Karlsruhe Institute of Technology

An Extensible Parallel Computing Framework for Ultra-Fast X-Ray Imaging

zur Erlangung des akademisches Grades eines

Doktors der Ingenieurwissenschaften

der Fakultät für Informatik
des Karlsruher Instituts für Technologie (kit)

genehmigte

Dissertation

von

Matthias Vogelgesang

aus Elsterwerda

Tag der mündlichen Prüfung:	18.11.2014
Erster Gutachter:	Prof. Dr. Achim Streit
Zweiter Gutachter:	Prof. Dr. Marc Weber

Ich versichere hiermit wahrheitsgemäß, die Arbeit bis auf die angegebenen Hilfen selbständig angefertigt, alle benutzten Hilfsmittel vollständig und genau angegeben und als kenntlich gemacht zu haben, was aus Arbeiten anderer und eigenen Veröffentlichungen unverändert oder mit Änderungen entnommen wurde.

Karlsruhe, den 17. Juni 2015

Matthias Vogelgesang

Zusammenfassung

Während das Moore'sche Gesetz weiterhin zu einer höheren Integrationsdichte und einer immer größeren Zahl an Transistoren auf einem Chip führt, entsteht durch die Verwendung von Architekturen mit immer höherer Parallelität eine Lücke zwischen der genutzten sowie der tatsächlich vorhandenen Leistung eines Mikroprozessors. Der Herausforderung, einerseits das Optimum an Leistung auf einem parallelen System herauszuholen, andererseits diese dem Entwickler und Nutzer möglichst transparent zur Verfügung zu stellen, widmet sich diese Arbeit.

Da sich viele Fragestellungen der Datenverarbeitung durch einen datenstromorientierten Ansatz beantworten lassen, wollen wir uns in dieser Arbeit auf eben diesen Anwendungsfall konzentrieren und Verbesserungen durch eine geeignete Softwarearchitektur sowie gezielten Optimierungen auf allen Ebenen erreichen. Als konkrete Anwendung betrachten wir hierbei bildgebende Röntgenversuche an Synchrotronstrahlungsquellen, die sich durch einen hochenergetischen Strahl und damit weitaus höherer räumlicher und zeitlicher Auflösung auszeichnen, als mit konventionellen Röntgenröhren möglich wäre. Die hierbei entstehenden Datenraten erfordern gerade die Nutzung hochparalleler Verarbeitungssysteme, sowie einer gezielten Optimierung der Datenakquisition und der Kontrollumgebung, um den Anforderungen aktueller Experimente zu begegnen.

Um das Problem der Hochdurchsatzdatenverarbeitung zu lösen, wurden im Rahmen dieser Arbeit Systemarchitekturen zur Aufnahme und Verarbeitung von Datenströmen auf parallelen und insbesondere heterogenen Systemen entwickelt, modelliert und evaluiert. Es wurden Methoden entwickelt um eine allgemeine Aufgabenbeschreibung auf einer heterogenen Rechenarchitektur abzubilden und diese optimal zu verarbeiten. Überdies wurden Ansätze aufgezeigt, die Aufgabenbeschreibung, unter Wahrung der Verarbeitungsleistung, weiter zu vereinfachen und mittels der gewonnenen Daten den Experimentaufbau in asynchroner Weise zu kontrollieren.

Im Ergebnis zeigt sich zum Einen, dass mit dem entwickelten Datenverarbeitungssystem in weicher Echtzeit sowohl tomographische Daten akquiriert werden können als auch auf einem heterogenen System aus CPUS und GPUS das Volumen zu rekonstruieren. Der nötige Aufwand für den Nutzer besteht dabei in der korrekten Formulierung seines Problems. Mit den aufgezeigten Systemarchitekturen haben wir die Grundlage für Experimente geschaffen, die sowohl Einblicke während der Datenaufnahme zulassen als auch einen intelligenten Experimentaufbau ermöglichen. Während bisherige Experimente von einer statischen Umgebung abhängen, können wir nun überdies den Experimentaufbau regeln.

Abstract

Moore's law stays the driving force behind higher chip integration density and an ever-increasing number of transistors. However, the adoption of massively parallel hardware architectures widens the gap between the potentially available microprocessor performance and the performance a developer can make use of. This thesis tries to close this gap by solving the problems that arise from the challenges of achieving optimal performance on parallel compute systems, allowing developers and end-users to use this compute performance in a transparent manner and using the compute performance to enable data-driven processes.

A general solution cannot realistically achieve optimal operation which is why we will focus on streamed data processing in this thesis. Data streams lend themselves to describe high-throughput data processing tasks such as audio and video processing. With this specific data stream use case, we can systematically improve the existing designs and optimize the execution from the instruction-level parallelism up to node-level task parallelism. In particular, we want to focus on X-ray imaging applications used at synchrotron light sources. These large-scale facilities provide an X-ray beam that enables scanning samples at much higher spatial and temporal resolution compared to conventional X-ray sources. The increased data rate inevitably requires highly parallel processing systems as well as an optimized data acquisition and control environment.

To solve the problem of high-throughput streamed data processing we developed, modelled and evaluated system architectures to acquire and process data streams on parallel and heterogeneous compute systems. We developed a method to map general task descriptions onto heterogeneous compute systems and execute them with optimizations for local multi-GPU machines and clusters of multi-GPU compute nodes. We also proposed an source-to-source translation system to simplify the development of task descriptions.

We have shown that it is possible to acquire and compute tomographic reconstructions on a heterogeneous compute system consisting of CPUs and GPUs in soft real-time. The end-user's only responsibility is to describe the problem correctly. With the proposed system architectures, we paved the way for novel *in-situ* and *in-vivo* experiments and a much smarter experiment setup in general. Where existing experiments depend on a static environment and process sequence, we established the possibility to control the experiment setup in a closed feedback loop.

Acknowledgements

The last three and a half years have been a lasting experience with many ups and certainly some downs. This thesis is the culmination of this time and would have not been possible without the help and support of so many kind people. First of all, I would like to thank Prof. Achim Streit and Prof. Marc Weber for supervising and reviewing my thesis. I would also like to thank my group leader Dr. Andreas Kopmann for setting an interesting research topic and having countless fruitful discussions with. I am grateful for the technical discussions I had with my colleagues Dr. Suren Chilingaryan, Dr. Michele Caselle, Uroš Stevanović and Timo Dritschler from IPE, Tomáš Faragó and Tomy dos Santos Rolo from IPS, with our Russian collaboration partners from St. Petersburg, Moscow and Tomsk as well as the Helmholtz groups from Hamburg and Dresden. I also have to thank my students and student assistants Andrej and Roman Shkarin, Timo Dörr, Yuemin Liang, Maria Matveeva and Sven Werchner for taking some of the work off of my back. Last but not least, I have to thank Anne for putting up with me for so long.

Contents

List of Figures

List of Tables

1 Introduction

Moore's law accurately predicts a doubling of the number of microprocessor transistors every one to two years [88]. For the first 40 years, the shrinking feature size and increasing number of transistors allowed the manufacturers to raise the clock frequency, the number of arithmetic floating point operations processed in a second and in total increase the complexity of a processor core. For this time an increase of compute power was solely technology-driven and allowed for an ever-increasing data throughput without additional software development efforts.

Since around 2004, technical obstacles such as leakage current and increasing difficulties to dissipate the heat stopped the trend for higher clock rates and faster Central Processing Unit (CPU) cores. As illustrated in figure 1.1, the stagnating clock frequency had a direct impact on the results of the single-threaded SPEC benchmarks [70]. As a consequence, microprocessor engineers used the ongoing technology scaling – from 65 nm in 2005 to 14 nm as of now – to develop multi-threaded and multi-core processor architectures. Besides multi-core CPUS, massively parallel accelerator architectures such as Graphics Processing Units (GPUS) and Field Programmable Gate Arrays (FPGAS) gained wide adoption in areas that require highly parallel data processing performance. Compared to CPUS, accelerators perform faster with large scale parallel data tasks and require less power for the same amount of work. However unlike general purpose CPUS they are limited to specific applications.

The focus on parallel compute architectures led to the "multi-core dilemma" that requires re-writing programs in order to gain noticeable performance improvements or even compensate for performance losses due to lower per-core clock frequencies. A loss of technology-driven performance increases means that "the free lunch is over" for programmers [120] who now have to adopt parallel programming models and technologies to satisfy application demands. For high performance applications, however, merely applying the parallel programming principles is not sufficient because any unused parallel potential has a direct impact on the performance according to Amdahl's law [3].

Up to now, there is no technological solution that allows programmers with varying parallel programming backgrounds to benefit from the performance of parallel accelerators and at the same time provide optimal performance on a wide range of parallel

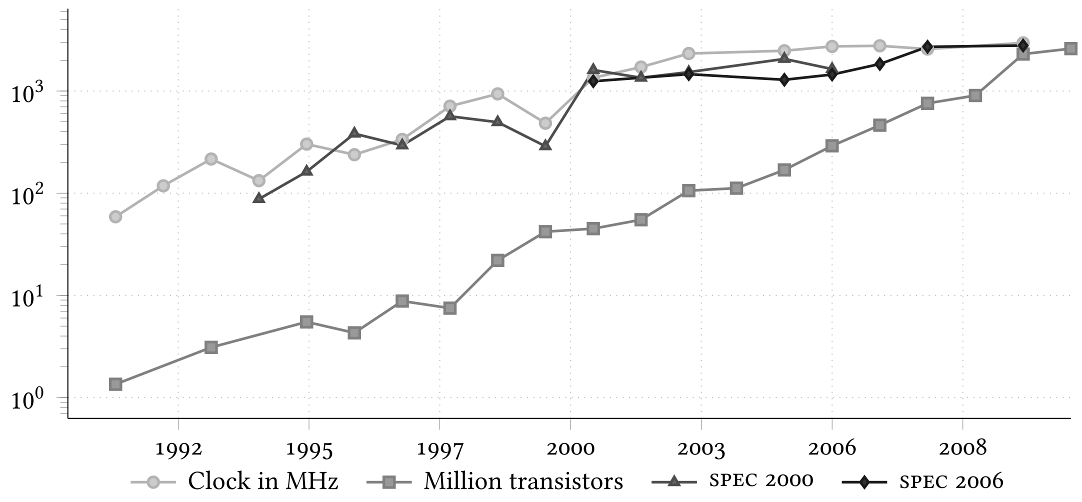


Figure 1.1 CPU clock frequency, number of transistors and SPEC benchmark floating point performance trend, with a clear convergence of frequency and single-threaded performance since 2004.[1]

architectures. If the abstraction level of the employed technology is too low, the programmer must make an effort and posses considerable hardware knowledge to exhaust the performance potential. If the technology level is too abstract, the overheads imposed by the abstractions will limit the maximum attainable performance. Because this dilemma cannot be solved in a general way, we will put the focus in this thesis on the specific class of *streamed* data processing on parallel architectures. Streamed processing covers a wide range of applications and is particularly suited for soft real-time data processing of large data sets.

In this thesis, we will apply the streamed data processing to X-ray imaging applications at synchrotron radiation sources which have particularly large requirements concerning image processing performance. Synchrotrons, such as KIT's Ångströmquelle Karlsruhe (ANKA), produce a powerful X-ray beam that is used for a wide range of analysis and manufacturing tasks. For example, ANKA's imaging beamlines TOPO-TOMO [104] and IMAGE use X-ray light to scan and analyse samples from life- [66] and material sciences [28] using non-destructive imaging experiments such as radio- and tomography. Unlike conventional video stream processing, X-ray imaging experiments have much higher demands on the acquired data rate, the flexibility of the data processing and the processed data throughput. With soft real-time data analysis in place, novel on-line experiments can use the experimental results to influence the experiment control itself. An important use case is the adaptive acquisition of data in order to reduce the X-ray dose applied to a sample to prolong the total scan time. Such experiments not only need

[1] With data from the public CPU-DB at `http://cpudb.stanford.edu` and official NVIDIA and Intel sources [29].

fast image processing but also a beamline control system that is able to supervise the entire experiment chain and integrate all aspects of data acquisition and processing. To support casual and expert operators, such a system must provide both a straightforward user interface as well as programmatic access.

In this thesis, we will present a systematic approach to leverage the performance potential of parallel, heterogeneous compute systems for experiment workflows with high performance demands. To accommodate for the large variety of heterogeneous environment parameters, we propose a generic and adaptive framework for optimized data processing. We also develop a generic, high-throughput data acquisition system and an asynchronous experiment control system to enable *soft* real-time data analysis and feedback-driven experiments using the high-performance compute system. A low-latency, high-throughput experiment pipeline allows us to implement novel *in-vivo* X-ray imaging experiments such as X-ray *cinetomography* [108]. In the remaining chapter, we will outline the problem space in detail and raise specific questions that need to be addressed in order to solve these problems.

1.1 Problem statement and research questions

X-ray imaging that uses a beam with an energy as high as at the IMAGE beamline and current detector technology permits data acquisition at a high sampling rate and a high spatial resolutions. On the one hand, this allows for the detection of small temporal and spatial changes that occur in dynamic processes and on the other hand allows to scan more samples per measurement phase than before. Current high-speed detectors can read out, digitize and transmit a full frame in milliseconds using vendor-specific Software Development Kits (SDKS) and a high-speed interconnect. However, there is no detector abstraction available that is capable of streaming at that data rate and provides a single abstract interface for all kinds of detectors. The latter is necessary to avoid hard application dependencies on specific detector functionality which may change with each new experiment setup.

Increasing the sample rate, spatial resolution and experiment throughput, requires larger compute resources to process the enlarged amount of data. Conventional data processing methods employed at synchrotron facilities do not utilize the available hardware resources to their full extent. Because the amount of image data cannot be processed *during* an experiment, acquisition and quality problems can only be revealed by subsequent data analysis long after the experiment has finished. Moreover, insufficient data processing prevents novel experiment types based on event-triggered feedback setups that depend directly on the data processing results.

Supporting fast, reactive experiments requires proper integration of high-throughput data acquisition, data processing as well as fast asynchronous device access on the experiment control level. None of the existing control systems provide the necessary integration nor the flexibility to model X-ray imaging experiments rapidly. Instead, *ad-hoc* solutions and inflexible setups are used to control an experiment. This approach prevents sophisticated experiment feedback loops and on-line monitoring of the data quality.

The operators of current control systems are trained personnel but often lack a software engineering background. Thus, to ease working with the experiment control system, experiment sequences are pre-defined and rarely changed. However, especially for X-ray tomography, a "rapid prototyping" approach is necessary to cope with unconventional experiment setups in a flexible way.

Based on these high-level problems, the three central questions that this thesis addresses are:

> *Which system architecture enables on-line data analysis and fast feedback X-ray imaging experiments and which components are necessary for optimal throughput and low latencies? Moreover, how can users and developers rapidly set up new experiment and data processing workflows?*

These central questions lead to a series of related problems that we are going to highlight in the remaining section. For each problem, we will pose specific research questions that need to be addressed in order to solve that particular problem.

Heterogeneous data processing

The majority of the parallel compute systems employed in the synchrotron context use homogeneous hardware architectures in order to process large amounts of data. Multi-threaded parallel computing on multi-core CPUS, GPUS and clusters of compute nodes is the dominant compute model for these system architectures. Neglecting the heterogeneity of mixed hardware architectures using CPUS *and* accelerators, however, prevents maximum throughput required for on-line monitoring and fast feedback purposes. A large body of research has already shown how to leverage heterogeneous compute systems for processing fixed-sized data or distribute a large number of processing tasks. *Streamed* data processing on heterogeneous compute systems, as required for X-ray imaging experiments on the other hand, has seen only minor investigations. Thus, to provide a strong foundation for heterogeneous computing with streamed processing in mind, we have to find appropriate architectural and compute models:

> *Question 1:* How can we model the flow of data streams and processing for use within a heterogeneous compute system?

The existing compute models cannot accurately estimate the run-time behaviour and suggest optimal resource allocation plans. Although system parameters such as CPU caches, slow data links and inter-processor affect the run-time and throughput of modern heterogeneous compute systems they are often not considered in large scale compute architectures. Moreover, because GPU and accelerator architectures such as Intel Xeon Phi are fundamentally different from CPU architectures, must be examined separately, thus:

> *Question 2:* How can we estimate system parameters and use them to improve heterogeneous data processing? How does this affect the system design itself?

Having a near-optimal heterogeneous architecture does not yield an optimally performing compute system that is required for an on-line imaging experiment. Thus, based on the compute model and estimated system parameters, we have to find and optimize pre-processing, reconstruction and post-processing imaging algorithms to give an answer to

> *Question 3:* What are the best performing implementations of imaging algorithms and technologies on existing and future compute architectures?

Generic, low-latency data acquisition

An X-ray imaging experiment requires a pixel detector with specific properties depending on the characteristics of the scanned sample or process. To cover a wide range of use cases and applications, a beamline operator usually chooses between detectors with high sensitivity and low frame rate or high frame rate and bad signal-to-noise ratio. Moreover, pixel sizes and pitches between pixels determine the resolution of spatial information and the dynamics. Due to a lack of software interface standards, each vendor uses a different approach to access these parameters. Thus in practice, to use n detectors with m client systems $m \times n$ different detector implementations are necessary.

Separating data acquisition and data processing on different machines improves the reliability and maintainability of the entire control system. Such a physical separation requires that the acquired detector data is streamed through the control system network. Streaming the data through ANKA's 10 Gigabit Ethernet (10GbE) Transmission Control Protocol/Internet Protocol (TCP/IP) network has two major drawbacks: First, the net bandwidth of a $10\,\mathrm{Gbit\,s^{-1}}$ connection does not suffice to transmit the data produced by the fastest detector and thus cannot be used for on-line and fast feedback purposes. Second, using the shared control network for full frame data transmission negatively affects other control processes which in turn could miss soft real-time requirements. Therefore we have to address

> *Question 4:* What does a detector interface for fast, generic and remote data acquisition look like?

Experiment control

A beamline experiment setup consists of hardware devices such as motors, shutters and detectors that are used to control beam parameters and acquire image data. To access these devices from a central location, state-of-the art synchrotron control systems model the entire beamline as a TCP/IP-based distributed system of compute nodes, each node accessing a particular piece of hardware. A client application sequences device access by communicating with these nodes to perform experiment processes. Even though a distributed system provides inherent asynchronous operation, the existing higher level experiment control systems were not designed to operate multiple devices in parallel. Such asynchronous device access, however, is mandatory to increase the experiment throughput.

Apart from a lack of parallelization, existing low-level control systems do not model devices in a semantically correct hierarchy of device families. Therefore, client applications cannot rely on common software interfaces to access different devices of the same family in a generic way. This prevents the development of maintainable software components for rapid experiment prototyping.

Although existing low and high-level control system provide mechanisms for rudimentary analysis of acquired system parameters and experiment data, none of them integrate high-throughput data processing to enable soft real-time on-line data analysis. On-line processing is a necessary to implement fast feedback loops for experiments that have to adapt to changes in the sample. Also, because the majority of data is processed off-line, any problems related to the experiment setup go unnoticed for the time of acquisition. To prevent these problems we need to address

> *Question 5:* How can we design an experiment control system to support experiments with high throughput demands and fast feedback loops?

Processing and control interfaces

Programming a heterogeneous compute system is a challenging task because different parallel programming models and Application Programming Interfaces (APIS) are required to build efficient programs for different multi-core architectures, clusters and GPU systems. To cover these platforms, a programmer must know about a low-level multi-threading API such as OpenMP for multi-core CPUS, MPI for cluster communication and an accelerator API such as CUDA or OpenCL to program a GPU. Considering the integration of such a high-performance compute system within a flexible experiment control system bears the following questions:

> *Question 6:* What does a low-level interface for a heterogeneous compute system look like and what is necessary to provide minimal interface for

developers and machines alike? How does the integration with a control system look like and how can a control system itself to ease the description of complex workflows?

With the previous question answered, an operator can easily *use* the system but not *extend* it. Especially the conceptual differences between a GPU and a CPU as well as the low-level APIS required to program GPUS prevent novice developers to modify and extend GPU-based algorithms. However, flexible experiment workflows also require flexible imaging algorithms and derived GPU implementations, hence:

Question 7: How can we provide high-level to modern GPU hardware?

1.2 Objectives and contributions

The problem statements and research questions provide a framework that set the goals for this thesis and limit its scope. In particular, we are going to present the results and contributions of the following objectives:

1. We will design and evaluate an open platform for parallel and distributed computing of data streams. The computing platform will target and scale with both locally available parallel hardware such as multi-core CPUS, many-core GPUS and small-scale clusters of individual nodes, which comprises typical imaging beamline hardware. We will also characterize system parameters for subsequent low- and high-level optimizations and provide a high-level integration for end-users.

   The UFO compute framework implements the proposed system design [130]. It uses novel mapping techniques to reduce the run-time by optimizing the utilization of multiple accelerator devices and compute nodes. The system is in use at ANKA and evaluated at DESY for X-ray imaging tasks [24, 69]. Despite being designed originally for X-ray imaging tasks, the system is currently being adopted for high-frequency X-ray beam diagnostics analysis [20] as well.

2. We will evaluate and optimize efficient imaging algorithms to pre-process X-ray images, reconstruct tomographic volumes and post-processes data on CPUS and GPUS for high throughputs as required by on-line experiments. The main objective is to find algorithms and algorithmic implementations that are suitable for on-line usage.

   Besides standard GPU optimizations for common pre- and post-processing tasks, we also integrated optimized algebraic and Fourier-based reconstruction algorithms into our heterogeneous compute framework [111, 112].

3. We will design a generic detector interface that uses low-level shared memory primitives for high throughput, a property system for generic access and a plug-in mechanism for modularity. It will be used to control arbitrary detectors and stream data at high speeds. On top of the generic interface, we will devise an extension for remote data acquisition.

   We designed and implemented this generic system as part of the UCA data acquisition framework. The remote extension uses a custom InfiniBand network component for high-throughput remote data acquisition [36]. Apart from accessing a dozen commercial detectors, it is used to control the UFO detector system [21, 117] and the grating-based phase contrast detector characterized at DESY's P07 beamline [82].

4. Based on modern asynchronous and concurrent programming paradigms, we will design and evaluate a control system for parallel device access and high-throughput data processing. The main objectives of this control system are on-line monitoring, fast feedback loops and rapid definition of new experiment and data processing workflows.

   The Python-based *Concert* control system integrates the UFO compute framwork and the Unified Camera Abstraction (UCA) acquistion system together with the low-level control systems [69, 131].

5. To aid developers writing GPU-based programs, we will propose a high-level source-to-source translation system that maps Python functions to functionally equivalent OpenCL kernels for execution on GPUs and other parallel accelerators. The main objectives are the reduction of code complexity and first-level optimizations on the functional description.

   We proposed a source-level annotation system used for just-in-time code generation. The system is used for quick prototyping of GPU code as well as fast execution of numeric Python code on a local multi-GPU machine.

The proposed system designs and subsequent results have also been presented at different workshops: The *Concert* control system was introduced at the *27th TANGO Workshop 2013* in Barcelona. The general architecture and implementation details of the data processing framework were presented at the *High Data Rate Initiative Workshop 2013* held at the German Synchrotron Radiation Source DESY and the *Science3D Workshop 2014* also held at DESY. Details on the tomographic reconstruction pipeline and the Python code generation were given in a talk for the *Fast Data Processing Workshop 2014* held at the CUDA Center For Excellence at Technical University Dresden.

The results and contributions of this thesis are part of the UFO-1 and UFO-2 BMBF research projects. Both projects are joint German-Russian collaborations between KIT's Institute for Data Processing and Electronics (IPE), Institute for Photon Science and Synchrotron Radiation (IPS), Laboratory for Applications of Synchrotron Radiation (LAS) and Institute of Experimental Nuclear Physics (IEKP) as well as the Shubnikov Institute of Crystallography in Moscow, the Saint-Petersburg State University for Civil Aviation and Tomsk Polytechnic University. Besides the topics covered by this thesis, UFO-1 and UFO-2 develop new X-ray imaging setups, detector systems, smart FPGA-based detectors and novel scientific applications and experiment types.

1.3 Outline

Chapter 2 reviews current technologies and performance models in the context of heterogeneous compute systems consisting of CPU and GPU as well as related work for streamed computing. We present X-ray imaging principles and algorithms giving background to the synchrotron use case that we evaluate in Chapter 6.

Chapter 3 describes the compute and hardware model for processing data streams on heterogeneous systems. We present algorithms to map data processing tasks to instruction-level parallelism, accelerator and multi-core CPUs thread-level parallelism as well as distributed task parallelism on clusters of distributed machines. We analyse the system based on an analytical model and optimized streaming-specific aspectes to improve the throughput on heterogeneous streaming systems.

Chapter 4 proposes a system architecture for fast data acquisition and the design of a system architecture for parallel experiment control. The data acquisition system reduces the number of memory copies between different sub-systems for low latency and high throughput. The control system uses the processing capabilities presented in chapter 3 to realize novel experiment types and improve the throughput of existing beamline processes. We use a tomography experiment as a case study for the full chain of data acquisition, data processing and control using online feedback.

Chapter 5 shows ways to simplify GPU programming and optimize of algorithms for current heterogeneous architectures. We will describe a compiler system that translates Python code to GPU programs and allows for rapid prototyping of GPU code suited for the system described in chapter 3. The compiler includes a run-time system which makes it also possible to compute existing numeric Python code on a local multi-GPU system. In the second part, we will present specific GPU optimizations concerning the reconstruction of tomographic data.

Chapter 6 presents quantitative results that characterize the proposed architectures from chapters 3 to 5. For this, we evaluate system constraints and measure the performance achieved with the data processing framework using real data. We also measure the impact of the optimization strategies we presented in chapter 3. We will estimate the improvements of an asynchronous control system by simulating experiment workflows and determine performance improvements gained by using the code generation tool.

Chapter 7 discusses the contributions and results from the preceding chapters in light of the objectives posed in 1.2 and compared with other systems proposed in the literature. Based on the discussion, we will give an outlook on possible future work and aspects that are not covered by this thesis.

2 Preliminaries and related work

Parallel and heterogeneous computing are the key factors for high performance data processing in the multi- and many-core era. In this chapter, we will review parallel compute and machine models and explore contemporary parallel hardware such as AMD and NVIDIA GPUS as well as the Intel Many Integrated Core (MIC) architecture. We will also review the OPENCL accelerator software interface. In the second part, we will introduce X-ray imaging principles and related technologies used in synchrotron X-ray imaging applications. We will present the foundations of a typical data acquisition chain, different reconstruction algorithms as well as filters and algorithms used in pre- and post-processing stages.

2.1 Parallel computing

Since the emerge of microprocessor-based compute systems in the early 1970s, programmers took advantage of ever-increasing performance gains made possible by the increasing clock frequency predicted by Moore's law [88]. Around 2005, technical problems with CPU architectures such as difficult heat transfer and current leakage stopped the development of higher clock frequencies and automatic performance increases. These problems caused a shift from complex uni-core microprocessor architectures to multi-threaded and multi-core processor designs [120]. In hindsight, parallel computing became an ubiquitous mainstream technology [17].

The parallel computing paradigm enables potential compute performance improvements by subdividing a large problem into smaller sub problems. By processing these smaller problems simultaneously on multiple processors, the overall compute time is reduced [124]. *Task parallelism* attains an execution speed up by decomposing a larger task into multiple independent tasks that are distributed across multiple processors for concurrent execution. *Pipeline parallelism* is a special form of task parallelism and achieves compute speed ups by overlapping the concurrent execution of dependent tasks that process a single data stream. Unlike task and pure pipeline parallelism, *data parallel* algorithms split a single large data item into smaller sized data that can be processed independently by the *same* task. This task is executed with a particular data item on one of the multiple processors [119]. Data parallelism helps to improve the *latency* for fixed-size data by reducing processing time and improve the *throughput* by increasing the amount of data that can be processed at the same time.

2.1.1 Compute and communication models

The Parallel Random Access Machine (PRAM) is a theoretical shared memory machine model that extends the random access register machine with an arbitrarily large set of p independent processors. Each processor can access the same memory instantaneously but a concurrency protocol may restrict read or write accesses. Such a protocol postulates which and if a processor has exclusive read or write access to a specific memory location [40]. Unlike a distributed system, the PRAM model does not model the time required to move data from one processor to another, thus assuming instantaneous access to a memory location.

Because of the assumptions of an unlimited number of processors and instantaneous access, the PRAM model is insufficient to describe real world compute systems. A more realistic compute model assumes a *fixed* number of processors $p \in P$, a defined execution time for a given task size on a single processor and a fixed communication time required to move data between two processors [76]. More formally, each processor $P_i \in P = \{P_1, \ldots, P_p\}$ requires T_i time units to compute a single computational task unit. For convenience, we define a function $T_i(n) := n \cdot T_i$ which denotes the total amount of time required to compute n task units. The *speed* s_i of processor P_i arises from the reciprocal of the computation time. The relative speed of each processors is obtained by normalizing the set of processor speeds $S = \{s_1, \ldots, s_p\}$ by constraining $\sum_i^p s_i = 1$ [76].

Given a problem of task size N, a task parallel strategy tries to find the best partition $\vec{n} = (n_1, \ldots, n_p)$ where $\sum_{i=1}^p n_i = N$, to minimize the total execution time. Assuming independent tasks and processors, the slowest processor bounds the total execution time by

$$T_p(\vec{n}) = \max_p T_i(n_i). \tag{2.1}$$

Per definition, executing all tasks of a partition $\vec{n}$ on a single machine takes

$$T_1(\vec{n}) = \sum_i^p T_1(n_i) \tag{2.2}$$

time units. Without loss of generality, we will use T_p and T_1 to denote compute times for tasks with fixed size parameter if the size is of no interest.

In parallel systems without shared memory, processors can exchange data only by sending messages through a connection link. The time for preparation, sending, transferring and receiving data incurs a communication overhead of $T_{i,j}$ time units between processors P_i and P_j. This overhead, and therefore the total communication time, depends on the size of the message m given in Bytes (B) and the communication link between P_i and P_j.

Scalability

The main purpose of parallel computing is the accelerated execution of a program. The *speed up* metric describes the improvement of a parallel implementation compared to its sequential implementation. More generally, the speed up measures any improvement that is the result of some optimization. In the parallel computing domain, we have to distinguish between *absolute* and *relative* speed ups. The absolute speed up compares the best *sequential* performance with the best parallel implementation on p processors. The *relative* speed up compares the execution of the *same* parallel implementation on a single and up to p processors. In general, the speed up $S(p)$ for p processors is given by

$$S(p) = \frac{T_{\text{initial}}}{T_{\text{optimal}}} = \frac{T_1}{T_p} \tag{2.3}$$

In a homogeneous system, $T_i = T_j$ for all $i, j \in \{1, \dots, p\}$. Thus the time to compute a problem on p processors is given by

$$T_p = \frac{T_1}{p}. \tag{2.4}$$

In this case, a hypothetical *perfect* speed up $S(p) = T_1/T_p = T_1/T_1/p = p$ that increases linearly can be obtained. The efficiency metric $E(p) \in [0, 1]$ of a parallel system characterizes the attained speed-up in terms of the hypothetical perfect speed up by

$$E(p) = \frac{S(p)}{p}. \tag{2.5}$$

Amdahl recognized that the sequential parts of a parallel program limits the speed up of the entire program [3]. Let $\alpha \in [0, 1]$ denote the program fraction that can be executed in parallel, then the time to compute a program is given by

$$T_p = (1 - \alpha)T_1 + \alpha\frac{T_1}{p}. \tag{2.6}$$

From that equation it follows that the speed up converges towards a limit

$$S(p) = \frac{T_1}{T_p} = \frac{T_1}{T_1\left((1-\alpha) + \alpha\frac{1}{p}\right)} = \frac{1}{1 + \alpha\left(\frac{1}{p} - 1\right)} \overset{p\to\infty}{=} \frac{1}{1-\alpha} \tag{2.7}$$

that depends on the sequential fraction of a program. Thus, the larger the sequential fraction of a program is, the smaller is the gain reached by adding more processors.

As we can see, the speed up is a measure to characterize the *scalability* of a parallel algorithm, which means how much better an algorithm performs if more processors are added to the system. *Strong* scaling, to which Amdahl's law applies, assumes that *all* processing units process a *fixed* amount of data. Thus in a perfectly strong scaling system, doubling the number of processors halves the execution time. On the other hand, the *weak* scaling model assumes that the amount of data *increases* at the same rate as the number of processors increases. Thus, a perfectly weak scaling system processes twice the amount of data without increasing the execution time. For a weak scaling system, Gustafson [50] quantifies the speed up as follows:

$$S(p) = \frac{(1-\alpha) + p\alpha}{(1-\alpha) + \alpha} = (1-\alpha) + p\alpha \tag{2.8}$$

Amdahl's and Gustafson's laws limit the algorithmic speed up for additional processors to a theoretical upper bound. In practice, external factors such as scheduling overheads, input/output (I/O) latencies and communication costs also limit the effective speed up that can be obtained on real systems. Thus when designing a parallel compute architecture, these factors have to be taken into account too.

2.1.2 Heterogeneous computing

A heterogeneous compute system consists of more than one type of processor that executes a *single* application at a time [76] in order to improve either latencies or throughput. Therefore, heterogeneous computing is a special case of parallel computing. An older, more restricted definition, limits the definition of heterogeneous compute systems to networks of heterogeneous machines thus excluding contemporary CPU-GPU machines [68]. In this thesis, we will concentrate on higher level heterogeneous systems which consist of a set of distinct processors that communicate locally via Peripheral Component Interconnect eXtended (PCIe) or remotely via a communication network, often referred to as *distributed computing*. Compared to homogeneous compute systems, heterogeneous systems can in principle designate the most appropriate processing resources to a certain sub-task of a problem. For example, FPGAS may acquire and pre-process raw data while subsequent GPU and CPU stages process massively parallel or branch-heavy problems. This requires that the problems can be partitioned properly.

Compared to the simplistic models from 2.1.1, we also have to take the time into account that is required to transfer data between two processors. Two major models estimate the transfer time $T_{i,j}$ between two processors P_i and P_j, the Hockney and the LogP model classes. Hockney's peer-to-peer communication model assumes a linear relationship between message size m and the time required to send it [55]. Given a startup latency $L_{i,j}$ and a maximum link bandwidth $B_{i,j}$ between P_i and P_j, the total time to transfer messages with a size of m bytes is

$$T_{i,j}(m) = L_{i,j} + \frac{m}{B_{i,j}}. \tag{2.9}$$

By sending messages of different sizes, the parameters $L_{i,j}$ and $B_{i,j}$ can be found by computing the linear regression of the bandwidth. For real distributed systems, the assumption of a strong linear relationship between message size and transfer time is not precise enough because the overhead of sending small messages compared to large bulk transfers is not considered.

The second class is the LogP model family first introduced by Culler [26]. The basic LogP model assumes an upper bound on latency L for small messages, an overhead o required for a processor to transmit or receive a message during which it cannot do anything else, the minimum gap interval g between two consecutive messages and the number of processors P. In the original model, the transfer time between two processors is given by

$$T_{i,j}(m) = L + 2o. \tag{2.10}$$

Alexandrov extended the LogP model by adding the gap per byte parameter G to capture arbitrarily sized message transmissions [1]. In this model $T_{i,j}$ is given by

$$T_{i,j}(m) = L + 2o + G(m-1) \tag{2.11}$$

Bandwidth and throughput

The performance of a communication channel between two processors P_i and P_j depends on the physical frequency *bandwidth* of the communication link. In data networks as well as in the remainder of this thesis, the term bandwidth denotes the *data rate* at which this communication link can transfer [101]. The achieved bandwidth not only depends on the employed physical layer technology but also on the number of bytes m that are transferred. Knowing this and the time required to transfer m bytes from P_i to P_j using one of the data transfer models, the bandwidth $B(m)$ is approximately

$$B(m) = \frac{m}{T_{i,j}(m)}. \tag{2.12}$$

For large m, the bandwidth converges towards the channel's maximum possible bandwidth B. Contrary to the physically possible bandwidth, the *throughput* of a system denotes the data rate perceived at the application level. This includes data transfer *and* data processing rates. The latter metric is thus useful to determine the performance of an algorithm's implementation that cannot be analysed in terms of floating point operations per second (FLOP s^{-1}). In this case, the processing throughput $B_i(n) = \frac{n}{T_i(n)}$ is a reliable metric for the performance of an algorithm. For certain algorithms we have to distinguish between the *inbound* and the *outbound* processing bandwidth. Inbound denotes the throughput measured in terms of processed *input* bytes, whereas outbound denotes the throughput measured in terms of *produced* bytes.

2.1.3 Streamed computing

A streamed data processing system processes a potentially infinite stream of data items using a fixed set of stages or *filters* arranged in an ordered *pipeline* structure. This model is particularly useful for data processing tasks such as audio or video transcoding as well as flexible transformation of line-based data such as UNIX pipes [105]. Except for the immediate data input, filter stages are entirely independent of their adjacent neighbour filters. This makes a pipeline ideal for task parallel execution of pipeline stages.

A pipeline that processes $n < \infty$ items with k filters requires

$$T_1 = n \cdot k \tag{2.13}$$

time units to process the entire stream on a single processor. On a PRAM, it takes k iterations to fill the pipeline, $n - k$ steps until the first filter processes the last item and $k - 1$ iterations until the last item is fully processed by all filters, hence for $p = k$ the parallel execution time is given by

$$T_p = k + n - 1. \tag{2.14}$$

Applying (2.3), the speed up $S(p)$ converges towards

$$S(p) = \frac{T_1}{T_p} = \frac{n \cdot k}{k + n - 1} \overset{n \to \infty}{=} k. \tag{2.15}$$

This performance model assumes that all filter tasks require equal amount of time to process their data items. In reality, however, the slowest filter will limit the pipeline throughput. Moreover, this model only considers the task parallel part of pipeline parallelism where in fact stages can be replicated for internal data parallel execution. Thus for a PRAM with $p = kn$ and n times replication of the pipeline, the entire stream can be processed in k time steps with a theoretical speed up of n. We will investigate a similar but weaker assumption in the next chapter to increase the overall parallelism on heterogeneous compute systems.

2.2 Parallel hardware architectures

Flynn's classification of hardware architectures considers the number of instructions that are processed simultaneously on a number of data items [43]. In total, four classes can be distinguished: Single Instruction Single Data (SISD), Single Instruction Multiple Data (SIMD), Multiple Instruction Single Data (MISD) and Multiple Instruction Multiple Data (MIMD). SISD architectures are classic uni-core architectures in which one instruction processes exactly one data item. A SIMD machine executes a stream of single instructions on multiple data items in parallel. This is the vectorized compute model of specialized instruction sets such as Intel's Streaming SIMD Extensions (SSE) in SISD processors. MIMD architectures execute multiple instructions on different streams of data. This model is implemented in multi-core processors with SIMD instructions, distributed systems and Very Long Instruction Word (VLIW) processors.

Flynn's coarse taxonomy explains microprocessor features found across different architectural levels. On the lowest level, Instruction-level Parallelism (ILP) describes the potential parallelism *within* the instruction stream: *In-order* instruction pipelines allow overlapping of executed instructions whereas *out-of-order* pipelines execute instructions truely parallel on superscalar execution units. Moreover, *speculative* execution issues instructions of different conditional branches in parallel and discards the result of the branch whose condition turned out to be false. The complexity of these execution units and the difficulties of compilers to find parallelism for VLIW processors within a program limit the potential parallelism within a single CPU core; a problem that is often referred to as the ILP *wall* [5].

2.2.1 Multi-threading on multi-core architectures

Instead of using chip area to support more fine-grained ILP, current microprocessor technologies try to emphasize thread-level parallelism (TLP). The TLP paradigm simplifies the single CPU core by replicating the logic and resources as multiple CPU cores. This moves the burden of parallelization from the compiler to the operating system and the programmer. On the microprocessor level, a *thread* is an abstraction for the execution of instructions and consists of its own register state; in particular its own copy of the instruction and the stack pointer.

Simultaneous Multithreading (SMT) is a hardware multi-threading technique to execute instructions of multiple threads in a single clock cycle using replicated control and execution units and was first introduced with Intel's NetBurst microarchitecture in 2003 [74]. Rather than true concurrency, SMT allows a thread to occupy execution units if another thread is waiting on a resource. For example, if a thread must access the slow Double Data Rate (DDR) memory in of a cache miss, another thread may execute on the unused Arithmetic Logic Unit (ALU) of a core. The increased utilization of core resources improves the overall run-time performance. On the other hand, if all threads are exclusively CPU or I/O bound, contention for shared resources will negatively impact the run-time performance.

Replicating the whole processing core within the same chip, results in chip-level multi-processing. These *multi-core* architectures are the dominant CPU feature since the introduction of the Intel Core Duo line of chips. Multi-core architectures are a special case of Symmetric Multiprocessing (SMP), in which separate chips share common resources. On both multi-core and SMP systems, multiple threads run independently from each other, with exclusive access to the resources provided by the core a thread runs on. The difference is that classic SMP systems are built from multiple CPU dies whereas a multi-core architecture is housed within the same die.

