.01
Computation	3.73	$\pm$0.13	4.19
Total	5.84	$\pm$0.14	6.33
Rectangular (Pipelined)	**mean**	**σ**	**max**
Polling	0.41	$\pm$0.01	0.43
Computation	3.63	$\pm$0.10	3.82
Total	4.17	$\pm$0.07	4.37
Hexagonal (Sequential)	**mean**	**σ**	**max**
Polling and Transfer	1.91	$\pm$0.05	1.99
Computation	4.94	$\pm$0.11	5.20
Total	7.07	$\pm$0.12	7.40

We built the system in such a way that it uses exclusively DMA to move data to and from the GPU. Transfer times were measured from within the GPU and showed comparatively stable transfer times of 2 μs for any dataset up to 8 kBytes. Data arrival was ascertained by means of reading a data flag from GPU main memory. This indicates that 2 μs measurement is largely influenced by the propagation of the flag to memory cache, rather than actual DMA transfer. Furthermore, polling times for the pipelined approach, in which data will have already arrived before checking the respective completion flag, show a highly stable 0.4 μs access time. This aligns with our preliminary measurements of GPU main memory access requiring roughly 300 clock cycles.

Computation and GPU Kernel design were aggressively optimized for latency, rather than throughput. This resulted in our system being able to reach computing times as low as 4 μs. But due to highly unfavourable thread-block distribution and the limited amount of shared memory of the GPU, we were unable to process all of the stubs of a single detector sector at once. This would have forced us to split even single detector segments across multiple GPUs, which would have added additional complexity and synchronization overheads to the system. The dataset transferred is still represented as a full dataset, for which we used simulated TTBar events of 140 and 200 Pileup (PU) containing a top and antitiop quark pair. The total number of stubs transferred was 160, which overestimates the actual number of stubs generated by such events and pileups by roughly a factor of 2. However, due to the aforementioned limitations in memory layout and kernel design, we were only able to process around 90 of these stubs in parallel. At the current expected CMS detector segmentation of 9×32 sectors, this would have made it necessary to use between 288 and 576 GPUs to process the data of the whole detector, depending on pileup.
We still deem the presented results as valid, due to the fact that all stubs get processed independently from each other in fully parallel fashion. This means that the mean computation time does not scale off the number of stubs in total, but rather off the longest individual processing time for any given individual stub. With regards to future generations of GPUs, we are hopeful that we will see an increase in the amount of available shared memory and/or significantly reduced GPU main memory access times, as well as an increase in available parallel execution units that would make it possible to process a higher number of stubs in parallel.

> **Note:** The current design proposals for the detector segmentation of CMS introduces necessary overlaps between sectors. This requires the front-end electronics and data concentrators to duplicate stubs in those overlapping regions and passing them on to more than one individual computing stage at a time. This introduces the need to remove the resulting duplicate tracks from the results, which is not implemented in our demonstrator.

Leveraging the floating-point computing capabilities of GPUs, we introduced a change to the Hough transformation from the conventional (rectangular) parameter space into a hexagonal one. The added computational complexity increased the average computing times by around 1 μs, or roughly 25% However, the benefits of the hexagonal parameter space could be shown to reduce the amounts of fake track candidates found by an amount of up to 35% in scenarios of high pileups ([Moh16],[Moh17]).

Overall, we were able to demonstrate that we could develop a commodity GPGPU-based DAQ computing system that comes astonishingly close to satisfying the demands of something like the CMS' track trigger specifications of 1 μs for data transfer and 4 μs for computing. Our system was capable of achieving 2 μs for data transfer and between 3.8–5.2 μs for computing, depending on the version of the implementation (Rectangular or Hexagonal Hough Space).

Table 5.2: Comparison of actual CMS L1 Track Trigger constraints against achieved performance of the Commodity GPGPU DAQ Demonstrator, by algorithmic variation. Times for the algorithms represent worst-cases.

	Data Transfer	Computing	Total
CMS Constraint	1 μs	4 μs	5 μs
Rectangular (Sequential)	2 μs	4.2 μs	6.2 μs
Rectangular (Pipelined)	0.4 μs	3.8 μs	4.2 μs
Hexagonal (Sequential)	2 μs	5.2 μs	7.5 μs

With these results, we are less than a factor of two away from the target specifications. Even though at the time of measurement, the amount of required GPUs to process an entire CMS dataset would have been prohibitively large, the results clearly indicate the feasibility and potential performance of the commodity DAQ concept, especially when taking potential future advancements in GPU technologies and general technology-extrapolation into account.

The results of this investigation and proof-of-concept implementation have been presented in the proceedings of the *Topical Workshop on Electronics for Particle Physics 2016*. They were published in the *Journal of instrumentation 12.04 (2017)* ([Moh17]).

6 Software-defined DAQ and future developments

When looking at the proposed technologies and system designs presented in the previous chapters a of this thesis, a trend towards software-programmable components becomes visible. A similar trend has already emerged in the general field of wireless transmission in the early 2000s. This trend is commonly known as *Software-defined Radio (SDR)* [Sad04].
The concept of SDR can be described as a movement to replace as many "hard-wired" analogue electronics in radio transmission systems as possible with programmable devices. SDR garnered widespread adaptation in the respective communities, with its main benefits stemming from software-system's high flexibility due to the ability of software defined systems to quickly adapt to and interoperate between multiple different standards.
Much like with SDR, the point can be made that the DAQ systems presented in this thesis achieve a similar level of flexibility due to their core principles of employing mainly software-programmable commodity computing components. As a result, we coin the term ***Software-defined DAQ (SDDAQ)*** for the systems presented in this thesis.

This chapter will show potential future prospects of fully embracing the software-controlled nature of SDDAQ with special emphasis on concepts such as *Routing and Fault-Tolerance*.

6.1 RDMA Network Routing for fault tolerance and load balancing

When computing systems need to offer more (computing) resources than a single processing node can supply, they often begin to scale their operations on a network level. By introducing multiple separate physical nodes to a processing scheme, connected by a network, one also necessarily introduces the challenge of *Routing*. Routing describes the process of (re)directing network traffic from a source to a destination through intermediary participants. It is a vital part of many large network installations today, and is one of the key technologies of the Internet as we know it. That is why the devices we typically use to connect to the internet are still today most commonly known as "Routers".
Routing can also be used to provide reliability in a distributed network service. When data packets are addressed towards a node that went into a failure state, they can be dynamically and transparently rerouted towards a fallback node that provides the same functionality. Equally, nodes and services can be replicated multiple times, and the routing towards these nodes can be selected based on the current workload of each node. This is commonly known as *Load Balancing* [Bar89]. Both of these techniques are common place in most modern data centers.
Many of these routing techniques rely, in essence, on the capability of changing the intended recipient data-field in a network package and then re-emitting the package towards that (new) destination. In this section, I want to present thoughts on how fault-tolerance and load-balancing in an RDMA capable network can be achieved.