Parallel programming

Compared to the hardware definition of a thread, an operating system thread has specific properties that are visible only to the application programmer. A software thread is always part of a process and shares memory and other kernel resources with all other threads of the same process. In the absence of hardware multi-threading, concurrent execution can be emulated by *cooperative* or *preemptive* multi-threading. Threads in a cooperative operating system deliberately suspend their execution to yield for other threads. A preemptive operating system rapidly switches between different threads giving the illusion of concurrent execution.

On a real multi-threaded operating system, low-level software abstractions such as POSIX threads (Pthreads) on UNIX systems [19] or the Win32 API provide access to the operating system thread primitives. To prevent *deadlock* and *race condition* problems that are typical for parallel programming [97], operating systems and higher level APIs allow programs to enforce separate memory spaces by communicating via message passing [53] and use synchronization primitives. The latter includes waiting for a condition or the completion of a thread as well mutex or semaphore data structures to lock access to a resource. Higher-level multi-threading models such as Open Multi-Processing (OpenMP) [77] reduce these problems by moving multi-threading into the language specification itself. The programmer marks appropriate sections of the code as parallelizable, for which the compiler then generates code that is responsible for thread management and data partitioning. Although these programming models cannot prevent race conditions in all cases, the compiler can in principle warn about suspicious code.

2.2.2 Many-core architectures

GPUS are multi-threaded hardware architectures that consist of a large number of compute cores, typically an order to two more than CPUS. Compared to CPU memory, GPU cores can access their on-board memory with much a higher memory bandwidth. However, the data must be transferred first from CPU to GPU memory through a PCIe con-

Table 2.1 Development of NVIDIA microarchitecture specifications from 2006 until 2012. SHM denotes shared memory per SM.

Year	Architecture	SM	SP/SM	SFU	LD/ST	SHM/SM	IEEE 754
2006	G80	16	8	2	—	16	1985
2010	GT200	16	15	2	—	16	1985
2011	Fermi	16	32	4	16/16	16–48	2008
2012	Kepler	15	192	32	32/32	16–48	2008

nection. Unlike CPU cores, GPU cores are simpler in design and focused on arithmetic throughput rather than complex logic. The larger number of cores and the distinctive memory address space require a specific programming model that is not compatible with current CPU architectures and existing multi-threaded approaches. Rather than programming separate multiple threads for each core, a programmer must write GPU programs with SIMD execution in mind. This requires that single instructions are mapped to all cores at the same time.

NVIDIA architecture

The NVIDIA GPU architecture arranges Texture/Processor Clusters (TPCs) in a processor array. Each TPC consists of a texture unit for fast interpolated data access and a number of Streaming Multiprocessor (SM) units that are managed by an SM controller. An SM consists of a number of single precision Streaming Processor (SP) cores, each composed of a 32-bit integer ALUs, a single and double precission Floating Point Units (FPUs), units for loading and storing data (LD/ST), a Special Function Units (SFUs) for fast computation of transcendental operations such as the exponential, logarithm and trigonometric functions and a block of shared high-speed on-chip memory. Table 2.1 shows the development of the NVIDIA microarchitecture features and how Moore's law had direct impact on the number of cores and available cache memory.

An SM follows the SMT execution principle that runs hundreds of threads of different programs in parallel. An SM executes a number of threads using their own execution and register state and is therefore able to execute independent code paths [80]. To efficiently schedule instructions, the SM scheduler groups 32 threads into *warps*. Each thread within a warp executes the same instruction in lock-step fashion with all the other threads but using its own register state. Within a warp only threads that satisfy a condition are executed at the same time while the rest is idle. This serialization of branched instructions causes performance degradation of heavily branched GPU code.

AMD architecture

AMD's pre-Graphics Core Next (GCN) architecture was based on a VLIW design. This design is well-suited for three-dimensional graphics operations but did not fully utilize the ALUS for typical compute workloads. This is due to the fact that a single VLIW instruction encodes four operations at once but not all of them may be executed concurrently. In compute-intensive workloads, stricter data dependency requirements can eventually reduce the number of parallel instructions.

The GCN successor microarchitecture replaced the VLIW architecture with a multithreaded approach focused on SIMD operations similar to the NVIDIA microarchitectures. A GCN GPU features a number independent Compute Units (CUS). Each CU consists of a single Scalar General Purpose Register (SGPR), four Vector General Purpose Registers (VGPRS), a L1 data cache and a 64 kB Local Data Storage (LDS). A VGPR is comprised of 16 Processing Elements (PES) and processes multiple data in SIMD fashion whereas the scalar unit is used for scalar instructions or memory fetch operations. Four CUS share a 16 kB read-only L1 scalar cache and a 32 kB L1 instruction cache.

Intel MIC architecture

Intel's MIC architecture originated from the former Larabee and Single-chip Cloud Computer (SCC) research projects [60, 110]. The Larrabee architecture was intended as a GPU replacement but is in fact a CPU-GPU hybrid. It featured two conventional CPU cores with out-of-order instruction pipelines and ten Pentium cores with simple in-order pipelines. To improve the performance of the older Pentium architecture, each core featured an additional 512-bit wide vector engine for highly parallel SIMD computation. The SCC research prototype had 48 Pentium cores for conventional CPU tasks. The entire chip is structured as a grid of 6×4 tiles, each tile covering two CPU cores. The cores communicate through message passing primitives thus the whole chip architecture resembles a cluster of single core machines with additional benefits from shared caches.

The Intel MIC architecture that originates from both projects shares features of these architectures, mainly the large number of simple Pentium cores. Similar to the Larabee processor, it features up to 60 Pentium cores, each with 32 512-bit vector registers which can in turn process up to 16 single precision operations per core per cycle. High-end models achieve a memory bandwidth of up to 320 GB s^{-1}. The MIC architecture is currently available as Intel Xeon Phi accelerator cards that are connected to a host CPU through a PCIe connection.

Analysis

From table 2.2, we can see that the heterogeneous accelerators outperform the Xeon CPU and achieve a peak floating point performance that is in the same order of magnitude. The gap between peak memory bandwidth and peak performance, often called

Table 2.2 Number of cores, memory bandwidth and peak floating point performance for a CPU and different accelerators. The last column compares the memory bandwidth from the second column with the single and double precision performance measure.

			GFLOP s^{-1}		FLOP B^{-1}	
Device	Cores	GB s^{-1}	Single	Double	Single	Double
Xeon E5-2690 CPU	8	51	371	186	7.27	3.65
GTX 580	512	192	1580	198	8.23	1.03
GTX 680	1536	192	3090	128	16.03	0.66
GTX TITAN	2688	288	4500	1312	15.63	4.56
Tesla K20x	2688	250	3950	1310	15.80	5.24
Firepro W9100	2816	320	5240	2620	16.38	8.19
Xeon Phi 5110P	60	320	2021	1010	6.32	3.16

the memory wall [137], becomes even more apparent in these architectures. This leads to the critical situation that the accelerators could process algorithms with large memory demands but cannot provide the data fast enough. This becomes a severe problem for applications with large memory demands and low computational intensity on high-performance accelerators. For example, a GTX 680 must compute 16 floating point operations for each byte that is transferred to GPU memory in order to saturate core peak performance and memory bandwidth.

2.3 GPU computing

As seen in figure 2.1 and outlined in section 2.2.2, a GPU performs an order of magnitude more floating point operations per second than a conventional CPU. Early applications leveraged the GPU performance potential by programming the pixel processing pipelines directly to perform arithmetic operations on data encoded in pixel textures [96]. The pixel pipelines were designed specifically for graphics operations and therefore adoption of GPUs was limited to applications that mapped well to the graphics paradigm. To ease usage and open up new application domains, GPU vendors extended the microarchitectures for full programmability [93]. Early research projects such as Cg [86] and Brook [18] abstracted the hardware details behind programming interfaces. Eventually, the vendor-neutral industry standard Open Compute Language (OpenCL) and NVIDIA's proprietary Compute Unified Device Architecture (CUDA) API emerged as two competing *de facto* standards. Unlike CUDA, which is limited to NVIDIA GPUS, OpenCL targets arbitrary accelerator devices [46] such as multi-core CPUS with SIMD processing units, GPUS by all vendors, Intel's Xeon Phi or FPGAS from Altera. Despite its device-agnostic programming model, OpenCL's API follows the microarchitecture im-

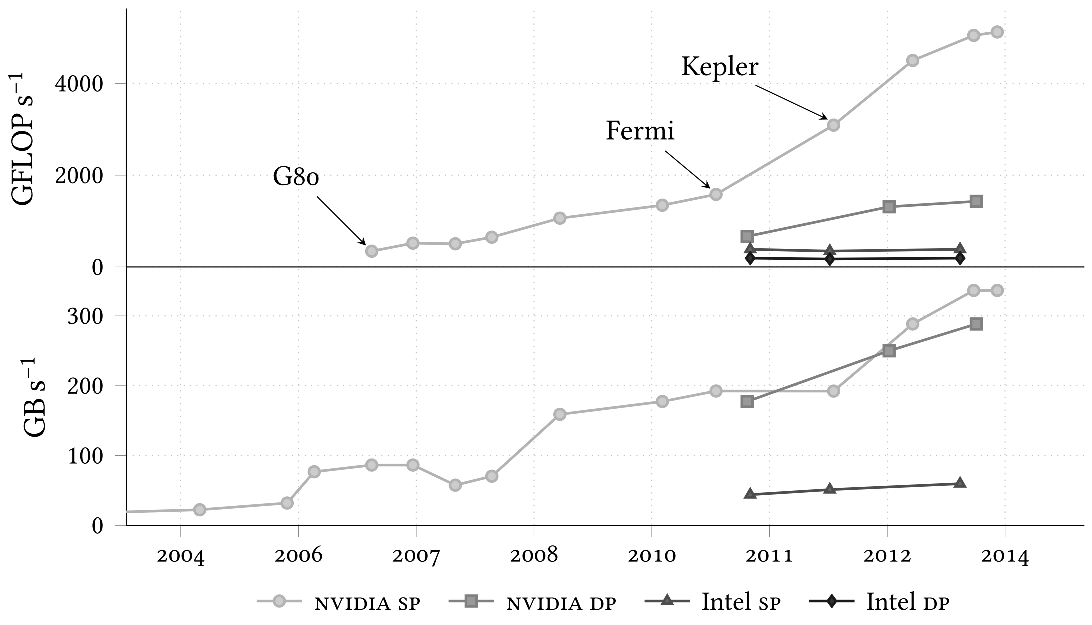


Figure 2.1 Performance and bandwidth gap between NVIDIA GPUs and Intel CPUs for single (SP) and double precision (DP) floating point operations.

posed by GPUs and resembles to a large degree CUDA.

Although CUDA and OpenCL currently account for the majority of GPU-accelerated programs, the development of higher level languages indicates the need for a simpler and less error-prone access to GPU performance. For example, OpenACC [136] and OpenMP 4 use the well-known pragma-based approach to shift the parallelization towards the compiler and reduce the burden on the programmer. The adoption of OpenMP for multithreaded programming has shown that this approach is a viable alternative to the lower level libraries. Nevertheless, at the moment the higher-level solutions are not yet mature enough for production, which is why we will focus on OpenCL in this thesis.

2.3.1 OpenCL programming model

A heterogeneous OpenCL system consists of the *host*, which is any regular C or C++ program accessing the OpenCL API and one or more abstract platforms. Each platform exposes *devices* that are attached to the host system and supported by a particular vendor. Each device consists of one or more CUs and each CU of one or more PEs. The host uses the OpenCL C API to communicate with the OpenCL run-time system and initiates execution of device programs written in OpenCL C. Within a single platform, the application programmer creates one or more *contexts* to group devices according to application demands. Devices of the same context can share memory and *event* information. Memory buffers are not specific to a device but transferred transparently to the device that

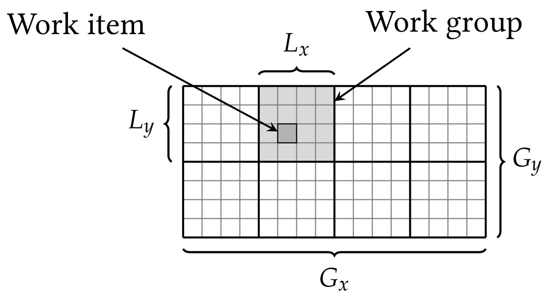


Figure 2.2 Work items on a global 16×8×1 grid, arranged in work groups of size 4×4×1.

wants to access it. To communicate with the attached devices, the programmer creates one or more *command queues* per device. These queues are used to address a device to submit commands for transferring data and launching programs.

GPU programs consist of a number of *kernel* functions written in OpenCL C, a subset of standard ISO C99 [102]. At run-time, the OpenCL C compiler compiles the kernel for a list of devices into device-specific byte or native code. For example, on NVIDIA platforms the code is compiled into device-specific Parallel Thread Execution ISA (PTX) code whereas the Intel compiler directly emits x86 machine code. The application programmer sets the required kernel parameters and submits a kernel launch command to the command queue of the device where the kernel should be executed.

2.3.2 Execution model

A GPU kernel is a function that describes a task in a data parallel, Single Program Multiple Data (SPMD)-like fashion. Although these functions are executed in lockstep by the CU their instruction paths may diverge due to branching. In this case, the hardware schedulers execute subset of threads, hence the execution model is more strictly classified between SIMD and SPMD [39]. Common algorithmic patterns that use this massively parallel execution model include parallel scatter, gather, map and reduce operations [96]. From these basic patterns, most data parallel algorithms can be derived straightforwardly. For example, a sequential *for* loop that does not posses any data dependencies can be replaced by a parallel map operation.

In OpenCL, GPU programs are launched by submitting a kernel on a multi-dimensional index space $\vec{G} = (G_x, G_y, G_z) \in \mathbb{N}^3$. Each grid point – or global ID $\vec{g} \in G_x \times G_y \times G_z$ – uniquely identifies one of the $G_x \cdot G_y \cdot G_z$ *work items* that execute the kernel function. Using the global ID and a global offset $\vec{F} \in \mathbb{N}^3$, a work item addresses the task or data item that it is working on. For example, in an image processing task each work item may use the global ID to adress the input and output pixel of an image. The global grid is further sub-divided into *work groups* of size $L_x \times L_y \times L_z$ that are addressed by their work group ID $\vec{W} \in \mathbb{N}^3$. Within a single work group, a work item is uniquely identified by a local ID $\vec{l} \in L_x \times L_y \times L_z$. The global ID corresponds to a linear combination of work group ID and local ID as shown in figure 2.2 and given by

$$\vec{g} = \vec{w}\vec{L}^T + \vec{l} + \vec{F}. \tag{2.16}$$

After submitting a kernel launch command to a command queue, the OpenCL run-time decides autonomously on which CU a work group and on which PE a work item is scheduled. Splitting functionality into concurrent work items and kernels, enables three levels of parallelism: Coarse-grained task parallelism with multiple devices, fine-grained task parallelism using concurrent kernel execution on the same device and fine-grained data parallelism through massively parallel SIMD execution of a single kernel.

2.3.3 Memory model

The memory model of OpenCL distinguishes between host and device accessible memory and requires explicit data transfers between those address spaces. Because kernels cannot return values on the call stack, the communication back to the host must happen through shared global memory.

Similar to the CPU memory system, the device memory is arranged in a hierarchy of memory levels. Each level is characterized by a different scope, access, size and access latency as shown in table 2.3. Generally, *global* memory is used to transfer data between host and device using explicit memory copy operations or memory mapping. With memory mapping, the OpenCL run-time maps a memory buffer on behalf of the driver to a user space memory location. The host program writes into this buffer and unmaps it when done. After unmapping, the host cannot access the data cannot anymore and the run-time transfers the data back to the device. Within the same work group, work items can share data through *local* user managed on-chip memory, which is usually orders of magnitude faster than global memory. Local variables are private to a work item and stored in on-chip registers, which provide the highest memory bandwidth available.

A special type of global memory is an image object. This object represent two- or thee-dimensional arrays of structured elements such as pixels with a specific color format. Looking up a pixel value from an image leverages GPU texture units to perform fast hardware interpolation. However, OpenCL images are created and accessed using special function calls and therefore are not directly compatible with ordinary OpenCL memory buffers.

2.3.4 Synchronization

To prevent race conditions affecting the correctness of the code, OpenCL provides mechanism to synchronize concurrent kernel and work item execution. The command queue structure ensures consistency of submitted commands by stipulating a policy on the order of kernel execution: the result of a kernel operation k_1 is visible to operation k_2 if

Table 2.3 Mapping of OpenCL memory locations to specific NVIDIA hardware features [94].

Memory	Where	Cached	Access	Scope	Lifetime
Register	on-chip	–	R/W	work item	work item
Local	on-chip	–	R/W	work group	work group
Global	off-chip	no	R/W	all work items + host	host allocated
Constant	off-chip	yes	R	all work items + host	host allocated
Texture	off-chip	yes	R	all work items + host	host allocated

and only if k_1 was enqueued before k_2. To ensure correct ordering of commands that were submitted 1) on different queues of the same context, 2) to an *out-of-order* queue or 3) to multiple queues accessing the same device, the application programmer can specify the order by chaining *events*. An event is implicitly created by API calls that enqueue a command such as kernel launches or data transfers on a queue. Event objects can then be passed to other enqueue commands which must wait until the passed event object has finished. To synchronize multiple kernels with an external trigger, user-defined events are marked as finished on purpose.

To ensure consistency between work items of the same work group, OpenCL provides barriers and memory fence mechanisms. Barriers guarantee that all work items reach a certain point within the program flow before continuing the execution. Memory fences ensure that reads or writes to specific memory locations have completed and that every work item sees the most recent value. Due to the hardware architecture of most GPUs, the OpenCL standard does not ensure correct synchronization with barriers and memory fences for work items located in different work groups. To permit synchronization across work groups, current OpenCL versions expose atomic arithmetic operations. They can be used to either implement synchronization primitives such as mutexes and semaphores or avoid explicit synchronization altogether, for example by accumulating scalar results.

2.3.5 Hardware implementation

OpenCL originates from GPU accelerators, hence most concepts map directly to the hardware features found on AMD and NVIDIA cards. A CU executing a work group maps its work items to the same AMD CU or a NVIDIA SM. A work group, however, is not necessarily mapped to the corresponding SM. A single work item is executed by at least one PE which per specification is an abstract scalar processor, for example an SP of a NVIDIA GPU or the SIMD core of GCN.

A work group is a set of work items that execute the same kernel on the same CU and share local memory and per-work-group barriers. Although the work items of a

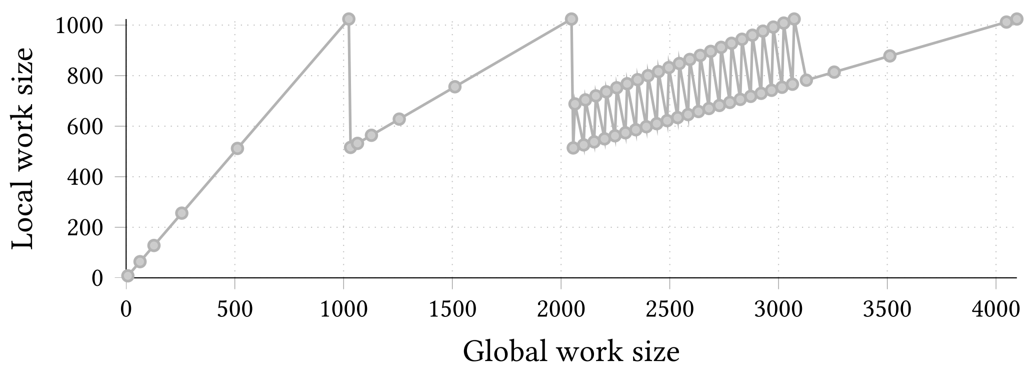


Figure 2.3 Local work size for a given global work size in x-dimension on NVIDIA platforms.

work group can be mapped exactly to a thread warp or wavefront, work groups are not necessarily sized into groups of 32 or 64 elements. On NVIDIA for example, the largest possible work group size is 1024×1024×64. However, no matter which dimension is chosen, the run-time only takes the first dimension into account. Depending on the width of the global work size, the NVIDIA platform chooses the total work group size for a one-operation kernel according to the function shown in figure 2.3. Up to 1024 elements, the local work size grows at the same rate as the global work size, after that it goes back until reaching a new power of two. Between 2048 and 3200 elements, the local work size fluctuates for no apparent reason before it rises in linear fashion until 4096 elements.

2.4 X-ray imaging

X-ray imaging is an analysis technique to study the inner structure of an object that is otherwise opaque to visible light. Unlike visible light which is obstructed by an object, the intensity of an X-ray beam is merely attenuated. Depending on the atomic number of the object's material, X-ray photons penetrating the object are either absorbed, scattered or pass through. Due to this effect, materials can be discriminated by measuring and comparing intensities. This allows the non-destructive determination of the structural composition of an object. Immediately after the discovery by Wilhelm Röntgen in 1895 [106], the medical examination of patients was established as one of the first X-ray imaging application that is still in wide-spread use today [115].

X-ray light has a short wavelength ranging from 50 nm down to 1 pm. This allows for a spatial imaging resolution at the micro and nano scale, which is not possible with visible light. The fast deceleration of a high-velocity electron beam causes the electrons to lose their kinetic energy in the form of X-ray photons. To produce an X-ray beam for small-scale desktop applications, an X-ray tube accelerates electrons by a magnetic force and stops it by hitting a metal target. At larger scale, synchrotron facilities such as ANKA accelerate the electrons until near the speed of light and keep them traveling in a storage ring [103]. To keep the electron beam in a ring, large magnets or undulator devices bend the beam using a magnetic force Due to this force, X-ray photons are emitted tangentially to the ring [2]. At ANKA, the storage ring keeps the initial stream of electrons at an energy level of about 2.5 GeV. At the TOPO-TOMO beamline, this energy is converted to an X-ray beam with an energy range of 6 keV to 40 keV. Monochromators narrow the bandwidth of the raw white light and filters and aperture devices reshape the monochromatic beam to achieve the desired beam properties. After leaving the vacuumed beam tube, the X-ray beam is shaped by experiment-specific devices and measured with an appropriate detector.

2.4.1 Radiography

In a *radiography* setup, the sample is placed in the direct line-of-sight of the X-ray beam. It attenuates the beam to the measured beam intensity I through different photon-matter interaction effects. For an initial intensity I_0, material thickness z and a material-specific attenuation coefficient μ, the non-absorbed X-ray beam intensity I is given by

$$I = I_0 e^{-\mu z}. \tag{2.17}$$

Photon counting pixel *detectors* measure this intensity either directly or indirectly. Detectors such as the Medipix detector chip [81] convert the X-ray light directly into digitized photon counts while indirect converting techniques use a separate thin *scintillator* screen to convert X-ray into visible light. This light is then captured by a conventional

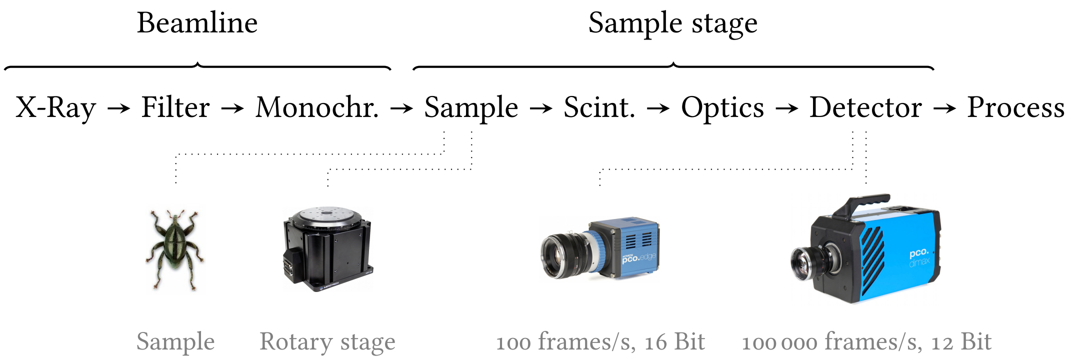


Figure 2.4 Overview of the data acquisition chain in an X-ray imaging experiment.

Charge-coupled Device (CCD) or Complementary Metal-oxide Semiconductor (CMOS) photon detector. Figure 2.4 shows the basic data acquisition chain that is used in all X-ray imaging applications.

Noise reduction

Pixel detectors inevitably add varying amounts of noise to the original signal during the digitization of the physical photon count value. In low-light situations, the discrete and Poisson-distributed nature of light leads to *shot noise* [11]. Increasing the photon counts through higher gain and exposure time, the thermal and electric circuit noise cause the image noise to approach a normal distribution. The amount and spatial distribution of Gaussian noise can be estimated by acquiring a *dark frame* $I_d(x, y)$, an image that is taken without any light source.

The manufacturing process of the detector chip and obstructions in the light path such as dirt on the lens and chip cause reproducible, distinctive patterns or *fixed pattern noise* that is independent of the sample and electrical properties of the chip. In radiography applications, the beam itself becomes visible as fixed pattern noise because of its inhomogeneous Gaussian-like shape. To remove this type of noise from an unprocessed frame $I_r(x, y)$ of size $w \times h$, *flat fields* $I_f(x, y)$ are acquired. These images are evenly illuminated or have a known content such as the beam without a sample in the view. Using the dark frame I_d the corrected image I_c is obtained by

$$I_c(x,y) = \frac{\sum_{x=1}^{w} \sum_{y=1}^{h} |I_f(x,y) - I_d(x,y)|}{w \cdot h} \cdot \frac{I_r(x,y) - I_d(x,y)}{I_f(x,y) - I_d(x,y)} \tag{2.18}$$

In second generation synchrotron facilities, the beam intensity decreases over time because photons get lost in the storage ring. For a beamline operator this results in series of radiographs whose mean pixel value is not constant. In this case, function I_f in (2.18) can be replaced by a time-dependent interpolation of a flat field that is computed from

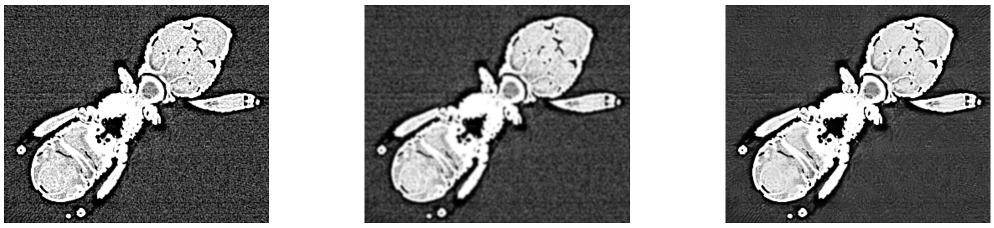

(a) Original **(b)** Gaussian **(c)** Non-local means (NLM)

Figure 2.5 Qualitative comparison of Gaussian and NLM noise reduction filters. The Gaussian smoothing kernel reduces the noise but fine detail gets lost. NLM keeps edges within the sample sharp.

flat fields that were acquired before the first and after the last radiograph.

Depending on the intended quality and performance impacts, different noise reduction algorithms are used to remove Gaussian noise. The most basic but often used methods reduce high frequent noise contributions by filtering the image with low-pass filtering kernels such as a mean, median or Gaussian smoothing. As seen in figure 2.5b, the main disadvantage of these approaches is that they also remove fine detail such as high gradients between different materials. To preserve the detail, computationally intensive filters trying to identify structures such as the NLM filter (figure 2.5c) are used [16]. The NLM filter reconstructs a pixel $I_c(x, y)$ using the weighted sum of all pixels

$$I_c(x,y) = \sum_{i=-n}^{n} \sum_{j=-n}^{n} w\left(\mathcal{N}_{x,y}, \mathcal{N}_{x-i,y-j}\right) \cdot I_r(x-i, y-j) \tag{2.19}$$

in a neighbour region of $2n$ pixels. $\mathcal{N}_{x,y}$ denotes a patch of $k \times k$ pixels around $I_r(x, y)$. The Euclidean distance between the grey level vectors $\vec{v}(\mathcal{N}_{x,y})$ defines the similarity weight of two neighbourhoods

$$w\left(\mathcal{N}_{x_i,y_i}, \mathcal{N}_{x_j,y_j}\right) = \frac{1}{Z(i)} \exp\left(-\frac{\|\vec{v}(\mathcal{N}_{x_i,y_i}) - \vec{v}(\mathcal{N}_{x_j,y_j})\|_{2,a}^2}{h^2}\right) \tag{2.20}$$

with $Z(i)$ denoting a normalization factor

$$Z(i) = \sum_j \exp\left(-\frac{\|\vec{v}(\mathcal{N}_{x_i,y_i}) - \vec{v}(\mathcal{N}_{x_i,y_i})\|_{2,a}^2}{h^2}\right). \tag{2.21}$$

Because the denoising algorithms have adverse performance and quality properties, the user must carefully choose the most appropriate algorithm. While an NLM denoising step provides superior quality, its computational demands are too high to be used in a

(a) Enhanced edges in a phase contrast image. **(b)** After phase retrieval and correction.

Figure 2.6 Result of phase retrieval and subsequent reconstruction. Image courtesy by D. Haenschke, V. Weinhardt, L. Helfen and J. Moosmann and produced at ID19 of ESRF, Grenoble.

soft real-time context with the expected image dimensions.

Phase retrieval

An X-ray beam that penetrates an object that consists of different materials is not only attenuated but has its phase shifted by the different refractive indices of the materials. Due to the wave propagation properties of an X-ray beam, the different phases become visible as edge gradients. The larger the distance between the detector and the sample, the stronger this effect is. This type of phase-contrast X-ray imaging allows discrimination of materials with similar low absorptivity [56]. Figure 2.6a shows a sample with bright edges around different features.

Using the exact known acquisition parameters, one can estimate a deconvolution kernel and reconstruct the original image as shown in figure 2.6b. Contrary, to regular absorption-based radiography the retrieved phase gives additional information about the composition of a sample.

2.4.2 Tomography

A *tomography*[1] experiment extends the radiography experiment by a vertical *rotation axis* along which the sample or the detector is rotated. Using the radiographic setup, projections are acquired at discrete angular steps θ. The step between two successive angles is small enough to avoid blurring caused by the rotational motion and large enough to avoid redundant projections. Scanning the sample for angles θ between 0 and 180 degrees accumulates enough spatial information to determine the inner structure using a reconstruction algorithm. The result of these algorithms is a *tomogram* of volume data containing the estimated densities within the object. Although medical applications for

[1] From Greek $\tau\acute{o}\mu o\varsigma$ (tomo), a *section* or *slice*. A single slice is the data at a fixed y coordinate within a reconstructed tomogram.

Computed Tomography (CT) exist since the 1970s, synchrotron radiation-based micro CT (SRμCT) only recently became an important technique for the analysis of materials and biological specimen [35].

In medical applications, the X-ray source and the detector are rotated around the patient. Due to the mechanics and the hazardous dangers of the X-ray beam itself, the total beam exposure is reduced, thus limiting the spatial and temporal resolution. In a SRμCT setup, however, the sample itself is rotated along θ and scanned with a high intensity beam. To compare the acquired data of different tomography setups, the reconstructed absorption grey values μ are normalized to the device-independent Hounsfield scale [65] given in Hounsfield units (HU)

$$H = 1000 \cdot \frac{\mu - \mu_{\text{water}}}{\mu_{\text{water}}}. \tag{2.22}$$

Using this scale, air has a value of roughly -1000 HU and water zero HU. Prior calibration helps to determine exactly of what a sample is composed of.

2.4.3 Laminography

A generalization of the tomography experiment is the *laminography* setup.[2] Instead of rotating the sample around a fixed axis that is aligned orthogonal to the detector plane, a laminographic rotation axis can be tilted by an angle ψ thus allowing the sample to swing in a non-linear motion [91]. This setup permits scanning flat objects such as microelectronic devices more evenly and with higher beam transmission thus allowing more efficient use of the beam. The intricate detection geometry, however, complicates the reconstruction of a volume and makes it computationally much more demanding than with a regular parallel beam geometry.

2.4.4 Synchrotron radiation-based micro CT

According to the Lambert-Beer-Law, the absorption of X-ray photons by an object causes an attenuation of the X-ray beam. In a plane, such an object is described by a two-dimensional function $f\colon \mathbb{R} \times \mathbb{R} \to \mathbb{R}$ that relates the spatial locality to unknown absorption coefficients. The beam intensity measured by a detector is a line integral that penetrates the object plane at a rotation angle θ and an offset t as shown in figure 2.7. Thus a single (one-dimensional) projection $P_\theta(t)$ is given by

$$P_\theta(t) = \int_{-\infty}^{\infty} \int_{-\infty}^{\infty} f(x,y)\delta(x\cos\theta + y\sin\theta - t)\, dx\, dy \tag{2.23}$$

[2] From Latin *lamina* that means to consist of something *thin* or being made of *layers*.

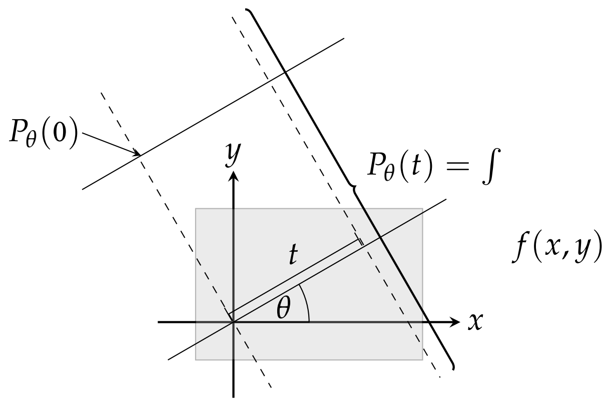


Figure 2.7 The Radon transform relates the a projection $P_\theta(t)$ to the line integral of the object function $f(x,y)$ along the line (θ, t).

where δ denotes the Kronecker delta symbol. The discrete function $P(t,\theta) := P_\theta(t)$ is called the *sinogram* of a particular slice. Equation (2.23) is valid only for a parallel beam, which is the typical beam geometry of synchrotron X-ray beams. For fan- and cone-shaped beams that occur with point-shaped X-ray sources such as X-ray tubes, additional parameters that relate the detector position to a fan angle must be incorporated. For data acquired with a fan-beam geometry, the fan-beam projections can be sorted to match a parallel beam geometry [65, p. 93]. For a fan-beam projection $R_\beta(\gamma)$ at angle β and beam spread γ taken with an X-ray source at a distance D from the origin, it is

$$\theta = \beta + \gamma \qquad t = D \sin\gamma.$$

Thus using the beam geometry relation

$$P_{\beta+\gamma}(D\sin\gamma) = R_\beta(\gamma) \tag{2.24}$$

we can find a fast algorithm 2.1 to transform fan-beam projection data to parallel projection data.

Algorithm 2.1: Re-arrangement of fan-beam data $R_\beta(\gamma)$ to parallel beam projections $P_\theta(t)$.

```
for $\theta \in [0, \pi]$ do
    for $t \in [-w, w]$ do
        $\gamma' \leftarrow \arcsin\left(\frac{t}{D}\right)$
        $\beta' \leftarrow \theta - \gamma'$
        $P_\theta(t) = R'_\beta(\gamma')$
```

From (2.23) it also follows that

$$P_\theta(t) = P_{\theta+\pi}(-t) \tag{2.25}$$

which tells us that it is enough to acquire projection data of a single half circle. This property is used to find the reconstruction axis in case it is not perfectly aligned with the middle column of the detector. In this case, a constant offset Δ is added to the t parameter for the entire tomographic scan. By determining the offset Δ, we can find the correct axis of rotation which is crucial for the subsequent reconstruction. From (2.25) it follows that

$$P_\theta(t+\Delta) = P_{\theta+\pi}(-t+\Delta). \tag{2.26}$$

From this equation, we can see that a general algorithm would try to minimize the difference between the two projections $P_\theta(t)$ and $P_{\theta+\pi}(t)$ for different Δ. A straightforward approach computes the convolution of both projections and uses the maximum to determine the offset via

$$\Delta = \operatorname*{argmax}_t P_\theta(t) * P_{\theta+\pi}(t) \tag{2.27}$$

This matching algorithm is highly susceptible to noise and does not detect the center of rotation for high speed acquisitions reliably. A reliable alternative to the direct determination of the axis is the iterative reconstruction-based detection [34]. This method provides higher precision by scoring a quality metric such as the sum of absolute differences or the entropy of the reconstructed *volume* instead of the similarity of original and shifted projection. On the downside, this approach requires multiple preliminary reconstructions until the estimated axis converges towards a sub-pixel precise axis. Although the performance of this method can be improved by selecting candidate axes via a binary search, the overall reconstruction time will still increase.

Reconstruction

The Fourier-slice theorem is the theoretical foundation to reconstruct an approximation $\hat{f}(x,y)$ from the acquired projections $P_\theta(t)$ of the real object plane $f(x,y)$ [65]. Given (2.23), the two-dimensional Fourier transform $\mathcal{F}$ of an object is

$$F(u,v) = \mathcal{F}(f)(u,v) = \int_{-\infty}^{\infty}\int_{-\infty}^{\infty} f(x,y)e^{-i2\pi(ux+vy)}\,dx\,dy. \tag{2.28}$$

Let (t,s)

$$\begin{pmatrix} t \\ s \end{pmatrix} = \begin{pmatrix} \cos\theta & \sin\theta \\ -\sin\theta & \cos\theta \end{pmatrix} \begin{pmatrix} x \\ y \end{pmatrix} \tag{2.29}$$

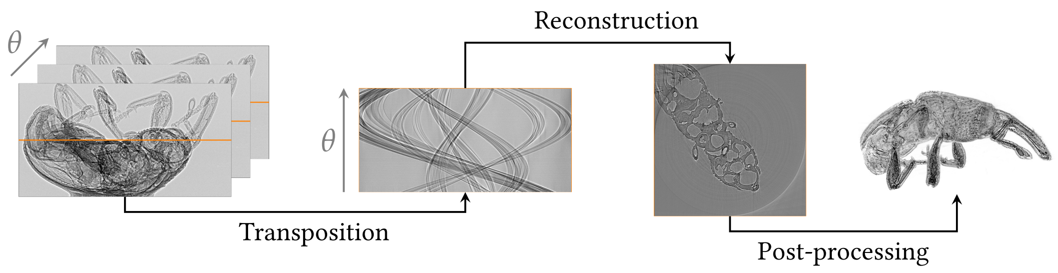


Figure 2.8 CT acquisition and reconstruction process: a series of projections is first transposed into sinograms to approximate the result of the Radon transform for a single detector row. Reverting the transform with a reconstruction algorithm yields a two-dimensional slice. Post-processing all slices reconstructs the final volume.

denote the rotated coordinate system of (x, y), then (2.23) can be re-written as

$$P_\theta(t) = \int_{-\infty}^{\infty} f(t, s)\, ds. \tag{2.30}$$

Its Fourier transform $S_\theta(w)$ is given by

$$S_\theta(w) = \int_{-\infty}^{\infty} P_\theta(t) e^{-2i\pi wt}\, dt = \int_{-\infty}^{\infty}\int_{-\infty}^{\infty} f(t, s)\, ds\, e^{-2i\pi wt}\, dt \tag{2.31}$$

and transforming the rotated coordinate system back yields

$$S_\theta(w) = \int_{-\infty}^{\infty}\int_{-\infty}^{\infty} f(x, y) e^{-2\pi w(x\cos\theta + y\sin\theta)}\, dx\, dy. \tag{2.32}$$

Replacing the right side with (2.28) and substituting u with $\cos\theta$ and v with $\sin\theta$ finally relates the one-dimensional Fourier transform of a projection with the two-dimensional Fourier transform of the object

$$S_\theta(w) = F(w\cos\theta, w\sin\theta). \tag{2.33}$$

With an infinite number of projections, the original object function $f(x, y)$ can be reconstructed by transforming F back to the spatial domain, therefore

$$f(x, y) = \int_{-\infty}^{\infty}\int_{-\infty}^{\infty} F(u, v) e^{2i\pi(ux+vy)}\, du\, dv. \tag{2.34}$$

Interpolation

Because detectors can only resolve a finite number w of discrete pixels along the x-axis, the reconstructed object function $\hat{f}(x, y)$ is a finite grid of absorption coefficients.

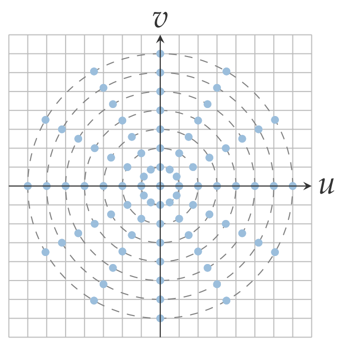


Figure 2.9 One-dimensional Fourier transforms placed on a Cartesian coordinate system results in sparsity of higher frequencies.

Moreover, to keep acquisition time and X-ray dose low, at most n_P projections are acquired so that each pixel in a frame rotates no more than one pixel in a single rotation step. The minimum rotation step for a frame width of w pixels is $\Delta_\theta = \arctan 2/w$, hence the minimum number of projections n_P is

$$n_P = \lceil \pi / \Delta_\theta \rceil. \tag{2.35}$$

Trying to reconstruct from a finite number of projections using (2.34) results in interpolation errors in the higher frequencies as shown in figure 2.9.