IP-based data communication uses a "port" information field to describe the software service on the recipient that is intended to process the received data package. Changing any of the addressing information of the package on the way to its destination will not interfere with this mechanism, as long as the new (think: rerouted) recipient of the communication has an identical software service listening on that same port. Therefore, an Ethernet IP package

can be rerouted both on the data-link-layer (Ethernet MAC address), as well as the network-layer (IP Address).

> **Note:** Remember, the mentioned Data-Link-Layer and Network-Layer pertain to layer 2 and 3 of the OSI Model, presented in table 2.1, in section 2.1.2 of this thesis.

If we imagine the same mechanism for RDMA network communication, specifically for InfiniBand / RoCE, we encounter a problem. Recall the introduction of InfiniBand in chapter 3 of this thesis. To perform RDMA operations on a remote node, that node first needs to grant access to its memory to a connected peer by means of generating an *access token*. This token is unique for each node-peer combination. InfiniBand uses both, data-link-layer and network-layer address information, and should therefore be routable just like IP based communication. However, the access tokens embedded in the data frame will be invalid if the package gets rerouted to a completely different recipient node. Therefore, InfiniBand traffic is capable of being routed through a network of networks (OSI Layer 3), but —at least not in its native form —can not be used to provide load balancing or fault tolerant redundancy. The same holds true for InfiniBand through Ethernet (RoCE), as RoCE merely replaces the data-link-layer and network-layer encodings of InfiniBand with that of conventional Ethernet and IP protocols.
To provide a solution to these issues, and make InfiniBand RDMA a suitable candidate for fault-tolerant and load-balanced routing, the following solutions are proposed.

Token Unification: The driver software for InfiniBand RDMA could be changed, so that it either always generates the exact same token for each allocated RDMA memory segment, or it could be changed to not rely on the tokens all together. This solution would provide the following benefits:

- Can be implemented fully and exclusively in software
- Conventional routing mechanisms of Ethernet / IP based systems can be retained

However, this solution also holds a number of disadvantages.

- Individualized changes in software packages maintained by other sources need continuous upkeep and maintenance whenever the software receives downstream changes
- Changes potentially need to be platform-dependent, increasing development complexity when using heterogeneous systems consisting of multiple different platforms

Furthermore, we observe another potentially severe drawback of this solution. Up until this point, we have only considered RDMA access tokens. However, the necessary information encoded in a data package for RDMA does not only include these access tokens. They also include the system-level memory address of the allocated RDMA segment on the recipient system. Where these memory segments reside is highly application specific and in many cases depends on the allocation mechanisms of the node's operating system. Even if we change the access tokens to be uniform across our network, the location of each memory segment on each peer would still vary. When rerouting RDMA traffic to a new destination, it would lead to the RDMA packages addressing memory locations that are not part of the provided access token.
Changing this behavior can be considered to not be generally possible. How much memory is available to each system, how the operating system segments the memory and if the desired memory segment is available at the time of allocation, can not universally be controlled, lest extensive changes are made to the node's operating system.

Custom RDMA Dispatcher: A different solution could be to develop what we shall call an *RDMA Dispatcher*. This intermediary device would be responsible to ensure the correctness of the RDMA access tokens as the data packages travel through the device. To do this, we could configure the device in such a way that it provides memory segments for each software service on the network. When a device wants to send data to one of those services, it instead sends data to the corresponding memory segment of the dispatcher. The dispatcher receives the communication, and upon transfer completion, re-emits the received data to the node on the network that actually implements the

corresponding service. For this, a software framework would be required that informs the dispatcher of each of the services on the network and exchanges the required access tokens for those service's memory segments with the dispatcher. The dispatcher can then dynamically choose which device to forward the received data to, based on any metric, like static configuration, faults or workload on the nodes. This process is symbolized in figure 6.1.

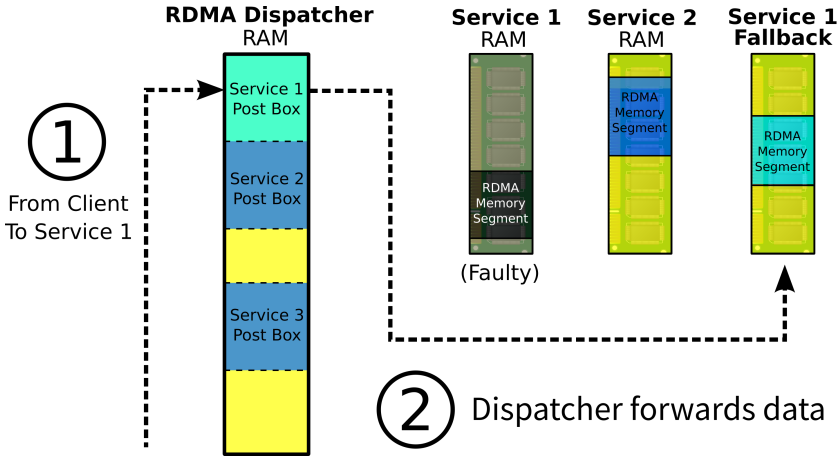

Figure 6.1: The working principles of an RDMA dispatcher. The dispatcher provides specific memory segments for each available service. A client that wants to send data to one of these services sends data instead to the corresponding memory segment of the repeater (1). After the transfer concludes, the repeater forwards the data to its final destination in the respective service's memory (2). Based on internal metrics, like faults or resource occupation, the dispatcher can decide to redirect data to a different service instead.