Direct Fourier Inversion

The Direct Fourier Inversion (DFI) algorithm reconstructs the object function $f(x, y)$ by direct application of the Fourier-slice theorem. Instead of back-projecting the filtered projections back into the slice space, the one-dimensional Fourier transforms of the projections are arranged on the two-dimensional Fourier space of the object domain. The two-dimensional reconstruction is then obtained by computing the two-dimensional inverse Fourier transform of the Fourier representation according to (2.34).

For real-world applications, the number of acquired projections is limited to a finite number. Thus, the main difficulty with this approach is to find an interpolation method that maps a limited number of Fourier transformed projections in polar coordinates onto the two-dimensional Cartesian coordinate system without introducing artifacts. The main reason for these artifacts is, as one can see in figure 2.9, the radial arrangement which causes an uneven contribution of the data points with higher density on lower frequencies and sparser density on higher frequencies. For any practical use, the nearest neighbour and linear interpolation methods yield an error that is too large and causes noticeable artifacts. To overcome the interpolation problems, Stark proposed the convolution of the Fourier data in a neighbourhood with a truncated two-dimensional sinc filter [116]:

$$F^i(u,v) = \sum_{(w,\theta)\in\mathcal{N}_{u,v}} S_\theta(w) \cdot \mathrm{sinc}_{2\mathrm{D}}(x-u, y-v) \tag{2.36}$$

where $\mathcal{N}_{u,v}$ denotes a neighbourhood of elements around (u,v).

Filtered backprojection

Replacing the rectangular coordinate system (u,v) in (2.34) with an equivalent polar coordinate system (w,θ), such as $u = w\cos\theta$ and $v = w\sin\theta$, yields the reconstruction

$$f(x,y) = \int_0^{2\pi}\int_0^{\infty} F(w,\theta)e^{2i\pi w(x\cos\theta+y\sin\theta)}\, w\, dw\, d\theta. \tag{2.37}$$

which by splitting in two halves and exploiting (2.25) can be rewritten to

$$f(x,y) = \int_0^{\pi}\int_{-\infty}^{\infty} F(w,\theta)|w|e^{2i\pi w(x\cos\theta+y\sin\theta)}\, dw\, d\theta. \tag{2.38}$$

Applying the Fourier-slice theorem from (2.33) results in

$$f(x,y) = \int_0^{\pi}\left[\int_{-\infty}^{\infty} S_\theta(w)|w|e^{2i\pi w(x\cos\theta+y\sin\theta)}\, dw\right] d\theta \tag{2.39}$$

which gives us a principal framework for a reconstruction using the Filtered backprojection (FBP) approach as outlined by algorithm 2.2. The term $|w|$ corresponds to a convolution of $P_\theta(t)$ with the inverse Fourier transform of $|w|$ which gives the algorithm its *filtered* name.

Algorithm 2.2: Steps of the FBP reconstruction

```
for each projection $P_\theta(t)$ do
    $S_\theta(w) = \mathcal{F}(P_\theta(t))$
    $S^f_\theta(w) = S_\theta(w)|w|$
    $P^f_\theta(t) = \mathcal{F}^{-1}\left(S^f_\theta(w)\right)$
for each $(x,y)$ do
    $\hat{f}(x,y) = 0$
    for each $\theta \in [0,\pi]$ do
        $t = x\cos\theta + y\sin\theta$
        $\hat{f}(x,y) = \hat{f}(x,y) + P^f_\theta(t)$
```

Using the Fast Fourier Transform (FFT) for the discrete Fourier transformation, the computation of the FBP algorithm requires $O(n^3)$ time steps, whereby each output pixel requires the evaluation of all projections.

Algebraic reconstruction methods

The class of Algebraic Reconstruction Technique (ART) reconstructs $\hat{f}(x,y)$ by modelling the detection process as a system of linear equations and solving for $\vec{f}$ (with $f_i = f_{yw+x}$)

$$\mathbf{W}\vec{f} = \vec{p} \tag{2.40}$$

where $\vec{p}$ is the m-dimensional projection vector consisting of $P_\theta(t)$, $M = n_P \times w$, $\vec{f}$ the $N = w \times w$-dimensional vector of unknowns and $\mathbf{W}$ the weight or system matrix, where one row weighs the unknowns for one projection entry. Solving this linear system directly becomes intractable for large w because the size of the system matrix cannot fit into main memory. For example reconstructing 1024×1024 slices requires a system matrix of 1 TB assuming one byte per entry. Compared to the Fourier-based approaches ART methods can incorporate additional information about the detector system, assumptions about the expected materials [6] and use appropriate solvers to reconstruct $\hat{f}$ with a limited number of projections which would otherwise violate the Shannon-Nyquist sampling theorem.

Ring artifact reduction

Fixed pattern noise such as spots with unusual attenuation or "hot pixels" manifests themselves by $P_\theta(a) \approx P_0(a)$ for all θ, hence $P(t = a, \theta) = b$ for some constant b. Reconstructing a slice from such a sinogram causes visible *ring artifacts* as seen in figure 2.10a. Because stripes in θ direction affect only the u components in the two-dimensional Fourier transform, we can devise a filter $H(u,v)$ with horizontal cut-off frequency f_c

$$H(u,v) = \begin{cases} 0 & u = 0 \wedge v \geq f_c \\ 1 & \text{else} \end{cases}. \tag{2.41}$$

Multiplying this filter with the spectrum of $P(t,\theta)$ and transforming back to the spatial domain yields an improved sinogram. Slices reconstructed from such a sinogram exhibit reduced ring artifacts as shown in figure 2.10b. This requires two two-dimensional or four one-dimensional Fourier transforms and $O(N^2)$ multiplications.

(a) Reconstructed from original sinogram. **(b)** Reconstructed from filtered sinogram.

Figure 2.10 Reconstruction result with ring artifacts on the left. On the right hand side removing the vertical stripes in the sinogram reduced the amount of artifacts.

3 Streamed processing on heterogeneous architectures

Three main properties characterize streamed data processing: strict dependencies between adjacent filter stages, a lock-step style flow of data and a high variance of the computational intensity of each task. Scheduling data processing pipelines on homogeneous compute systems does not satisfy the required performance demands because a high task heterogeneity causes underutilized processors. Contrary to homogeneous systems, heterogeneous compute systems consist of a variety of processors. This heterogeneity can reduce the performance problems by mapping each pipeline task to the most appropriate processor [68]. In this chapter, we will present the theoretical and practical foundations of a processing framework for streamed data. We will present scheduling and mapping techniques that utilize the advantages of heterogeneous architectures to provide X-ray image processing for soft real-time purposes.

3.1 Requirements

The main contents of this chapter is the analysis, design and implementation of an image processing system for streamed data. The main requirement of this system is the complete use of the parallel potential of arbitrary heterogeneous compute systems composed of CPUs and GPUs. We will use a set of abstract principles to model this system and constrain it by system parameters to meet the following goals, which were derived from the requirements stated in section 1.2:

1. The system should be designed as a flexible *framework* that is able to process streams of multi-dimensional floating point data with at least single precision. In practice, the expected data sets will be arrays of up to three dimensions containing gray value pixel data.

2. On a single machine node, the system must use all local CPU and GPU resources. The performance of the system must scale adaptively with the number of available resources. In a cluster environment consisting of multiple compute nodes, the system should scale with the available communication bandwidth.

3. The performance of the system should allow for *soft* real-time data processing, which means that data produced at the beamline must be processed immediately without significant on-disk buffering. This requirement is necessary for quick on-line assessment of the data and conduction of fast feedback experiments. This includes the partial tomographic reconstruction from all projections.

4. The system must provide a straightforward low-level API that allows human end-users to describe tasks and facilitate the system. The same API must allow other compute system programmatic access in order to build end-user applications. Ideally, the task description is independent of a particular platform and portable between heterogeneous systems.

5. The design should allow users and developers to *customize* the run-time behaviour and *extend* the system with additional tasks. To simplify the deployment adding additional processing tasks processing-related function should not require changes to the core run-time system.

To validate the fulfillment of these requirements, we will first define a formal model of the structure and the execution of our compute system. It will be based on a task and architecture graph notion. Using this compute model, we will propose algorithms to map specific tasks to specific processes. After modelling the system architecture using UML, we will present implementation-related problems and solutions.

3.2 System model

The principles of heterogeneous compute system given in sections 2.1.1 and 2.1.2 are the basis of our system and a performance model describing this system and which we are going to present in this section. As we have seen, conventional heterogeneous system models assume that a compute problem can be broken down into smaller pieces and distributed among a set of p processors. In the context of streamed data processing tasks, this model neglects the fact that the *repeated* task execution depends on the order of their arrangement and the implied data flow described in section 2.1.3. Mapping these tasks using conventional scheduling heuristics will inevitably lead to suboptimal execution.

In the remaining section, we will use a graph approach to describe heterogeneous compute architectures as well as the algorithmic computation. Unlike linear pipelines, graphs allow for compute configurations with multiple data inputs thus providing larger expressiveness.

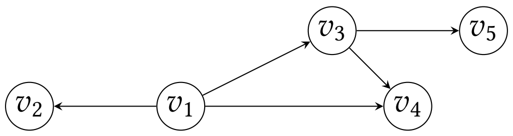


Figure 3.1 Graph $G = (V, E)$ with five vertices $V = \{v_1, \ldots, v_5\}$ and five edges $E = \{(v_1, v_2), (v_1, v_3), (v_1, v_4), (v_3, v_4), (v_3, v_5)\}$.

3.2.1 Graphs

A graph G is a tuple (V, E) where V denotes a set of *nodes* or vertices and $E \subseteq V \times V$ a set of *edges*.[1] Each edge $e \in E$ is a tuple of elements from V that relates two *adjacent* nodes. Nodes forming an edge are *incident* to that edge. If the direction of an edge does not matter, which means $(a, b) = (b, a)$, the graph G is called *undirected*. Otherwise, the graph is *directed* and called a digraph. In this case, an edge (a, b) is called an *arc* with b being the successor of a and a being the predecessor of b. A graphical representation of a directed graph is given in figure 3.1. The union and disjunction of two graphs G and G' are defined as $G \cup G' := (V \cup V', E \cup E')$ and $G \cap G' := (V \cap V', E \cap E')$ respectively. If $V' \subseteq V$ and $E' \subseteq E$, then $G' \subseteq G$ and G' a subgraph of G.

The degree $\deg(v)$ of a node v denotes the number of edges incident to v. It is formally defined as $\deg(v) = |\{(x, y) \in E \mid x = v \vee y = v\}|$. In directed graphs, $\deg^-(v)$ and $\deg^+(v)$ denote the number of edges going in and out of v:

$$\deg^+(v) = |\{(x, y) \in E \mid x = v\}|$$
$$\deg^-(v) = |\{(x, y) \in E \mid y = v\}|.$$

A path P of a directed graph G is a sequence of edges

$$P = (e_1, e_2, \ldots, e_n) = \left((v_1^s, v_1^t), (v_2^s, v_2^t), \ldots, (v_n^s, v_n^t)\right)$$

for which $v_i^t = v_{i+1}^s$ holds $\forall i \in \{1, \ldots, n-1\}$. In this case n denotes the *length* of the path. If $v_n^t = v_1^s$ then the path is called a *cycle*. A directed graph that does not contain a cycle is a Directed Acyclic Graph (DAG). If for any two nodes of a graph G there is a path $P \subseteq G$ that connects both nodes, then G is called *connected*.

3.2.2 Machine model

Let $G_A = (V_A, E_A)$ denote a directed *architecture* graph that describes the communication between $P_c \in \mathbb{N}$ CPUs and $P_g \in \mathbb{N}$ GPUs. V_A corresponds to the set of heterogeneous processors, hence $|V_A| = P = P_c + P_g$. We further distinguish the disjunct sets of CPU and GPU processors, i.e. $V_{A_c} \cup V_{A_g} = V_A$. An edge (P_i, P_j) exists

[1] We only give brief overview on graph theory; for a thorough introduction refer to [33].

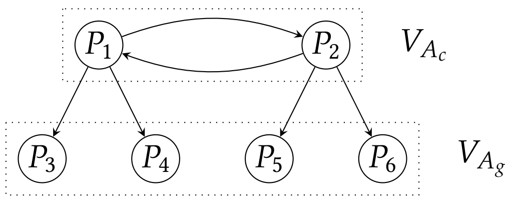


Figure 3.2 Machine model with two compute nodes, each consisting of one CPU and two GPUS.

in $E_A \subseteq V_A \times V_A$, if P_i can send data to P_j. For each edge, we assign an edge weight $w_A : E_A \rightarrow \mathbb{R}$ that denotes the relative bandwidth between two processors. We do not specify explicit values for this bandwidth but will use existing models such as Hockney's model or the LogP model family introduced in section 2.1.2 when appropriate. Nevertheless, within this model we can safely assume that data that is present on a processor and used on the same processors does not incur any costs, thus $\forall P_i \in V_A \colon w_A(P_i, P_i) = 0$.

Figure 3.2 shows an exemplary machine model of a heterogeneous compute system that consist of two machines or compute nodes. Each machine is equipped with one CPU and two GPUS. In the remaining thesis, we will consider CPU cores as individual processors. This is a valid assumption, because in all modern CPU architectures, a CPU core has exclusive access to its own resources and can executed code autonomously from all other cores.

3.2.3 Task model

A streaming compute system consists of independent *tasks*, each processing input data based on an internal state. A task receives input data uni-directionally from one or more *predecessor* tasks and outputs data consumed by a single *successor* task. Receiving data from multiple sources allows to model non-trivial systems, for example an adder that accumulates received operands data. For this reason, we model the data flow and execution of tasks by a task graph $G_T = (V_T, E_T)$. Each node $v \in V_T$ is an atomic processing unit that receives data from a fixed number of tasks and sends data via a fixed number of outputs. The number of input tasks corresponds to $\deg^-(v), v \in V_T$ and the number of output tasks to $\deg^+(v)$. Tasks v for which $\deg^-(v)$ is zero are called *source* tasks and tasks v for which $\deg^+(v)$ is zero are called *sink* tasks. Because nodes denote the tasks and edges the dependencies between nodes, it can be classified as a *task precedent* graph model [75].

An edge $(t_i, t_j) \in E_T \subseteq V_T \times V_T$ denotes the flow of data from task t_i to task t_j. A weight or argument map function $w : E_T \rightarrow \mathbb{N}^+$, maps an edge to the input position or port of the target, i.e. $\forall e = (t_i, t_j) \in E_T \colon 1 \leq w(e) \leq \deg^-(t_j)$. To model the time required to finish processing an item, we use the function $c_t : V_T \times V_A \times \mathbb{N} \rightarrow \mathbb{R}$ that denotes the run-time a task requires for a certain size m on a given processor $p_i \in V_A$.

If $t \in V_T$ cannot be executed on some $p \in V_A$, for example because I/O operations require execution on a CPU, then $c_t(t, p) := \infty$. Given a single task t, the variance of run-times is an indicator that determines a *machine's heterogeneity* [14]. It can be estimated by

$$\frac{1}{P-1} \sum_{p \in V_A} (c_t(t,p) - \overline{c_t}(t))^2, \tag{3.1}$$

where

$$\overline{c_t}(t) = \frac{1}{P} \sum_{p \in V_A} c_t(t,p). \tag{3.2}$$

The *task heterogeneity* can be estimated similarly by determining the variance of run-times using a single processor over all tasks. According to Braun et al. a heterogeneous system is *consistent* if $c_t(t_1, p_1) < c_t(t_2, p_2) \Rightarrow \forall t \in V_T \colon c_t(t, p_1) < c_t(t, p_2)$ and *inconsistent* otherwise [14]. A *partially consistent* system has a subset of processors p_i which by themselves classify as consistent. Due to the architectural differences between CPUs and GPUs, a heterogeneous system composed of these processor is inherently inconsistent [78].

As we pointed out in section 2.1.2, the input data size does not necessarily match the data size produced by a task. For example, a high quality DFI reconstruction requires zero-padding and oversampling of the projection data. This increases the effective image dimensions by up to four times. On the other hand, computing a metric such as the mean value of an image reduces the data to a single point. To denote the data reduction of a task, we use function $\phi_i \colon \mathbb{N} \to \mathbb{N}$, which returns the number of bytes produced by task t_i for a given input size.

3.2.4 Scheduling

The result of a general scheduling algorithm is an *allocation* of tasks to processors and the subsequent *execution* of tasks in such a way that the precedence of the tasks is maintained. The main objective of a scheduling algorithm is to minimize the time for the last task to finish by reducing the overall schedule length or so-called *makespan*. The NP-completeness of the problem has been proven even in light of restricting assumptions such as task scheduling on an *arbitrary* number of processors [126]. Hence, unless $P = NP$, there is no algorithm that can compute a solution for the general scheduling problem in polynomial time. Considering the intractability of the problem, the majority of the research uses heuristics to sacrifice optimality in favor of efficient computability.

In this thesis, we will restrict the task graph generality in order to map tasks more efficiently onto the processing resources and to match the intended application use cases more closely. First of all, we only consider *non-preemptive* scheduling of tasks. This means that task execution cannot be interrupted on one processor in the midst of a computation, moved to another processor and resume execution. Second, we only consider task graphs that do not have conditional branches. These are task graphs with edges that followed only with a certain probability.

Streamed systems

Applying the general task graph based model on a streaming system means to schedule the DAG independently for each iteration. This requires that the sink tasks of iteration n finish before the source tasks of iteration $n+1$ start.

There are two approaches to use the existing task graph based scheduling algorithms for streamed applications. First, we could construct a *super* task graph that contains the same number of copies of the original task graph as there are iterations. To simulate iterations through that graph, the sink task of task graph n would be connected to the source tasks of task graph $n+1$. Second, we could also schedule a single time using the original task graph and then re-use the result for subsequent iterations. Both approaches do not take into account that tasks which finished must be re-scheduled immediately. This will yield low performance, especially on systems with low number of tasks and high number of processors.

To describe our scheduling approach, we define a *process iteration* $s \in \mathbb{N}$ as a single completed step. This step starts with the production of data from all source tasks and ends with the completion of all sink tasks. Figure 3.5 shows five iterations required to process three data items. For each iteration s, we have to find an optimal mapping between the system's architecture graph V_A and the task graph V_T, that means a schedule that minimizes the total time required to execute one iteration. We denote this by the iteration-dependent allocation function $M_s : V_T \to V_A$. The *inverse image* of M_S that means $M_s^{-1} : V_A \to \mathcal{P}(V_T)$ denotes all tasks that are assigned to a particular processor.

Actor model

Given our basic model of computation, we assume that data flows in lock step fashion through the task graph. The reasons for this is that a node can only process another data item as soon as it receives data from its preceding task nodes and as soon as the succeeding node is ready to receive data. On a per task-level, it is therefore useful to describe the computation stages as a finite state machine.

Over the course of multiple iterations, task nodes are not only characterized by the time required to process an item but also by the *number* of data items they produce. This depends on the number of items they receive and is denoted by function $\gamma_i \colon \mathbb{N}^+ \to \mathbb{N}^+$, which computes the total number of items produced by task t_i given a number of input data items. From the definition it follows that tasks with $\gamma_i(n) = c$ produce a new item at every s-th iteration, $s = \frac{n}{c}$.

Depending on the function γ_i, we can group tasks into three disjunct classes. Tasks $t \in V_T$ for which $\gamma_t(0) = k, k > 0$ and $\gamma_t(n) = 0,\ n > 0$ produce a stream of k

items without taking any input. We call these types of tasks source tasks or *generators*. For example, readers loading data from files and inserting them into the stream are classified as generators. Tasks for which $\gamma_t(n) = n$ consume and produce an item per iteration. These tasks are called *processors* and represent typical processes such as pixel or arithmetic stream operations. Any task with $\gamma_t(n) = \frac{n}{k}$ and $k < n$ is called a *reductor* and produces an item every k-th iteration. Different averaging strategies fall into this category: If $k = n$, hence $\gamma_t(n) = 1$, such a task averages across the whole data stream. On the other hand, an averaging filter with $k < n$ could be thought of as a sliding window averaging process with window size k.

3.3 Mapping algorithms

To reduce the amount of data transfers and execute tasks on the most suitable processor, finding the best mapping between task and processor is crucial for high throughput. Thus, a per-iteration mapping resembles scheduling algorithms that minimize the total length of computation and communication for a set of tasks. Fernandez et al. proved that the problem of finding the *optimal* mapping is NP-complete [42]. Hence heuristics are necessary to find a mapping that optimizes the makespan in polynomial time. In this section, we will present three strategies to determine these mappings.

3.3.1 Local scheduling

Our first mapping strategy assumes an actor execution model in which each task decides autonomously on which CU it will execute. For the scheduling decision, it uses only locally available information. The processor node sets are placed into synchronized First In First Out (FIFO) queues which the tasks use to determine their next location. Fetching the processors by the time they are finishing, implies an earliest-finish-time-first scheduling policy of the task nodes.

Although this strategy is simple to implement it has serious drawbacks. The strategy neglects any data transfer metrics, thus in the worst case data could be swapped unnecessarily frequently between different processors. A technical disadvantage of this approach is that data cannot be shared between tasks that work on different subsets of the same memory block. This is a necessary mechanism for tasks where data does not fit into the processors memory, for example when reconstructing larger volumes using a laminographic geometry. We solved this problem by *grouping* all task nodes that execute the same task and have the same path length from the root of the task graph. The scheduling policy for a single group then assigns an incoming data item to *all* nodes within in the same group.

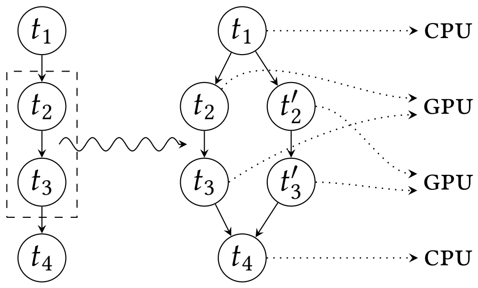


Figure 3.3 Duplication of two tasks, effectively splitting the data stream in two sub streams, each processing approximately one half of the original stream.

3.3.2 Sub-graph duplication heuristic for multiple GPUs

During the execution of a task graph, the structure of the graph itself does not change and a fixed schedule can yield optimal run-time, if all resources are equally utilized. On a multi-GPU system, we can assume that $w_A(P_b, P_c) < w_A(P_c, P_g) < w_A(P_g, P_h)$ where $P_b, P_c \in V_{A_c}$ and $P_g, P_h \in V_{A_g}$ are not identical.

Because $w_A(P_i, P_j) = \infty$ for all $P_i, P_j \in V_{A_g}$, $P_i \neq P_j$, GPUs cannot communicate directly with each other and must share data via a CPU node. Using the general mapping model requires that the work of a single iteration is distributed among all GPUs. This is prohibitive because data would need to be transfered between the GPUs with at least two PCIe transfers [62].

Algorithm 3.1: Expansion of a task graph by duplicating the longest path of GPU nodes.

```
Function expand($V_T$, $V_A$) // Algorithm modifies the original graph in-place
    $V_{\text{cand}} \leftarrow \{v \in V_T \mid \exists p \in V_{A_g} : c_t(v, p) < \infty\}$
    $E_{\text{cand}} \leftarrow \{(v_i, v_j) \in E_T \mid v_i \in V_{\text{cand}} \wedge v_j \in V_{\text{cand}}\}$
    $P \leftarrow$longest_path ($V_{\text{cand}}$, $E_{\text{cand}}$)
    $n \leftarrow |P|$
    $v_x \leftarrow v \in V : (v, P[0]) \in E$
    $v_y \leftarrow v \in V : (P[n], v) \in E$
    for each $v \in V_{A_g}$ do
        $P' \leftarrow$ duplicate of $P$
        Add $(v_x, P'[0])$ and $(P'[n], v_y)$ to $E$
```

In this case a data parallel approach, in which GPUs process *different* items from the stream is more efficient because it avoids any data transfers. This requires the stream to be split in P_g sub-streams whose results are merged at the end accordingly. Path duplication works because tasks are stateless and do not depend on any other task except for receiving input and sending output data from and to adjacent tasks. Hence we can distribute data among a group of receivers. The advantage of duplicating paths is that data buffers reside on the same GPU device as long as possible. Therefore, our main metric for determination of a good path is the length of the path itself. In general, solving the longest path problem is NP-hard [25, p. 987], however our task graph is a directed acyclic graph for which linear time algorithms for the longest path problem exist [109, pp. 661–666]. Conceptually this means duplicating a path of nodes in the task graph and map nodes of the same path onto the same GPU as shown in figure 3.3. In our heuristic, we assume that it is only worth to duplicate the *longest* path of common tasks as shown in algorithm 3.1.

Algorithm 3.2: Longest path determination used by algorithm 3.1.

```
Function longest_path($V, E$)
    Initialize map $L\colon V \to \mathbb{N}$ with $\{(v, 0) \mid \forall v \in V\}$
    $S \leftarrow$ topo_sort($V, E$)
    for $v_c \in S$ do
        if $deg^-(v) > 0$ then
            $v_p \leftarrow$ Predecessor $v$ with maximum $L(v)$
            $L \leftarrow L \cup \{(v_c, L(v_p) + 1)\}$
    $P \leftarrow$ empty, ordered set
    $v_c \leftarrow \mathrm{argmax}_{v \in V}\, L(v)$
    while $deg^-(v_c) > 0$ do                // Track back to find longest path
        Prepend $v_c$ to $R$
        $v_c \leftarrow v\colon (v, v_c) \in E$ with maximum $L(v)$
    return $P$
```

3.3.3 Sub-graph fusion for remote execution

Thermal problems and construction restrictions limit the number of GPUs that can be used reliably in a single machine. Thus to increase the number of GPUs, introducing additional machines with their own local GPUs is the only way to increase the total number of GPUs in the system. In our machine model, we encode the entire heterogeneous system in a single architecture graph G_A with edge weights denoting the transfer time or mean bandwidth between processors. In a system that consists of a network of machines, the edge weight of connections between processors in different machines is higher than between processors or cores within the same machine. We can therefore partition[2] G_A into smaller subgraphs G_{A_i} with $i \in [1, n_r]$ representing one of the n_r

[2] Which should not be confused with the general graph partitioning problem.

Algorithm 3.3: Topological sort algorithm used by algorithm 3.1.

```
Function topo_sort($V, E$)                                  // Topological sort [64]
    $R \leftarrow$ empty, ordered set
    $S \leftarrow \{v \in V \mid \deg^{-}(v) = 0\}$
    while $S \neq \emptyset$ do
        Remove a $v_n \in S$
        Append $v_n$ at end of $R$
        for $e = (v_n, v_m) \in E$ do
            Remove $e$ from $E$
            if $\nexists\, v_x \in V\colon (v_x, v_m) \in E$ then
                $S \leftarrow S \cup \{v_m\}$
    return $R$
```

local machines. Algorithm 3.4 computes the partition set $\mathcal{G}$ for G_A and a lower bound t_l of the transfer time between networked machines

In the system, a designated *master* compute node keeps the initial task graph description as well as a list of remote machines. After the master expands the graph locally, the scheduler transforms the graph in a second pass as shown in figure 3.4. For each potential execution path, the scheduler inserts a *remote task* into the master graph and sends a serialized description of the replaced sub-graph to the corresponding remote compute node. In the local case, each task is processed in a dedicated thread. The inserted remote nodes in the task graph serve as proxies to a daemon process that runs on a distant compute system.

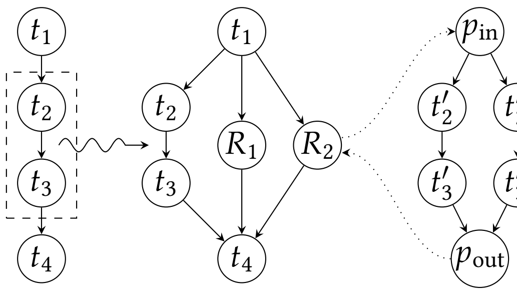


Figure 3.4 Two proxy nodes R_1 and R_2 are added to the original graph on the left, mirroring the functionality of t_2 and t_3 on the remote compute nodes. On the right side, data forwarding from the local R_2 task to the remote is shown. On each remote, the duplication heuristic is applied just like on the master.

Algorithm 3.4: Partitioning G_A into remote sub-graphs.

```
Input: Architecture graph $G_A = (V_A, E_A)$, transfer bound $t_l$
Output: Partition set $\mathcal{G}$
$C \leftarrow V_A$
$\mathcal{G} \leftarrow \emptyset$
while $\exists v_c \in C$ do
    Function visit $(v, E_c)$
        for each $(v, n) \in E_A$ do
            if $n \in C \land w(v, n) < t_l$ then
                $V_c \leftarrow V_c \cup \{v, n\}$
                $E_c \leftarrow E_c \cup \{(v, n)\}$
    $V_c \leftarrow \emptyset$
    $E_c \leftarrow \emptyset$
    visit $(v_c, E_c)$
    $\mathcal{G} \leftarrow \mathcal{G} \cup \{(V_c, E_c)\}$
    for $v \in V_A \cap C$ do
        Remove $v$ from $C$
```

3.3.4 Task graph replication with data partitioning

In 3.3.3, we assume that data originates only from a single source – the machine node which started the system and hosts the master scheduler. This is a valid model for single-source data and data that is produced on the fly such as frames streamed from a 2D detector. Nevertheless, there are applications in which all data is available to all remotes prior to the execution. For example, each remote node can read pre-recorded frames independently if the files are stored in a network or distributed file system such as NFS. In this case, streaming the initial data from the master to all remotes is unnecessary overhead and can be avoided by each remote node reading or generating its own subset of the data.

In this execution model, the master scheduler replicates the entire task graph G_T $n_r - 1$ times. To avoid redundant work, the master sends a uniquely identifying tuple (i, n_r) that consists of the partition index and number of processed data items to all source tasks of G_{A_i}. The source tasks then use this information to determine the partition of data that they are going to produce. In the file reading example, source task i would produce a stream of files numbered from $(i-1)n_f/n_r + 1$ to in_f/n_r, where n_f is the total number of files. Apart from that, the general structured does not change and each G_{A_i} maps the hardware to the tasks locally as presented in 3.3.2. Because each node reads its data locally, no further communication happens between remote nodes and master node except for signaling the end of the data processing. In this mode, execution is similar to the MapReduce pattern for cluster machines [32].

In practice, the sink nodes must know how to partition the result data set. If the data stream of the given example is written back to the network file system, already written files will be overwritten $n_r - 1$ times. Because this setup is fixed prior to execution, $\gamma_i(0)$ for all source $v_i \in V_T$ is known. Hence, from the shortest path $(i, x_{n-1}, \ldots, x_1)$ between source and task t_i we can compute the number of expected items

$$\nu_i = \gamma_i \circ \gamma_{x_{n-1}} \circ \ldots \circ \gamma_{x_2} \circ \gamma_{x_1}(0). \tag{3.3}$$

Hence, each t_i produces data with an index ranging from $(i-1)\nu_i/n_r + 1$ to $i\nu_i/n_r$. In the case of file writing, this index can be used to determine a valid filename.

Analysis

Contrary to a classic heterogeneous system, the data size sent between processors does not only depend on the initial problem size but also on the transformation function ϕ_i of each task. Without loss of generality, let $\psi_i(m)$ denote the final data size sent from task t_i to a succeeding task given an initial data size m sent by the source

$$\psi_i(m) = \phi_i \circ \phi_{x_{n-1}} \circ \ldots \circ \phi_{x_2} \circ \phi_{x_1}(m) \tag{3.4}$$

and $(i, x_{n-1}, \ldots, x_1)$ denoting the shortest path the item takes from source task t_{x_1} to t_i. The number of processors P determines the total time for a single iteration. In case $|V_T| \leq P$, the per-iteration run-time τ is bound by the longest time that a specific task needs for receiving[3] and computing an item as well as the overhead o of computing the mapping

$$\tau = \max_{t \in V_T} \left(c_t\left(t, M_s(t)\right) + \psi_t(m) \cdot L \right) + o \tag{3.5}$$

If, on the other hand, the task graph contains more nodes than processors, one or more processors must execute more than one task in a sequential order. In this case, the maximum per-iteration run-time is the longest execution time taken on a particular processor. From this observation, it follows that the per-iteration run-time is bound by

$$\tau = \max_p \sum_{t \in M_s^{-1}(p)} c_t(t, p) + o \tag{3.6}$$

Using this equation, we can give a performance estimation based on the idea of the processor pipeline model from section 2.1.3. Let $k = |V_T|$ tasks and n data items, the total execution time

$$T_p = \tau \cdot (k + n - 1) \tag{3.7}$$

[3] Accounting the receiving task for the data transfer is arbitrary but can be motivated by the fact that the data processing finishes an iteration.

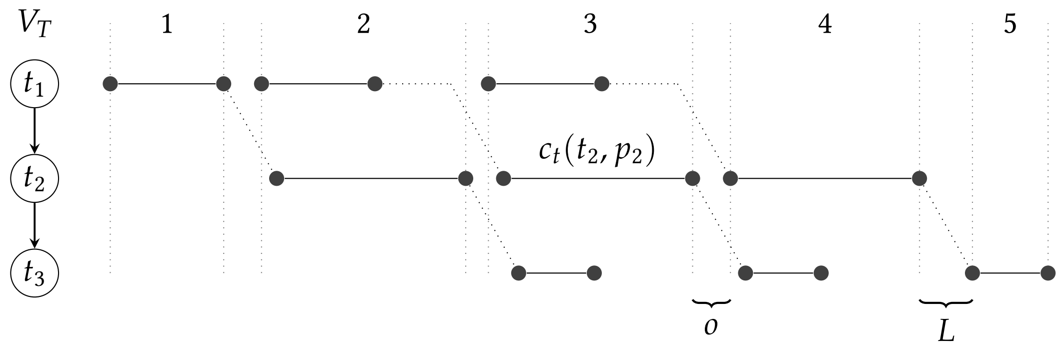


Figure 3.5 Execution of three tasks, mapped to two processors processing three input data items results in a global latency of two and total number of five iterations.

if $P \leq k$. In this case, the speed up becomes

$$S = \frac{T_1}{T_s} = \frac{\tau \cdot nk}{\tau \cdot (k + n - 1)} \overset{n \to \infty}{=} k \tag{3.8}$$

otherwise

$$T_s = \sum_{t \in V_T} c_t(t, M_s(t)) \tag{3.9}$$

For static schedules, $M_{s+1} = M_s$ for all time steps, whereas M_s could change on each iteration for dynamic schedules trying to minimize .

3.3.5 Discussion

Widespread static scheduling heuristics such as Minimum Execution Time, Minimum Completion Time, Simulated Annealing, Genetic Algorithms and A* search assume a set of *independent* tasks [14] scheduled on a system of homogeneous or heterogeneous nodes. This assumption conflicts with our data stream model, in which tasks can only be scheduled as soon as all *immediate* predecessors have finished. Belkhale and Banerjee presented parallelizeable dependent task scheduling but for homogeneous compute systems [7]. One problem that all task graph based scheduling algorithms have in common is the *underutilization* of processing resources. In case of short pipeline task graphs and a high number of processors, the existing heuristics would leave processors unused. With our duplication mechanisms and the streaming mode of operation, we can duplicate as many times as there are processors thus utilize processing resources optimally. Our sub-graph duplication heuristic has superficial similarity with Duplication Based Scheduling (DBS) algorithms [30, 100]. However, the duplicated tasks in DBS serve the purpose of executing sub-branches of the task graph in parallel with *identical* input data. Considering the high data volume that originates from a single source, DBS algorithms will fail to speed up the execution because tasks require approximately equal time to process the same data.

3.4 System architecture

Figure 3.6 shows a reduced Unified Modeling Language (UML) model of our proposed system architecture. It is based on three main abstractions that match the core elements of the abstract model presented in the previous section: Graphs, tasks and schedulers, which we will explain in detail in the remaining chapter. Based on this UML model, we devised a C library framework using the GObject toolkit as a basis for object-oriented features such as classes, inheritance and interfaces and convenience abstractions such as properties, closures and generic values [134].

3.4.1 Graphs and nodes

The basic `Graph` class represents generic graphs and provides methods to add `Node` types and connect them by `Edges`. The `Node` type is a real class that provides methods to copy it and compare it with other nodes. Edges, on the other hand, are simple C structures that contain a reference to their source and target nodes as well as a pointer to arbitrary edge data. The `Graph` class provides graph operations such as retrieving the root and leaf nodes as well as determining the predecessors and successors of a particular node. To simplify the modification of task and architecture graphs, the `Graph` class also provides methods to copy itself and determine paths based on filter predicates. Graph copies can be either deep or shallow. Deep graph copies contain copies of the nodes from the original graph while the shallow copies keep references to the original nodes.

Task graphs

`TaskGraph`'s derive from the basic `Graph` class but store only nodes of type `TaskNode` and their derivatives. When connecting two nodes in a `TaskGraph`, the edge weight is either set implicitly to zero to denote connection to the *first* input port or, if specified, to an arbitrary, valid input port of the target node. Apart from that, a `TaskGraph` extends the basic graph with operations for task expansion and serialization from and to JavaScript Object Notation (JSON) format. While `TaskNodes` represent a task and associated data within the task graph they do not provide methods to execute code by themselves.

To implement the code of a processing task, a `Task` derives from `TaskNode` and implements the `TaskIface` interface. This interface defines the call signatures to query task specific properties and process a *single* data item. The properties characterize the task's role, i.e. if it is a processor, generator or reductor, the number of input ports and the accepted data dimensionality on each input port. To define a task's behaviour, a processor must implement the `process` method, the generator the `generate` method and

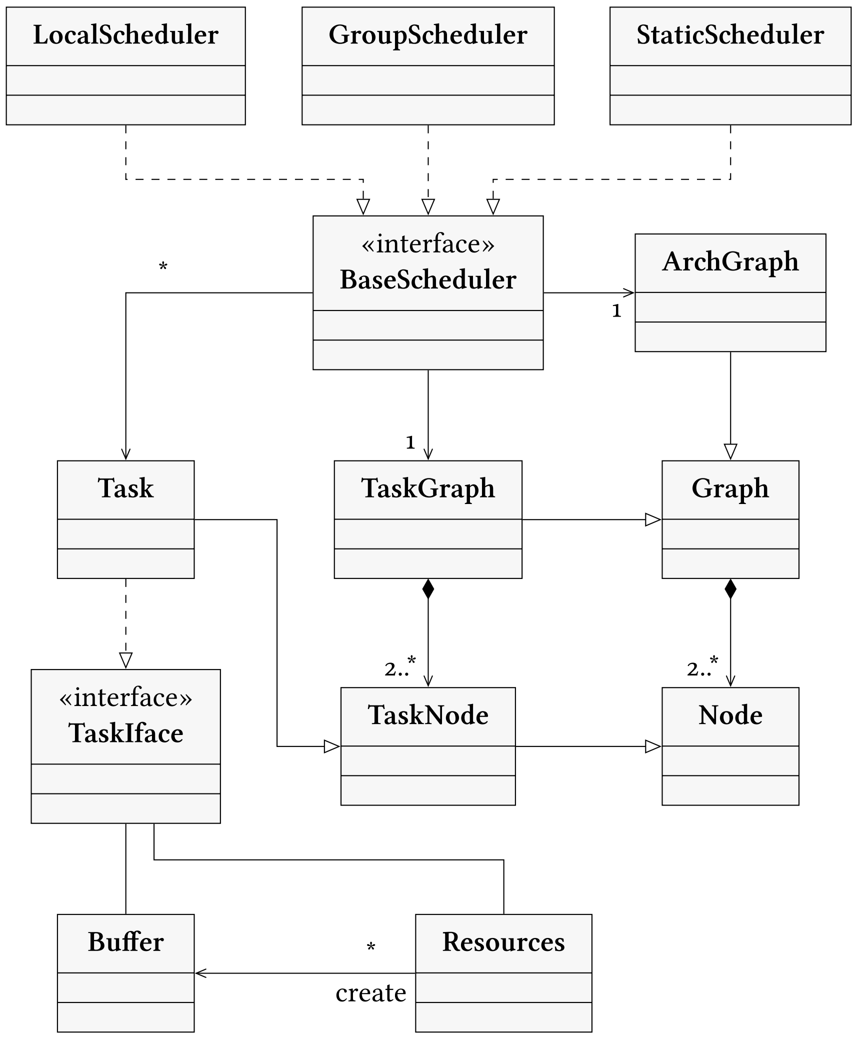


Figure 3.6 Reduced UML class diagram of the high level system architecture.

Listing 3.1 JSON serialization of a reader-writer copy pipeline.

```
 1 {
 2   "edges" : [
 3     {"from": {"name": "reader"}, "to": {"name": "writer", "input": 2}}
 4   ],
 5   "nodes" : [
 6     {"plugin": "reader", "name": "reader",
 7      "properties" : { "path": "/home/user/data/*.tif", "count": 5 }},
 8     {"plugin": "writer", "name": "writer"}
 9   ],
10   "version": "1.0"
11 }
```

the reductor both. Additionally, all types must implement a method that returns the output data size given the current inputs. The `process` method receives both input and output, the `generate` method just an output buffer. Both methods signal the end of the stream by returning `false`.