This solution would have the following benefits:

- Hardware could be set up using only standard commodity components
- Connections between dispatcher and participating nodes will use only conventional RDMA schemes
- No packet manipulations is required

- Required routing tables and metrics can be implemented in software

However, the system would also have drawbacks

- Receiving data on the dispatcher first, and then retransmitting the received data, introduces one additional round of systemic transfer overhead
- Depending on the amount of services on the network, the dispatcher would require large amounts of memory
- Concurrent access to the same memory segment on the dispatcher could cause data corruption
- Dispatcher performance would be limited by memory bandwidth and throughput

Custom RDMA Router: Finally, we propose the solution of a custom RDMA router. Modern programmable hardware devices, such as FPGAs with Gigabit transceivers, and "Smart" Switches with embedded FPGAs have become available over the recent years. It is assumed that it should be technically feasible to use such a device to read and modify network communication as it passes through the device, not unlike conventional Ethernet routers operate. Such devices could be used to fully and transparently rewrite the entirety of the RDMA access tokens and address information in an InfiniBand / RoCE packet. For this to work, the router would need a database of the currently available RDMA services on the network, and their respective access tokens. When a node on the network sends an RDMA package to any destination on the network, the router would analyze the target address, and replace RDMA access information based on desired metrics. Note that in this case, not only the Access Token would be replaced, but the entire RDMA package header, except for the data payload. In the previously discussed solution (RDMA Dispatcher), the dispatcher itself was a conventional endpoint of the RDMA communication. In case of the proposed RDMA router, the router would *not* be an endpoint. Packages would travel through the device and get modified as they get passed along. The working principle of such an RDMA router is visualized in figure 6.2.

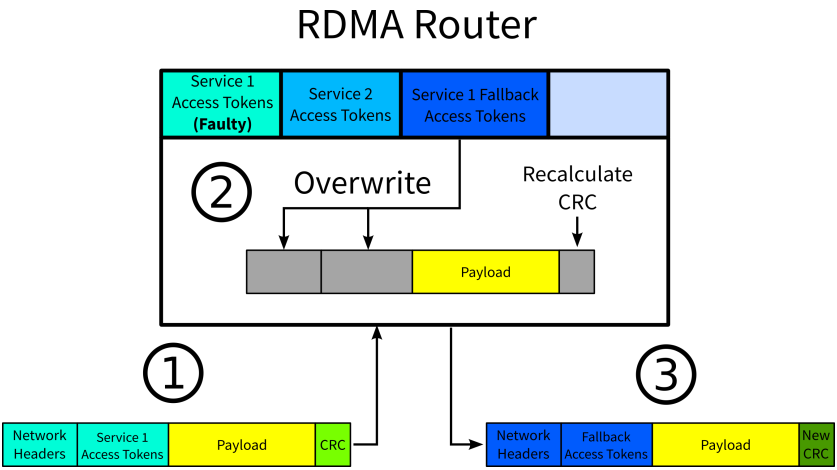

Figure 6.2: The working principles of an RDMA router. The router is made aware of the specific memory segments for each available service, and their respective RDMA access tokens. All communications directed towards the network are funneled through the router. The router observes a data package addressed towards one of the services (1). Based on internal metrics, the router then rewrites parts of the package to change its routing information and path through the network (2). This may change the previously intended recipient of the package. Afterwards, the rewritten package is emitted towards the rest of the network (3) where conventional transfer schemes are used to deliver the package to its (potentially new) destination.

This solution would have the following benefits:

- Only marginal added communication latency, since data is not fully placed in memory before being re-emitted
- Concurrent access would be a non-issue, since packages are not stored, but merely rewritten
- Commodity hardware, capable of implementing such actions, are already commercially available (e.g.: Xilinx Vitis devices)

However, this solution also features a list of complex drawbacks:

- Implementation in HDL is much more complex, error-prone and less flexible than pure software solutions

- Correctly decoding, modifying and encoding the *checksum* of each package may not be possible if the checksum algorithm of the protocol being used is unknown
- Reliability and retransmission mechanisms of protocols like InfiniBand would make it necessary to reimplement parts of their protocol stack on the router, increasing the device's complexity

In summary, all of the proposed solutions have individual strengths and weaknesses. It is not currently possible to choose one of them as the clearly optimal solution to implement the required routing mechanisms. Implementation of these solutions, as well as performance measurements remain subject to future investigations.

6.2 Future of DAQ in the Cloud

Cloud Computing has gained increased significance over the recent years. When speaking of a computing "cloud", we generally mean any heavily distributed computing service that is made available to a user on-demand. Large companies, like Microsoft, Amazon, Google, Facebook and others offer highly specialized resources to users, billed on some form of per-use basis. Popular cloud services include data storage, audio and video transcoding, image processing and more. More generalized computing resources, like many-core CPUs, machine-learning accelerators and other forms of specialized computing hardware are also commonly available *in the cloud*.
This has given rise to services that live exclusively in the cloud. Companies leveraging cloud computing services benefit from not having to set-up, operate and maintain their own data centers, freeing up resources. Furthermore, they can leave development and optimization of such services in the hands of the cloud providers, avoiding having to bundle the necessary technical expertise on their own roster of personnel.

With the newly coined concept of **Software-defined DAQ**, we envision services specialized for scientific data processing eventually adopting a similar paradigm. Many research groups (such like universities) could benefit from

operating a single data-center that provides access to multiple small and mid-sized research teams. Considering possible solutions to the RDMA routing problems (as proposed in the previous section) it may soon become possible to fully realize such DAQ data centers in a local context (e.g. per University Campus). As mentioned in chapter 2.1.1 of this thesis, conventional metal wire cables can already bridge distances of hundreds of meters at up to 200 Gbps. Optical fibres can even provide dozens of kilometers range at up to 400 Gbps. For the local context of a single research campus it should therefore be feasible to link local research installations to the campus' data center without suffering substantially reduced networking performance.

Another possibility could be the decentralization and sharing of DAQ computing resources through the internet. While conventional internet connections to services worldwide are currently not sufficient to handle the data-rates and latencies depicted in this thesis, it is not unreasonable to assume that these constraints might one day disappear. As can be seen in the "Global Speedtest Index" by *Ookla* (one of the leading providers of online internet performance tests worldwide), average internet performances around the globe are on the rise. Figure 6.3 shows how average worldwide internet throughputs have increased from just below 40 Mbps in mid 2017 to almost 110 Mbps by the same time in 2021. Mobile internet download throughputs have increased by an even larger margin. While these graphs only show average download speeds, neglecting average upload speed and average latencies, the presented data can still be seen as an indication that average internet key metrics will likely keep increasing in the coming years.

Furthermore, while these key performance metrics are orders of magnitude below the performances demonstrated by the systems in this thesis, it is important to keep in mind that not all experiments have such challenging requirements. Many projects operate at data rates much lower than the terabit- or even gigabit- domain, and with event rates in the order of lower kilohertz. For such experiments, typical internet throughputs and latencies of a few milliseconds are already sufficient, even today. However, as the internet is built on a "best-effort" basis, data passing through the internet is not guaranteed to arrive at its destination. Protocols like TCP can provide resilience towards

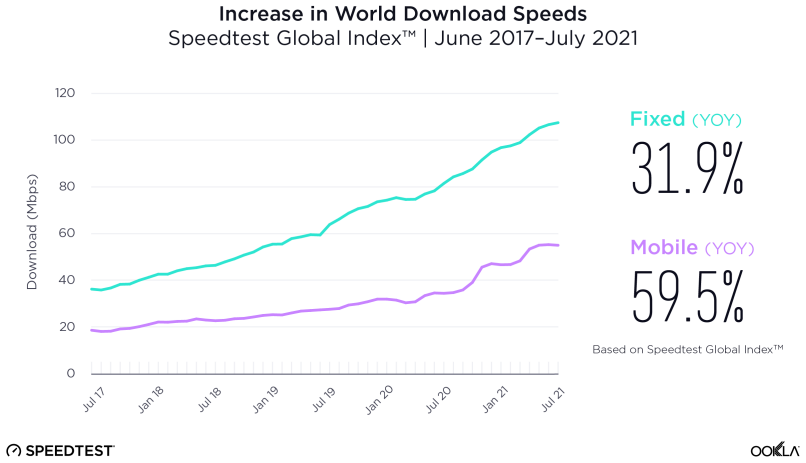

Figure 6.3: Average global internet download speeds as measured by Ookla, from July 2017 to July 2021, as part of their yearly Speedtest Global Index. [b]

[b] Provided by Ookla inc.

lost data packages, but do so by repeatedly sending packages until the recipient has confirmed their arrival. On particularly unreliable connections, such a mechanism introduces unpredictable latencies and vastly varying times-of-arrival. Therefore, even if data-rates of an internet connection might be sufficient for certain experiments, real-time conformity remains an open challenge.

In summary, we believe that the systems and concepts presented in this thesis bring us further towards a future in which experiments can enjoy substantially decreased development efforts for their DAQ. Fuelled by centralized cloud-DAQ services, systems configured and defined exclusively through software, and ever-increasing key performances, due to continues industry developments in the fields of High-Performance-Computing.

7 Conclusion and Research Results

This work investigated the potential of using commodity computing and data-transfer components to build high-performance DAQ systems that are capable of satisfying the needs of even highly challenging scientific experiments. Summarizing the results of this research, the questions formulated in the introduction of this thesis can now be answered as follows.

Research question 1:
Which commodity computing technologies can be used to aid the integration of heterogeneous DAQ components?

The requirements for modern high-performance DAQ have developed into a direction that became increasingly similar to those of modern high-performance computing (HPC) installations. However, both domains differ in their specific core focus. In many cases, HPC installations can afford to put lower emphasis on transfer and processing latencies, as many HPC applications lack real-time requirements. For the creation of fast feedback-control-loops, as they are increasingly more commonly found in modern scientific experiments however, low transfer and computing latencies are of much higher importance. Furthermore, most HPC applications do not require connections to external custom electronics, whereas such connectivity is essential for DAQ system. These subtle but important differences in design priorities are what sets apart the DAQ systems proposed in this thesis from conventional HPC installations. Essentially, the systems resulting from this work are distributed, heterogeneous HPC systems, optimized for lowest possible data transfer and computing latencies, while offering convenient connectivity to any arbitrary custom electronics through FPGAs.

> **Answer:** Contemporary HPC systems provide a suitable blueprint for heterogeneous DAQ systems but need to be augmented with connectivity solutions for custom electronics at low latencies. FPGAs can serve as middle-grounds between fully custom electronics and standardized interfaces to PC systems. Interconnect technologies that support DMA aid with the optimization towards low-latency communication in large heterogeneous systems.

Research question 2:
Which RDMA capable network standards are available? How do they differ from each other, and how well are they suited to modern DAQ needs? How can existing Ethernet DAQ infrastructure benefit from RDMA?

To realize high-performance commodity DAQ systems, (Remote) Direct Memory Access (DMA/RDMA) was shown to be an invaluable asset. As the complexity of distributed systems grows, so does the number of intermediary memories across those systems. The capability to move data directly into target memory layers on interconnected systems while bypassing unrelated memory structures significantly reduces critical data path lengths and thereby shortens systemic data transfer latencies. It was demonstrated how RDMA communication could bypass bottlenecks in existing systems at the example of a high-speed tomography DAQ system through the integration of a newly developed RDMA networking library, called *KIRO*. KIRO aims to mimic the behavior of conventional UNIX network sockets, while providing additional conveniences, such as self-synchronizing remote ring-buffers, and many-to-many messaging. It supports InfiniBand and RoCE RDMA data transfer, and was shown to be able to provide adapter bandwidth utilizations of up to 95% (38 Gpbs throughput of 40 Gpbs bandwidth) with systemic latencies averaging 6 μs. With the help of this software, it became possible to add RDMA to any networking application with just a few lines of code [Dri14].

> **Answer:** Two most commonly used (R)DMA-capable interconnects are PCI-Express (PCIe) and InfiniBand. PCIe is optimized for peripheral interconnectivity inside of a single PC system.

It is not optimized for networking. In contrast, InfiniBand is built specifically to be a Network-Layer routable protocol. Both standards are well-suited for low-latency interconnectivity. PCIe serves as intra-system data bus. InfiniBand serves as inter-system network solution.
Ethernet does not natively support RDMA, but can profit from it by means of the *RoCE* protocol. RoCE places InfiniBand "instructions" into Ethernet data frames by replacing Data-Link and Network Layer information in an InfiniBand data frame with Ethernet and UDP addressing headers respectively.

Research question 3:
How can Remote Direct Memory Access be combined with traditional High-Performance Computing methods, such as GPU computing?

It was shown how Graphics Processing Units (GPUs) can benefit from (R)DMA to receive processing data directly into their graphics memory, rather than having to rely on the CPU to copy data from system main memory over to the GPU. This reduced transfer latencies by an order of magnitude from 20 μs to as low as 1.8 μs. GPUs are typically valued for their very high computational throughput, and RDMA data transfer therefore additionally enables the utilization of GPU's high computing performance for distributed and networked applications with extremely low latency requirements.

Answer: GPU manufacturers provide drivers that enable direct bus-level access to GPU memory directly on the device. This supports bypassing conventional CPU-centric data transfer schemes and enables DMA capable devices to target GPU memory directly. Using RDMA extends these capabilities onto the network level.

Research question 4:
How can general purpose commodity DAQ systems benefit from the application of FPGAs, and how can FPGAs be integrated into such systems?