To exchange task graphs across platforms, a task graph is serialized in JSON format. This serialization format allows the system to *store* task graphs on the disk, reconstruct a `TaskGraph` object from it, *exchange* task graphs between different target languages and *transmit* sub task graphs to remote compute nodes. The data format specifies a `nodes` and an `edges` array (refer to lines 2 and 5 in listing 3.1). The `nodes` array contains JSON objects with a `plugin`, `name` and `properties` field. The `plugin` key has a string value that references the task plugin, whereas the `name` key uniquely identifies the node within *this* graph. The `properties` key maps to a JSON object containing GObject property names as keys and the corresponding values frozen at the time of serialization. The `edges` array contains edge objects with `from` and `to` keys. The values are objects with at least a `name` referring to the names found in the `nodes` array and optionally an `input` number specifying the input port. To change the format in the future, we also add a `version` key (line 10) which specifies the serialization format.

Architecture graph

To represent G_A, we use the non-modifiable `ArchGraph` class that derives from `Graph` and contains nodes of type `CpuNode` and `GpuNode`. During construction, an `ArchGraph` checks the local system for the available number of CPU cores using the GNU-specific `get_nprocs` function and the number of GPUS using a `Resources` object to create a corresponding number of `CpuNode` and `GpuNode` objects. It associates a `CpuNode` with a specific CPU, by assigning the affinity mask of the corresponding CPU core using the Linux-specific `CPU_ZERO` and `CPU_SET` macros. During execution, the scheduler pins a task to a specific CPU with a call to the `sched_setaffinity` Linux system call and the assigned affinity mask. To associate `GpuNodes` with a specific GPU device, the `ArchGraph`

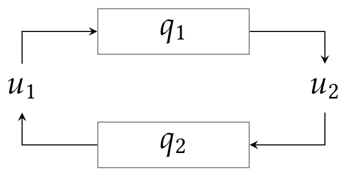


Figure 3.7 Double dequeue used for sharing data buffers.

assigns the command queue that is unique to the device.

3.4.2 Resources

All system resources are managed through a single `Resources` object. During the construction, the `Resources` object, initializes the OpenCL run-time and devices, and creates contexts and command queues for a list of `GpuNodes`. Moreover, the `Resources` object loads, compiles and caches OpenCL kernels read from strings and files. During the execution, the `TaskNodes` request buffers either implicitly by asking for the data of an input or explicitly to store intermediate results. Given size and type constrains, the `Resources` object returns a `Buffer` that encapsulates the location and data in order to abstract from the data transfers between host and device. Besides accessing the data, the user can copy, duplicate and resize `Buffers` as well as converting between different image formats.

Data transfers

As outlined, `Buffer` objects manage the direct access of CPU data arrays and OpenCL `cl_mem` structures. Depending on the needs, a `TaskNode` queries for data in either format after which the buffer will move the data transparently between devices or hosts. Although the amount of inter-device data transfers is reduced by the sub-graph duplication heuristic presented in 3.3.2, there is always the need for host-to-device transfers. The throughput between host and device (less than 16 GB s^{-1} for PCIe 3.0 x16), however, is an order of magnitude lower than intra-device data accesses to the on-board DDR memory (up to 320 GB s^{-1} on a high-end GPU). To avoid unnecessary data transfers between a GPU and a CPU, we improve the data locality by restricting data accesses of producer and consumer tasks. Because our execution model assumes lock-step pipelining, producers must be able to process a data item that is used by the consumer in the next iteration. This is realized by using a double buffering scheme implemented by a *double deque* data structure.

A regular double-ended queue (deque) is a queue data structure that exposes a FIFO policy: data is pushed into one end and popped off the other. This model does not restrict the number and the way users access such a queue. Thus we semantically extended this data structure by two distinct *users* u_1 and u_2. From both perspectives, the structure is accessed using normal queue semantics that means u_1 and u_2 push and pull data in a regular FIFO manner. Unlike a regular queue, an item that was pushed by u_1 must be pulled first by u_2 and pushed back before u_1 can pull it off the queue again. A double ended queue is not sufficient to implement this behaviour. We can, however, use two queues q_1 and q_2 and a protocol as shown in figure 3.7: u_1 always pulls from q_2 and always pushes to q_1 whereas u_2 always pulls from q_1 and always pushes to q_2. Therefore, u_1 and u_2 never directly communicate with each other through a single queue. We apply this pattern and restrict the access of the data depending whether u_1 and u_2 are producers or consumers: in case of producers, data is only written and never read, in case of consumers, data is only read and never written.

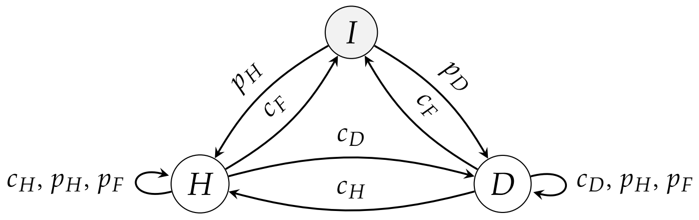

$$\begin{tikzcd} & I \arrow[dl, bend right, "p_H"'] \arrow[dr, bend left, "p_D"] & \\ H \arrow[ur, "c_F"] \arrow[rr, bend left, "c_D"] \arrow[loop left, "{c_H, p_H, p_F}"] & & D \arrow[ul, "c_F"'] \arrow[ll, bend left, "c_H"] \arrow[loop right, "{c_D, p_H, p_F}"] \end{tikzcd}$$

Figure 3.8 The transition table of data locations within buffers.

To enforce buffer locality, we formalize a buffer's location by a Finite State Machine (FSM). The FSM $F = (\{p_H, p_D, p_F, c_H, c_D, c_F\}, \{I, H, D\}, I, \delta, \emptyset)$ is a tuple where H and D specify that the current data location is on the host or device, whereas I denotes an invalid and unallocated location. $p_{\{H,D\}}$ represent the intention of a producer to write data to host (H) or device (D) memory and $c_{\{H,D\}}$ represents the intention of a consumer to read from these memory locations. p_F and c_F denote the action after producer and consumer finished writing and reading. The state transition table δ is given implicitly by the graphical representation shown in figure 3.8. Initially, neither device nor host allocate memory for a buffer, thus the corresponding state is set to I. When producers ask for output memory, the data is allocated and the state transitions to either D or H. When a consumer requests a buffer for a location and the location matches the current one the state is kept, otherwise the state transitions and the data is transferred. When a consumer finishes using a buffer, the state is set to I but the allocated memory location is kept intact. This has the net effect that we do not transfer data to a location that will be overwritten by the producer anyway thus saving one memory copy.

Filters

Although the compute system is designed for general streamed processing algorithms, the majority of the filters are only useful in a X-ray imaging context. The available filters can be grouped into I/O, point- and region-based image processing, frequency and reconstruction filters. The I/O filters allow to read and write TIFF and EDF files as well as to capture detector frames with the UCA framework (see 4.1). The group of image processing filters includes image arithmetics, denoising and sub- and supersampling processes. The last group contains projection and frequency filters to reconstruct volumes from tomographic and laminographic data sets as well as the phase of of phase images.

3.4.3 Scheduler

The `BaseScheduler` interface defines methods to execute a `TaskGraph` on a given `ArchGraph` and to configure itself for remote execution. Concrete implementations such as `LocalScheduler`, `GroupScheduler` and `StaticScheduler` implement a processing loop in the `run` method. Due to the streamed processing model, the loop follows the protocol outlined in algorithm 3.5.

Algorithm 3.5: General scheduler loop protocol.

```
Input: TaskGraph $G_T$, ArchGraph $G_A$
for each task node $t \in V_T$ do
    Call setup() on $t$
    Read get_num_inputs
while not all $t \in V_T$ finished do
    Pick unscheduled $t \in V_T$
    Determine predecessors of $t$
    Fetch output data $I$ of predecessors
    Ask for $t$'s output data size requirements given $I$
    Create or re-use a suitably sized output Buffer $o$
    Call process($I$, $o$) on $t$
    Mark $t$ as finished if process signals end of data
    Mark $t$ as processed
```

Because all $p \in V_A$ execute their tasks concurrently, the order of completion is non-deterministic and the timely arrival of an item cannot be guaranteed. In a pipeline this is not a problem because tasks receive input in-order and have to wait for their successors to complete. In split graphs, this is not a problem, if the only common preceding task forwards data to a single input port. For example, in a task graph that computes the arithmetic addition of two independent streams, the addition task itself synchronizes by fetching the operands appropriately. The split strategy from 3.3.2, however, causes problems if multiple tasks push their results to the *same* input port. This can cause a wrong result order because the preceding tasks run concurrently and finish processing their items in a non-deterministic order. To avoid a wrong ordering, each `Buffer` is assigned a unique, monotonously increasing ID. The scheduler uses this ID to sort input buffers from multiple streams that join at the same input port.

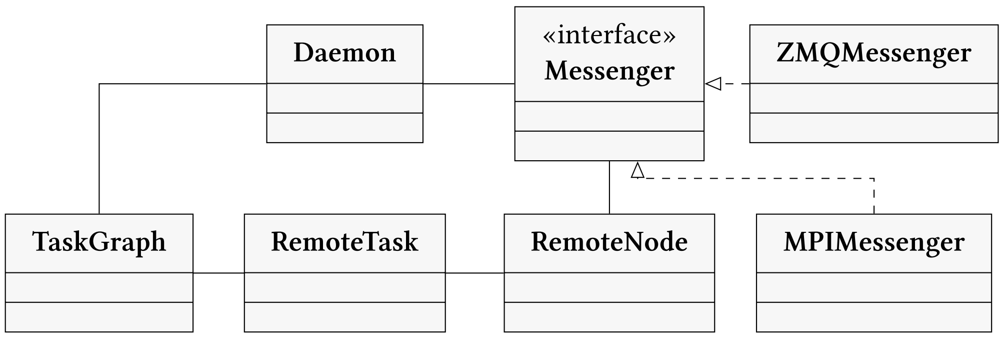


Figure 3.9 Remote extension of the UML model given in figure 3.6.

Remote execution

Figure 3.9 shows an extension of the fundamental UML model from 3.4 for remote execution. The `RemoteTask` class implements the `TaskIface` interface and represents a proxy node in the *local* task graph. It uses a `RemoteNode` – which is part of an `ArchGraph` – to send messages within the `process` method and write the result from the remote into its output buffer. A `RemoteNode` uses the `Messenger` interface to transmit data via some communication link. At the moment, two messenger implementations are available: the `ZMQMessenger` uses the low-latency TCP/IP-based ZeroMQ [54] messaging library, while the `MPIMessenger` maps send and receive calls to corresponding primitives defined in the Message Passing Interface (MPI) standard [113].

On the remote machine, the `Daemon` uses the `Messenger` interface to receive and handle requests from `RemoteNodes`. Both `Daemon` and `RemoteNode` use a common protocol to set up the computation and process data. Piggy-backing on the initial hand-shaking request, the master node queries each remote daemon for the number of locally attached GPU devices. Using this information, the master updates its own architecture graph and determines a suitable sub graph according to the heuristic defined in 3.3.3. In case, the user enables full replication the sub graph corresponds to the original task graph. The master serializes the sub graph in the common JSON format and sends it to each remote `Daemon`. The daemons receive this JSON description and deserializes it back into a local `TaskGraph`. If the task graph is a real sub graph, the `Daemon` inserts dummy source and sink tasks which it uses to insert buffer received from the master and read out the result. After the setup phase, the `Daemon` starts a local `Scheduler` in a new thread and waits for compute requests from the master node.

The master node begins execution as in the local case. As soon as the `RemoteNode` receives data, it serializes it into a compact binary format, sends it to the remote `Daemons` via the `Messenger` interface and waits for a response. The `Daemons` deserialize the data back into a `Buffer` and forward it to the dummy source task of the local sub-graph. The `Daemon` waits for the result of the dummy sink tasks and returns the serialized buffer back that to the requesting `RemoteNode`. If the last compute result is empty because of a

Listing 3.2 Low-level fair scheduling barrier.

```
int remote_barrier(volatile int* pending, int n_remotes)
{
    int index = g_atomic_int_add(pending , 1);

    while (g_atomic_int_get(pending) % n_remotes != 0)
        g_thread_yield();

    return index % n_remotes;
}
```

finished data stream, it sends a corresponding message back and cleans up the existing `TaskGraph`. After that, the `Daemon` handles new setup requests.

With full replication enabled, the remote `Daemon` uses the serialized JSON format to set up a copy of the received `TaskGraph` that does not contain additional dummy tasks. To assign local tasks the correct data partition, it uses the partition ID that the master node has sent. After the setup phase, the `Daemon` starts immediately with the execution and responds back to the master as soon as the local computation has finished.

Fair remote scheduling

The network throughput is lower than the bandwidth of a PCIe connection, thus it is necessary to send multiple data items to remote nodes in large bursts. For this, the local master uses input and output queues of varying lengths for each `RemoteNode` that are limited by default to 20. To maximize the utilization of a remote node, the input queues are filled during the computation. If the capacity of the output remote node queue is reached further fetching from the input queue is stopped until the capacity of the output queue falls below the output queue limit.

Because a daemon cannot bound its input and output queues, the capacity within the proxy node is limited to n_f elements and defaults to 10. A remote node will send up to n_f items at once to the daemon and immediately request results from the remote system. If at that point the remote node's output queue is empty, sending more items is blocked until a finished item becomes available in the output queue. This is repeated until the number of items fetched from the remote node equals the number of items previously sent to it.

To avoid preference for a single remote node, fair access to the send operation by multiple remote nodes must be guaranteed. In that case, a remote node performs the send-fetch-repeat loop before other threads would get a chance to send their data. This causes an underutilization of the remote daemons' resources. To ensure fairness amongst the remote nodes and minimize the amount of time an input queue is empty a fair scheduling scheme based on global busy-wait barrier is used (refer to listing 3.2). Before sending, each proxy node must acquire a global lock. After n_f items have been sent, the lock is released and the `remote_barrier` function called. When all remote nodes reach the barrier, each node has to sent up to n_f work-items. The same combination of global mutex and barrier is now used to fetch data from the remote daemon. An explicit barrier is set after every request to retrieve the results from the remote nodes in a balanced way and keep their output queues at the same level.

3.4.4 Related work

StreamIt is a Domain Specific Language (DSL) with a C-like syntax that is designed to express general streamed processing problems [47, 123]. It uses a compiler to turn stream constructs such as *splits* and *joins* into valid C code. Depending on the back end, the code is executed on a a multi-threaded run-time or in the case of Sponge on GPUS [59]. Sponge uses the StreamIt description to compile custom CUDA code. By using optimizations such as stage fusion and memory mapping according to the code's intensity, Sponge is 3.2 faster than a hand-written CUDA implementation and 20 faster than a CPU baseline implementation of benchmarks from the StreamIt suite. StreamIt does not support multiple GPUS which is crucial for maximum performance on a single compute node.

Lime [38] and GStream [138] are library approaches to express computing pipelines written in Java and C++. GStream is a thin framework around the GPU run-time and requires the developer to write CUDA kernels manually. Lime, on the other hand, is able to generate OpenCL kernels from the annotated Java sources. GStream allows multi GPU task distribution via MPI, which scales accordinglingy but has performance drawbacks when using multiple GPUS on the same machine. Lime does not support task distribution on multiple GPUS.

Neumeyer et al. investigate massively scaling streamed data processing with the S4 system [92]. It is a Java-based distributed system and set out as an alternative to Hadoop cluster applications. Contrary to the preceding systems, it is throughput-oriented targeted towards high frequent data processing of small data items. The authors use the processing of ad-related click events on Yahoo's website as an example. At 20 000 events per second the relative error increases to 4.2 per cent. Recent work has shown how embedded systems-on-chips process streamed data [57, 89].

Reconstruction packages that cover the same SRμCT use case as our intended system are focused on usability rather than performance. This includes PITRE which is written in the proprietary scripting language IDL [23] and TomoPy which is written in Python and single-threaded C code [49]. Apart from these SRμCT specific packages, ASTRA and CONRAD are specific frameworks for accelerated tomographic reconstructions. The ASTRA tomography toolbox is a high-performance, domain-specific MATLAB that provides FBP, ART and other iterative techniques [98]. CONRAD is a cone-beam reconstruction library for medical use and incorporates FBP and ART reconstruction algorithms running on a Java/OpenCL platform [84].

3.5 Optimization

Providing a GPU kernel to process data does not suffices to meet the soft real-time requirements if data becomes larger and the data processing system is not optimized. In the following sections, we will present methods and optimization strategies to use the full potential of a GPU-based heterogeneous compute system.

3.5.1 Batched data transfer

In a pipelined system, tasks may receive up to one data item per process iteration. In case of adjacent CPU and GPU tasks, the data item must be transferred through the PCIe bus according to the protocol given in 3.4.2. Generally, due to startup latencies and transfer overheads of the PCIe bus, transferring smaller blocks of data requires more time per transferred byte than larger blocks of data. Thus by transferring single data buffers, the available bandwidth capacity might not be utilized to its full extent. One possible solution to improve the throughput, would be to enlarge the data size, however, there are limitations given by the data source itself.

Instead of inflating the data size artificially, another solution to improve the throughput involves queueing buffers and transferring multiple items in a single large *batch*. On the scheduling level, this changes a processor task with $\gamma_i(n) = n$ to a reductor task with $\gamma_i(n) = k, k < n$. Integrating these data transfers requires changes in the processing protocol of the scheduler and the data handling. First, the scheduler must defer execution of the receiving task until it has collected the required number of input buffers. This is obviously the case for an n for which the bandwidth $B(nm) < nB(m)$. Second, the original receiving task cannot use the transmitted OpenCL memory buffer as it is because it is larger and contains more data than expected. Changing the task to cope with larger buffers is not possible but one can create reduced views on OpenCL buffers using `clCreateSubBuffer()`. Calling this function and setting up the data structures is then conveniently hidden behind the general `Buffer` interface.

3.5.2 Kernel fusion

With a small number of available GPUs and long paths $P = (t_1, \ldots, t_k)$, it is likely that the same GPU executes the same tasks. If the tasks themselves exhibit only low computationally intensity, the overhead of launching each task's kernel can mitigate the performance gain of a GPU. Moreover, a high memory-access-to-computation ratio caused by the each kernel's read and write to global memory reduces possible processing throughput. To reduce the overhead of launching individual kernels and improve memory locality, we investigate the fusion of GPU kernels of $t_1, \ldots, t_k$ into a single combined kernel. This single kernel is used by a single task t_{new} with $\deg^{-}(t_{\text{new}}) = \deg^{-}(t_1)$ and $\deg^{+}(t_{\text{new}}) = \deg^{+}(t_k)$.

For this approach, we can assume the following performance model. Each individual kernel needs time to read input from write back the results to global memory as well as execute on a GPU. For a set of n kernels, we can determine three sets of time information for an input of m bytes: $R = \{R_i(m), \dots, R_n(m)\}$ for reading data from global memory, $W = \{W_1(m), \dots, W_n(m)\}$ for writing back to global memory and $E = \{E_1(m), \dots, E_n(m)\}$ for the execution. Furthermore, the host needs time L to start a kernel. We measured the startup latency and found no significant latency difference depending on the size of the kernel once the OpenCL run-time caches the compiled kernel object. We intentionally leave out the time for transferring the data from host to device. With these three parameters, the time T_i to run kernel i is bound by

$$T_i(m) = L + R_i(m) + W_i(m) + E_i(m). \tag{3.10}$$

The total time to run all kernels on a single GPU is thus $T(m) = \sum_i^n T_i(m)$. Fusing all kernels into a single kernel reduces the number of kernel launches to one and the number of global memory accesses to two. Instead of having each individual kernel read and write once, the single kernel must only read what the first kernel in the chain reads and write what the last kernel in the chain writes. Thus, the total time for the fused kernel reduces to

$$T_F(m) = L + R_1(m) + W_n(m) + \eta \sum_{i=1}^{n} E_i(m). \tag{3.11}$$

$\eta \in \mathbb{R}$ describes the performance gain (or penalty) due to larger kernel. Given (2.3), the speed up will be positive if

$$\sum_{i=1}^{n} T_F(m) < T_i(m) \Leftrightarrow \sum_{i=1}^{n} E_i(m)(\eta - 1) < \sum_{i=2}^{n} R_i(m) + \sum_{i=1}^{n-1} W_i(m) \tag{3.12}$$

in other words, if the compute to data access ratio is low.

To implement kernel fusion in the system, we added a pre-processing step to the scheduler. In this step, paths of `TaskNodes` that implement GPU code are analyzed and the underlying kernel source extracted. The scheduler calls out to a Python module that receives the kernel sources and parses it into a *pycparser* C Abstract Syntax Tree (AST) [8]. In order to parse the OpenCL kernel code correctly, we modified pycparser's lexer and parser stages so that it is able to recognize OpenCL data types and address space qualifiers. In the next step, a merge algorithm matches the ASTs of producer and consumer kernels and rewrites read and written memory locations. Instead of letting a producer write into its own output buffer, the write expression is modified such that it writes into the consumers input buffer. After analyzing memory access patterns, barriers and memory fences are inserted in order to ensure consistency. In the last step, the final AST merged from all n kernels is used to generate OpenCL code that is used in a single OpenCL task instead of the original GPU tasks.

3.5.3 Buffer views

Reconstruction algorithms for non-parallel-beam geometries such as cone-beam CT and laminography require access to large amounts of data. While a CPU-based system with large amount of main memory can compute these algorithms, it is still impossible to fit all data on a GPU's on-board memory.[4] For these applications, the data must be split up and processed iteratively in blocks containing a subset of the original data. To reduce the amount of data copies, we use different strategies depending on the data location.

First, let an d-dimensional *array* $A := N_1 \times N_2 \times \cdots \times N_d$, $N_i \in \mathbb{N}$ be defined as a block of $\prod_1^d N_i$ elements laid out contiguously in memory in row-wise fashion. That means that the linear memory address a of an element $(n_1, n_2, \ldots, n_d), n_k \in [0, N_{k-1}]$ is given by

$$a = n_1 + N_1(n_2 + N_2(n_3 + \cdots + N_{d-1}n_d)) = \sum_{k=1}^{d} n_k \cdot \prod_{l=1}^{k} N_k \tag{3.13}$$

A *view* v of a d-dimensional array A with size $\vec{s} = (s_1, s_2, \ldots, s_d)$, $s_k \leq N_k$ and origin $\vec{o} = (o_1, o_2, \ldots, o_d)$, $o_k \leq N_k - s_k$ is a d-dimensional sub array of A, where for each index $(i_1, \ldots, i_d)$, $i_k \leq s_k$ v is defined by

$$v[i_1, \ldots, i_d] = A[o_1 + i_1, \ldots, o_d + i_d] \tag{3.14}$$

If $\exists k \colon s_k < N_k$, v cannot be substituted but must be copied. The size difference $N_k - s_k$, however, causes a gap (also called stride or pitch) between consecutive rows of v, which is why a simple `memcpy` over the total size of v is not possible. Instead of naively copying all elements in d nested loops, we optimize for the common cases. From (3.13) we see that smaller indices change faster, hence if $\exists i \leq d \colon s_1 = N_1 \wedge \ldots \wedge s_i = N_i$, then we can use a $\prod_{j=i+1}^{d} N_j$ `memcpy` operations to copy $\prod_{j=1}^{i} N_j$ elements per operation on the host or an equivalent number of `clEnqueueWriteBuffer` or `clEnqueueReadBuffer` operations if data crosses devices. In case the data is already located on the GPU, we use the `clEnqueueCopyBufferRect` function which takes the origin o and pre-computed pitches, to copy memory directly on the device.

[4] Currently available high-end GPUS provide a maximum of 16 GB.

3.6 Summary

In this chapter, we presented a compute model that maps streamed data processes onto heterogeneous compute architectures. It uses task graphs to describe streaming algorithms and task mapping heuristics to distribute work among the system's processors. We propose a mapping strategy that replicates task nodes and maps them statically to the available processing resources. The proposed mapping heuristic guarantees data locality and reduces unnecessary data transfers. By replicating proxy nodes instead of sub-graphs, we can scale with cluster nodes. A UML model completes the description of the architecture.

Based on the proposed architecture, we presented optimization strategies to improve the run-time and bandwidth usage: Grouping data transfers in batches improves utilization of the PCIe bus and a proposed kernel fusion approach eliminates data transfers on subsequent kernel invocations.

4 Data acquisition and process control

Modern synchrotron X-ray imaging experiments provide flexible setups to acquire data for a wide range of applications. This requires a coordinated effort to manage high-speed data acquisition, data processing and device access and guarantee a continuous and sustained data flow. Most experiment workflows, however, are defined in an *ad hoc* fashion and do not focus on the integration aspect thus missing important optimization potential. In this chapter, we will present the core components for data acquisition and experiment control. The UCA architecture is a generic, low-latency and high-throughput data acquisition interface used for contemporary high-speed detectors. The *Concert* experiment control system spans a bridge between low-level device access and high-level experiment workflows. By exposing asynchronous device access, the data processing framework presented in 3 and the UCA architecture through a simple user interface, beamline operators are able to define arbitrary experiment workflows.

4.1 High-speed data acquisition

X-ray imaging applications have specific detector requirements concerning maximum frame rate, spatial and temporal resolutions, dynamic range and noise behaviour. CCD detectors provide the best signal-to-noise ratio for slow sample scans, while CMOS and hybrid SCMOS detectors perform best for high-speed recordings of dynamic processes. Table 4.1 lists commercial and research detectors as well as their technical specifications.

For high-speed data streaming, commercial vendors implement the CameraLink (CL) industry standard that specifies an electrical LVDS connection [12] between the detector and a PCIe *frame grabber* connected to the acquisition machine [114]. CL's full configuration profile specifies the transmission of 64 bit of pixel data at a clock speed of 85 MHz resulting in a total a bandwidth of about $5.44\,\mathrm{Gbit\,s^{-1}}$. Besides the electrical protocol, connector and cable specification, CL stipulates a low-level serial communication protocol between frame grabber and detector. Although the CL standard foresees the serial protocol as a means to control the detector firmware, the standard does not prescribe specific application semantics. Hence, all vendors define their own application-specific protocol. These protocols are often proprietary and incompatible with the detectors of other vendors.

Table 4.1 CMOS- and SCMOS-based detectors used for fast X-ray imaging. The third column denotes the frame rate achieved at the maximum resolution, higher frame rates are possible with a reduced field of view. The data rate is the maximum possible transfer rate from detector to machine.

Detector	Resolution	Frames/s	Bit/pixel	MB/s	Interface
UFO	2048 × 1088	330	12	1402.5	direct PCIe
pco.edge	2560 × 2160	100	14	1054.7	CL + PCIe
pco.dimax	2016 × 2016	1279	12	250.0	CL + PCIe
Andor Neo	2560 × 2160	30	16	316.4	CL + PCIe
PF MV2	1280 × 1024	488	8	610.0	CL + PCIe

In contrast to the frame grabber approach, IPE's hardware group for embedded systems developed a Xilinx Virtex 6 FPGA-based pixel detector (codenamed UFO). This system combines a custom sensor readout logic with a direct PCIe cable connection for high-speed data transmission [21]. With the on-board Direct Memory Access (DMA) logic and an eight lane PCIe 2.0 configuration, the detector can stream the data from the sensor directly into the user memory at up to 12 Gbit s^{-1}. Although, modern frame grabber cards use on-board FPGA modules for basic pre-processing, a custom FPGA-based read-out electronic allows for internal feedback loops taking the sensor into account [117]. To parameterize these feedback loops and configure the data acquisition, the UFO FPGA defines a register-based hardware model.

Both, the insufficiently specified CL serial communication protocol and the register-based hardware model of the UFO detector prohibit a common access interface. Such an interface, however, is necessary to build data acquisition systems that are independent of detector-specific details. Therefore, we have to devise an *adapter layer* between the control application and the hardware interface to provide a generic interface. Figure 4.1 outlines the core UCA architecture.

4.1.1 Requirements

To satisfy the demands of X-ray imaging experiments embedded in a fast and flexible control system, the adapter layer must fulfill the following list of requirements:

1. The adapter must provide mechanisms to start an acquisition and read out frame data consisting of a pixel arrays with bit depths between 8 and 16 bits. The exposure of the sensor may be triggered by a software stimulus of the adapter layer, automatically by the detector according to a set frame rate or by an external hardware signal. To cover all use cases, it must be possible to read the data from the detector using blocking, synchronous calls or asynchronous callbacks.

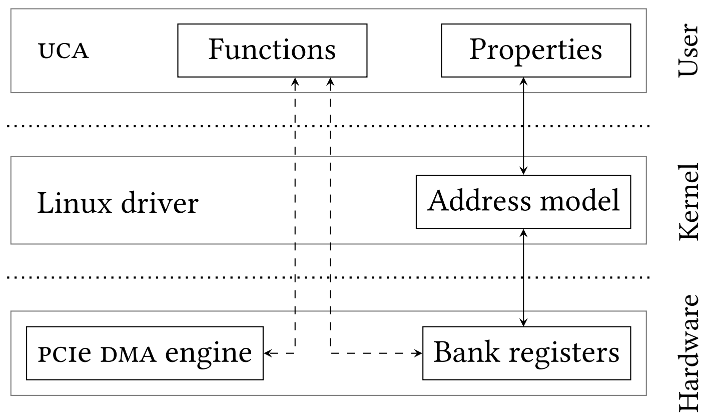


Figure 4.1 For the UFO detector platform, UCA maps the register model to a list of properties and accesses control functionality via the driver's event structure. For other cameras, UCA wraps the vendor-specific libraries.

2. It must be possible to add arbitrary detector-specific parameters without changing the core adapter interface. The parameter can be primitives such as characters and numbers but also structured data like arrays and strings. Minimum meta data about the parameters must include the data type, a range or list of possible values, units and a description.

3. The adapter must support reading data from streaming and buffering detectors using a single access interface. Buffering detectors, such as the pco.dimax, are used for the highest frame rates, in which case it is not feasible to transmit the data at the maximum frame rate using the employed data link. Thus, the adapter must be able to separate the acquisition phase from the readout[1] phase. Streaming detectors combine both phases and transmit the data on-the-fly. Mixed operation must be possible for detectors that use streaming for a live preview.

4. To enable high throughput and low latencies, the adapter must not limit the bandwidth provided by the low-level detector library or vendor SDK. In particular, it must be possible to let the frame grabber or low-level driver write directly into a supplied user memory buffer without requiring additional memory copies, i.e. allow zero-copy data transfer.

5. In order to meet security and stability demands, it must be possible to acquire data on a remote machine. Only the network bandwidth or the detector frame rate itself should limit the effective data throughput between acquisition and processing

[1] With *readout* we mean the transmission of data from the detector to the host machine and not the transmission of pixel data from the sensor to the detector readout logic. Initiating the transmission sensor readout is caused by *triggering* an exposure.

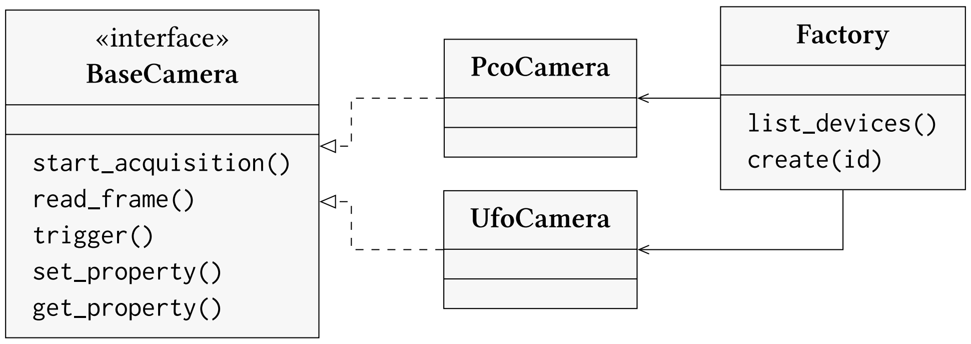


Figure 4.2 UML class diagram of the UCA architecture with concrete implementations of a PCO and the UFO detector.

machine. Moreover, existing control system interfaces should control the remote detector as far as possible. Ideally, a remote detector can be used transparently as a drop-in replacement for a local detector.

4.1.2 Unified detector access

The core abstractions of the UCA architecture are based on the basic UML class hierarchy shown in figure 4.2. All vendor-specific detectors implement the `BaseCamera` interface. This interface provides generic methods to initialize a detector, start an acquisition, do a software trigger and initiate a data readout. These methods call private virtual methods that are overwritten by the derived detector classes which in turn dispatch to the corresponding vendor-specific library or driver. In the example, the `PcoCamera` class is an implementation of the `BaseCamera` providing functionality to control the pco detectors listed in table 4.1. In order to control the application behavior, each public method checks the current state against an implied FSM. In case of an invalid state transition, for example when the user tries to stop an acquisition before starting it, the call is aborted. If the state transitions succeed, the call is dispatched to the private method.

Although, all detectors follow the general interface defined by the `BaseCamera` interface, detectors also have additional device-specific options and parameters such as sensor gain or temperature values. A *property* system maps these parameters to unique identifiers that are accessed by the user via single `get` and `set` methods. Each property consists of a name, a type, value ranges and a string describing the parameter. The detector class implements the access to a property, by overriding the `get` and `set` methods and handle the supplied property ID. In addition to the standard property meta data, we associate optional units with a property to describe its physical representation. Units can be real SI units such as second and meter or descriptive meta data such as the ordinality, for example the number of recorded frames. The advantages of a flexible property system compared to simple object attributes are programmatic enumeration and the

Listing 4.1 Creating a new detector object for a pco detector and setting exposure time to 1.2 seconds and reducing the frame width to 1024 pixels.

```
UcaCamera *detector = uca_plugin_manager_get_camera("pco");
g_object_set(detector, "exposure-time", 1.2, "roi-width", 1024, NULL);
assert(uca_camera_get_unit(detector, "exposure-time") == UCA_UNIT_SECOND);
```

possibility to register property change handlers. The latter allows a front end user interface to implement the model-view-controller pattern in order to update a graphical or textual representation whenever a property value changes.

The core and the detector-specific code are separated on the binary level in order to increase modularity, permit independent development of new detectors and simplify the deployment of compiled code. This is achieved by linking derived detector implementations into separate shared libraries. However, this also means that applications using the adapter cannot directly instantiate a new detector. Instead, an application uses an instance of `Factory` class to load the shared library at run-time and instantiate the contained detector object using a plug-in mechanism. During its construction, the `Factory` object determines the available plug-ins by checking known and configurable locations for shared libraries. The `Factory` then assigns a unique identifier string derived from the filename to each library. For example `libpcocamera.1.2.so` becomes the plugin name `pco` that the client application passes to the `Factory`. The `Factory` provides an enumeration function to list the available detector identifiers. Listing 4.1 gives an example how to set up a new detector, change properties such as exposure time and region of interest and assert that the unit of exposure time is seconds.

Memory management

To cover different application use cases, detector and frame grabber implementations typically support a mixture of synchronous and asynchronous calling behaviour as well as blocking and non-blocking data readout modes. While *synchronous* reads block the caller until data is ready and returned, *asynchronous* readouts call a previously registered callback function whenever data is ready to be processed. *Non-blocking* or *buffered* readout refers to the capability of a detector to trigger and read out the data without user intervention. For high-speed recordings, the connection bandwidth often does not suffice to transmit the data in real-time. Hence, high-speed detectors that support this mode posses a large *in-camera* memory storage to buffer frames[2] for the acquisition time. After the acquisition completed, the frame grabber transmits the frames to the caller either frame by frame or as a sequence. *Unbuffered* or *blocking* data readout are the main readout modes to provide live streaming of data. To provide a unified inter-

[2] The pco.dimax is able to record thousands of frames per second with an internal memory capacity of 36 GB

Table 4.2 Minimum and maximum number of CPU memory copies required for all acquisition modes combinations.

Synchronous	Blocking	n_{min}	n_{max}
yes	yes	0	1
no	yes	0	1
yes	no	2	3
no	no	1	2

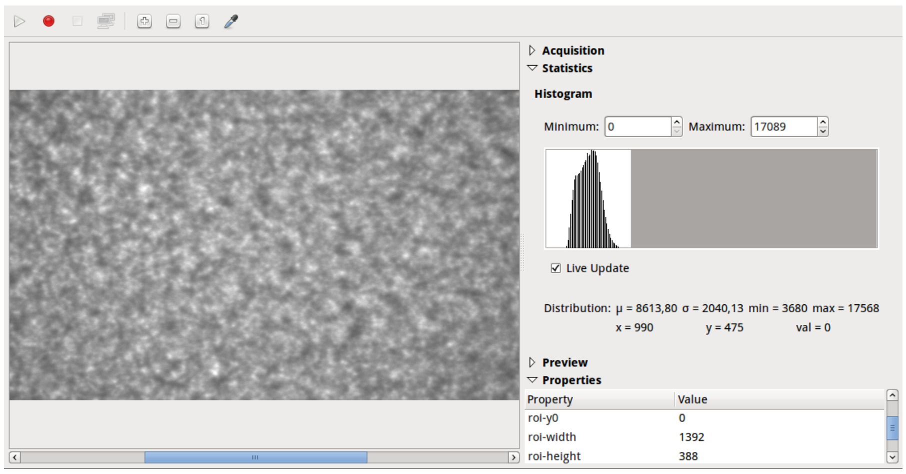
face to all detector types, the UCA core implementation uses software fall backs for all modes that are not implemented directly in hardware.

A software-side ring buffer that stores the most recent frames provides non-blocking behaviour on the client side. This allows us to separate the acquisition of data from the detector and the transmission of data to the client as per the third requirement. With this mode enabled, an acquisition thread reads out data continuously from the detector to fill the ring buffer. The user of the detector either reads out the data synchronously or registers a callback that is notified asynchronously when a frame is ready. Synchronous readout requires an additional memory copy for the user supplied buffer which can be avoided in the asynchronous callback handler if the buffer is used as is. If the detector vendor allows writing directly into the ring buffer, a single memory copy from the ring buffer to the client memory suffices. In synchronous, blocking mode, the client initiates the data transfer and supplies a suitably sized memory buffer. In this case, the vendor can write directly into the client memory without an additional CPU memory copy. If the hardware or driver does not support a direct copy, the CPU has to perform an intermediate memory copy.

Table 4.2 summarizes the calling behaviour and readout modes as well as the required minimum and maximum number of memory copies that the CPU must perform. Due to the intermediate buffering, non-blocking readout guarantees a consistent readout rate, at the cost of increased latency, as long as the ring buffer is not filled completely. Synchronous blocking guarantees lowest possible latencies and potential zero-copy behaviour as long as the client can process the data fast enough and the vendor supports writing into the supplied buffers.

Implementation details

The UCA interface is implemented as a C library called *libuca*. Like the UFO compute framework, it is based on the GObject framework from which it uses the type, class and property system. The GObject introspection mechanism allows for integration with scripting and high-level languages such as a Python. Because Python code cannot access raw pointers cannot directly, we provide a small module that converts the raw C



Minimum: 0 Maximum: 17089

☑ Live Update

Distribution: μ = 8613,80 σ = 2040,13 min = 3680 max = 17568
x = 990 y = 475 val = 0

Property	Value
roi-y0	0
roi-width	1392
roi-height	388

Figure 4.3 Basic preview GUI using libuca with a pco.dimax acquiring X-ray projections of a liquid.

data array to a NumPy array.[3]

Up to now, the UCA framework provides access to all detectors given in table 4.1, slower X-ray detectors from Dexela and Basler as well as direct X-ray converting Medipix and Timepix boards developed at IPE [15]. In the case of the UFO detector platform, we use the zero-copy transfer mechanism to decode and decompress the frame data on-the-fly and directly into the user's memory. By using a custom data format based on the 12-bit dynamics of the sensor and vectorized SSE instructions, we can decode three 12-bit pixels simultaneously into three 16-bit words, giving a speed up of up to three compared to the naïve conversion that uses manual bit shifts.

Due to the flexible and generic approach, a wide range of tools uses the UCA implementation for different use cases. Command-line programs aid in the diagnostics and profiling of specific detectors or generate off-line documentation of detector properties. For quick previews, simple acquisition purposes and basic data assessments, beamline operators use the generic Graphical User Interface (GUI) shown in 4.3. For slow data acquisition purposes, a Taco Next Generation Objects (TANGO) device server[4] allows remote access to the frame data and parameters. Figure 4.5a shows the core architecture of this setup.

4.1.3 High-speed remote data transfer

To ensure stable control and reliable data transfer at the beamline, the data is acquired and processed on separate machines. Although, we can cover this scenario using the aforementioned TANGO server, high latencies and low data throughput inhibits soft

[3] NumPy is the *de facto* Python matrix and array library for scientific computation.
[4] TANGO is a beamline control system that we will introduce in more detail in the next section.