To provide the necessary interface between custom detector front-end electronics and commodity DAQ systems, it was demonstrated how RDMA-capable FPGAs can be integrated into our systems to write digitized data straight to the memory of any connected computing accelerator (e.g. GPUs). An example of this is the DMA capable FPGA-based "High-Flexibility DAQ Platform" (High-Flex) developed at KIT [Cas14]. Interconnectivity between the High-Flex platform and a PC is provided through PCIe (Gen. 3, x8 or x16) as well as Gigabit Ethernet.

> **Answer:** FPGAs provide high levels of control over external interconnectivity by means of discrete logic blocks like Analogue-to-Digital converters and general purpose high-speed transceivers. This makes them well-suited as interfacing devices to custom electronics. High-Speed transceivers can be used to implement (R)DMA capable interconnects and protocols to standardize integration with other commodity PC components at low communication latencies.

Research question 5:
Which suitable commodity DAQ system design facilitates both low data communication latencies and high- performance computing? Which standards should be chosen to aid maintainability and scalability?

It was shown how the combination of FPGAs and GPUs inside of DAQ is particularly powerful, as both technologies compensate each others weaknesses. FPGAs provide excellent connectivity and electronic interfacing, as well as precise control over timing and digital logic, whereas GPUs provide computation capabilities at higher performances than FPGAs. Furthermore, the demonstrated integration of heterogeneous computing components through the use of RDMA was shown to result in a general purpose computing system that was able to provide turnaround times of only $4.1 - 5.8\ \mu s$. Interconnectivity was provided through PCIe Gen. 3 x8, with potential peak point-to-point throughputs of up to 52 Gbps (as presented in [Rot16]). This was demonstrated by the implementation of a Hough-Transformation Track-Finding algorithm, as it is projected to be used by the CMS experiment after the LHC's

high-luminosity upgrade. These times include data transfer and computing (however, no full loop was implemented, which needs to be considered in future implementations). This example illustrates the feasibility of the commodity DAQ concept by showing performance of the same order of magnitude as those required by the CMS Track Trigger specifications, which is projected to be 5 μs for a full turn-around trip. The presented times of our system are predicted to further decrease in the future, as newer generations of networking adapters and GPUs become available, which are likely to provide increased performance metrics and reduced intrinsic latencies [Moh17].

> **Answer:** The proposed systems use DMA-capable FPGA platforms to implement interfaces to external systems and custom electronics. For computing, the proposed systems use GPUs because of their high computing performances and DMA capabilities. Communication between components is realized through PCIe, as the de-facto standard for peripheral interconnectivity in contemporary systems while being fully DMA-capable. Scalability can be achieved through a distributed design. The necessary network communication can be provided through InfiniBand or Ethernet + RoCE networking adapters, which also interface with the PCIe bus, easing integration.

Research question 6:
How can DAQ benefit from being defined mainly through software, and how could such a Software-defined DAQ system look like?

For practical applications, this work proposes the novel concept of "Software-defined DAQ". As most of the proposed commodity components are software programmable, it is possible to model the data flow and the processing schemas across the proposed commodity DAQ systems completely in software. Typically, this is done by defining a chain of network services that exchange data with each other by means of addressing their respective memories. To this end, we envision a modular system of heterogeneous processing nodes, interconnected by an RDMA capable network. Each node in these

systems provides a specific "service" to the DAQ chain, e.g.: Filtering, Compression, Track Reconstruction, etc. The distribution of data is managed by orchestration software, running on "smart" networking device that will be capable of dynamically routing data according to the current experiment requirements. Deep-Package-Inspection could be used to rewrite RDMA address and access information in networking packages to dynamically reroute traffic based on system metrics like occupation or system faults. The necessary software configuration could be provided in the form of containerized software, which would ease the creation and integration of additional computing nodes into the system.
In the future, advances in networking technologies may reduce global internet latencies while simultaneously increasing available bandwidths and reliability, to potentially provide generalized Software-defined DAQ services in the form of a computing cloud. This could open up the possibilities to provide shared access to highly specialized DAQ processing chains to research groups that do not possess the required resources to operate such systems on their own. These cloud DAQ services could also provide access to emerging technologies, such as Neuro-Accelerators, Quantum-Processors or arrays of FPGA computing cards.

> **Answer:** Software-defined DAQ provides high-level control over the behavior of computing and data paths in a distributed DAQ system with a high degree of flexibility and adaptability. With the addition of the proposed routing mechanisms for RDMA it should become possible to build decentralized DAQ services, as well as providing fault-tolerance and load-balancing to such systems.

Overall, it was shown that with careful system design, using commodity computing components, it is possible to create highly performant and flexible software-defined DAQ systems, capable of satisfying the needs of even highly challenging experiments.

Bibliography

[Agg17] Aggleton, R. et al.: "An FPGA based track finder for the L1 trigger of the CMS experiment at the High Luminosity LHC". In: *Journal of Instrumentation* 12.12 (Dec. 2017), P12019–P12019. doi: 10.1088/1748-0221/12/12/p12019. url: https://doi.org/10.1088/1748-0221/12/12/p12019 (cit. on p. 121).

[Ams16] Amstutz, C. et al.: "An FPGA-based track finder for the L1 trigger of the CMS experiment at the high luminosity LHC". In: *2016 IEEE-NPSS Real Time Conference (RT)*. 2016, pp. 1–9. doi: 10.1109/RTC.2016.7543102 (cit. on p. 123).

[Ams17] Amstutz, Christian: "Evaluation of an Associative Memory and FPGA-based System for the Track Trigger of the CMS-Detector". 54.02.02; LK 01. PhD thesis. Karlsruher Institut für Technologie (KIT), 2017. 183 pp. doi: 10.5445/IR/1000071572 (cit. on p. 122).

[And17] Andre, J-M et al.: "The CMS Data Acquisition - Architectures for the Phase-2 Upgrade". In: *Journal of Physics: Conference Series* 898 (Oct. 2017), p. 032019. doi: 10.1088/1742-6596/898/3/032019. url: https://doi.org/10.1088/1742-6596/898/3/032019 (cit. on pp. 2, 10, 117).

[Arr01] Arregui, M; Carena, W; Chapeland, S; Csato, P; Denes, E; Divia, R; Eged, B; Jovanovic, Predrag; Kiss, T; Lindenstruth, V et al.: "The ALICE DAQ: current status and future challenges". In: *Computer physics communications* 140.1-2 (2001), pp. 117–129 (cit. on p. 10).