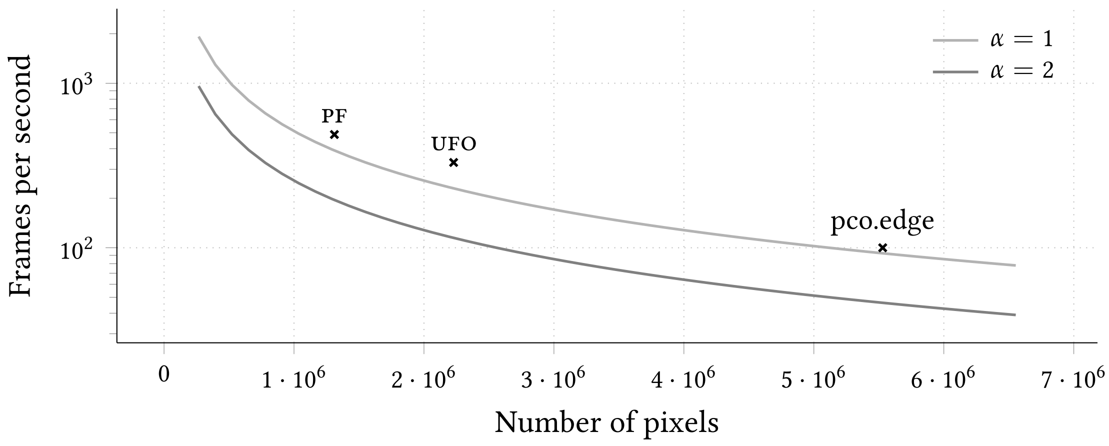


Figure 4.4 Theoretical frame rate achieved with TCP/IP on a 10GbE network for frame sizes between 512^2 and 2560^2. Each detector requires more bandwidth than is available for sustained streaming.

real-time process control. These latencies and throughputs are caused by the TCP/IP-based Common Object Request Broker Architecture (CORBA) [129] implementation that is used for remote method calls. The 10GbE network technology limits the maximum attainable bandwidth to 10 Gbit s^{-1}. The protocol overheads and processing demands of the network interfaces limit the 10GbE net peak bandwidth between 4.1 Gbit s^{-1} and 7.6 Gbit s^{-1} [41, 61]. However, not only the network technology but also CORBA itself limits the attainable bandwidth. CORBA communication is susceptible to congestion if the number of concurrently communicating peers increases [63]. In an X-ray imaging experiment, this communication cannot be avoided because the experiment control system controls motors and shutters the entire scan time.

Assuming the reported worst case throughput of 4.1 Gbit s^{-1} and neglecting the impact of CORBA congestion, a 10GbE network has a throughput of about $M := 488\,\mathrm{MB\,s^{-1}}$. Given typical detector data streams with 8 to 16 bit per pixel, we can estimate the attainable frame rate $f(n)$ for a given number of pixels n by $f(n) = M/\alpha n$, where $\alpha \in [1, 2]$ denotes the number of bytes per pixel. Figure 4.4 shows simulated results of $f(n)$ for frame sizes between 512^2 and 2560^2 pixels. The data points in figure 4.4, indicate the required bandwidth of the fastest detectors given in table 4.1. As we can see from the figure, 10GbE is not sufficient to stream the data at full frame rate and frame size for any of those detectors. Not only are we unable to stream frames at rate the detector outputs the data but we also saturate the communication network and negatively affect other services that share the same network.

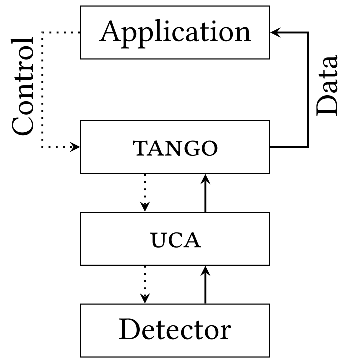
To solve the throughput problem, we use a secondary data channel for exclusive transmission of frame data to the recipient machine [36]. The TCP/IP connection is used solely for transferring handshaking, configuration and control messages thus reducing the overall load on this channel. As we have shown, the bandwidth restrictions of a 10GbE ethernet connection prevent transmission of data acquired by the fastest streaming detectors. Consequently, we use InfiniBand interconnect that provides a higher link bandwidth and can scale more easily by aggregating multiple links [73]. Moreover, InfiniBand Remote Direct Memory Access (RDMA) allows an InfiniBand Host Channel Adapter (HCA) to circumvent the CPU and read from and write directly into the memory buffers, thus reducing the load on the sender and receiver CPUS.

Figure 4.5b shows the extension of the existing data acquisition architecture necessary for remote acquisition. The original TANGO server uses the UCA interface solely to acquire data from a detector. A Kserver TANGO server replaces the original TANGO server to handle regular TANGO requests and accept InfiniBand connection requests for data transmission. The Kserver maps the detector parameters by mapping TANGO attributes programmatically to the UCA parameter infrastructure to provide remote control of the detector. On the receiving end, a UCA plug-in uses the existing TANGO connection to negotiate connection details for the data transmission and provide end-user control of detector parameters. After configuring the detector, the UCA Kclient plug-in connects to the Kserver component and receives the data stream via RDMA. The data is written either directly into the client buffer or into the ring buffer described in the previous section. By providing a wrapper UCA plugin that contains both the InfiniBand and TANGO clients, the user can access the remote detector transparently and indistinguishably from a local detector. If the detector supports writing into the supplied memory buffers, transmitting data from the remote detector to the client memory via RDMA requires only a single explicit memory copy.

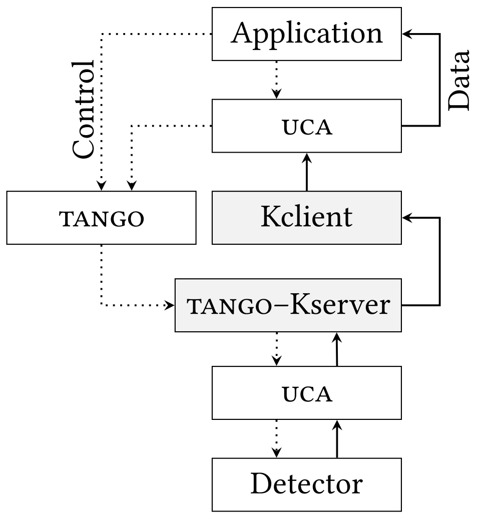
As before, we can estimate the maximum possible frame rate but for an InfiniBand network. At IPE, an InfiniBand 4x Quad Data Rate (QDR) network provides a theoretical peak bandwidth of $40\,\mathrm{Gbit\,s^{-1}}$ according to the specifications of the vendor. The nominal bandwidth of the network is $32\,\mathrm{Gbit\,s^{-1}}$ as per the encoding of 8 bit data in 10 transmitted bits. Profiling using the provided InfiniBand benchmark tools has shown that the maximum bandwidth saturates at about $31\,\mathrm{Gbit\,s^{-1}}$. Protocol and operating system overheads cause this performance impact. Assuming a bandwidth of $31\,\mathrm{Gbit\,s^{-1}}$, the maximum frame rate that can be attained is shown in figure 4.6. It shows that each detector can stream its projected maximum frame rate by a safe margin. In particular, the network bandwidth exceeds the detector throughput of the Photon Focus MV2, the UFO detector and the pco.edge by factors of 29.6, 2.6 and 3.5.

4.1.4 Related work

Similar to the UCA architecture, the Library for Image Acquisition (LIMA), developed at ESRF, unifies the specific detector interfaces. Unlike UCA, LIMA integrated additional functionality such as data processing and storage in different formats [58]. The monolithic architecture consists of a detector hardware interface that corresponds to the base



(a) Conventional acquisition.



(b) Fast remote acquisition.

Figure 4.5 Comparison of state-of-the-art acquisition using TANGO for both control and data transfer via a 10GbE interconnect and improved throughput with an InfiniBand data channel. On the detector side, UCA communicates with the hardware, on the client side, a UCA client plug-in provides transparent access to the remote detector.

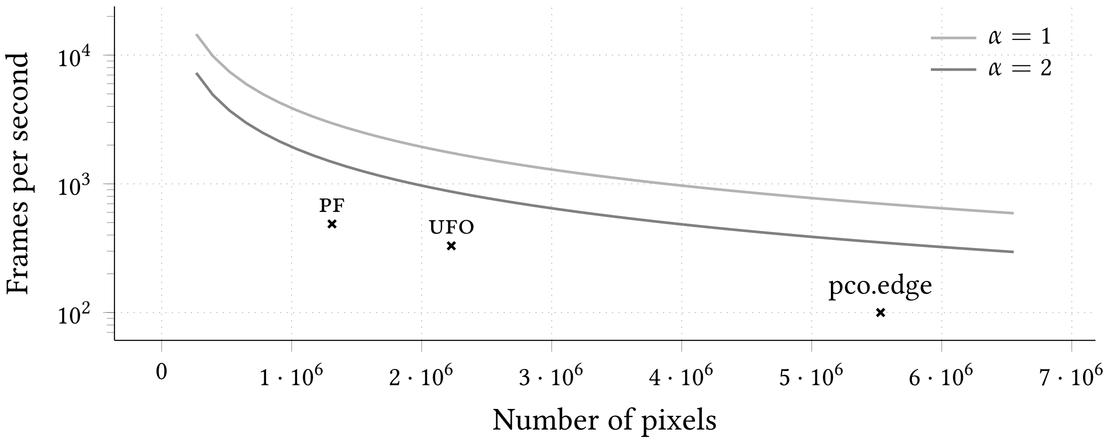


Figure 4.6 Simulated frame rate using QDR InfiniBand model parameters and frame sizes between 512^2 and 2560^2. The bandwidth suffices to stream the data directly from the detector to the acquisition machine.

interface of the UCA architecture as well as a control layer and data processing modules built on top of the lower layers. Contrary to our proposed minimal interface, LIMA's hardware and control interface uses fixed methods to provide detector-specific capabilities. Because these must be known at compile time, the generality and straightforward extensibility as stated by the second requirement suffers. Although, each detector is linked into its own shared library, LIMA does not provide a mechanism to enumerate and instantiate detectors dynamically at run-time. Instead, one must compile and link directly against the specific detector library. Due to the static nature of LIMA, it is impossible to write generic and dynamic client applications. Instead, the authors decided to provide flexibility at the process level and deploy a TANGO server for each detector.

Unlike LIMA, UCA deliberately avoids data processing and storage on the acquisition level. This enforces a strong separation of concerns and increases modularity. We also argue that specialized tools and libraries are faster for processing and storing data and easier to maintain than a single, monolithic library. To support this model, we promote a low-latency data transfer mechanism which is on par with the performance provided by the monolithic model. However, because LIMA is a generic detector interface, we were able to integrate a LIMA detector within UCA.

4.2 Experiment control

As outlined by question 5, an X-ray imaging experiment control system must provide enough flexibility to adapt quickly to an evolving experiment environment and perform fast enough in order to manage hundreds of hardware devices, is required to model all experiment types conducted at an X-ray imaging beamline. To enable feedback processes it is necessary to integrate both data acquisition and data processing. In this section, we propose a generic control framework to describe experiment processes and workflows. It integrates our high-throughput data acquisition and the high-performance compute frameworks to enable quick data assessment and fast feedback processes for SRμCT experiments. Compared to existing solutions, data is kept in fast main memory at all times and only relevant results are written out to disk. This allows us to keep up with the high data rate produced by the detector [131].

4.2.1 Requirements

To meet the objectives stated in 1.2 and answer question 5, we state the following requirements for a high-level experiment control system:

1. The control system must provide a standardized, generic high-level device hierarchy and an API that is independent of the underlying hardware and suitable for the rapid development of automation processes. The main focus lies on calibration and adjustment tasks as well as the acquisition and processing of radio- and tomographic

images for SRμCT experiments.

2. Asynchronous device control is mandatory to reduce the time spent on synchronization and therefore increase the throughput of processes and experiments. Accessing the devices in an asynchronous manner should be transparent to the user.

3. Existing technologies should be re-used wherever possible to reduce development time and increase interoperability. If new developments are necessary, the intended architecture must provide open interfaces for additional extension.

4.2.2 Low-level control

The low-level control of a beamline is split into safety-critical (for example door locks) and experiment-related device access. For safety-critical applications, Siemens SIMATIC PCS 7 is used in most installations [9] whereas beam- and experiment-related hardware control is developed and provided by the accelerator communities. Here, TANGO [48], the Experimental Physics and Industrial Control System (EPICS) [27] and the Distributed Object-Oriented Control System (DOOCS) [52] are the main low-level control systems. The experiment control systems distribute access across a network of machines to increase scalability and improve fail safety. Each machine accesses a particular device with a device-specific protocol and exposes data and control through control-system-specific and device-agnostic network protocol. DOOCS and TANGO represent data by remote object attributes and control via Remote Procedure Calls (RPCS) communication primitives. DOOCS uses a custom C++ RPC implementation, whereas TANGO relies on the open CORBA standard [129]. EPICS uses process variables exclusively to abstract device access. To expose these process variables on a remote machine, EPICS distributes the data using a custom messaging protocol. Although both EPICS and TANGO are used at ANKA, we will focus on TANGO for the remainder of this section because it is the *de facto* standard low-level experiment control system in use, whereas EPICS is used to control the machine operating the beam storage ring.

As indicated in 4.1.3, TANGO is a thin abstraction layer between the hardware devices and the application. Each device is controlled and represented by a *device server* that *client* applications connect to in order to gain remote access [122]. A central SQL-based device server keeps a list of known hardware device servers and resolves requested server names from a hierarchical namespace to negotiate a connection request from the client to the server. Due to the variety of devices, the TANGO server model provides a generic access interface consisting of *commands* and *attributes*. Commands initiate an action on the device; attributes represent device-specific parameters or process variables. Although, each device is part of a device class, TANGO does *not* provide nor enforce a strict semantic device class hierarchy. In practice, each concrete device implementation inherits directly from the root device class which impedes type polymorphism between devices of the same "family". This leads to the situation that TANGO

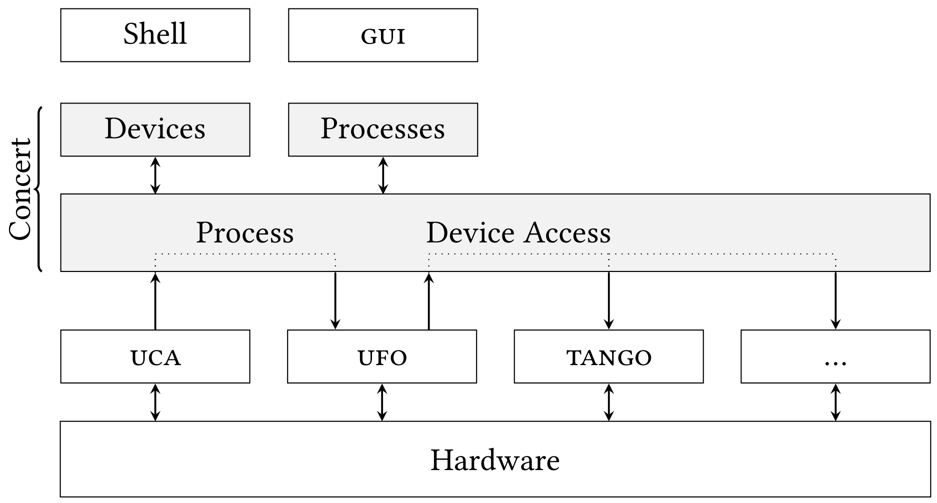


Figure 4.7 The control layer integrates the low level data acquisition, process and control interfaces and provides device and process abstractions. On top of that generic shell or graphical user interfaces are built.

applications are written for specific use cases. As a result, this makes it impractical to describe flexible experiment types in a generic manner and requires large efforts to implement tomographic scans. In these circumstances, the experiment setup changes depending on the sample and required experiment condition.

4.2.3 Proposed system architecture

Because TANGO is widely adopted by a large number of synchrotrons, a full replacement of it is neither desirable nor realistic. Thus, we designed a high-level control system layer, called *Concert* that negotiates between the lower-level control systems such as TANGO or EPICS and the user or control applications as shown in figure 4.7. By abstracting from the low-level access details, we can structure devices in a semantically strict hierarchy and provide common data processing capabilities. Moreover, we can define device-independent *processes* working with a large class of device types to reduce code redundancies. The top-most representation layer uses *Concert* to provide either programmatic user access or a user interface such as a command-line shell or a GUI. By abstracting the differences of the low-level control systems, end-users and application developers can use a beamline and the employed devices in a common way.

The central idea of a control system is to map physical hardware devices to software components. An operator uses these software entities either manually or as part of a program to describe process sequences used for an experiment. In the most general sense, the control system abstractions expose *parameters* representing hardware values that can always be read and may be written, as well as *methods* to change the state of a device. Although methods could be represented by a protocol of parameter changes, separating method *policies* from access *mechanisms* helps hiding low-level details from the user and simplifies development of different, co-existing policies [79].

Parameters

Figure 4.8 illustrates the core architecture as a UML model. The foundation of the system is the separation of `Parameter` classes from the device itself. A `Parameter` encapsulate access to a single parameter, whereas devices expose auxiliary methods based on those parameters. The `Parameter` classes have mandatory getter methods to read values from a device and optional setter methods to write them. A documentation string describes the purpose of that parameter and how it relates to the device itself. To enforce the generality of the parameter concept, `Parameter` objects are not restricted to specific use cases and instead represent *arbitrary* device values.

To encapsulate digitized analog data that corresponds to a physical quantity as measured by a device, we use the `Quantity` class. This class derives from a `Parameter` and adds an associated unit to the `Parameter`'s value that is converted internally to a hardware-specific target value. Potentially wrong or malicious user input is checked for unit compatibility, e.g. setting seconds on a motor position throws an error. If this test passes, the `Quantity` value is converted to the device's base unit and checked if it falls into the user-defined *soft* limit range. If the second check turns out positive, the converted value is set on the device using native hardware access or by writing the corresponding TANGO attribute. The `Parameterizable` class groups `Parameter` objects and provides auxiliary methods to lock writing of parameters or save all current values on a stack for later resets. The latter is used to preserve a certain device state across seemingly idempotent processes. For example, after adjusting the focus of an objective lens, other devices that were involved in the process of adjusting the lens, should not be affected by the process.

Devices

`Device` objects are derived from the `Parameterizable` class and add additional locking mechanisms to prevent changing a device during operation. `Device` objects also enforce correct operation sequences by an implicitly defined FSM. The FSM is based on a hidden per-device state that is associated with one or more state transitions. A transition is either specified by annotating state-changing methods using a `@transition` decorator or by passing `Transition` objects to the constructor of a `Parameter` object. Listing 4.2 shows how possible states are assigned to the position `Quantity` property of a linear stepper motor and how arbitrary methods can trigger a state change. The construction arguments to the `Transition` is a list of valid entry states and a target state. Contrary to a formal FSM, a transition allows to change a `Parameter` to an intermediate state *during* the execution of a method or set operation. After successful completion of the operation, the `Parameter` is set to its final target state.

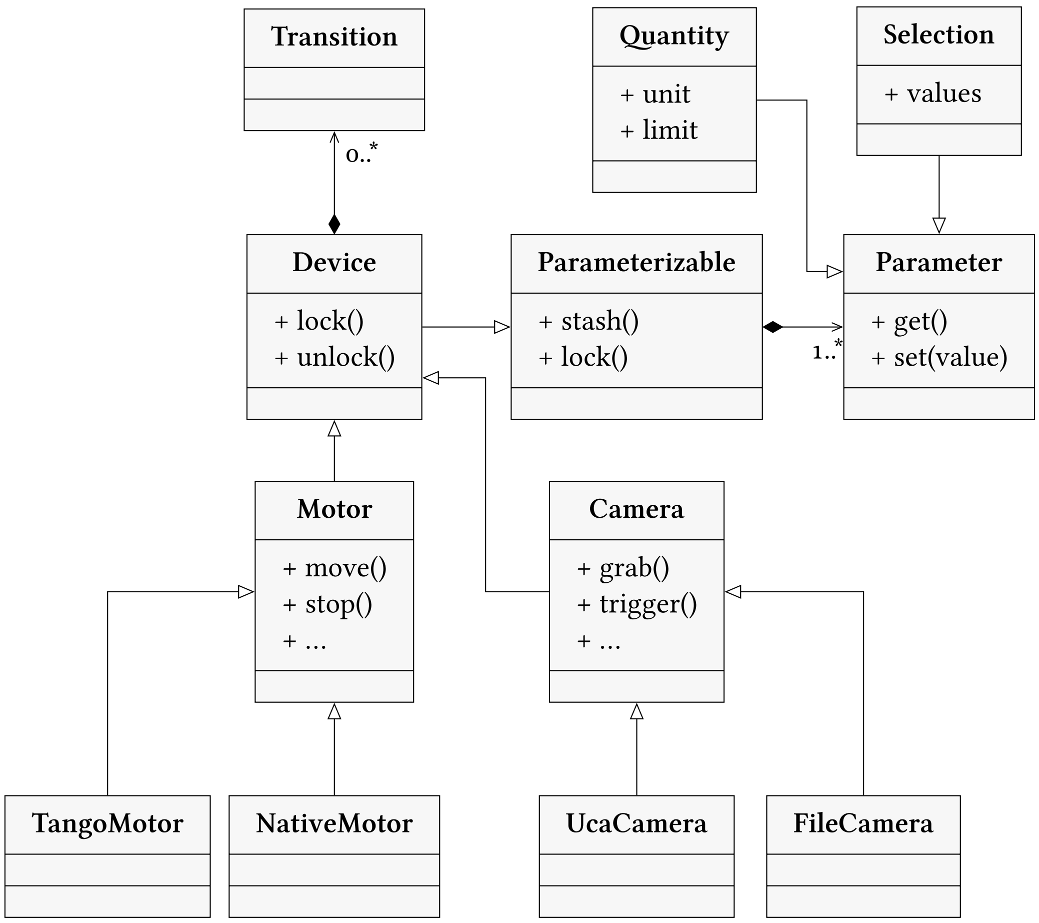


Figure 4.8 Simplified class hierarchy. `Parameter` and `Device` classes form the basis to build device types.

Listing 4.2 Implied state annotation for `Parameters` and methods.

```
class Motor(Device):
    position = Quantity(q.meter,
                        transition=transition(source=['moving', 'halt'],
                                              target=['moving', 'halt']))

    @transition(source=['halt'], target='moving')
    def move(self, delta):
        self.position += delta
```

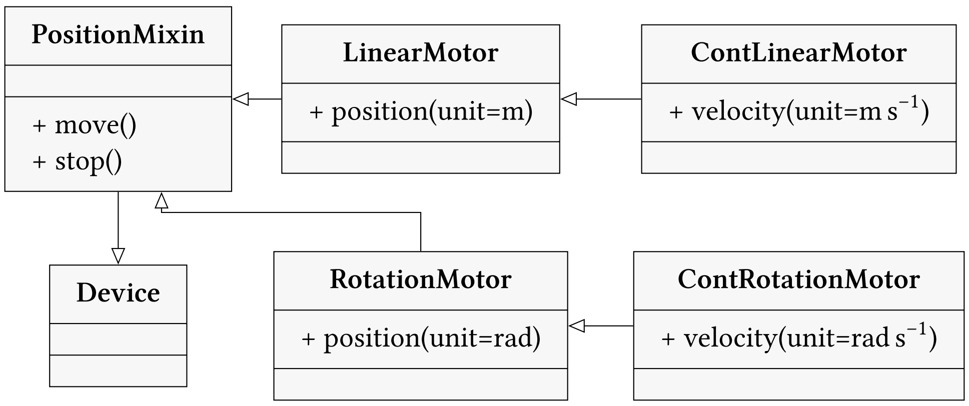


Figure 4.9 Motor classes and mixins to provide a common interface for lateral and angular stepper and continous motors.

To avoid the weak inheritance tree of the TANGO device classes, every class representing a hardware device *must* derive from a specific *device class* class. A device class[5] is a group of devices with common functionality. At the moment, the device class hierarchy specifies base classes for detectors, detector systems composed of detectors, motors, controllers, I/O cards, light sources, monochromators, positioners, pumps, scales, shutters and storage rings. From these device classes, hardware-specific classes are derived. For example, a concrete TANGO motor derives from the `Motor` device class and implements the imposed interface. The net effect is that client applications can rely on the polymorphism of types and use a TANGO motor in the same way as a native motor. The base class functionality is chosen to cover behaviour expected from a derived class. For more complex cases, deeper base hierarchies are provided as shown in figure 4.9. For example, both linear and rotational stepper motors inherit the `PositionMixin` interface which itself inherits from `Device` and uses the common position attribute to provide high-level functionality such as moving. On the second inheritance level, continuous motors add velocity properties to describe continuous motion.

Asynchronous operation

In any synchrotron experiment, multiple devices must be manipulated in the correct order to achieve a desired process sequence. However, manipulating the devices sequentially is a slow process due to high startup latencies and slow motor movements. Similar to parallel data processing, parallelizing the device access can decrease the overall process time and in turn improve the experiment throughput. The pre-scan phase of an X-ray tomography experiment illustrates the potential gains: After taking flat field images for noise reduction, a shutter is closed to obstruct the beam, while at the same time the sample can be moved into the field of view and the rotary stage is spun up.

[5] Which is not identical to a `Device` class.

Listing 4.3 Difference between synchronous and asynchronous parameter access.

```
# Synchronous access
motor.position = 1 * q.mm

# Asynchronous access returns a future
future = motor.set_position(1 * q.mm)
future.wait()
```

As with parallel computing, correct synchronization is crucial and necessary to avoid devastating effects such as physical device crashes.

To enable parallel, asynchronous device access, parameter accesses and device methods are encapsulated within *future* objects. A future represents a value that is produced by an asynchronous operation and becomes available at some arbitrary point in the future [132]. Asking for the `result` of a future either returns the value immediately or blocks as long as it is not yet ready. Apart from that basic requirement, a future also provides methods to query if it is `running`, `cancelled` or `done` computing. A user can also attach a callback to a future and be notified asynchronously about the completion.

Futures are created by submitting a regular function to an *executor* which uses any mechanism to implement asynchronous execution of the function. Within our architecture, we use Python's default `ThreadPoolExecutor` to run tasks asynchronously. As the name suggests, the `ThreadPoolExecutor` manages a limited pool of threads to process a submitted task. Additionally, we built a custom executor based on the *gevent*[6] event loop system that uses single-threaded kernel-level polling to provide asynchronous I/O. Because it can be difficult to identify problems and errors during asynchronous execution of code, we also provide a `NoFuture` implementation which executes the submitted tasks sequentially one by one. The user can set a run-time switch to choose between any of the three executor and future implementations.

By decoupling the parameter access from de-referencing the value, we can access multiple devices at the same time. The future objects themselves can then be used to synchronize with other asynchronous operations by chaining callbacks or waiting explicitly for a future to finish. As shown in listing 4.3, attribute-like parameter access is always executed synchronously because setting the parameter cannot return a future. Similar to parameter accesses, methods can be executed asynchronously. To denote methods for asynchronous execution, an `@async` decorator is placed before the method definition. The decorator wraps the low-level call and provides the same asynchronous interface as parameter accesses. This has the positive effect that device developers do not need to care about *how* parallelism is implemented but merely specify which methods should be executed asynchronously as shown in listing 4.4.

With concurrent operations, there is always the danger that two independent code

[6] `http://gevent.org`

Listing 4.4 Annotation of asynchronous methods.

```
# Partial class definition of Motor
class Motor(Device):
    @async
    def move(self, delta):
        self.position += delta

# Moving the motor asynchronously
f = motor.move(-2 * q.mm)
f.wait()
```

Listing 4.5 Locking of two related devices using the `with` statement.

```
with motor, detector:
    # Other processes cannot access neither motor nor detector
    motor.move(1 * q.mm)
    data = detector.grab().result()
```

paths access the same device asynchronously. This can lead to race conditions between device accesses with potentially dangerous outcome. To prevent such situations, each device has a lock that is activated manually or by accessing the device within a `with` block as shown in listing 4.5. Because the `with` statement is executed atomically from Python's point of view, deadlocks with two devices and two processes are not possible.

Data acquisition

To integrate the data acquisition framework presented in 4.1, we derived a custom UCA-based detector class. The constructor of this class receives the unique identifier string which is used to instantiate a low-level detector object shown previously in listing 4.1. During the instantiation, all properties are enumerated and wrapped in corresponding `Parameter` and `Quantity` objects. For the latter, we map UCA units to control system units. The real low-level UCA access object is hidden and only accessed by interface methods of the detector. In case of the frame grabbing method, we use the Python module described in 4.1.2 to return a NumPy data array that is used for image analysis and calibration purposes.

In certain cases, we have to specialize the general UCA detector class. For example, the UFO detector provides temporal frame compression that is controlled by a set of parameters [21]. Determining the correct parameter values is non-trivial, hence we provide additional functionality in the detector class to set the correct parameters after a calibration run. For this we use the fact that the detector can control itself. Since we use the UCA interface we immediately profit from low local latencies and high throughput from a remote detector.

Listing 4.6 A tomographic scan that changes the rotation axis and acquires frames at each set position.

```
def acquire():
    return detector.grab()

scan = Scanner(stage['angle'], acquire)
future = scan.run()
positions, frames = future.result()
```

4.2.4 Process Control

The basic device and parameter abstractions can be used to control devices manually. Although these mechanisms could be used to write simple scripts to perform certain experiment tasks, the tasks often provide similar functionality and often differ only in some parameters. Hence, we devices a high-level process control module containing abstract process skeletons derived from the decomposition of recurrent logic from hardware-specific operations.

A common procedure is a scan of a device parameter along a range of values and the evaluation of a measure at each scan point. For example, while moving a motor the pressure of a pump is measured or while changing the exposure time of a detector, the image response is evaluated. Using the `Parameter` objects of a device, the `Scanner` object from the process module can provide this in an abstract way. By passing `Parameter` objects instead of `Devices`, we can model every type of scan, for example a tomographic scan as shown in listing 4.6. Although scanning allows processing the measured data, it is not suitable for *feedback-based control* because the feedback loop itself is missing.

Feedback-based control is necessary for beamline tasks that need to evaluate a measure and react on the outcome of the result. In tomographic environments, the control algorithms often require an image-based feedback, for example when focusing objective lenses or aligning samples. This logic can be decoupled into image-based metrics, control algorithms and feedback mechanisms. We can find different metrics for diverse problems, for example gradient- or variance-based metrics to determine the sharpness during a focusing process. Data assessment based on such a metric is used by a control algorithm which manipulates parameters to optimize the measured metric. The computed parameters are then set on the hardware by a high-level device API, which closes the feedback loop. For example, a simple focusing process is just the optimization of a `Parameter` object with respect to a measured value such as the sharpness of the current detector frame. A sample procedure is shown in listing 4.7.

Because of the asynchronous approach, we are able to use the feedback loops in a *continuous* mode. This prevents waiting for the process steps to finish and instead allows us to sequence device accesses in parallel.

Listing 4.7 Finding the optimal focus position by maximizing the standard deviation of the current detector frame.

```
def measure():
    return np.std(detector.grab())

def on_finish()
    detector.stop_recording()

maximizer = Maximizer(motor['position'], measure, bfgs)
detector.start_recording()
f = maximizer.run()
f.add_done_callback(on_finish)
```

4.2.5 Data flow descriptions

As shown in the previous chapter, a data flow oriented processing model is necessary to describe and schedule streams on heterogeneous system architectures. So far, we used task graphs composed of connected nodes to specify the data flow. This model is sufficient for programmatic access but becomes burdensome for experiment workflows. As stated in the first requirement, we need high-level abstractions to integrate data processing within the experiment. In this section, we present two approaches to describe the work flow in two different ways.

Functional description

Each task t_i processes an arbitrary number of inputs to produce a task-specific output. Thus, we can give a task function $f_{t_i} : \mathbf{I}^n \to \mathbf{I}$, where $n = \deg^-(t_i)$ denotes the arity of the function and $\mathbf{I}$ describes an abstract data type covering possible input and output types. An arc that describes the data flow from one task to the numbered input of another task is thus the composition of the associated functions. Thus for a task t_i, the data flow is defined by the function application $f_{t_i}(f_{x_1}, f_{x_2}, \ldots, f_{x_n})$, where $(x_k, t_i) \in V_T$. This property is symmetric, which is why we can derive the graph structure from function applications. In that case, the task graph is effectively the static call graph of all involved task functions [107].

The lower-level `TaskNode` objects represent nodes in a task graph (see 3.4.1). To abstract from these graph-based tasks, we introduce the higher-level *Concert* `Task` objects which represent a task's function. During the construction of a `Task` object, the name and construction arguments of the corresponding `TaskNode` are associated with the `Task`. Moreover, all `Task` objects hold a reference to a global `Environment` and a common, initially empty `TaskGraph` object.

To derive the task graph structure from the call chain, we override the `__call__` method of the `Task` object. This method is called whenever an object is accessed with

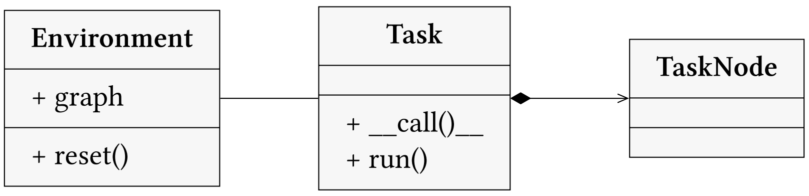


Figure 4.10 Relationship between high-level `Task` and low-level `TaskNode`. The `Task` is created with an implicit `Environment` object which it uses to connect itself to its `__call__` arguments.

Listing 4.8 Set up and execution of a pipeline implementing the FBP algorithm.

```
write(backproject(ifft(ramp(fft(read()))))).run().join()
```

parenthesis syntax, i.e. `Task.__call__(object, x, y)` is called when `object(x, y)` is used. During such a call, we instantiate a new internal `TaskNode` using the arguments that were saved from the construction of the `Task`. We require that each argument to the `__call__` is either a `Task` object itself, hence we can associate the position of an argument with the input port, or a terminating iterable that is inserted into the task graph. For example, calling `z(x, y)` maps `x` to input port 0 and `y` to input port 1 of `z`. To allow nested composition of function calls connections, the `__call__` method always returns the object itself.

The `Task` object exposes a `run()` method that creates a new `Scheduler` object with which it executes the current state of task graph. Therefore, to start the entire pipeline, the `run()` method must be called on the outermost function which at that point guarantees that the task graph is complete. To allow for parallel control and computation, the `Scheduler`'s main loop is started in a different Python thread. To control the execution the `Task` object exposes a `join()` method that blocks execution until the computation has finished. Listing 4.8 shows how a linear reconstruction pipeline is constructed using nested function calls and how the last returned object is used to start the execution and wait for its completion.

Coroutines and generators

Coroutines offer an elegant approach to describe data flow oriented processing such as pipelines by generalizing the concept of a *subroutine*. A subroutine is defined as a program function that enters at the first instruction of the subroutine body and exits *once* at a `return` statement or at the end of the body. Thus, by themselves subroutines can not hold any state across multiple invocations. Contrary to subroutines, *coroutines* can exit prematurely by suspending the execution and *yielding* to another coroutine. By subsequent yields of other coroutines, they can *resume* execution where the coroutine left before [90]. Hence, unlike subroutines, coroutines are able to persist the execution

```
def numbers():
    i = 0
    while True:
        yield i
        i += 1

def show(producer):
    for number in producer():
        print(number)

show(numbers())
```

(a) `numbers` *produces* items

```
def numbers(consumer):
    i = 0
    while True:
        consumer.send(i)
        i += 1

def show():
    while True:
        number = yield
        print yield

numbers(show())
```

(b) `printer` *consumes* items

Figure 4.11 Yielding on the left side of a generator suspends the execution to the caller, whereas yielding on the right side suspends until a new value is pushed into the generator.

state across multiple invocations. The difference between passing control to another coroutine and merely calling it, lies in the symmetry and cooperative relationship between two coroutines. *Generators* are asymmetric or semi coroutines which return a value and suspend to the caller without explicitly specifying a target coroutine [90]. Because items are produced on-demand and piece by piece, coroutines are the ideal approach to build light-weight *iterators*.

Per default, any occurrence of the `yield` statement in a Python function declares it automatically as a generator of data items and forbids further `return` statements. Figure 4.11a shows an example of a generator that yields a monotonically increasing number sequence as well as an iterator that prints the sequence *ad infinitum*. In this case, the coroutine itself does not know who will receive the yielded numbers, hence it is a typical asymmetrical coroutine. Values can also be sent *into* a generator by using the `yield` statement on the right side of an expression. Figure 4.11b illustrates how the roles of producer and consumer from the previous example are swapped: The `numbers` function is an ordinary function that produces a stream of items and sends each item to some consumer. The consumer suspends execution as long as no item is sent into it. As seen in the examples, both generator coroutine types can describe linear data processing pipelines. For producing generators, the data flows from the inside producers to the outer consumers whereas consumer generators push data into the arguments. Because producers can only yield data to a single caller, we cannot describe complex data flows such as branching and diversion. A consuming coroutine, however, allows arbitrary arguments and thus enables us to explicitly specify the flow from sources to destinations. Tasks which are already present as regular producing generators can be converted to the

```
def inject(producer, consumer):
    for item in producer:
        consumer.send(item)
```

(a) Injecting items from a generator

```
def broadcast(consumers):
    while True:
        item = yield
        for c in consumers:
            c.send(item)
```

(b) Forwarding items to multiple consumers

Figure 4.12 Auxiliary coroutines to bridge producing and consuming coroutines shown on the left and forward a single item to a list of consumers on the right.

Listing 4.9 Splitting the reconstructed data stream for writing and viewing at the same time using the broadcasting mechanism from figure 4.12b.

```
acquire(sinograms(backproject(broadcast(write(), view()))))
```

send-style coroutines by an intermediate injection coroutine as shown in figure 4.12a.

Due to the flexible arrangement of coroutines we use them to map high-speed data processing graph execution from chapter 3 to the control workflows. A data processing task graph is wrapped in a *Concert* specific `InjectProcess` object. This object keeps a reference to a `TaskGraph` that was defined manually or constructed through the `Task` mechanism described in the previous section. If the `TaskGraph` has an input task with $\deg^-$ larger than zero we insert an appropriate dummy source task as described in 3.3.3. To insert data into the stream, the user calls `inject` method with the desired input data. If $\deg^+$ of `TaskGraph`'s source task is larger than zero, we append an output task to read the result from the data processing. In order to receive a result data item, the user must call the `result` method explicitly.

To integrate the `InjectProcess` into the coroutine-based workflow system, we used the `inject` and `result` primitives to implement the iterator protocol. For this, we override the `__call__` method and yield to receive an item, process the data item by inserting it into the `TaskGraph` and forwarding the `result` to the consumers. Because the task graph is synchronized implicitly by the data queues, the execution is suspended for the time the `Task` yields to the caller.

Integrating data processing as a coroutine allows us to re-arrange data processing and control structures in a flexible way. For example, using the broadcasting coroutine shown in figure 4.12b, we can display the slices that were reconstructed on the heterogeneous system during the reconstruction process and at the same time write them to disk using the coroutine pipeline shown in listing 4.9. Although, the coroutine-based approach allows for flexible data flow arrangements, they have to be written for the same applications over and over again. Therefore, we wrap common coroutine pipelines in classes and function. Additionally, classes include the necessary logic to *restart* the wrapped coroutine without flattening the generator results into a list. This is necessary because Python generators produce data only once until exhaustion.

4.2.6 Related work

Generic beamline control

SPEC is a commercial X-ray diffraction control system extended for X-ray imaging applications and TANGO interfacing and comes with a custom scripting language. MXCUBE is European Synchrotron Radiation Facility (ESRF)'s TANGO-based control system for X-ray crystallography [45]. Another recent development is Sardana,[7] a distributed, Python- and TANGO-based general purpose experiment control system. Although all control system front ends are capable of flexible device control, we argue that these systems are either too rigid or too slow due to the emphasis on distributed acquisition, control and processing and thus cannot withstand the continuous data stream required for fast and feedback-driven experiment control.

High-throughput tomographic reconstruction

At the BM-2 beamline of the Advanced Photon Source (APS), Wang et al. measured acquisition times of about an hour for 720 projections using a CCD detector with a resolution of 1024×1024 and a reconstruction time of about 17 minutes on a SGI Origin 2000 using 80 out of 96 compute nodes at the 2-BM beamline at APS [133]. In a follow-up article, De Carlo and Tieman replaced the SGI machine with a 32-node compute cluster made of off-the-shelf hardware and reduced the acquisition and reconstruction of 720 projections at 1024×1024 down to about five minutes [31]. At the BL47XU and BL20XU beamlines of the SPring-8 synchrotron, Uesugi et al. use a Hamamatsu CCD detector with a resolution of 4000×2624. With a 2×2 binning readout it takes 20 minutes to acquire 1800 projections at 2000×1311 pixels in a continuous mode [125]. Using multi-threading, it takes about ten seconds to reconstruct a single slice of 2000×2000 pixels from the 1800 projections on a Core2-Duo machine. A fully automated software-controlled experiment setup for high-throughput sample scanning is used at the TOMCAT beamline at the Swiss Light Source [83]. Based on EPICS and Python, it automatically exchanges samples, computes the sample position and angular offset, corrects the alignment and acquires the data.

4.3 Summary

In this chapter, we proposed the designs of components required for feedback-driven experiments. We proposed a low-latency and high-throughput data acquisition system

[7] `http://sardana-controls.org`

architecture that provides a minimal, generic interface to control detectors and read out image data. Using optimized zero-copy memory transfers, we are able to reduce overheads by the system itself. We also proposed a novel remote data acquisition scheme based on InfiniBand that decouples the detector from the control machine. On the one hand, this scheme reduces load on the low-level control system and on the other hand allows us to stream the data from the fastest available detectors.