[ATL12] ATLAS Group: "Observation of a new particle in the search for the Standard Model Higgs boson with the ATLAS detector at the LHC". In: *Physics Letters B* 716.1 (2012), pp. 1–29. doi: https:

//doi.org/10.1016/j.physletb.2012.08.020. URL: https://www.sciencedirect.com/science/article/pii/S037026931200857X (cit. on p. 117).

[Ban12] BANERJEE, S: "CMS Simulation Software". In: *Journal of Physics: Conference Series* 396.2 (Dec. 2012), p. 022003. DOI: 10.1088/1742-6596/396/2/022003. URL: https://doi.org/10.1088/1742-6596/396/2/022003 (cit. on p. 127).

[Bar17] BARTZ, EDWARD et al.: "FPGA-Based Tracklet Approach to Level-1 Track Finding at CMS for the HL-LHC". In: *EPJ Web Conf.* 150 (2017), p. 00016. DOI: 10.1051/epjconf/201715000016. URL: https://doi.org/10.1051/epjconf/201715000016 (cit. on p. 122).

[Bar89] BARAN, Mesut E and WU, Felix F: "Network reconfiguration in distribution systems for loss reduction and load balancing". In: *IEEE Transactions on Power delivery* 4.2 (1989), pp. 1401–1407 (cit. on p. 136).

[Bel16] BELLALTA, Boris: "IEEE 802.11ax: High-efficiency WLANS". In: *IEEE Wireless Communications* 23.1 (2016), pp. 38–46. DOI: 10.1109/MWC.2016.7422404 (cit. on p. 17).

[Ber20] BERTELS, K.; SARKAR, A.; HUBREGTSEN., T.; SERRAO, M.; MOUEDENNE, A. A.; YADAV, A.; KROL, A. and ASHRAF, I.: "Quantum Computer Architecture: Towards Full-Stack Quantum Accelerators". In: *2020 Design, Automation and Test in Europe Conference and Exhibition (DATE)*. 2020, pp. 1–6. DOI: 10.23919/DATE48585.2020.9116502 (cit. on p. 49).

[Bis01] BISDIKIAN, Chatschik: "An overview of the Bluetooth wireless technology". In: *IEEE Communications magazine* 39.12 (2001), pp. 86–94 (cit. on p. 16).

[Bri17] BRITT, Keith A.; MOHIYADDIN, Fahd A. and HUMBLE, Travis S.: "Quantum Accelerators for High-Performance Computing Systems". In: *2017 IEEE International Conference on Rebooting Computing (ICRC)*. 2017, pp. 1–7. DOI: 10.1109/ICRC.2017.8123664 (cit. on p. 49).

[Bro75] Brooks, Rodney A and Di Chiro, Giovanni: "Theory of image reconstruction in computed tomography". In: *Radiology* 117.3 (1975), pp. 561–572 (cit. on p. 96).

[Buc04] Buchanan, W. J.: "RS-232". In: *The Handbook of Data Communications and Networks: Volume 1. Volume 2*. Boston, MA: Springer US, 2004, pp. 239–274. doi: 10.1007/978-1-4020-7870-5_11. url: https://doi.org/10.1007/978-1-4020-7870-5_11 (cit. on p. 24).

[Bud04] Budruk, Ravi; Anderson, Don and Shanley, Tom: PCI express system architecture. Addison-Wesley Professional, 2004 (cit. on p. 25).

[Can15] Canzian, Luca and Van Der Schaar, Mihaela: "Real-time stream mining: online knowledge extraction using classifier networks". In: *IEEE Network* 29.5 (2015), pp. 10–16 (cit. on p. 10).

[Cas14] Caselle, M. et al.: "Commissioning of an ultra-fast data acquisition system for coherent synchrotron radiation detection". In: *5th International Particle Accelerator Conference (IPAC 2014), Dresden, 15. - 20. Juni 2014. Paper THPME113 JACoW, 2014*. 55.51.10; LK 02. JACoW Publishing, 2014, pp. 3497–3499 (cit. on p. 148).

[Cas17] Caselle, M.; Perez, L.E. Ardila; Balzer, M.; Dritschler, T.; Kopmann, A.; Mohr, H.; Rota, L.; Vogelgesang, M. and Weber, M.: "A high-speed DAQ framework for future high-level trigger and event building clusters". In: *Journal of Instrumentation* 12.03 (Mar. 2017), pp. C03015–C03015. doi: 10.1088/1748-0221/12/03/c03015. url: https://doi.org/10.1088/1748-0221/12/03/c03015 (cit. on p. 72).

[Chi20] Chilingaryan, Suren; Ametova, Evelina; Kopmann, Anreas and Mirone, Alessandro: "Reviewing GPU architectures to build efficient back projection for parallel geometries". In: *Journal of Real-Time Image Processing* 17.5 (2020), pp. 1331–1373. doi: https://doi.org/10.1007/s11554-019-00883-w (cit. on p. 116).

[CMS12] CMS Group: "Observation of a new boson at a mass of 125 GeV with the CMS experiment at the LHC". In: *Physics Letters B* 716.1 (2012), pp. 30–61. doi: https://doi.org/10.1016/j.physletb.2012.08.021. url: https://www.sciencedirect.com/science/article/pii/S0370269312008581 (cit. on p. 117).

[CMS17] CMS Group: "The CMS trigger system". In: *Journal of Instrumentation* 12.01 (Jan. 2017), P01020–P01020. doi: 10.1088/1748-0221/12/01/p01020. url: https://doi.org/10.1088%2F1748-0221%2F12%2F01%2Fp01020 (cit. on p. 119).

[Dal05] Dalessandro, Dennis; Devulapalli, Ananth and Wyckoff, Pete: "Design and Implementation of the iWarp Protocol in Software." In: *IASTED PDCS*. 2005, pp. 471–476 (cit. on p. 33).

[Dal91] Dalesio, Leo R; Kozubal, AJ and Kraimer, MR: EPICS architecture. Tech. rep. Los Alamos National Lab., NM (United States), 1991 (cit. on p. 94).

[Die08] Dierks, Tim and Rescorla, Eric: The transport layer security (TLS) protocol version 1.2. Tech. rep. 2008 (cit. on p. 39).

[Dri14] Dritschler, T; Chilingaryan, S; Farago, T; Kopmann, A and Vogelgesang, M: "InfiniBand interconnects for highthroughput data acquisition in a TANGO environment". In: *Proc. PCaPAC'14* (2014), pp. 161–163 (cit. on pp. 103, 107, 108, 146).

[Eid] Eidson, Stevan; Gaines, Brett and Wolf, Paul: "30.2: HDMI: High-Definition Multimedia Interface". In: *SID Symposium Digest of Technical Papers* 34.1 (), pp. 1024–1027. doi: https://doi.org/10.1889/1.1832462. url: https://sid.onlinelibrary.wiley.com/doi/abs/10.1889/1.1832462 (cit. on p. 23).

[For02a] Forouzan, Behrouz A: TCP/IP protocol suite. McGraw-Hill Higher Education, 2002 (cit. on pp. 33, 34).

[For02b] Forouzan, Behrouz A: TCP/IP protocol suite. McGraw-Hill Higher Education, 2002 (cit. on p. 35).

[Göt03] Götz, A; Taurel, E; Pons, JL; Verdier, P; Chaize, JM; Meyer, J; Poncet, F; Heunen, G; Götz, E; Buteau, A et al.: "TANGO a CORBA based control system". In: *ICALEPCS2003, Gyeongju, October* (2003) (cit. on pp. 94, 98, 99).

[Gro96] Gropp, William; Lusk, Ewing; Doss, Nathan and Skjellum, Anthony: "A high-performance, portable implementation of the MPI message passing interface standard". In: *Parallel Computing* 22.6 (1996), pp. 789–828. doi: https://doi.org/10.1016/0167-8191(96)00024-5. url: https://www.sciencedirect.com/science/article/pii/0167819196000245 (cit. on p. 45).

[Guo16] Guo, Chuanxiong; Wu, Haitao; Deng, Zhong; Soni, Gaurav; Ye, Jianxi; Padhye, Jitu and Lipshteyn, Marina: "RDMA over commodity ethernet at scale". In: *Proceedings of the 2016 ACM SIGCOMM Conference*. 2016, pp. 202–215 (cit. on p. 33).

[Gus88] Gustafson, John L: "Reevaluating Amdahl's law". In: *Communications of the ACM* 31.5 (1988), pp. 532–533. doi: 10.1145/42411.42415 (cit. on p. 46).

[Hou17] Houzeaux, Guillaume; Borrell, Ricard; Fournier, Yvan; Gasulla, Marta Garcia-; Göbbert, Jens Henrik; Hachem, Elie; Mehta, Vishal; Mesri, Youssef; Owen, Herbert and Vázquez, Mariano: "High-Performance Computing: Dos and Don'ts". In: *Computational Fluid Dynamics*. Ed. by Ionescu, Adela. Rijeka: IntechOpen, 2017. Chap. 1. doi: 10.5772/intechopen.72042. url: https://doi.org/10.5772/intechopen.72042 (cit. on p. 47).

[HPL02] HPL, Steve Corrigan: "Introduction to the controller area network (CAN)". In: *Application Report SLOA101* (2002), pp. 1–17 (cit. on p. 24).

[Jai18] Jain, Amit Kr.; Acharya, Rupesh; Jakhar, Saroj and Mishra, Tarun: "Fifth Generation (5G) Wireless Technology "Revolution in Telecommunication"". In: *2018 Second International Conference on Inventive Communication and Computational Technologies (ICICCT)*. 2018, pp. 1867–1872. doi: 10.1109/ICICCT.2018.8473011 (cit. on p. 17).

[Jei14] Jeitler, M: "The upgrade of the CMS trigger system". In: *Journal of Instrumentation* 9.08 (Aug. 2014), pp. C08002–C08002. doi: 10.1088/1748-0221/9/08/c08002. url: https://doi.org/10.1088/1748-0221/9/08/c08002 (cit. on p. 119).

[Kam11] Kamp, Thomas van de; Vagovič, Patrik; Baumbach, Tilo and Riedel, Alexander: "A Biological Screw in a Beetle’s Leg". In: *Science* 333.6038 (2011), pp. 52–52. doi: 10.1126/science.1204245. eprint: https://www.science.org/doi/pdf/10.1126/science.1204245. url: https://www.science.org/doi/abs/10.1126/science.1204245 (cit. on p. 96).

[Kon09] Konorov, I; Angerer, H; Mann, A and Paul, S: "SODA: Time distribution system for the PANDA experiment". In: *2009 IEEE Nuclear Science Symposium Conference Record (NSS/MIC)*. IEEE. 2009, pp. 1863–1865 (cit. on p. 94).

[Kop16] Kopmann, Andreas et al.: "UFO — a scalable platform for high-speed synchrotron X-ray imaging". In: *2016 IEEE Nuclear Science Symposium, Medical Imaging Conference and Room-Temperature Semiconductor Detector Workshop (NSS/MIC/RTSD)*. 2016, pp. 1–5. doi: 10.1109/NSSMIC.2016.8069895 (cit. on pp. 97, 99).

[Lan17] Langley, Adam; Riddoch, Alistair; Wilk, Alyssa; Vicente, Antonio; Krasic, Charles; Zhang, Dan; Yang, Fan; Kouranov, Fedor; Swett, Ian; Iyengar, Janardhan et al.: "The quic transport protocol: Design and internet-scale deployment". In: *Proceedings of the conference of the ACM special interest group on data communication*. 2017, pp. 183–196 (cit. on p. 34).

[Lei85] Leiner, Barry; Cole, Robert; Postel, Jon and Mills, David: "The DARPA Internet protocol suite". In: *IEEE Communications Magazine* 23.3 (1985), pp. 29–34 (cit. on pp. 33, 34).

[Man14] Mankar, Jayant; Darode, Chaitali; Trivedi, Komal; Kanoje, Madhura and Shahare, Prachi: "Review of I2C protocol". In: *International Journal of Research in Advent Technology* 2.1 (2014) (cit. on p. 24).

[Moh16] Mohr, Hannes: "Evaluation of GPU-based track-triggering for the CMS detector at CERN's HL-LHC". 2016 (cit. on p. 134).

[Moh17] Mohr, H. et al.: "Evaluation of GPUs as a level-1 track trigger for the High-Luminosity LHC". In: *Journal of Instrumentation* 12.04 (2017). 54.02.02; LK 01, p. C04019. DOI: 10.1088/1748-0221/12/04/C04019 (cit. on pp. 125, 130, 134, 149).

[Mun09] Munshi, Aaftab: "The opencl specification". In: *2009 IEEE Hot Chips 21 Symposium (HCS)*. IEEE. 2009, pp. 1–314 (cit. on p. 54).