Our proposed control system concept uses the data acquisition interface to give quick access to the raw X-ray images but also to forward the incoming data stream to the heterogeneous compute system presented in chapter 3. Wrapping device accesses in future objects enables hardware parallelism that improves the experiment throughput. To improve the accessibility for end-users, the control system maps data stream processing to coroutines. These help to define arbitrary workflows and easily exchange built-in or external data processing components at run-time. The system provides high-level feedback processes that use the processed imaging data to calibrate, align and adapt the experiment setup on the fly.

This approach provides a flexibility that allows for rapid prototyping of experiments which is crucial in X-ray tomography setups that must adapt to the requirements of the sample environment.

5 GPU programming

In the previous chapters, we investigated the basic foundation for streamed data processing on arbitrary heterogeneous compute systems and how we can incorporate those into an experiment control system. The heterogeneity of the compute system allows us to use arbitrary compute resources, however, in order to maximize the throughput of these resources we need optimized low-level accelerator code. Writing this code is error-prone and burdensome. To reduce the burden on the beamline operator and increase his productivity, we will present a source-to-source translation system that aids writing high-performance GPU code without having knowledge about the underlying hardware. In the second part, we will investigate architecture-dependent optimizations of processing stages found in high-throughput reconstruction pipeline.

5.1 Just-in-time GPU code generation

The compute system that we presented in 3 reduces the amount of work that is necessary to set up and use a heterogeneous multi GPU compute system. To achieve optimal performance requires that a GPU implementation of the employed tasks exists. Because the run-time does not write the GPU code, a developer still has to translate existing CPU code to GPU kernels by hand. This is a tedious task that requires a considerable amount of low-level hardware knowledge and source code that is written in compiled error-prone languages such as C.

Contrary to a compiled language, a DSL can hide the low-level hardware details using a concise, abstract syntax that is designed for a specific application domain [128]. A DSL compiler transforms this language either into an intermediate language or directly to the target execution platform. The declarative properties of DSLS increase the productivity of programmers, reduce failure rates and may increase the performance potential by imposing restrictions on the programming constructs. Well-known real-world examples include the Make dependency system, the TEX document typesetting system or Structured Query Languages (SQLS) for uniform data base access. However, the time required to develop and maintain a new DSL as well as the additional effort that is needed to integrate such a language into the host program or framework can be disadvantageous [127].

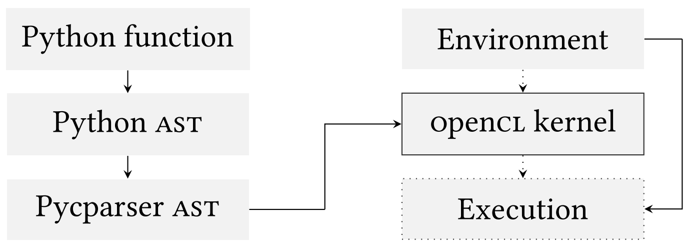


Figure 5.1 Outline of the translation process from Python to OpenCL code

As an alternative to real DSLS, DSL-like languages can be devised in dynamic, general purpose *host* languages such as Python, Ruby or the Lisp family. Expressive naming and domain-specific semantics allow such languages to feel like a DSL and provide expressiveness to implement systems such as Rake, a Make-like language written in Ruby. Because only features available in the host language can be used, the expressiveness is still limited to that of the host language.

In case of GPU computing, the target language is not the host language itself but a lower-level language such as OpenCL or CUDA. This means that we have to translate a piece of code in a source language such as a DSL or a dynamic language to the lower-level kernel. Because Python has a comprehensive AST representation of *itself* and integrates well with external C libraries, such as our heterogeneous compute framework, we will use it for source-to-source translation of Python *functions* to OpenCL *kernels*. In the remaining section, we will present the translation architecture, optimization methods and a run-time system that complements our heterogeneous architecture for standalone computation.

5.1.1 System architecture

Translating Python code to an OpenCL kernel is the main requirement of our source-to-source translation system which is made possible for two reasons. First, the OpenCL standard requires that kernel code can be compiled at run-time[1] and not restricted to the time the host program is built [121, p.138], thus avoiding any complicated just in time (JIT) compilation techniques. Second, Python is capable to modify code before this code is seen by the Python lexer. This allows us to parse source code at run-time, generate corresponding low-level code and provide wrapper code that calls this code instead of the wrapped code.

The three blocks on the left side of figure 5.1 outline the source-to-source translation. In the first phase, the textual source code of a live function is extracted using the `getsource` function of the standard `inspect` module. The AST representation of the function is derived by parsing the code with the standard Python `ast` module. For the code gener-

[1] Implementations are free to cache intermediate binary results as long as the kernel source does not change.

Listing 5.1 Decoration of an addition function with the `@jit` decorator.

```
@jit
def add(x, y):
    return x + y
```

ation, we translate the Python AST into the C AST structure of pycparser [8]. In this step, an emitter visitor walks the Python AST and emits corresponding C AST nodes. Python-specific syntax is translated into corresponding valid C expressions (for example the Python expression `x**2` is translated into a C call to `pow(x, 2)`) and type information associated with variables. In 5.1.1, we will go into detail how certain Python constructs are translated. In case of dynamic translation, we incorporate run-time information such as input data types and the current hardware environment into the translation process. Eventually, the system uses a modified pycparser generator function to emit valid C99 OpenCL code from the C AST.

User annotations

From the user perspective, a function is translated by annotating it with a *decorator* as shown in listing 5.1. A decorator is a function that receives a function as its argument and returns a new function that is assigned to the name of the original function. In our system, the decorator function starts the translation process on the wrapped function and returns a kernel string or a kernel object that can be called. The string is returned with the `@static` decorator that is called on import time. Because of this, it cannot incorporate information about the environment or arguments in the translation process. Calling a function decorated with `@static` returns an OpenCL kernel string that can be used directly within other compute systems.

The `@jit` decorator, as the name implies, delays the transformation of the passed function into an AST until the decorated function is called. During the call, the supplied input data as well as the hardware environment is analyzed and suitable kernel code generated. The run-time system then executes the kernel on the available hardware instead of calling the original function. Compiling for the target device at run-time has two advantages compared to the usual case of fixing the kernel code at run-time. First, the translation system can generate code that is best suited for the target architecture present at run-time. This is necessary because OpenCL provides source but not performance portability [37]. For example, OpenCL CPU code performs best when using vector types to exploit SIMD instructions such as SSE. Applying this policy to GPUs negatively impacts the performance and a scalar approach generally performs better. Second, we can analyse the input data and improve memory access times using optimal memory allocations.

Listing 5.2 Parameterized decorator for one scalar and two vector arguments.

```
@jit(float, np.double, np.double)
def multiply_accumulate(a, x, y):
    return a * x + y
```

Semantic translation

To avoid breaking assumptions about the semantics of the host language, the syntactic elements must be mapped closely to the target language. In case of Python, the execution model is a sequential virtual CPU, whereas OpenCL, as we have seen before, employs a massively parallel SPMD machine (see 2.3.2). Because these execution models do not match, we employ a middle ground inspired by NumPy. Similarly to MATLAB, NumPy auto-vectorizes scalar operations if the arguments are array or matrix types [95]. That means if a and b are array types, the c = a + b will compute the point-wise addition in a fast C loop. We assume that Python programmers are familiar with this semantic difference to the original Python operators.

Python is a dynamically typed language, which means that a programmer does not need to explicitly annotate variables with type information but instead is inferred during assignment and usage. On the other hand, OpenCL kernels – just like any other compiled language – require type information that must be known at compile time. Moreover, array arguments of OpenCL kernels require an address space qualifier that denote their location and implicitly specify their run-time behaviour. Moreover, the qualifiers may optionally restrict access to the contained data. The AST translation system uses a "best effort" approach which assumes that data is always stored as floating point arrays. In case this assumption is not valid, types and qualifiers can be specified by *parameterizing* the decorators. Each decorator argument is a type that corresponds to the argument of the decorated function. Listing 5.2 shows type annotation of three arguments, two of which are vector types and one which is scalar.

```
@static(int, np.int32)
def scale(s, x):
    return s * x
```

(a) Scalar multiplication

```
kernel void
scale(int s, global int *x, global int *y)
{
    int idx = get_global_id(0);
    y[idx] = s * x[idx];
}
```

(b) Resulting kernel code

Figure 5.2 Translation of a simple scalar multiplication expression to the corresponding kernel.

Return statements

Contrary to functions, kernels behave like procedures and cannot return data due to the missing call stack, thus the `return` statement cannot be used inside an OpenCL kernel. Instead, all outgoing data must be passed through an explicit output buffer that is transferred back to the host.

When translating Python functions, an additional `out` parameter is inserted AST subtree representing the argument list of the function's signature. Although it is not visible in the signature of the Python function itself, users of the `@static` decorator must provide a suitable OpenCL buffer and call the translated kernel accordingly. In case of a `@jit` decorated function, this parameter is handled transparently by the run-time system. During the call the run-time allocates a suitably sized output buffer and passes it to the kernel function. After completion the run-time transfers the buffer from the device back to the host and returns that as a NumPy array to the caller.

Array indexing

We motivated the use of implied vector accesses by the familiar syntax and concise expressiveness. In the standard case, this means that accessing an array translates to an access of a single element of that array by exactly one work item. Figure 5.2 illustrates the access translation for the function in figure 5.2a that scales an input array x by a scalar s to the kernel in figure 5.2b. This syntax is sufficient to implement point-based image processing functionality but does not allow threads to access neighbouring elements which is crucial for spatial filters.

Python's default array indexing semantic uses the index as an offset from the zeroth element. This means, for example, that $x[1]$ refers to the second element and $x[-1]$ to the last element of x. To simplify writing filter code, we allow the user to specify *relative* indices by prepending $+$ or $-$ to the constant index. Thus on a one-dimensional array, accessing $x[-1]$ refers to the left neighbour of the current element and $x[+1]$ to the right neighbour. Generally, this transformation is applied to all dimensions, therefore

$x[-1,+1]$ refers to the top-left element. One will notice that negative indexing does not yield the same result as the default Python behaviour which is a trade-off we had to make.

Out-of-bounds accesses, such as the first work item accessing its left neighbour element, must be handled correctly in order to preserve predictable results. By default, the translator zeroes any access that cross borders, i.e. $x[i] = 0 \; \forall i < 0$. By passing parameters to the decorator, the behaviour can be customized for the required application. For example, the border element may be extended ($x[-i] = x[0]$) or mirrored ($x[i] = x[-i]$).

Loop transformation

In Python a *for* loop always iterates over an *iterable* object. An object is iterable if it has a `__iter__` method that returns an iterator. Iterators are objects that return an item when their `next` methods are called. In principle, all data structures that behave like collections (lists, sets and so on) implement the iterator protocol. The `range(start, stop, step)` iterator is a built-in iterator that returns a step-wise sequence of numbers between start and stop. Rather than constructing manual *for* loops, idiomatic Python iterates over an iterator object. The translator detects these situations and emits C *for* loops as follows: For each `range` iterator, a corresponding *for* loop is generated. In case of iterations over objects, the translator tries to identify the dimensions of the object and construct a suitable loop. This requires that the object itself has a length, iterating using arbitrary iterators such as generators is not possible.

5.1.2 Run-time optimization

As mentioned before, functions decorated with the `@jit` decorator are translated prior to execution at run-time. The run-time system itself is transparent to the user and at the time of this writing uses PyOpenCL by as backend for interacting with the OpenCL platform [71]. Both, the current environment as well as the provided data is taken into account when generating the corresponding kernels. For each input and output array, one OpenCL buffer is created and cached using the dimension and type of the array as its hash key. This is necessary optimization to avoid excessive re-allocations of OpenCL buffer resources that happens when a decorated function is called in a tight loop. Besides this basic optimization, we added optimization strategies that work on the AST level.

Constant memory

By default, any input array is classified as both readable and writeable global memory. For data that is read only, it may be beneficial to place it in the constant memory address space to improve access latencies. There are two restrictions to the placement of kernel arguments into constant memory: the constant memory address space is limited – it can be as small as 64 kB – and kernels may only support as low as eight constant kernel arguments [121, p.43].

Our constant memory optimization strategy consists of two steps: First, the translation system determines suitable candidate arrays and second it allocates them to the available constant memory. In the first pass, we mark any array as constant that never occur on the left side of an assignment. Unless excluded by the `restrict` keyword pointer aliasing[2] of an array can hide the fact that a memory location was modified. Thus, we remove any candidate array that aliases with a variable that is modified in the body of the function. In the second pass, the system determines the maximum number bytes n_{const} available in the constant memory of the *smallest* device to prevent over-allocations. The translation system also determines the maximum number constant arguments n_{args} that can be submitted to a kernel at once. When calling the Python function, we determine the number of bytes n_i used by each of the N arguments. With this information we would need to solve a linear program for $\vec{x} \in \{0, 1\}^N$, for which

$$\begin{pmatrix} n_1 & n_2 & \cdots & n_N \\ 1 & 1 & \cdots & 1 \end{pmatrix} \cdot x \leq \begin{pmatrix} n_{\text{const}} \\ n_{\text{args}} \end{pmatrix} \tag{5.1}$$

maximizes $\sum_0^N x_i$. For each x_i that equals to 1, the corresponding parameter is assigned the `constant` qualifier. Solving the binary integer problem is one of Karp's original 21 NP-complete problems [67]. Hence we cannot find an optimal solution in polynomial time. To find a solution that is good enough in most situations, we use the greedy pack algorithm 5.1. This works well, except for pathological cases. Alternatively, one could use a dynamic programming approach to yield an optimal solution in pseudo-polynomial time, however, as we have noted the simple algorithm suffices for all our problems.

Algorithm 5.1: Greedy constant memory allocation scheme.

```
size $\leftarrow 0$, args $\leftarrow 0$, $i \leftarrow 0$
$s \leftarrow$ parameters sorted by size, smallest first
while $size < n_{\text{const}} \wedge args < n_{\text{args}}$ do
    apply constant qualifier to $s[i]$
    size $\leftarrow$ size $+n_i$
    args $\leftarrow$ args $+1$
```

[2] Pointers referring to the same memory location.

Table 5.1 Overview of expression substitution by equivalent OpenCL builtins.

Expression	Substitution
$\sin/\cos/\tan(\pi x)$	$\mathrm{sinpi}/\mathrm{cospi}/\mathrm{tanpi}(x)$
$\mathrm{asin}/\mathrm{acos}/\mathrm{atan}(x) \cdot \pi^{-1}$	$\mathrm{asinpi}/\mathrm{acospi}/\mathrm{atanpi}(x)$
$a \times b + c$	$\mathrm{mad}(a, b, c)$
$a \times b - c$	$\mathrm{mad}(a, b, -c)$

Substitution of mathematical expressions

OpenCL provides functions which are faster to compute at the expense of accuracy as well as functions which combine multiple operations in a single operation. The arithmetical structure is directly encoded in the AST, hence we can substitute known expressions by an equivalent function call. One type of substitution replaces function calls like $\cos(x\pi)$, where x can be an arbitrary expression, by the equivalent OpenCL built in function $\mathrm{cospi}(x)$. This is an optimization step that would have to be implemented by the user because the OpenCL standard lacks an abstraction for the π constant. Another common mathematical expression often used in the field of digital signal processing is the multiply-accumulate operation $a \cdot b + c$. OpenCL offers a function $\mathrm{mad}(a, b, c)$ that provides an approximation of that expression in a single clock cycle. An overview of all substitutions is given in table 5.1.

The optimization pass walks the AST in breadth-first search manner and tries to match the target expressions shown in the table. If it finds a match, it stops walking that branch and replaces the expression node by a new node with the corresponding to the substitution, i.e. a call to the function in question.

5.1.3 Multi GPU execution

Typical image filters and point-based algorithms are suitable for execution on multiple GPUs because no or only a small amount of communication is necessary between individual work items. Thus, when executing a `@jit` decorated function we leverage the performance of multiple available devices by splitting the input data into smaller items and scatter the small items among the GPUs.

For multi-dimensional data sets it is necessary to split and partition the data along an axis that reduces irregular data accesses by the kernels. To use only linear data accesses, we use a heuristic that splits the input data along the *shortest* axis. The idea is that for most input data the shortest axis also changes the slowest. For example the shortest axis of a reconstructed volume is in most cases the z-axis of the slice stack, hence we distribute slices in z direction. This ensure that the elements are accessed along the fastest changing x and y axes thus exhibiting better cache behaviour. The new data size is then used to create multiple smaller input and output buffers and to determine the

```
@cu
saxpy_copperhead(a, x, y):
    return [a * xi + yi
            for xi, yi in zip(x, y)]
```

(a) Copperhead

```
@jit
def saxpy(a, x, y):
    return a * x + y
```

(b) Our approach

Figure 5.3 Comparing the implementation of the SAXPY kernel with our approach and with Copperhead.

appropriate global work sizes and offsets for each kernel.

To avoid run-time decisions and branching in tight loops, we generate and compile one kernel with fixed data addressing for each GPU. After compilation, we run the kernels in parallel on the devices and wait for the kernels to finish. In the order of finishing kernels, we transfer the result on the device back to a single host buffer. This allows us to overlap on-going computation with data transfers. The final result array is then returned to the user for further processing. Depending on the OpenCL vendor, frequent resource allocation leads to performance degradation. Thus, for each GPU and decorated function we cache allocated buffers using the argument index and dimension as the hash key.

5.1.4 Related work

Membarth et al. describe a CUDA pseudo-DSL that transforms a C++ description of kernels and execution using Clang/LLVM into CUDA and OpenCL kernels [87]. The execution model does not foresee multiple GPUs.

Like our system, Copperhead is Python-based system that uses decorators to annotate functions for execution on parallel system architectures. Instead of OpenCL Copperhead targets CUDA, OpenMP and Intel's Threading Building Blocks [22] for general purpose high-performance computing, whereas our system has a clear focus on high-throughput image processing. The result is that Copperhead uses a functional style (see the comparison in figure 5.3) with manipulations that must be free of side effects, whereas our system explicitly allows changing data in place. Moreover, Copperhead does not automatically distribute data and computation among multiple GPUs but requires the user to specify where kernels should be launched.

5.2 Kernel statistics

The characteristics of typical GPU compute kernels gives an insight into high-performance compute application patterns and environment requirements. Using static kernel information, we can optimize our heterogeneous compute system for the common

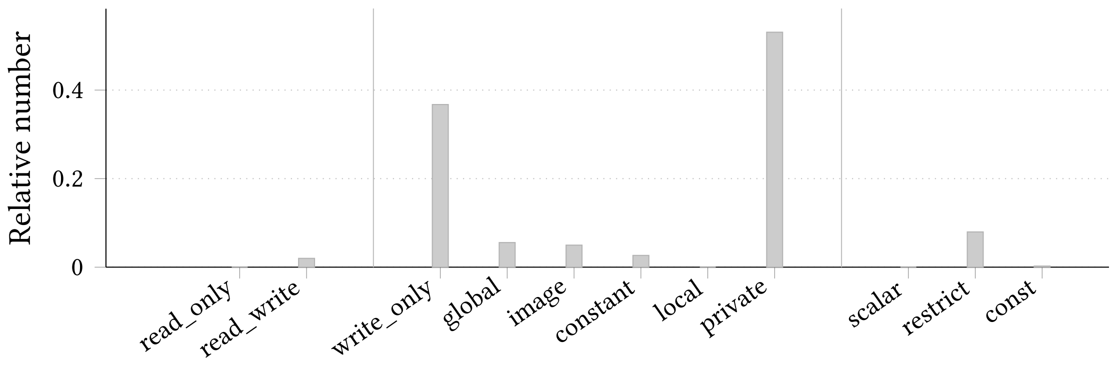


Figure 5.4 Relative number of kernel argument qualifiers grouped into access restrictions, address space and type qualifiers. The image type includes both `image2d_t` and `image3d_t`.

case.

A kernel is characterized by its API that consists of function declaration and type annotations as well as run-time parameters that determine how the kernel is executed. The latter depends on the hardware device for which the kernel is compiled and can be estimated with off-line compilers provided by the GPU vendors. To get a comprehensive overview, we collected OpenCL kernels from our own projects as well as from projects publicly accessible via the collaborative development platform GitHub.[3] In total, we collected 112 kernel files with close to 600 kernels, covering applications from image processing, cryptography and large-scale physics simulations.

For each kernel, we analyzed the function declaration and determined the number of arguments. For each kernel argument, we determined access, address space and C99 type qualifiers of each argument. Access qualifiers specify if an argument can be read or written from the kernel, whereas address space qualifiers restrict the memory locality (see 2.3.3). The C99 type qualifiers denote pointers that do not alias (`restrict`) and may inhibit optimizations by the compiler, that cannot be written (`const`) and that can be changed even without intervention of the program (`volatile`). Using the AMD CodeAnalyzer, we also compiled each kernel to the AMD Tahiti microarchitecture and determined the number of used SGPRS and VGPRS per kernel (see 2.2.2).

As shown in figure 5.4, of all kernel arguments 53.07 % are scalar arguments, which means there is no specific memory buffer associated and the argument is transfered by value to the GPU kernel. 36.73 % of the remaining arguments reference global memory and 10.53 % designate either OpenCL image types or constant memory. Of all pointer typed arguments, 18.46 % are read-only and 10.8 % write-only. Moreover, none of the pointer arguments is limited to non-aliasing pointers by the `restrict` type qualifier. Figure 5.5 shows the relative number of kernels, SGPRS and VGPRS for a given number

[3] `https://github.com`

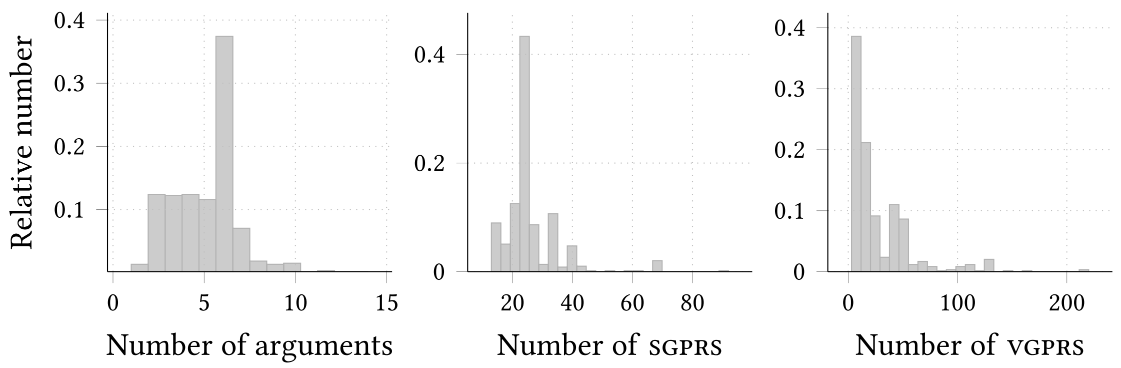


Figure 5.5 Histograms of SGPR, VGPR and number of kernel metrics measured for 592 kernels.

of arguments. As we can see, 37.42 % of all kernels use six kernel arguments, whereas kernels with two to five arguments have a similar share of about 12 %. The distribution among the number of allocated SGPRS and VGPRS indicates a slight tendency towards scalar code.

5.3 Algorithmic optimization

We have presented a heterogeneous compute system and a supervising control system to integrate tomographic reconstruction and other X-ray imaging related processes into experiment workflows. Nonetheless, to enable soft real-time performance we have to optimize individual components and algorithms. For non-iterative image processing tasks computing the result of a single output pixel does not depend on the result of other output pixels. Hence per-pixel operations can be executed straightforwardly in a data-parallel fashion and are particularly suitable for execution on GPUS. However, with the exception of simple point operations such as global thresholding and point-wise image arithmetics, an operation requires multiple input pixels that may also be shared with other output pixels.

5.3.1 Filtered backprojection

The FBP algorithm 2.2, consists of a one-dimensional Fourier transform of the projection data, filtering in the frequency domain, subsequent inverse Fourier transform and the back-projection in the slice space. The performance of the Fourier transforms largely depends on the employed GPU FFT libraries (see 6.3.1), but can be improved by transforming the whole *sinogram* in a batch process instead of single one dimensional projections. Because the image data is always represented as real numbers, it is possible to compute two real FFTS using a single complex FFT [10]. However, this is currently only possible with NVIDIA's *cuFFT* library. The frequency filtering is a single point-wise multiplication with frequency coefficients. The main optimization measures are the pre-computation of the coefficients and subsequent storage in `constant` memory as well as work group size optimization to maximize the DDR memory throughput.

Instead of back-projecting each projection $P_\theta^f(t)$ along the ray path, computing $\hat{f}(x,y)$ for each (x,y) allows for massive parallelism suited for GPUS. Since for each (x,y), $\cos\theta$ and $\sin\theta$ will be computed for all $\theta \in [0,\pi]$, we can reduce the number of trigonometric operations from $O(n_P \cdot N^2)$ to $O(n_P)$ by pre-computing these values. From table 4.1, we know that current detectors have a maximum horizontal resolution of 2560 pixels. Using (2.35), the estimated worst case number of projections is bound by $\lceil \pi/(\arctan 2/2560) \rceil \approx 3220$. As per the standard at least 64 kB of constant memory must be available on all devices [121, p. 43], thus we can easily store the required $3220 \times 2 \times 4$ B in the faster constant memory space.

For each projection we must access element $P_\theta^f(t)$, $t \in \mathbb{R}$. Because P_θ^f is a finite, discrete grid, we have to interpolate the final value using t. Because GPUS are optimized for fast interpolated texture access, we use the OpenCL image type for projections and use the hardware interpolation units of the GPU to do the nearest, linear or bilinear data lookup. The kernel given in algorithm 5.2 computes the pixel value of a slice from a filtered sinogram.

Algorithm 5.2: Backprojection kernel

```
Input: Filtered $M \times N$ sinogram matrix, position $(x,y) \in \mathbb{N}^2$.
Output: Reconstructed pixel value $s$ at $(x,y)$
$s \leftarrow 0$
for $\theta \in [0,\pi]$ do
    $r \leftarrow x\cos(\theta) + y\sin(\theta)$
    $p \leftarrow$ fetch value via GPU interpolation at $S[r,\theta]$
    $s \leftarrow s + p$
```

5.3.2 Noise reduction

Gaussian smoothing on a GPU is a common operation and involves a convolution of the input image with a two-dimensional kernel approximating the impulse response of a Gaussian function. For arbitrarily large kernel sizes, just like the filtering stage of FBP, both input and kernel can be multiplied in the Fourier domain and transformed back. For smaller kernels typically used in noise reduction applications, the smoothing is implemented in the spatial domain. To reduce the number of required operations, the Gaussian separability property is used to convolve the input with two one-dimensional kernels in x- and y-direction instead of convolving with a single two-dimensional kernel. Implementing this within a single GPU kernel is challenging, because work items can only synchronize within a single work group but the data required by boundary work items crosses the work groups. Hence, we use one GPU kernel for each direction. To reduce the number of repeated evaluations of the exponential function, the one-dimensional kernel weight vector is pre-computed and stored in the constant address space.

To reduce the noise with NLM algorithm, we use a GPU implementation that computes equation (2.19) for each denoised output pixel. The main kernel iterates over the search window and computes the weighted pixel value. To reduce the number of global memory accesses, the work items of a single work group pre-load the image data into shared, local memory.

5.3.3 Projection transposition

In a SRµCT scan, each projection at angle θ is a two-dimensional image of size $w \times h$. The Radon transform $P_\theta(t)$, however, is the projection of a single slice along *all* angles θ. Thus, to reconstruct from a representation used by the reconstruction algorithms, we must first transpose all n_p projections of size $w \times h$ to h sinograms of size $w \times n_p$. If all projections fit entirely into system memory,[4] we can use a task as outlined in algorithm 5.3 to produce a stream of sinograms from a stream of projections. As we can see, the algorithm is bound entirely by memory throughput. Instead of copying w projection items from P_i to S in the inner *k for* loop, we can use a single `memcpy` operation. In our tests, these showed a much better copy throughput because of vectorized move instructions. Moreover, we can see that the i and j indices cause exclusive read and write accesses, thus loop iterations are independent of each other. Hence, by executing them in parallel, we can hide memory access latencies caused by cache miss stalls.

A disadvantage of sinogram pre-processing is that a task that implements that behaviour effectively degenerates to a *deferred* reductor. This property has the negative side effect that all subsequent reconstruction stages have to wait until the last projec-

[4] As a rough estimate, just the 3220 projections at 2560×2160 pixels require about 34 GB memory at 16 bit per pixel.

Algorithm 5.3: Transposing a stream of projections into a stream of sinograms

```
Input: $n_P$ projections $P_i$ of size $w \times h$
Output: Linearized sinogram $S$ of size $h \times w \times n_p$
for $i \in [0, n_P - 1]$ do
    index = $i \cdot w$;
    offset = $w \cdot n_P$;
    for each sinogram $j \in [0, h - 1]$ do
        for $k \in [0, w - 1]$ do copy projection row
            $S[\text{index} + j \cdot \text{offset} + k] = P_i[j \cdot w + k]$;
```

tion is processed and the sinogram volume filled completely. Only then can the first sinogram be pushed to the filter stage. By reconstructing directly from the acquired projections, one can avoid this pre-processing latency. However, this means that all projections must be distributed among all reconstruction tasks and that each reconstruction task must keep the entire reconstructed volume in memory. With current GPU resources, this becomes excessively large and prohibitive except for toy examples.

5.4 Summary

In this chapter, we presented a source-to-source translator to provide beamline operators a straightforward interface to high-throughput GPU accelerators. The translator parses ordinary Python functions and emits OpenCL code, which is either inserted into the streamed system described in chapter 3 or used on the fly with an optimizing runtime system. The functions are written in a vectorized stencil-fashion that is compatible with NumPy and allows beamline operators to re-use existing code.

In the second part, we gave an insight on how to optimize the transposition of projections as well as the FBP and NLM algorithms, which are parts of a real-time tomographic reconstruction setup. These optimizations are necessary to keep latencies low and enable feedback-driven experiments with the system introduced in chapter 4.

6 Performance evaluation

In this chapter, we will evaluate the proposed system architectures that we introduced in the preceding chapters. We will conduct throughput and speed up experiments on different heterogeneous compute systems to validate if we were able to meet the given requirements from chapter 1. We will also characterize the systems in terms of bandwidth, throughput and profiling performance to evaluate how the system parameters affect the acquisition, processing and control stages. In the last part, we will investigate the remote data acquisition performance and study simulated control results.

6.1 Measurement method

To compare different compute systems and processing architectures, we use the speed up and bandwidth metrics defined in 2.1.1 and 2.1.2. To determine these metrics, we measure the time to execute entire application processes as well as specific code paths. The UNIX `time` tool returns the wall clock, system clock and user clock time that elapsed during the execution of an application process. We estimate the elapsed process time T_p by the *wall* clock time which in fact represents the duration that the user perceives. The cumulative *user* clock time that a process executes on all cores corresponds to T_1. For precise in-process measurements, we use the `GTimer`[1] facility from the cross-platform GLib library. On our Linux systems, these timers depend on the POSIX `clock_gettime` call which use the best available timer source such as CPU time stamp counters or a dedicated high performance event timer. To measure the run-time of OpenCL kernels, we profile kernel launch commands via the command queue profiling facilities. We will investigate pitfalls of using the profiling mechanism in the following section.

To reduce the measurement noise from external factors such as the Linux scheduler, we execute an experiment for each parameter combination a fixed number of runs n_{runs}. For each run, the time was measured using one of the aforementioned methods and the average run-time T_p/n_{runs} determined. In case of large run-time variances, we also determined the standard deviation to estimate the measurement error.

We conducted all experiments presented in the remaining chapter on different hetero-

[1] `https://git.gnome.org/browse/glib/tree/glib/gtimer.h`

Table 6.1 Single machine and multi-node compute systems used for performance evaluations. The performance numbers in the last column are rough estimates from accumulating CPU and GPU single precision figures of that system.

Name	CPU	n_{cores}	GPUS	GFLOP s^{-1}
desktop	Core i7-950	1 × 4	2× GTX 580	1776
ufosrv	Xeon X5650	2 × 6	6× GTX 580	9735
compute1	Xeon E5540	2 × 4	4× GTX 590 + 1× GTX 680	13204
compute2	Xeon E5-2640	2 × 6	2× GTX TITAN + 2× Tesla	17380
compute3	Xeon X5650	1 × 6	4× R9 290	19655
cluster1	Xeon X5650	4 × 2 × 6	4 × 2× GTX 580	13660
cluster2	Core i5-4670	6 × 1 × 4	6 × 1× GTX TITAN	28305

geneous systems with mixed CPU and GPU configurations, ranging from single compute nodes with multiple GPUS to small-scale InfiniBand clusters (see table 6.1). For the data processing tests we used the performance metrics introduced in section 2.1.1.

6.1.1 Impact of command queue profiling

Multi-GPU scheduling decisions rely on run-time information gathered *during* the execution of a GPU kernel. Although, the overall run-time of a GPU kernel corresponds to the wall clock time measured by the surrounding host code, the asynchronous execution of the GPU kernel prevents accurate timings of the kernel itself. Hence, we cannot infer any conclusions about the execution time, including any overheads incurred by the OpenCL run-time.

To measure exactly when a kernel starts and finishes, we query profiling information from the associated kernel launch command. For this, the command queue on which the kernel was launched must be switched to profiling mode. If enabled, the enqueue functions return an event with four associated time stamps [121, p.192]: *queued* and *submitted* timestamps which denote when the event was placed on the command queue and when it was submitted on the host, as well as *started* and *finished* timestamps which denote the beginning and the end of the command execution. According to the specification, the event time stamp precision is given in nanoseconds, whereas the measured accuracy depends on the specific hardware platform. On our systems, the accuracy ranges from 1 ns resolution for AMD Radeon R9-290 GPUS and Intel Xeon CPUS up to 1 µs for NVIDIA GPUS and Intel Xeon Phi.[2]

We hypothesize that system profiling during the execution affects the overall performance caused by additional overheads within the OpenCL run-time system. Significant

[2] The accuracy can be determined by calling `clGetDeviceInfo()` with the `CL_DEVICE_PROFILING_TIMER_RESOLUTION` parameter on each device.

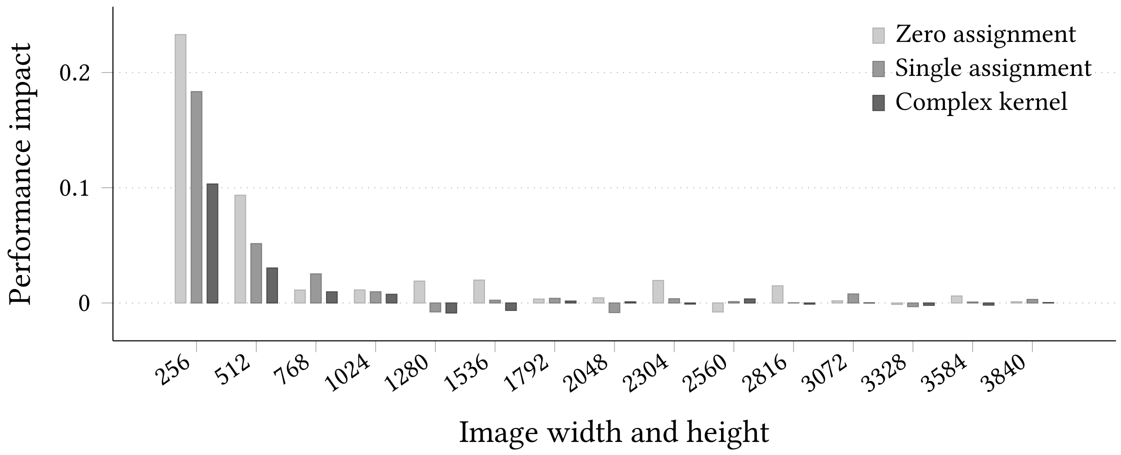


Figure 6.1 Relative performance impact of execution with command queue profiling enabled for two simple and one complex backprojection kernel.

influences on the performance would prohibit any measurements of run-time performance data. To measure this overhead, we executed three kernels, each simulating a different application pattern. Two kernels performed basic arithmetic operations thus exhibiting low computational intensity. The third kernel computed the tomographic backprojection step which is an order of magnitude more compute intense. To estimate the profiling overhead, we measured the *host* wall clock time that elapsed between launching the kernel and the `clFinish()` call that ensures that the command has finished on the specified queue. We measured time T_0 that denotes the elapsed time with command queue profiling disabled and T_1 with profiling enabled.

Figure 6.1 shows the relative performance impact $I = 1 - T_0/T_1$ for data sizes between 256×256 and 3840×3840 pixels. The higher the value, the more time is lost spent on profiling. For the smallest data sets, the performance impact for all three kernels ranges from 11 % to 24 % for 256×256 pixels and 4 % to 10 % for 512×512 pixels. For larger input data, the overhead varies between −2 % to 2 % and lies within the noise floor of the measurement itself. Although we have significant performance impacts for data sizes less than 512×512 pixels, these effects are negligible because the projections in typical SRμCT experiments are an order of magnitude larger. Hence, we can profile applications the entire time and monitor the system for re-scheduling decisions.

6.1.2 Concurrent kernel execution with OpenCL

According to 2.3.4, the OpenCL execution model foresees three levels of parallelism: Coarse-grained parallelism using multiple accelerators, fine-grained SIMD parallelism on the kernel level and concurrent kernel execution somewhere in-between. While

OpenCL implementations are free to support concurrent execution on different CUs of the same device, the standard itself does not define how multiple kernels can be started concurrently. The OpenCL standard merely guarantees that two subsequent calls to `clEnqueueNDRangeKernel` on a single in-order command queue cause the first kernel to finish before the second kernel has started [121, 5.11, p.191]. On the other hand, the OpenCL standard does not specify the execution behaviour of *multiple* command queues per device. Depending on the implementation of the vendor, accessing the same device using different queues may allow concurrent kernel execution.

To determine if concurrent execution is possible, we measured the efficiency of the speed up for work group sizes between 2 and 128. We let $2 \leq n_k \leq 7$ identical kernels execute a piece of code that updates its own global memory in a tight loop. To avoid overlapping kernel execution with kernel launches, we enqueue all n_k kernels without starting them immediately. A user event is passed as a prerequisite to each kernel and set to completed as soon as all kernels are properly set up. We used three enqueue approaches to determine under which circumstances a device executes kernels concurrently. The reference approach uses a single in-order queue and should not be any faster. The second approach uses a single out-of-order queue, which means that the run-time can decide how to schedule the kernels. The third approach creates one queue for each enqueued kernel. Via command queue profiling, we determined the start and finish timestamps s_i and e_i for each kernel. Using the earliest timestamp $s_m = \min_{2 \leq i \leq 7} s_i$ we computed the normalized start and end timestamps $s'_i = s_i - s_m$ and $e'_i = e_i - s_m$. The run-time for kernel i is given by $\Delta_i = e'_i - s'_i$. The latest end time stamp $T_{n_k} = \max_{1 \leq i \leq n_k} e_i$ must necessarily denote the total run-time of all n_k kernels. If the sum of the per-kernel run-times is smaller than T_{n_k}, the kernels must have executed in parallel. In this case, the parallel speed is given by $S(n_k) = T_{n_k} / \sum_i \Delta_i$. If the sum is larger than T_{n_k}, the overhead of scheduling multiple kernels exceeds the time required for execution.

Figure 6.2 shows the *average* speed up using all three queueing approaches for work sizes between 2 and 1024 and $n_{\text{runs}} = 1000$. Except for the AMD Radeon R290 cards, in-order queues do not affect the run-time when scheduling multiple kernels. Although all devices listed in table 6.1 reported support for out-of-order queues,[3] we can see that none of the devices shows a noticeable speed up $S(n_k) > 1$ using out-of-order queues. A significant speed up is measured using multiple command queues on GTX TITAN and the AMD CPU implementation on the Xeon X5650 although with large variances. The speed up on the NVIDIA architecture can be explained by the TITAN's grid management unit that provides dynamic parallelism. Although one may assume that $S(n_k) \approx 1$ for devices that do not support concurrent kernel execution, it is in fact considerably lower on AMD Radeon R290 cards. This behaviour is caused by a high ratio of dead time between two kernels and the total execution time $\sum_{i=2}^{6} e'_{i+1} - s'_i / \sum_i \Delta_i$. On this par-

[3] This was verified by checking the `CL_DEVICE_QUEUE_PROPERTIES` property with `clGetDeviceInfo`.

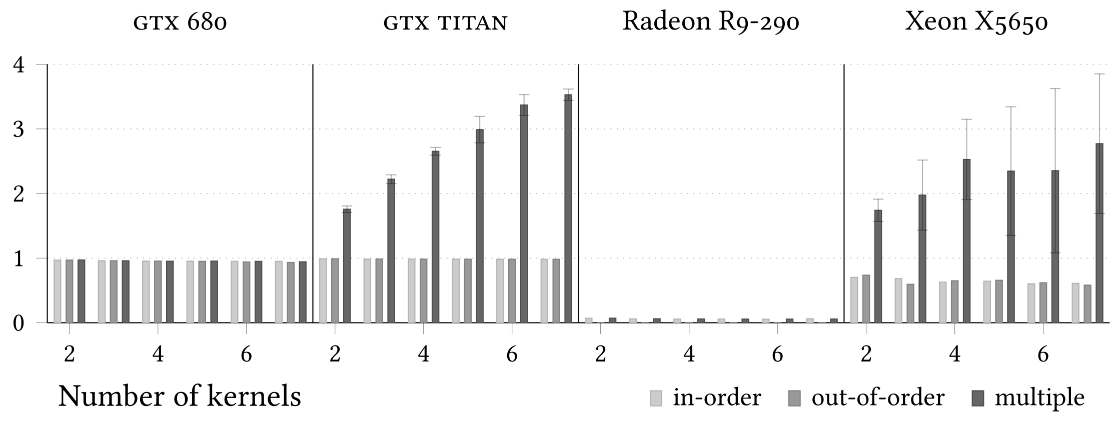


Figure 6.2 Speed up measured for two to seven kernels and three different queueing approaches. The speed up is taken as an average over all work group sizes, error bars show the deviation.

ticular architecture, the ratio ranges between 2.6 and 14.1. The overhead of scheduling is also the reason why the speed up is not exactly 1 for the in-order queueing strategy. The exact reasons for this ratio on AMD platforms is not known up to now but it is likely caused by an insufficient implementation of either the OpenCL run-time or the GPU driver.

Due to the extreme variance in performance gain – and loss in some cases –, the performance of concurrent kernel execution is not portable across architectures. For this reason, higher run-time management requirements and due to the fact that the majority of our kernels saturate the GPU, we conclude that concurrent kernel execution is not a viable performance optimization step that must be considered at the time of this writing.

6.2 Performance figures

Using the methodology described in the previous section, we will characterize the heterogeneous compute systems according to their memory and communication bandwidth in this section.

6.2.1 Single GPU memory bandwidth

As shown in 2.2.2, the data transfer between CPUs and GPUs has a significant impact on the overall performance depending on the application pattern. The main reason for this is the asymmetrical memory bandwidth of PCIe 2.0 and 3.0 transfers between CPU and GPU (8 GB s^{-1} to 16 GB s^{-1}) and on-board GPU memory accesses (up to 320 GB s^{-1} as per table 2.2). In a streamed data processing system with short pipelines or pipelines with

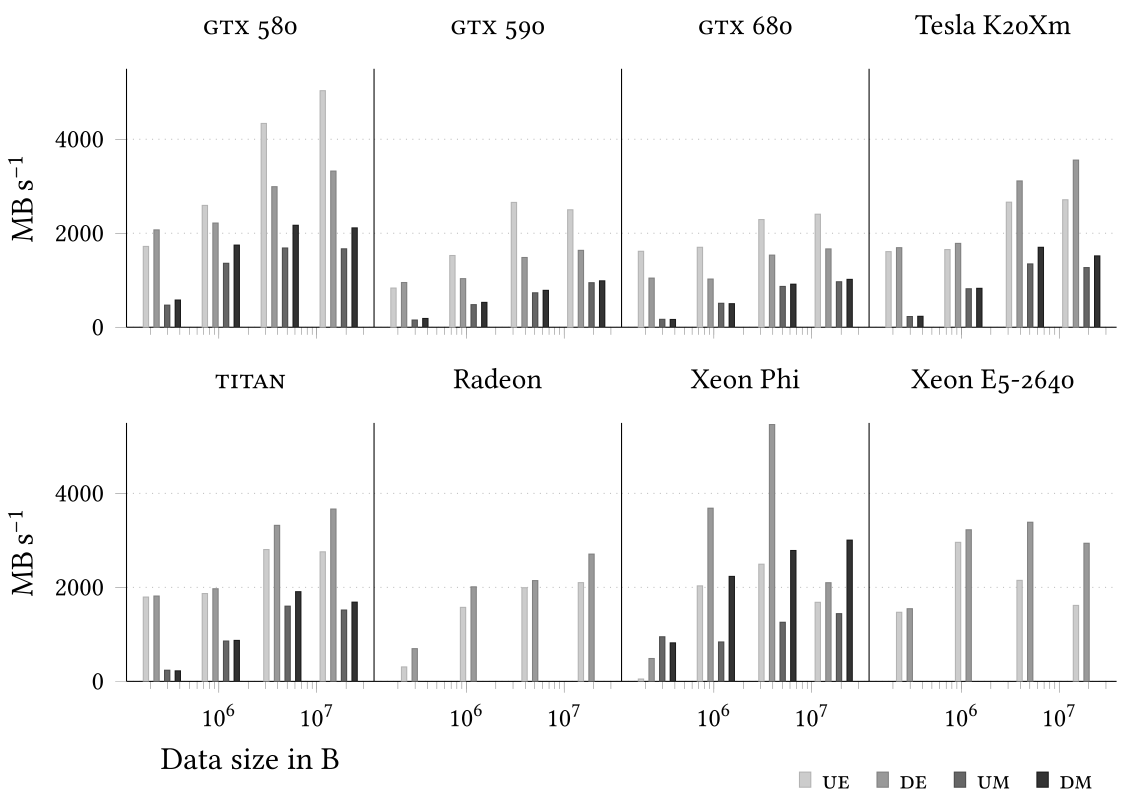


Figure 6.3 Memory bandwidth of up (U) and download (D) using explicit (E) and mapped (M) calls measured on different hardware architectures for a data sizes between 256 kB and 16 MB.

diverse heterogeneity, this bandwidth asymmetry becomes a bottleneck for the overall throughput.

In order to optimize the data flow, we quantify the edge weights w_a between CPU and GPU nodes. The OpenCL programming model provides two mechanisms to transfer data between host and device (see 2.3.3). To initiate explicit data transfers between host and device, the `clEnqueueWriteBuffer` and `clEnqueueReadBuffer` API calls are used. To initiate implicit data transfers, the pages of a memory buffer are blended into the user's address space using `clEnqueueMapBuffer` and read and written. After usage, the pages must be unmapped via `clEnqueueUnmapBuffer`. To determine the bandwidth of the link between the CPU and a single GPU, we measured the up- and downlink memory bandwidth using both methods for data sizes between 256 kB and 128 MB on *compute1*, *compute2* and *compute3* as well as on the Xeon Phi system.

Figure 6.3 shows the measured up- (U) and downlink (D) bandwidth for explicit (E) and mapped (M) transfer of 256 kB, 1 MB, 4 MB and 16 MB. From 16 MB on the bandwidth

Table 6.2 Hockney model parameters T and L in seconds for upload data transfers to the GPU and downloads to the CPU for data shown in figure 6.3.

	Upload		Download	
Type	T	L	T	L
GTX 580	$2.76 \cdot 10^{-10}$	$2.11 \cdot 10^{-4}$	$5.51 \cdot 10^{-10}$	$2.91 \cdot 10^{-4}$
GTX 590	$5.45 \cdot 10^{-10}$	$4.71 \cdot 10^{-4}$	$1.10 \cdot 10^{-9}$	$5.55 \cdot 10^{-4}$
GTX 680	$5.31 \cdot 10^{-10}$	$4.91 \cdot 10^{-4}$	$1.09 \cdot 10^{-9}$	$6.69 \cdot 10^{-5}$
Tesla K20Xm	$2.86 \cdot 10^{-10}$	$-6.99 \cdot 10^{-5}$	$7.83 \cdot 10^{-10}$	$-1.30 \cdot 10^{-4}$
TITAN	$2.86 \cdot 10^{-10}$	$-1.90 \cdot 10^{-4}$	$6.46 \cdot 10^{-10}$	$-8.84 \cdot 10^{-5}$
Radeon R9-290	$4.54 \cdot 10^{-10}$	$-1.13 \cdot 10^{-3}$	$9.54 \cdot 10^{-11}$	$9.79 \cdot 10^{-5}$
Xeon Phi	$8.53 \cdot 10^{-10}$	$-7.99 \cdot 10^{-3}$	$4.13 \cdot 10^{-10}$	$4.95 \cdot 10^{-3}$

stabilized and did not change for larger data sizes. From 256 kB to 16 MB, mapped and explicit up- and downlink bandwidths increase on all GPU platforms. However, mapped transfers are consistently lower than explicit transfers and stabilize between 4 and 16 MB. On the Xeon Phi, UE and DE drop from 2400 MB s^{-1} and 5790 MB s^{-1} to 1470 MB s^{-1} and 2090 MB s^{-1} for data sizes larger than 12 MB while the data rate keeps increasing for UM and DM. The Xeon E5-2640 CPU UE bandwidth drops right after 4 MB to a value that is lower than the possible CPU memory bandwidth. The CPU bandwidth was measured with the STREAM benchmark.[4] We tested the data transfer for one to three-dimensional data structures but could not measure any differences. It is likely that data is transfered as one-dimensional linearized arrays.

Single CPU-GPU system are simple heterogeneous compute systems and as such can be described by their Hockney model parameters (see (2.9) in section 2.1.2). We fitted the data transfer times to the model parameters with the results shown in table 6.2. The model parameters agree with a mean squared error smaller than 10^{-4} from the true data across all architectures. Similar measurements and model fittings were performed by Boyer et al. using the CUDA system solely for NVIDIA GPUs and then used to estimate the run-time of single kernels [13].

6.2.2 Multi GPU memory bandwidth

Multiple GPUs are the core components of a single compute node of a heterogeneous compute system. Although they scale perfectly in terms of compute power, the limited number of PCIe lanes that are physically available on a CPU and supported by the PCIe switch reduce the attainable memory bandwidth of a single GPU. Thus, to get a complete picture of the system parameters, we have to determine the aggregate memory bandwidth of all GPUs.

[4] `http://www.cs.virginia.edu/stream`

We measured the memory bandwidth on the *compute1* system which consists of a single NVIDIA GTX 680 and four NVIDIA GTX 590. Each GTX 590 consists of two independent GPU cores with their own separate memory. The OpenCL run-time detects each core as a single device. Although, all cards are connected via PCIe gen2 x16, only the GTX 680 attains the full transfer speed of 5 GT s^{-1}. Two GTX 590 *cards* are connected via the same PCIe bridge, thus both cards have to share PCIe lanes. In these cases, the transfer speed is reduced to 2.5 GT s^{-1}.[5] We split data between 256 kB and 16 MB according to the number of chosen GPUs and measured the total aggregate memory bandwidth.

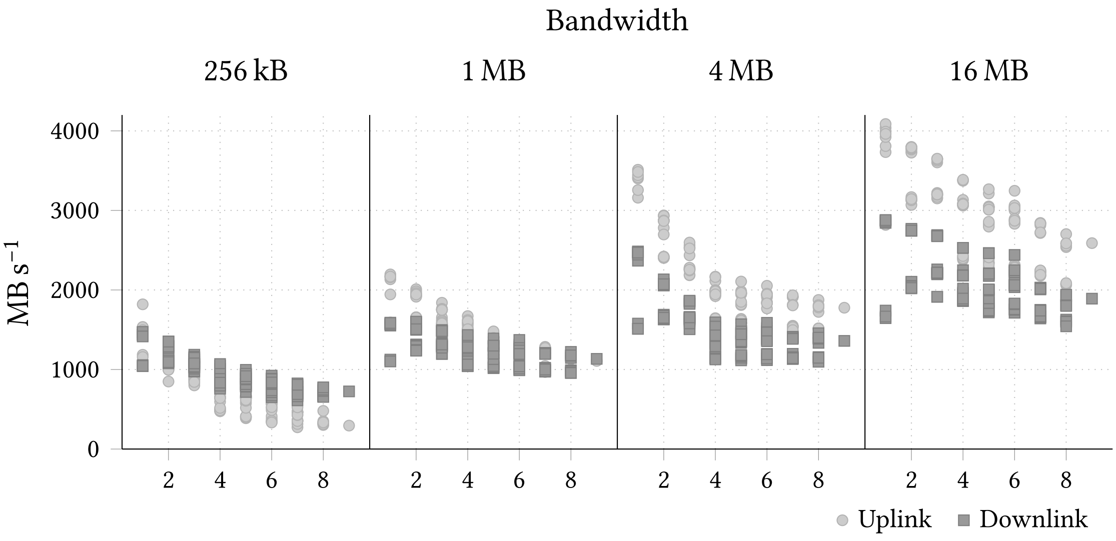


Figure 6.4 Total memory bandwidth measured for one to nine GPUs and four different data sizes.

Figure 6.4 shows the memory bandwidth achieved with multiple GPUs at four different data sizes. For data sizes smaller than 1 MB, the memory bandwidth from device to host is almost always higher than from host to device independently of the number and combinations of GPUs used. For any larger data size, the situation reverses and memory bandwidth from host to device is higher. Across all data sizes, the memory bandwidth reduces with an increasing number of GPUs. This can be attributed to the fact that more GPUs share a limited number of PCIe lanes. As with a single GPU, transferring larger data utilizes the full memory bandwidth of the PCIe bus up to a saturation point of 16 MB per transfer. The Hockney model is too simple to capture the performance decrease of multiple GPUs, instead we can use a modified LogP model such as

$$T(m) = L + P \cdot o + \frac{m-1}{P} \cdot G + L + o. \tag{6.1}$$

[5] The figures were collected from the output of the Linux `lspci` PCIe analysis tool.

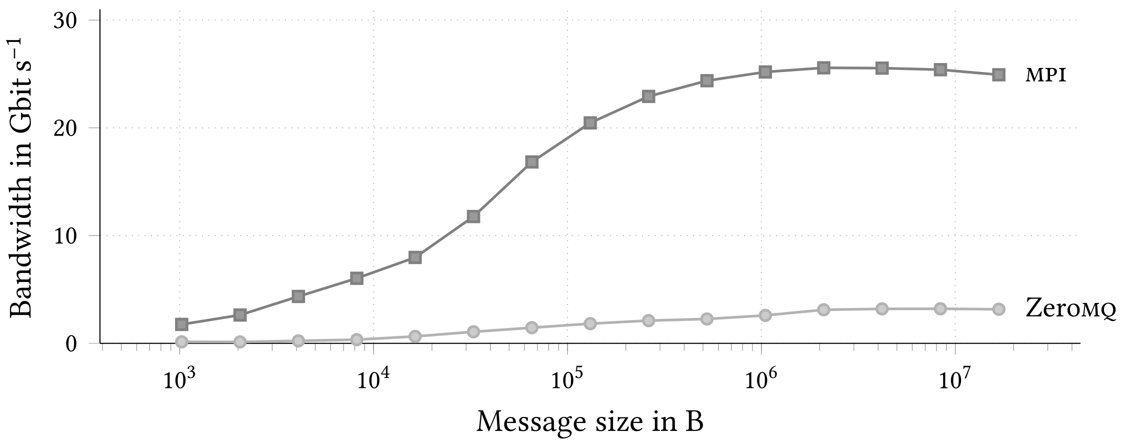


Figure 6.5 Bandwidth on *cluster1* using ZeroMQ and MPI. The theoretical bandwidth of the system is 40 Gbit s^{-1}.

Van Werkhofen [135] used the LogP family models to estimate the performance of data transfers using CUDA on three different GPUs.

6.2.3 InfiniBand performance of MPI and ZeroMQ

The performance of the network interconnect plays a crucial role when scaling the number of compute nodes in a heterogeneous compute system because the aggregate bandwidth of a star topology is even worse than on a single machine. Moreover, different higher-level software layers have a performance impact to a varying degree.

The *cluster1* system is a small cluster located directly at ANKA's IMAGE beamline. It consists of four GPU nodes connected via a Mellanox ConnectX PCIe 2.0 HCA to an InfiniBand switch. *cluster2* is a small research cluster located at the IPE and consists of six GPU nodes. The nodes are each equipped with a Mellanox ConnectX 3 PCIe 3.0 host channel adapters and connected to a Mellanox IS5022 InfiniBand switch. For this measurement, we used the vendor benchmark tools of MPI and ZeroMQ. ZeroMQ is a high-throughput socket abstraction to implement communication patterns besides the basic server-client model.

Figure 6.5 shows the achieved network bandwidth of *cluster1* for an increasing message size. At 1 MB, MPI saturates the maximum possible link bandwidth of 24.9 Gbit s^{-1} provided by the InfiniBand interconnect. ZeroMQ, however, is limited by the InfiniBand-over-IP overhead to a maximum bandwidth of 3.16 Gbit s^{-1} which is about 10 % of the net bandwidth. For high volume data processing, there is no alternative to MPI or direct programming of the InfiniBand HCA. Although ZeroMQ provides a conceptually fitting communication model, the performance is too low for any soft real-time reconstructions at the beamline.

6.3 Local data processing

In this section we are going to present performance results of X-ray imaging processes gathered on a single compute node.

6.3.1 Tomographic reconstruction

The tomographic reconstruction of pre-processed input projections (see 2.4.2) is the most compute-intensive X-ray imaging task of an on-line experiment workflow. To react quickly on changes *within* a sample low processing latencies and to process the expected data volumes in a reasonable amount of time, high throughput is required. In this section, we will evaluate suitability of the heterogeneous compute framework for soft real-time on-line reconstructions by estimating data throughputs and comparing them to the expected experiment data rates.

To evaluate the throughput of the FBP algorithm, we constructed a pipeline consisting of six filter stages: a sinogram generator, the forward FFT, the one-dimensional ramp filter,[6] the inverse FFT, the backprojection and an output filter stage. To remedy the influence of I/O delays, the input stage merely returns sinograms without writing any data and the output stage immediately discards any received input data. Using this pipeline, we processed 128 to 3968 sinograms increasing in steps of 128 on *ufosrv*. The sinogram dimensions ranged from 128×128 to 1920×1920 pixels in increments of 128 pixels. For each parameter configuration, we estimated the *in-bound* reconstruction throughput – the data rate in terms of processed number of sinogram bytes – instead of the produced number of slice pixels or bytes.

Figure 6.6 shows the throughput for a subset of sinogram widths and heights: The plot on the left side shows the throughput for fixed widths, the plot on the right side for fixed heights. As expected, the overall throughput is dominated by the width of the sinogram because the sinogram width has direct impact on the final width *and* height of the reconstructed slice. The height of sinogram, on the other hand, merely determines the number of reconstructed slices. Thus, increasing the width of the sinograms reduces the amount of reconstructed slices.

Despite the general trend, we can also see partial performance increases for widths from 384 to 512, 640 to 1024 and 1152 to 2048 pixels. This effect is caused by the padding of the input to the next power of two required for the FFT. The larger the difference to the next power of two, Moreover, we notice that the work group utilization on the GPU increases for sizes that approach a power of two. In general, the bandwidth ranges from 220 MB s^{-1} for 1152×896 pixels down to 2840 MB s^{-1} for 128×1920 pixels. We can also see, that the sinogram height has a minor effect on the overall reconstruction performance. Although for a width of 128 pixels the performance increases from 1250 MB s^{-1}

[6] The filter attenuates the frequencies on *all* one-dimensional projections or rows of the Fourier transform of the sinogram.

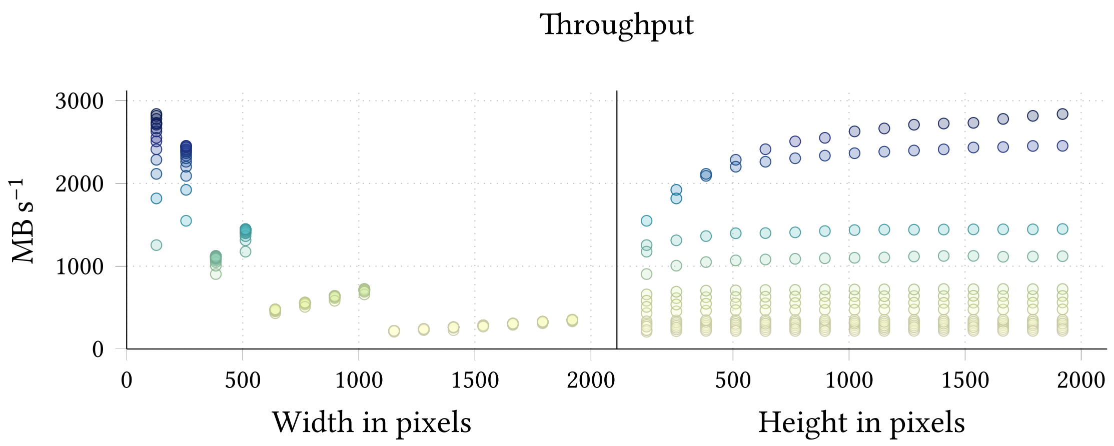


Figure 6.6 In-bound reconstruction throughput on *ufosrv* for different numbers of projections projected onto width and height axis.

to 2840 MB s^{-1}, the standard deviation drops quickly from $\sigma = 422.3$ to 240.1 for 256 pixels and down to 3.9 for 1536 pixels. Thus we can conclude, that the number of acquired projections is not the bottleneck of the reconstruction.

Influence of hardware architecture

The previous experiments have shown the throughput and scalability of the heterogeneous run-time system on a single hardware platform. To find the best platform for high-throughput reconstruction required for feedback-based experiments, we measured the throughput for the FBP and DFI algorithms which need only a single pass and thus can reconstruct with lowest possible latencies.

Figure 6.7 shows the in-bound throughput for a fix-sized dataset of real data. The dataset consists of 1700 projections each with a size of 1778×2000 pixels. Therefore, instead of reconstructing from sinograms we can simulate the on-line reconstruction chain by reconstructing directly from the projections. This requires an additional step to transpose the projections into sinograms (see). After this step, the regular DFI and FBP algorithms algorithms reconstructed the slices. The reconstructions were carried on GTX 590 and GTX 680 GPUS of *compute1*, on the GTX TITAN and Tesla K20Xm of *compute2* as well as on a single AMD R9 290 of *compute3*. As expected, DFI outperforms FBP by a factor of two across all platforms and has a maximum throughput of 1654 MB s^{-1} on a GTX TITAN.

From this number we can conclude that we are able to provide *soft* real-time data processing for the data rates expected from the detectors.

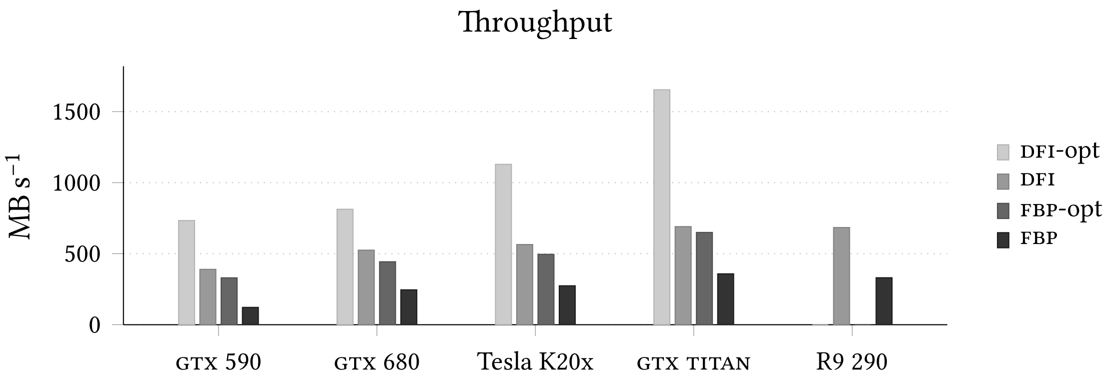


Figure 6.7 Reconstruction bandwidth of standard and optimized versions of FBP and DFI running on different single GPUS.

ART comparison with ASTRA

ASTRA is a GPU-accelerated MATLAB-based toolbox [98] for tomographic reconstruction that we introduced in 3.4.4. Although it was not designed for integration into SRμCT experiment workflows and control systems, it is a mature tool. Hence, we use ASTRA as a baseline to compare the performance of our heterogeneous compute system for qualitatively advanced reconstruction methods. We measured the time to reconstruct slices of size 512×512 pixels using the SART method of the ASTRA CT framework and our heterogeneous pipeline system on a GTX TITAN. From the measured times we computed the speed up of our system against ASTRA for four different iteration settings. ASTRA did not scale in the same way as our framework for low number of iterations which is caused by I/O problems on behalf of ASTRA. To get a clear picture on the influence of I/O operations, we measured the speed up once with and once without taking I/O into account.

From figure 6.8 we can see, that our platform provides positive speed ups across all iteration settings. Without I/O, the speed up ranges from 1.6 for one iteration and converges towards 2.8 for more than one hundred SART iterations. A better filesystem cache utilization caused by the pipeline is the most likely explanation for an improvement of 11.6 % and 14.9 % better speed up achieved when taking I/O into account.

Scalability with multiple GPUS

The preceding experiments used a single GPU to compute the steps of a tomographic reconstruction. To evaluate the performance on multiple GPUS, we measured the runtime taken by the same FBP, DFI and ART reconstruction pipelines on an increasing number of GPUS. The pipelines were executed unmodified in order to determine the effect of the subgraph duplication strategy (see 3.3.2) on the scalability. The single GPU

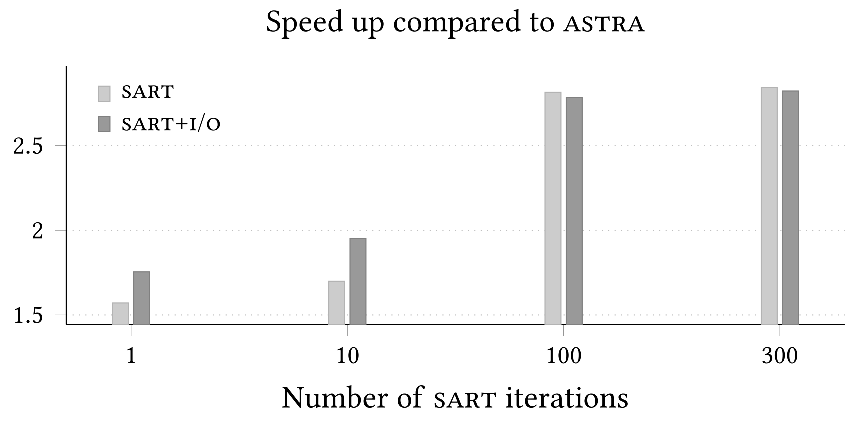


Figure 6.8 Speed up of UFO compared to ASTRA on a GTX TITAN.

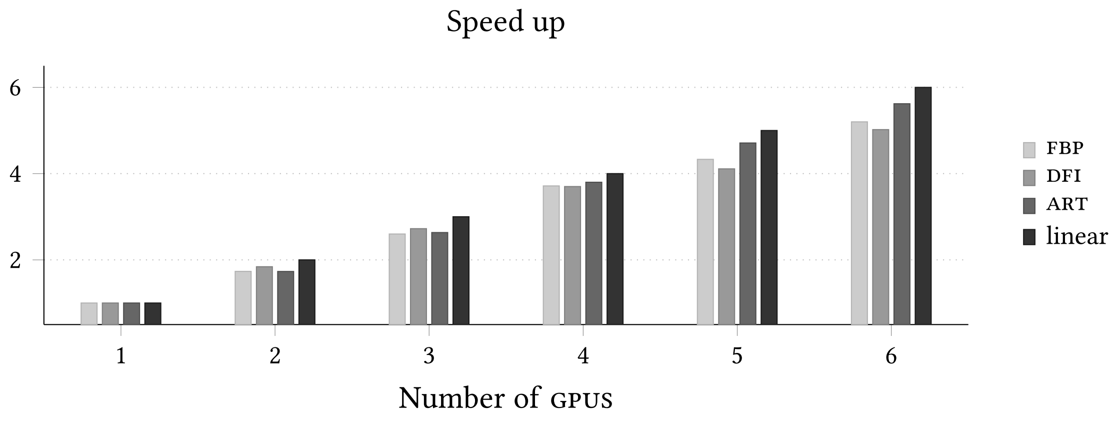


Figure 6.9 Speed up of FBP, DFI and ART reconstructions for one to six GPUS. For comparison, perfect linear speed up is shown.

case was measured with a single NVIDIA GTX 580 of the *ufosrv* machine. For each data size, we increased the number of GPUS with a maximum number of to six NVIDIA GTX 580. We measured the run-time and computed the relative speed up compared to the single GPU.

Figure 6.9 shows the speed up for the three reconstruction methods and a bar that scales linearly with the number of GPUS. The data shows that all three reconstruction methods scale almost linearly. On average FBP has a parallel efficiency of 89.9 % (with standard deviation $\sigma = 0.050$), DFI 90.2 % ($\sigma = 0.059$) and ART 92.9 % ($\sigma = 0.045$). The slightly better scalability of ART stems from the fact that ART is considerably more compute-intensive and less susceptible to the data transfer limits that we observed earlier. The data proves that our sub-graph duplication heuristic provides scalable reconstruction performance on a local machine. Figure 6.10 shows a short excerpt of an

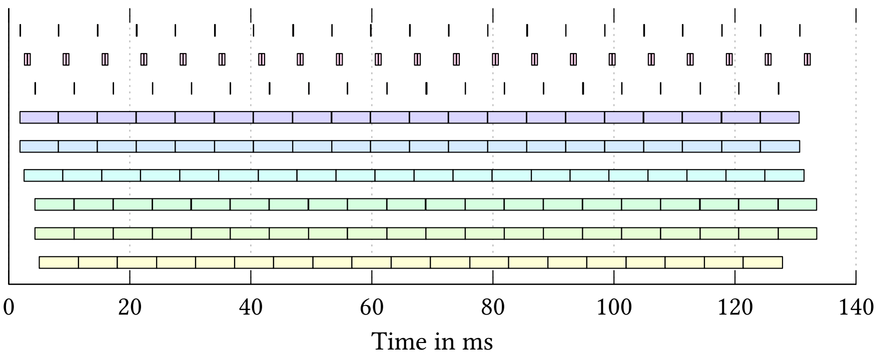


Figure 6.10 Overlapped execution of an FBP pipeline on two GPUS.

execution trace recorded for the FBP reconstruction on two NVIDIA GTX GPUS. As before, the subgraph duplication strategy distributed the work among both GPUS. From the figure we can see that this allows a tight overlapping of duplicate tasks that run on distinct GPUS.

Sinogram transposition

Section 6.3.1 introduced measures to improve the throughput of the sinogram transposition operation. To validate the individual approaches, we generated one thousand projections with a size of $w\times 1024$ pixels. The projection width w ranged from 512 to 2560 pixels. We measured the time to transpose the synthetic projections on the *desktop* machine and estimated the throughput given in number of sinogram bytes produced in a second. The default *baseline* implementation uses a *for* loop to write each row of a projection to the corresponding sinogram volume location. The MC implementation replaced this inner loop with a `memcpy`. The MT implementation uses OpenMP to run the outer loop in parallel and distribute partitions to different threads. The MC+MT approach combines both MC and MT.

Figure 6.11 shows the *relative* transposition throughput of the three optimization approaches. The baseline performs worst and peaks at about 1.05 GB s^{-1} across all projection sizes. Copying inner rows with `memcpy` improves this performance by a factor of up to 2.54 and peaks at 2.94 GB s^{-1} for the largest size of 2560×1024 pixels. The MT approach exhibits a speed up compared to the baseline of up to 3.77 and reaches its peak throughput of 4.06 GB s^{-1} at 2048×1024 pixels. Combining both approaches results in the highest transposition performance for the largest data sizes of 7.25 GB s^{-1} which is 5.9 times faster than the baseline implementation. On this system we measured a peak memory bandwidth of 9.64 GB s^{-1} using the STREAM benchmark. Hence, the MC+MT approach has a minimum memory bandwidth impact of about 25 %.

From these results we can estimate if we can reliably transpose streamed projections. From table 4.1 we see that the fastest streaming detector produces 330 frames per second.

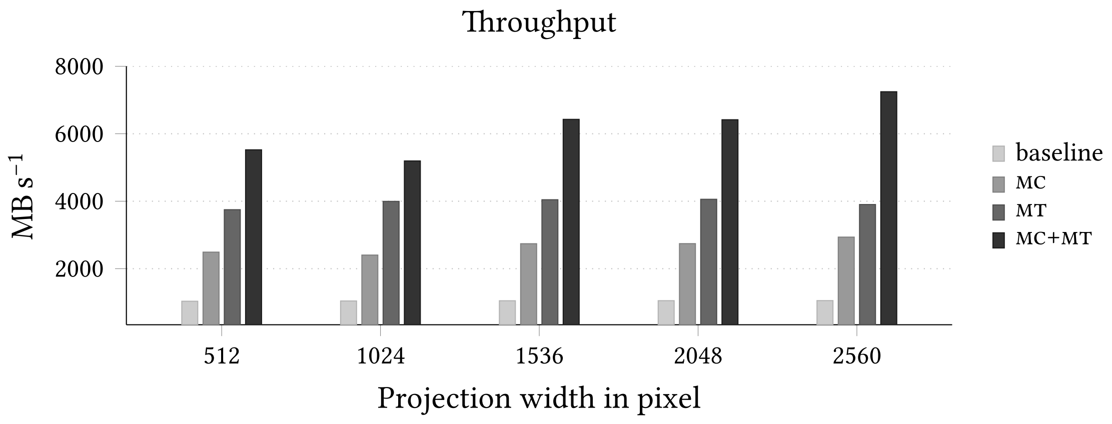


Figure 6.11 Transposition performance using combinations of `memcpy` (MC) and multithreading (MT) optimizations. The height and number of projections was fixed to 1024 and 1000.

Each full frame has a resolution of 2048×1088 pixels at 16 bit per pixel. At the maximum frame size and rate, data is streamed with a throughput of $2048 \cdot 1088 \cdot 2 \cdot 330 / 1024^2 \mathrm{B\,s^{-1}}$ or $1.37\,\mathrm{GB\,s^{-1}}$. At 2048×1024 pixels, the respective transposition throughputs for baseline, MC, MT and MC+MT are $1.04\,\mathrm{GB\,s^{-1}}$, $2.40\,\mathrm{GB\,s^{-1}}$, $4.00\,\mathrm{GB\,s^{-1}}$ and $5.20\,\mathrm{GB\,s^{-1}}$. Thus with MC, MT and MC+MT, we can transpose the projections at 57 %, 34 % and 26 % of the peak transposition bandwidth.

Fast Fourier transforms on GPUS

The Fourier transform is an integral part of the DFI and FBP reconstruction algorithms (see 2.4.4). To compute the Fourier transform as fast as possible, we pad the arbitrarily sized input data with zeros to a size equal to the next power of two and use the FFT algorithm with $O(n \log n)$ complexity. Due to the regular data access patterns and basic arithmetic operations used in the algorithm, computing the FFT for large data is particularly suited for GPUS.

At the time of this writing, two mature and freely available OpenCL FFT implementations by Apple[7] and AMD[8] are available. Both libraries generate the kernel code at run-time for the target architecture and the intended input data size. In order to determine the most suitable implementation for our reconstruction purposes, we measured the time to compute the FFT for a varying number of dimensions, including the time to transfer the data between device and host. We measure the speed up compared to the FFTW3 library [44], which claims to be the fastest CPU implementation available.

[7] `https://developer.apple.com/library/mac/samplecode/OpenCL_FFT/Introduction/Intro.html`
[8] `https://github.com/clMathLibraries`

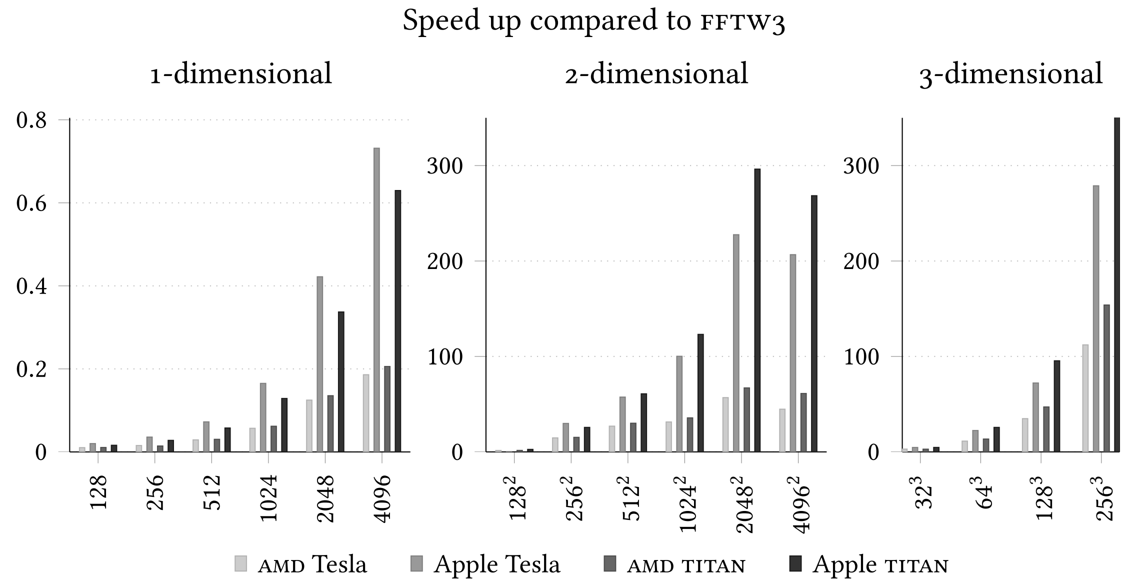


Figure 6.12 Speed up for 1-, 2- and 3-dimensional FFT implementations by AMD and Apple compared to FFTW3 and measured on *compute1* and *compute2*.

Figure 6.12 shows the speed up measured on *compute1* and *compute2*. For small non-batched one-dimensional data sets, transferring the data and computing the FFT on the GPU is always slower than the CPU-based FFTW3. One exception is the transform of 4096 elements using the Apple FFT on *compute1* that has a nominal speed up of 1.3. The low performance for the smaller data sizes is caused by the inherent overhead of transferring the data from host to device and back and relatively little computations required by the one-dimensional FFT. For two- and three-dimensional data, this overhead is mitigated by the amount of necessary arithmetic operations. Here, the GPU implementations perform much better. For larger data sizes the speed ups of both libraries peak at 300. On the given NVIDIA-based systems, the Apple FFT performs always better than the AMD FFT. For the three-dimensional data set of size 256×256×256, the Apple FFT is more than twice as fast as the AMD FFT. We suppose that the difference stems from one-sided optimizations towards AMD hardware.

6.3.2 Non-local means denoising

We outlined a GPU-based algorithm to compute the NLM noise reduction algorithm in 5.3.2. We implemented this algorithm as part of our compute framework and estimated the performance improvements compared to the reference implementation by Buades [16]. This baseline implementation uses OpenMP multi-threading to compute partitions of denoised pixels in parallel. We measured the time required to execute the reference as well as our GPU implementation on the *desktop* for input images ranging from sizes between 256×256 and 2048×2048 pixels using identical denoising parameters $\sigma = 10$, patch size $k = 20$ and window size $w = 2$.

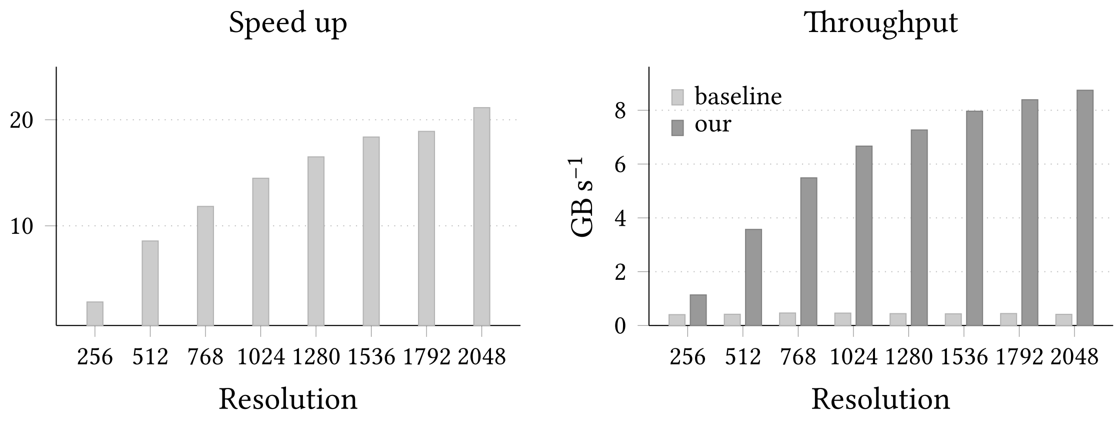


Figure 6.13 Speed up of our implementation compared to Buades' baseline implementation on the left and absolute throughput for both implementations on the right.

On the left of figure 6.13, we can see the nominal speed up of our GPU-based implementation against the run-time of the reference implementation which utilized 100 % of all CPU cores. For smaller data, the data transfer between host and device inhibits larger speed ups. For larger data sizes, the speed up increases from initially 2.8 until it peaks with a speed up of 21.1 at 2048×2048 pixels. On the right side of figure 6.13, we can see the absolute throughput achieved with both implementations. The reference implementation has an average throughput of 0.43 GB $\mathrm{s^{-1}}$ and peaks with 0.46 GB $\mathrm{s^{-1}}$ at 1024×1024 pixels. The throughput of the GPU implementation varies between 1.1 GB $\mathrm{s^{-1}}$ at 256×256 pixels and 8.7 GB $\mathrm{s^{-1}}$ at 2048×2048 pixels.

The NLM algorithm is suited for execution on both multi-core CPUs and many-core GPUs. Nevertheless, the performance on a GPU is an order of magnitude better on a GPU. Considering the data rates produced by contemporary detectors, we can conclude that NLM is only applicable for soft real-time applications using GPU.