[Pfi01] Pfister, Gregory F: "An introduction to the infiniband architecture". In: *High performance mass storage and parallel I/O* 42.617-632 (2001), p. 102 (cit. on p. 30).

[Pos80] Postel, Jon: RFC0768: User Datagram Protocol. 1980 (cit. on p. 33).

[Pur03] Purschke, ML: "The DAQ and online system of the PHENIX experiment at RHIC". In: *2003 IEEE Nuclear Science Symposium. Conference Record (IEEE Cat. No. 03CH37515)*. Vol. 2. IEEE. 2003, pp. 1151–1152 (cit. on p. 10).

[Rie15] Riechelmann, Max: "Optimizing high-speed data transfer and processing of DAQ systems with NVIDIAs GPUDirect RDMA". Karlsruhe, KIT, Institut für Technische Informatik, Dipl.-Arb., 2015. 2015 (cit. on p. 84).

[Ros04] Roshan, Pejman and Leary, Jonathan: 802.11 Wireless LAN fundamentals. Cisco press, 2004 (cit. on p. 17).

[Rot15] Rota, L.; Caselle, M.; Chilingaryan, S.; Kopmann, A. and Weber, M.: "A PCIe DMA Architecture for Multi-Gigabyte Per Second Data Transmission". In: *IEEE Transactions on Nuclear Science* 62.3 (2015), pp. 972–976. DOI: 10.1109/TNS.2015.2426877 (cit. on p. 72).

[Rot16] Rota, L.; Vogelgesang, M.; Perez, L.E. Ardila; Caselle, M.; Chilingaryan, S.; Dritschler, T.; Zilio, N.; Kopmann, A.; Balzer, M. and Weber, M.: "A high-throughput readout architecture based on PCI-Express Gen3 and DirectGMA technology". In: *Journal of Instrumentation* 11.02 (Feb. 2016), P02007–P02007. doi: 10.1088/1748-0221/11/02/p02007. url: https://doi.org/10.1088/1748-0221/11/02/p02007 (cit. on pp. 72, 148).

[Rub53] Rubinoff, Morris: "Analogue vs. Digital Computers-A Comparison". In: *Proceedings of the IRE* 41.10 (1953), pp. 1254–1262. doi: 10.1109/JRPROC.1953.274277 (cit. on p. 14).

[Sad04] Sadiku, M.N.O. and Akujuobi, C.M.: "Software-defined radio: a brief overview". In: *IEEE Potentials* 23.4 (2004), pp. 14–15. doi: 10.1109/MP.2004.1343223 (cit. on p. 135).

[San10] Sanders, Jason and Kandrot, Edward: CUDA by example: an introduction to general-purpose GPU programming. Addison-Wesley Professional, 2010 (cit. on p. 54).

[San14] Santos Rolo, Tomy dos; Ershov, Alexey; Kamp, Thomas van de and Baumbach, Tilo: "In vivo X-ray cine-tomography for tracking morphological dynamics". In: *Proceedings of the National Academy of Sciences* 111.11 (2014), pp. 3921–3926. doi: 10.1073/pnas.1308650111. eprint: https://www.pnas.org/doi/pdf/10.1073/pnas.1308650111. url: https://www.pnas.org/doi/abs/10.1073/pnas.1308650111 (cit. on p. 96).

[Spu00] Spurgeon, Charles E: Ethernet: the definitive guide. Ö'Reilly Media, Inc.", 2000 (cit. on p. 28).

[Ste18] Stevanovic, Uros: "A High-Performance Data Acquisition System for Smart Cameras in Science". 54.02.02; LK 01. PhD thesis. Karlsruher Institut für Technologie (KIT), 2018. 136 pp. doi: 10.5445/IR/1000079814 (cit. on p. 98).

[Tho88] Thompson, S.: "VGA—sign choices for a new video subsystem". In: *IBM Systems Journal* 27.2 (1988), pp. 185–197. doi: 10.1147/sj.272.0185 (cit. on p. 23).

[Tor22] Tortorella, Yvan; Bertaccini, Luca; Rossi, Davide; Benini, Luca and Conti, Francesco: "RedMulE: A Compact FP16 Matrix-Multiplication Accelerator for Adaptive Deep Learning on RISC-V-Based Ultra-Low-Power SoCs". In: *2022 Design, Automation and Test in Europe Conference and Exhibition (DATE)*. 2022, pp. 1099–1102. doi: 10.23919/DATE54114.2022.9774759 (cit. on p. 49).

[Vog12] Vogelgesang, Matthias; Chilingaryan, Suren; Tomy, dos Santos Rolo and Kopmann, Andreas: "UFO: A Scalable GPU-based Image Processing Framework for On-line Monitoring". In: *2012 IEEE 14th International Conference on High Performance Computing and Communication 2012 IEEE 9th International Conference on Embedded Software and Systems*. 2012, pp. 824–829. doi: 10.1109/HPCC.2012.116 (cit. on p. 98).

[Vog13] Vogelgesang, M; Farago, T; Santos Rolo, T dos; Kopmann, A and Baumbach, T: "When hardware and software work in concert". In: *Proc. IPACLEPCS'13* (2013), pp. 661–664 (cit. on p. 98).

[Wul95] Wulf, Wm A and McKee, Sally A: "Hitting the memory wall: Implications of the obvious". In: *ACM SIGARCH computer architecture news* 23.1 (1995), pp. 20–24. doi: 10.1145/216585.216588 (cit. on p. 45).

List of Figures

List of Tables

Listings

Acronyms

bps	Bits per Second
CERN	European Organization for Nuclear Research
CLB	Configurable Logic Block
CMS	Compact Muon Solenoid
CORBA	Common Object Request Broker Architecuter
CPU	Central Processing Unit
DAQ	Data Acquisition
DMA	Direct Memory Access
FPGA	Field Programmable Gate Array
HDL	Hardware Description Language
HLT	High Level Trigger
HPC	High Performance Computing
IP	Internet Protocol
IPS	Internet Protocol Suite

KARA	KArlsruhe Research Accelerator
L1T	Level 1 Trigger
LHC	Large Hadron Collider
LUT	Lookup Table
MAC	Media Access Control
OSI Model	Open Systems Interconnection Model
RDMA	Remote Direct Memory Access
RoCE	RDMA over Converged Ethernet
SDR	Software-defined Radio
SFP	Small Form-Factor Pluggable
SIMD	Single-Instruction-Multiple-Data
TCP	Transmission Control Protocol
UDP	User Datagram Protocol
UFO	Ultra fast X-ray tomography with Feedback control loops and Online reconstruction