6.3.3 Optimizations

In 3.5, we proposed low-level optimizations to improve the throughput and bandwidth of the heterogeneous compute architecture. In this section, we will evaluate the impact of these optimizations on the run-time and present possible speed ups.

Batched transfer

In section 3.5.1, we proposed a batched memory buffer method to reduce the overhead of small data transfers between host and device. To evaluate the method, we measured

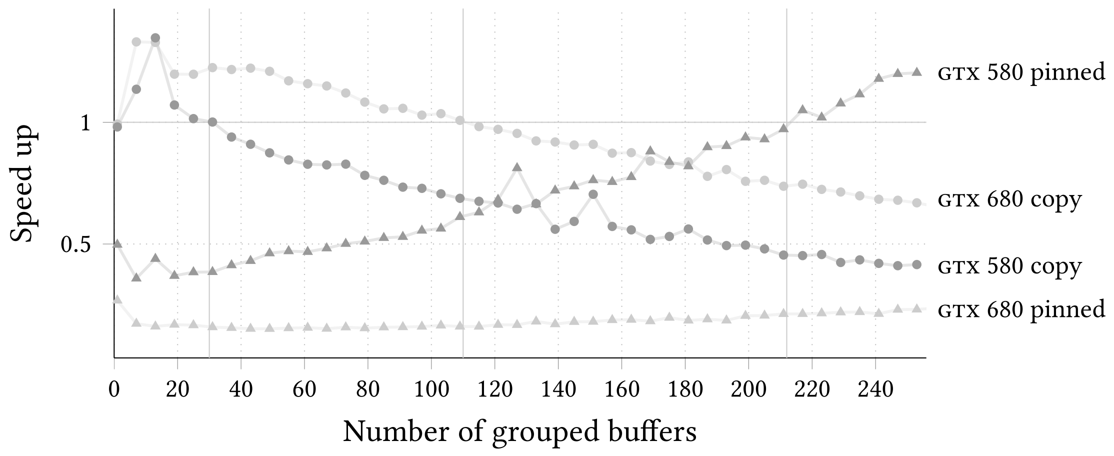


Figure 6.14 Sub-sampled data points of relative speed up for batched transfers versus single transfers as a function of the number of buffers.

the time required to transfer a single fixed size data buffer and its equivalent split into into 2 to 256 smaller buffers as well as the time needed to execute a compute-intensive kernel. In the case of the single buffer, we initiated the same number of kernel launches as in the case of the multiple buffers. Both single and multiple buffers were transferred explicitly using `clEnqueueWriteBuffer` (copy) and implicitly using `clEnqueueMapBuffer` (pinned) as explained in 6.2.1. We ran this experiment on the *desktop* consisting of two GTX 580 Fermi cards and a machine equipped with a GTX 680 Kepler card. From the measured time we computed the relative speed up of time required to transfer the large buffer against the time required to transfer the smaller buffers.

Figure 6.14 shows the trend of the relative speed up for partitions of up to 256 buffers. Although the figures are related to the bandwidth presented in 6.2.1, we have to consider the overheads of managing multiple smaller buffers and sending data repeatedly to the GPU. As expected, manually transferring a single buffer with a copy is faster than sending multiple smaller buffers for both cards. The speed up on the GTX 590, however, decreases for a larger number of buffers and drops below 1 at 30 buffers. The speed up of the GTX 680 remains above one until 110 buffers are reached. The reduced speed up can be attributed to the smaller work group sizes that can be scheduled on the GPUS and cause better cache and memory access behaviour for smaller data buffers. In case of pinned data transfers, single large data transfers perform worse on both cards but gradually improve. For the GTX 580, mapping smaller buffers becomes beneficial at a number of 212 buffers. The results indicate an increased run-time performance of up to 32 % using explicit data transfers compared to regular data transfers. Although, this is a significant speed up, the latency introduced by grouping the buffers has a considerable impact on the subsequent on-line control.

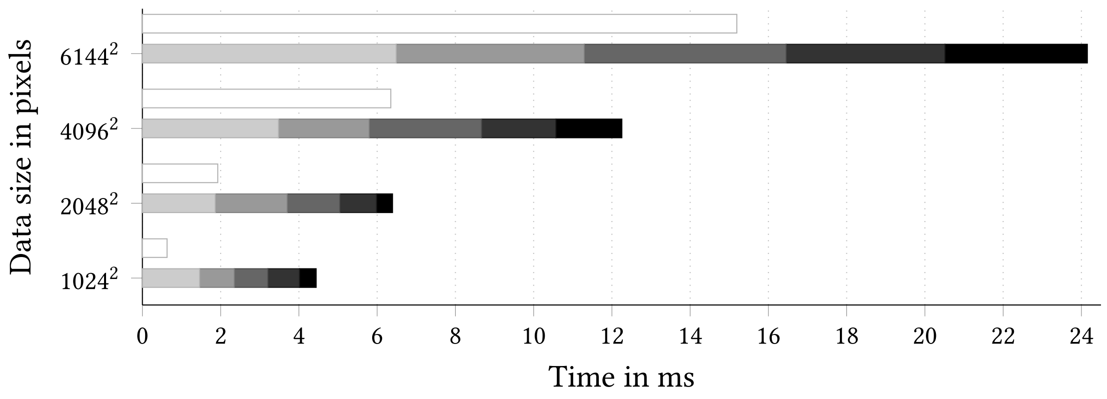


Figure 6.15 Computation time for multiple kernels against a fused kernel of all five kernels.

Kernel fusion

In section 3.5.2, we motivated the principal benefits of reduced kernel launches and separated data buffers by fusing groups of subsequent kernels into a single kernel. To evaluate the effect of the kernel fusion optimization, we first built an image processing pipeline consisting of five imaging tasks with similar computational intensity. We executed the pipeline on *compute1* with image sizes between 1024×1024 and up to 6144×6144 pixels. We measured the time required for each pipeline stage as well as the fusion and execution time for the functionally equivalent single kernel.

In figure 6.15 the run-times evaluated at four different image sizes is given for the separate and fused kernel represented by by the gray and white bars. For all image sizes, fusing and executing the single kernel is faster than executing individual kernels. The relative speed up of the fused kernel depends on the input size and ranges from 7.04 for 1024×1024 pixels sized input data down to 1.59 for the largest image size of 6144×6144 pixels. The decrease in speed up is in agreement with the performance model discussed in section 3.5.2: For smaller data sizes, the overhead of launching five kernels instead of one outweighs the time required for computing the individual tasks. For larger data sizes, the time required for computing individual or combined tasks dominates the total execution time, thus diminishing any gains.

We see that kernel fusion can reduce the run-time of *pipelined* operations significantly. However, there are two reasons not to enable this optimization for all applications. First, pipelines that consist of a mixed set of CPU and GPU tasks or tasks that use external third-party libraries to use GPUS cause multiple fusion steps and fused kernels. In this case, the benefit of potentially shorter run-times is mitigated by the time needed to fuse the kernels. Second, in pipelines with strict data dependencies, which means *each* kernel has to wait for its predecessor to complete, and compute-intensive smaller kernels, the fused kernel performs worse than their individual counterparts. In

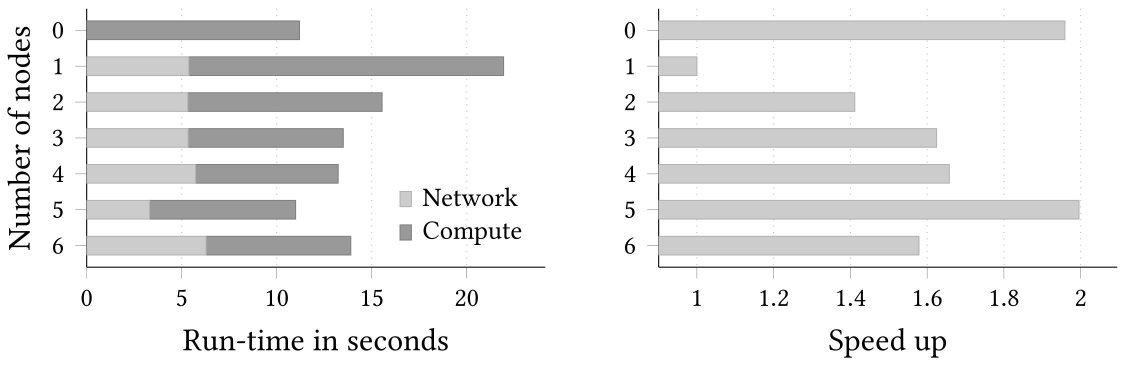


Figure 6.16 Left: Run-time split into network communication and compute for the FBP algorithm on up to six nodes of *cluster2*. Right: corresponding speed up compared to one master node.

this case, the synchronization between the task must happen *within* the fused kernels, hence blocking the entire task on a single data item. If the kernels were not fused, a predecessor can already start working on the next data item and overlap execution with its successor. This became a problem for the FBP pipeline with a very long-running backprojection step that blocks during the filtering stages.

6.4 Distributed data processing

In 3.3.3, we outlined the extension of the heterogeneous compute system for small-scale clusters which we will evaluate on *cluster2*.

Figure 6.16 illustrates the run-time and the relative speed up of a tomographic reconstruction. The topmost bar denotes execution using a single local node without network or MPI involvement. The remaining bars show the execution time with regard to the number of remote nodes used in the process. The light part of each bar depicts the fraction of the run-time that is spent performing network operations, while the darker shaded parts of the execution time spans the parallel computation of the program.

From the data we can tell that the time spent for computation equals the time required to process the data locally, using only a single remote node. This is expected behavior as the two nodes are identical in terms of hardware and therefore computation of the same input data takes the same amount of time on both machines. When using remote computing nodes instead of local computation, the required time for sending and receiving the data is added to the total runtime.

Because the amount of transferred data remains the same, one would expect the network time to remain constant when increasing the number of remote nodes. Instead, it can be observed that the network time increases with the number of remote nodes.

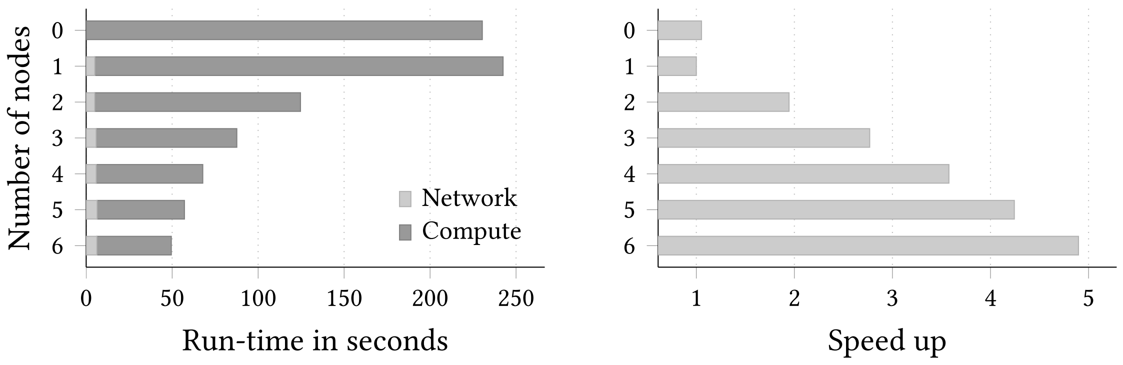


Figure 6.17 Left: Run-time for noise reduction using NLM and subsequent FBP reconstruction. Right: Corresponding speed up.

Even though the source of this increase can not be identified for certain, it is most likely attributed to the naive busy-waiting implementation of the thread barrier described in section 3.3.3. The more remote nodes participate in the process, the more threads invoke the barrier function and busy wait on resources. Although the threads yield thus preventing starvation of other threads, the permanent cyclic yielding of threads increases CPU utilization. This assumption is supported by two additional observations. First of all, the increase of network time becomes prominent when using four or more remote nodes. As the central node in the evaluation system uses a CPU with four cores, the increase may be a result of CPU over-subscription. When using four remote nodes, at least three threads busy wait on a single dedicated core, while the last core is used for other activities. When using five remote nodes, all four idle proxy threads use a single core and the problem worsens. Second, the increase of network time could not be reproduced on *cluster1* when using a six-core CPU and four remote computing nodes.

The network time contributes to the serial amount $1 - \alpha$ of the program runtime. Using (2.6), we can deduct a lower bound for the execution time of at least 5.32 s. Local execution time of 11.20 s can be halved to 5.6 s by adding a second GPU device of the same type. The lower bound predicted is then just 0.3 s faster. For this application, such low advantage may not be worth the effort to utilize a cluster system and leads to the conclusion that local processing should be favored for fast reconstruction purposes.

We evaluated the remote execution with a computationally more intensive processing pipeline that prepends an NLM denoising stage before the reconstruction stage, hence increasing the compute-to-transfer ratio. The results depicted in figure 6.17 are structured the same way as figure 6.16. Compared to the previous evaluation, the amount of data that is sent through the network link of the central node remains the same, but the time to process a work item becomes significantly higher. Overall processing time is now in the range of minutes when using a single node. The negative impact of

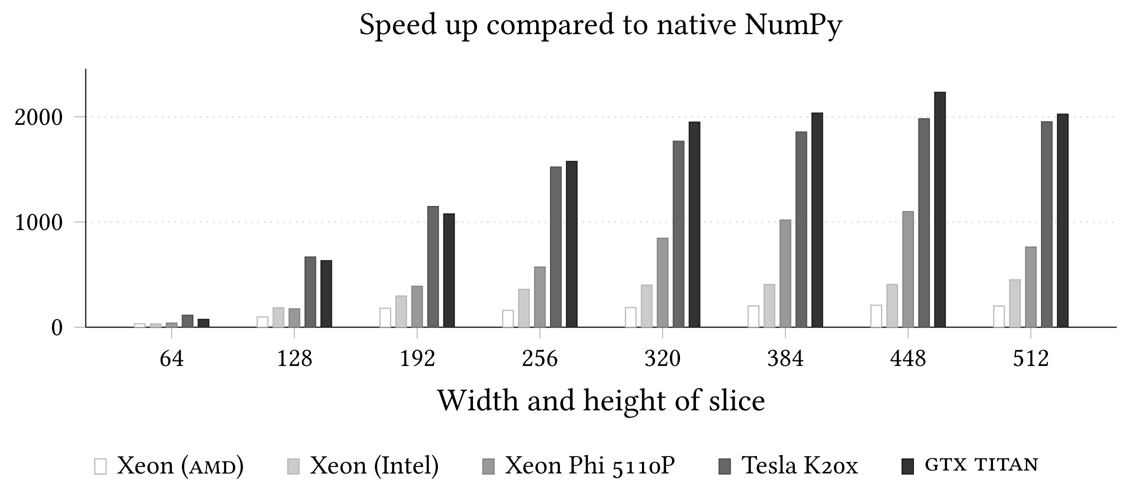


Figure 6.18 Reconstruction speed up for FBP across different hardware architectures compared to optimized NumPy code.

increased networking time is negligible, as the computing time is two orders of magnitude higher than the amount of network time. The actual processing time when using multiple nodes comes close to the predicted processing time when assuming an ideal speed up.

6.5 Just-in-time OpenCL code generation

The system presented in 5.1 translates Python functions into OpenCL kernels that are either passed to our heterogeneous run-time system or executed using its own run-time system on the fly. This enables programmers to benefit from the performance of GPUs without having deep knowledge about the hardware peculiarities.

We used three experiments to measure different aspects of the run-time system. The first experiment compares the execution time of the translated function – including the time required for translation – using all optimizations to the unmodified function. In particular, we use the fairly compute-intensive backprojection step of the FBP algorithm in this experiment. The unmodified function uses optimized NumPy code paths that employ C code rather than interpreted Python. We measured the time to backproject data into slices of size 64×64 up to 512×512 pixels and computed the speed up of accelerated execution.

In 6.18, the speed ups are shown for five different hardware platforms. Two OpenCL CPU implementations by AMD and Intel running on a Xeon E5-2640, an Intel implementation executed on a Xeon Phi and two NVIDIA GPU platforms. Across all platforms, we achieve one to three orders of magnitude in speed up. The AMD implementation ranges between 33 and 202, whereas the Intel SDK achieves speed ups between 32 and 451 on

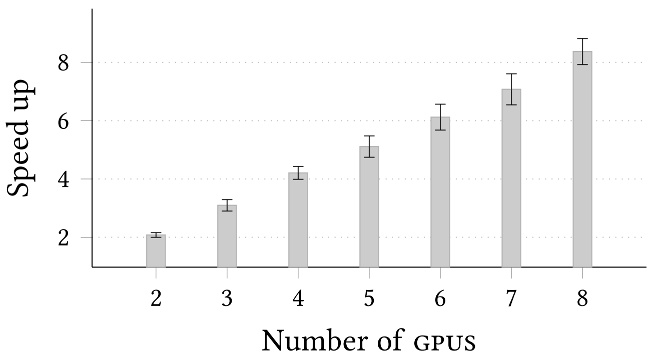


Figure 6.19 Mean relative speed up and standard deviations for the FBP algorithm with image sizes ranging from 4096×1024 up to 4096×4096 in steps of 256 run on up to eight GTX 590 cores.

the same hardware. The Xeon Phi MIC architecture has comparatively moderate speed ups between 40 and 763. The speed ups of the NVIDIA cards range from 40 and 75 for the smallest data size up to 1954 and 2026 for the largest slices.

As outlined in 5.1.3, the run-time system of the translator is able to partition the input data, generate appropriate kernels and distribute the partitions to the corresponding GPUS. After all GPUS finished computing the run-time collects the results and reconstructs the final result buffer on the host side. To estimate the effectiveness of the run-time, we used the backproject function from the previous experiment and executed it on up to eight NVIDIA GTX 590 cores of *compute1* for input data sizes between four and sixteen million pixels.[9]

Because the speed ups did not change significantly between different sizes, figure 6.19 shows the average scalability together with the standard deviation for all data sizes. As we can see, the system scales super-linearly but with a larger variance for a higher number of GPUS. We suspect that improved caching causes this behaviour. Contrary to the streaming system, the multi GPU implementation of the code generation system splits input data into smaller pieces to provide data parallelism on a multi GPU system. Since each GPU has to process smaller items, it is more likely to fetch data already present in the GPU's cache.

In 5.1.2, we introduced low-level micro optimizations that transform and replace entire sub-trees of the translated function's AST with instructions that are built into OpenCL. Thus, the programmer can freely describe an algorithm without having to care too much about optimization. To evaluate the effect of the optimizations, we compared the times for microbenchmarking $\cos x\pi$ and $\arccos x\pi$ substitutions for input data sizes between 512×512 and 4096×512 pixels on a single NVIDIA GTX 680 from *compute1*.

[9] In this experiment we are not limited by the slow backproject time of the native NumPy code paths and could increase the data sizes significantly.

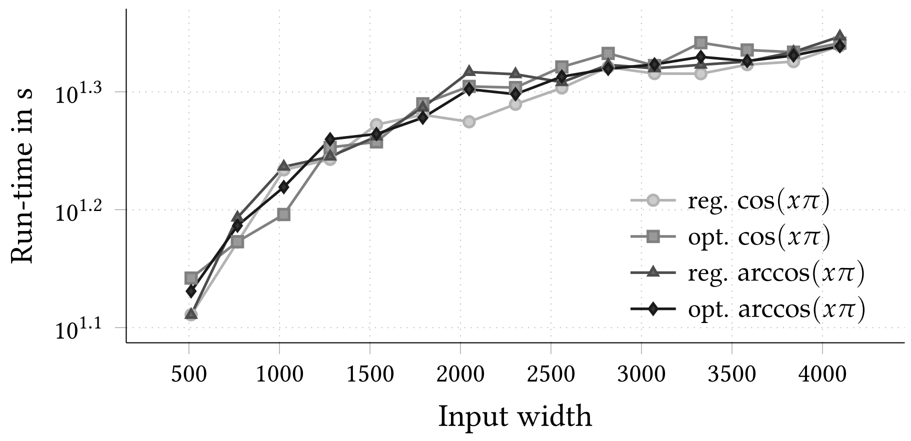


Figure 6.20 Speed up with and without substitution of cos and arccos by their optimized counter parts.

As we can see from figure 6.20 the influence of the micro optimizations on the run-time is marginal at best but in general does not improve the run-time significantly. These results have two causes. First, we include the time for the optimization phase which mitigates any potential performance increases. Second, the operations themselves do not execute in one clock cycle on the GPU but perform the same instructions as if they were written by hand. We investigated the binary compiler output of the `clBuildProgram` API call which on NVIDIA platforms is intermediate PTX assembly code [99]. The specialized instructions were broken down into simpler instructions that are computed by the GPUS ALU units instead of the SFU units also available.

6.6 Evaluation of asynchronous control

6.6.1 Low-level primitive overhead

In section 4.2.3, we presented the future concept to wrap asynchronous device access. With Python 3.2, futures became part of the standard library.[10] The standard future implementation provides both thread- and process-based executors to submit tasks. Our target platform is the *de facto* standard CPython, which is a single-threaded implementation of Python. Hence, any non-I/O operations will not profit from the parallel execution. In our case, we cannot use the process-based executor because tasks submitted to a process executor must be serializable in the default Python format and cannot share state.

Although CPython's thread implementation is based on real system threads managed

[10]Backports to earlier versions of Python are available, hence we can assume that they are available in virtually all Python version.

Table 6.3 Number of operations per second given for the dummy and the realistic random software motor. η is the relative performance compared to the baseline.

	Dummy			Random		
Primitive	Baseline	Future	η	Baseline	Future	η
thread	716.78	286.44	0.397	199.3	161.36	0.81
gevent	718.36	668.38	0.93	146.3	144.1	0.985

by the operating system, only one thread can execute at the same time. This is caused by Python's Global Interpreter Lock (GIL) which ensures a consistent state within the interpreter. This means that a parallel program that uses threads has no advantage over a sequential implementation except for an easier parallel programming model such as mapping device communication to a thread. On the contrary, the run-time may increase due to context switches and thread management. To overcome these limitations, we implemented the future interface using the *gevent*[11] event loop system which uses a single thread and kernel-level polling for fast I/O dispatching.

In order to determine the amount of overhead of futures based on threads and event loops, we measured the time to set random positions $x_k \sim \mathcal{N}(5, 3^2)$ on two linear software motors. The *dummy* motor compares the set position with the limits and stores the value if this is the case. The *random* motor approximates the dead time by waiting $|x_n - x_{n-1}| \cdot z$ seconds, where $z \sim \mathcal{N}(0.25, 0.01^2)$. For $n = 1000$ positions and three runs, we measured the number of set operations per second on *desktop* using synchronous operations and asynchronous futures. The thread primitive uses a pool of 128 threads – the default value of our production system.

From table 6.3, we can see that the pathological motor – which does nothing – performs 60 % worse than the sequential version using threads but only 7 % worse using the event loop. With the realistic motor model, the thread-based futures exhibit a performance impact of nearly 20 % whereas the event loop system performs close to 99 % of the baseline performance.

6.6.2 Concurrent device access

Figure 6.21 shows the efficiency of parallel device access according to (2.5). The chart shows the efficiency for two to eight independent processes, each consisting of up to nine sequentially ordered device accesses. Device accesses were simulated as explained in the previous section and were executed five times. In general, the efficiency is higher than 90 % ($0.002 \leq \sigma \leq 0.258$) in all cases except for one device access per eight independent processes.

[11] `http://gevent.org`

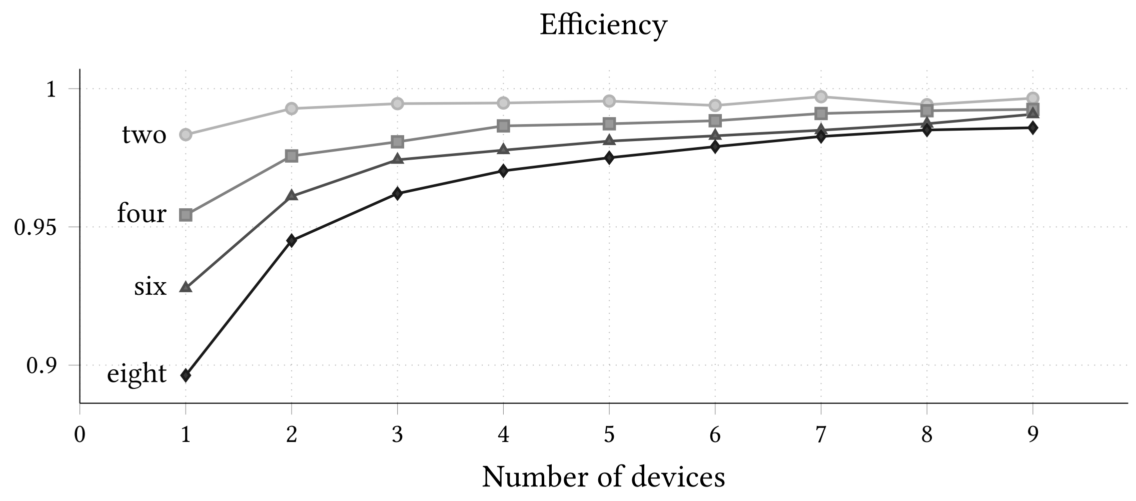


Figure 6.21 Efficiency of parallel device access for two, four, six and eight processes.

Although the results prove that the control system exhibits only low parallel execution overhead one has to take a real experiment into account. As with parallel programs, the amount of effective parallelism is bound by Amdahl's law. For complex motor motions, it is always necessary to synchronize motors to avoid catastrophic accidents causes by unintended crashes.

6.6.3 Remote data acquisition

In 4.1.3, we proposed an extension to the UCA framework that allows for transparent remote data acquisition through a secondary data channel. This channel is based on an InfiniBand connection to allow for independent data transfer at technologically highest possible throughputs. In this section, we want to evaluate if this extension allows us to stream the data using our fastest detectors.

We validated the performance impact of the remote data acquisition by transferring blocks of data with a size of up to 1 GB. For each data size, a dummy camera streams 1000 frames from the server machine to the host which controls the detector. We use a dummy camera to avoid accounting for overheads induced by the camera SDKS of the vendors by streaming directly from memory.

Figure 6.22 shows the measured bandwidth for data sizes from 1 MB to 1 GB. The InfiniBand HCAS are connected via InfiniBand 4 x QDR, delivering a theoretical bandwidth of 40 Gbit s^{-1} or an effective bandwidth of 32 Gbit s^{-1} caused by the 8 bit/10 bit encoding used on the link layer. As we can see, we achieve a net bandwidth efficiency from 82.5 % to 92.2 %. From these numbers, we conclude that we are able to stream data of the fastest contemporary detectors without disturbing the remaining control network.

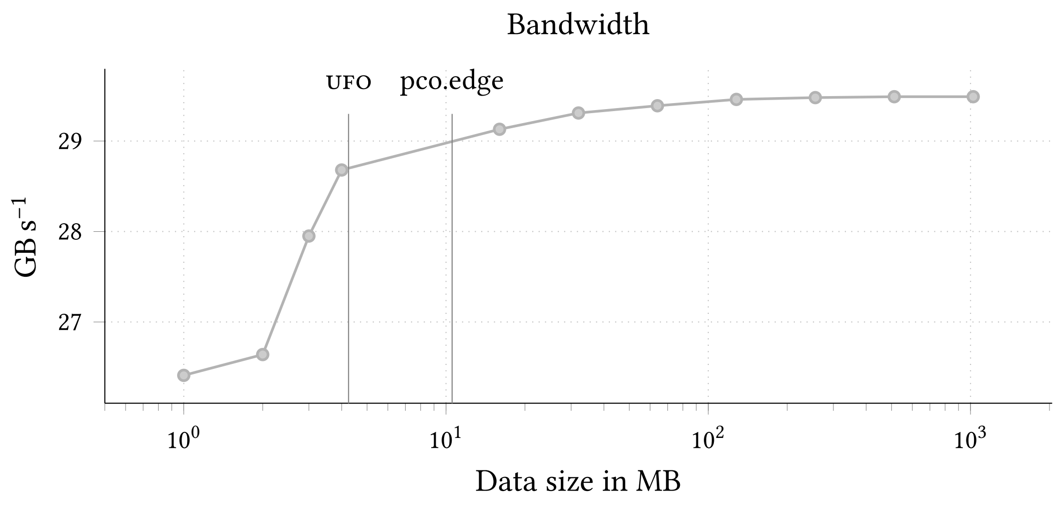


Figure 6.22 Bandwidth of RDMA data transfer of a remote dummy detector. Maximum data sizes of the UFO and pco.edge detectors mark possible bandwidths.

6.7 Summary

In this chapter, we analyzed the performance capability of the compute systems listed in table 6.1 that are used within the UFO projects. We have shown how the PCIe bus limits the data transfer bandwidth between host and devices, with severe performance drops in multi-GPU systems attached via an external box. Across all architectures, explicit memory transfers utilized the PCIe bus best. The Xeon Phi, however, suffers from a performance drop with data larger than 12 MB. For multi GPU setups, the same observations of single GPU data transfers hold, however, the bandwidth drops with more GPUs used. Enabling profiled queues is not harmful on the system performance as long as the data is sufficiently large and any overheads mitigated by the processing time. On the other hand, running multiple kernels concurrently can in the best case scale linearly with the number of kernels. We can also conclude that using multiple in-order queues is the only way to enforce concurrent kernel behaviour.

In the second part, we evaluated our proposed heterogeneous compute framework introduced in chapter 3 on the characterized hardware platforms. On a single GPU, the pipelining enables an FBP reconstruction throughput between 220 MB s^{-1} to 2840 MB s^{-1} of sinogram data. Despite the performance impacts encountered on multi-GPU systems, using the subgraph duplication heuristics introduced in 3.3.2 allow near linear scalability with scalability efficiencies of 90 % for FBP, DFI and ART reconstructions. A direct comparison with the CUDA-based ASTRA framework has shown, that we can reconstruct up to 2.8 times faster for SART reconstructions. A small study has shown that the free Apple FFT library provides enough bandwidth for on-line reconstruction. The proposed transposition techniques from 6.3.1 yield a speed up of 3.8 compared to the original strategy and peaks at a throughput of 5.20 GB s^{-1}. On GPUS we achieve a maximum NLM peak throughput of 8.7 GB s^{-1}, which is a speed up of 21.1 compared to the multi-threaded reference implementation. The batched data transfer presented in yields a potential speed up of 1.32 compared to regular single data transfers. The kernel fusion technique provides a speed up between 1.6 and 7. All together the proposed processing framework and optimization techniques permit soft real-time data processing required for on-line evaluation experiment purposes.

The final section contains evaluations concerning the experiment control and data acquisition purposes. The data acquisition framework has a minimal overhead of ...percent compared to raw memory copies, hence on a local machine we are limited only by the vendor SDK. We have also shown that our abstraction can efficiently utilize InfiniBand interconnects at up to 96.9 % and provide transparent remote access to all of our high-speed detectors.

7 Discussion

Our core research question is concerned with the design of a system architecture for online analysis of X-ray imaging data and the construction of fast feedback experiments. This requires a generic, low-level data acquisition framework to acquire data that is fed into a parallel computing framework that processes high-volume data streams on arbitrary heterogeneous compute systems. In the preceding chapters, we presented specific approaches how such a system can look like and how it affects the processing and experiment tasks. In the remaining chapter, we will discuss the results in light of the problem statement and research questions asked in chapter 1 and give an outlook of future work.

7.1 Impacts

High-performance streamed data processing

Question 1 asked for a processing model that is equally suitable to describe the flow of data streams and is easily mapped on heterogeneous system architectures, whereas question 2 more specifically asks how we can use architectural parameters to improve the performance of the overall system. Using the core framework for heterogeneous computing we can find the most appropriate algorithms suitable for X-ray image processing for SRμCT to answer question 3

In this thesis, we proposed a system model and an architecture based on arbitrary heterogeneous compute systems consisting of CPUS and GPUS to process large data streams. The system uses a graph of filter nodes to describe the data flow and map individual tasks at run-time to the available hardware resources. We fulfilled the objectives for a data streaming architecture given in section 1.2. In particular, we were able to scale the performance with increasing processing resources such as additional GPUS and compute nodes and achieved speed ups on local systems that were at most 15 % lower than a perfect speed up. We are also able to reconstruct tomographic data fast enough to provide instant feedback to the user and influence the control decisions.

Assuming that Moore's law will be valid for the foreseeable future, we can scale with upcoming hardware without changing the underlying software components. Moreover, using our system model we can adjust parameters and choose specific system configurations according to the demands of an X-ray imaging experiment. We designed the system using generic abstractions, which allows us to apply both the model and the implementation to streamed data processing outside the field of X-ray imaging. Initial efforts have led to prototype applications for beam monitoring and high energy physics trigger systems.

Because the foundation of our compute model is a DAG, data must flow from a source to a sink and cannot follow loops. Feedback-controlled experiments, however, require that the processing result flows back to one of the processing nodes. We argue that this is not a general problem that can be solved on only one of the two levels. On the lower level, we can always model feedback by encapsulating the loop *inside* a single task. We use this approach to provide iterative ART methods which repeatedly update the tomographic volume until a certain quality criteria is reached. At this point, the next filter stage receives the reconstructed volume. On a higher control level, we can model feedback by either *restarting* the data flow once the result reaches a sink or *inserting* the processed results using an appropriate input filter stage. This is the primary approach that we use for the feedback-based workflows.

The library and wrapper approach that we chose to expose the functionality favors composition of small modules. This architecture follows the principles of the UNIX operating system [105] but may not use the entire optimization potential. As always, abstractions lead to overheads and specific solutions may yield better performance. In particular, a data streaming DSL with a restricted language scope may offer better optimization opportunities. On the other hand, we can always use our proposed system as a *backend* to a DSLS.

Data acquisition

Question 4 asked how a detector interface must be designed to enable generic and fast access to locally and remotely attached 2D pixel detectors. In section 4.1, we presented a generic concept to acquire streams of two-dimensional detector data with both low latencies and a high throughput. We are able to achieve all objectives, specifically we provide a generic interface that covers a dozen different detectors, allows zero-copy data transfers if supported by the vendor and is able to stream data from a remote detector machine to a local acquisition and processing machine.

We discussed the advantages and disadvantages of the monolithic approach chosen by ESRF's detector system LIMA. Instead of LIMA's "kitchen sink" approach that allows for comprehensive image manipulation and data storage, we exclusively focus on data acquisition and the detector properties. The processing and control aspects were moved to other sub systems that were also subject of this thesis. The strict separation of concerns lets us optimize the data acquisition process, improves the modularization of software components and ensures better maintenance.

From a purely functional point of view both approaches are nearly identical. Although there are superficial similarities, the integration of new detectors and detector functionality accounts for the majority of the work rather than new functionality in LIMA's and UCA's core. Adapter layers that bridge between LIMA and UCA can help library users to benefit from *both* projects. Moreover, a TANGO server interface that exposes detectors with a common interface can be a viable alternative to adapter layers and does not require deep API changes.

Control system

Question 5 asked for a control system architecture that supports X-ray imaging experiments that allow for high-throughput and fast feedback loops. The *Concert* control system introduced in section 4.2, is the glue component that integrates the acquisition and processing sub-systems in high-level workflow-oriented experiments. Using this systems, we could fulfill all objectives concerned with on-line and feedback-based control experiments. In particular, asynchronous device access allows for high throughput experiment automation and a coroutine-based workflow description eases development of tight feedback loops.

The field of experiment control systems is diverse and integration of data processing has been achieved to a reasonable degrees [4]. Nonetheless, we integrated heterogeneous high-performance compute systems for the first time into an experiment control system and enabled soft real-time data processing of experiment data which was no possible before. As with the data acquisition framework, our experiment control system superficially resembles ALBA's Sardana. Python integration and a strong focus on ease-of-use are the only similarities between Sardana and *Concert*. Unlike our system, Sardana itself is a *distributed* experiment control systems that maps the interaction of devices on the TANGO communication model. This approach prevents quick feedback and the design of on-line systems. On the contrary, our modular, asynchronous approach is a major step towards fast and flexible experiment control that enables novel experiment types.

High-level rapid prototyping

Question 6 asked how the interfaces between the low-level heterogeneous compute system and the higher-level control system look like. More specifically, in question 7 we asked how we can support users and operators who may not have deep GPU programming knowledge but still want to benefit from the available hardware resources. In chapter 5, we introduced a high-level translation system that simplifies the accessibility of heterogeneous compute systems. The system translates Python functions following stencil-type semantics into valid OpenCL kernel code and either returns that for further use in our heterogeneous compute system or executes it on the fly.

Using this translation system, the scientist can explore and rapidly prototype solutions applicable to the image processing tasks encountered at the beamline. Instead of using a DSL we use high-level Python decorators to annotate translatable functions. This eases integration with the other acquisition and processing components and allows the beamline operator to stay within a common development platform. Although the run-time system is not optimized for processing large data streams and only incorporates execution on multiple GPUS, we attain a significant speed up of up to three orders of magnitude compared to the native execution in Python. This allows the users to benefit from the hardware resources available at the beamline without having to use lower-level programming languages and APIS. Because we focused on the needs of beamline scientists and operators we had to make compromises concerning the generality. Especially the stencil computation semantics of the translation step cannot cover all data processing needs.

7.2 Future work

Although, we made significant advances towards a fully parallel on-line X-ray imaging setup, we had to leave out post-processing steps for the sake of the scope of this thesis. In particular, we concentrated on the "front-end" parts of the parallel imaging pipeline including data acquisition, data processing and experiment control, yet storage and meta data handling as well as unusual hardware architectures are equally important.

The insights and results gathered in this thesis pave the way for high-level extensions such as the UFO follow-up project ASTOR.[1] The Arthropod Structure revealed by ultra-fast Tomography and Online Reconstruction (ASTOR) project will continue with the segmentation of the reconstructed volumes in a zoological context, clarify long-term storage options and develop end-user management and visualization concepts.

Storage

To write the raw data stream onto disks, we currently use a hierarchy of storage systems. The first level consists of large main memory with up to 512 GB of storage capacity. After the user decides which data is stored, the system moves it from the volatile main memory to a RAID-0 system of four SSDS. The SSDS provide a combined write throughput of up to $1.3\,\mathrm{GB\,s^{-1}}$ but have only a limited capacity of 250 GB. We use a RAID-6 array of regular hard disks with a total capacity of 20 TB to store data sequences of the same experiment data. The final level of the storage hierarchy is data archival at the Large Scale Data Facility (LSDF) [118].

[1] ASTOR is the *Verbundprojekt* for *Arthropod Structure revealed by ultra-fast Tomography and Online Reconstruction.*

Given our current acquisition and reconstruction system, the write performance suffices to store the raw and reconstructed data of an on-line tomography experiment. Multiple output streams, however, exceed the available data storage throughput which is not unlikely if any intermediate data must be stored alongside the final result. Hence, more work towards streaming-aware file systems or file system layers is necessary to replace our naïve storage solution.

Our current data storage approach writes flat files in pre-determined directory hierarchies or as grouped data in Hierarchical Data Format 5 (HDF5) files according to the NeXus specification [72]. We are able to store setup-related meta data alongside the acquired data but miss the integration of *user*-related meta data and do not follow a specified structure. This means that the beamline operator has to associate user and pre-acquired sample meta data (for example sample species or material) with the acquired and processed data in a separate, manual step. Moreover, he is responsible to move data by hand from the local disk storage to long-term storage systems such as the LSDF. For future systems, we need full integration with meta data providers and automatic archival for long-term storage.

Visualization

Besides writing results to disk, giving the beamline operator immediate feedback on the qualitative results has been one of this thesis' objectives. Using our data processing framework, the user can preview the reconstructed 2D slices and a static 3D projection of the whole volume. The 3D visualization method additively reprojects the volume [51], an algorithm implemented on GPUS still lacks the performance for real-time analysis. Furthermore, the visualization does not provide segmentation necessary for a comprehensive assessment and analysis. For full in-depth analysis, the user employs professional post-processing 3D analysis tools such as Amira[2] that still require manual copying of the reconstructed slices.

In the future, we will integrate 3D real-time graphics API such as OpenGL as part of the experiment control system to interact with the reconstructed volume on the fly. Additionally, we will continuously update the volumetric representation during the acquisition and reconstruction to give the user a three-dimensional impression of the acquired data. Real-time segmentation tasks to discriminate background from the sample and partitioning surfaces [85] will also be a part of this topic. The ASTOR project will provide a virtualized analysis architecture to ease transition and visualization of data. The remotely accessible system will multiplex software licenses and visualization hardware to multiple users who do not have to have physical access to a high powered workstation. The main benefits of this approach will be in reduced costs for the users and better utilization of available software licenses and hardware resources.

[2] `http://www.vsg3d.com/amira`

Architectural heterogeneity

Although, we designed the system architecture for general heterogeneous compute systems, we validated its characteristics solely on current CPU and GPU architectures. Due to the emerge of a large mobile market however, high-powered embedded system-on-chip GPU architectures such as ARM's recent Mali GPU architecture *Midgard* appeared. While not as powerful as desktop or server GPUS, mobile processors require less power and provide new architectural features such as shared memory caches between CPU and GPU. Shared caches are the first steps towards a unified address space and change the way a high-performance heterogeneous compute system can be implemented.

Sustainability

We have proven that the processing and control concepts work as we envisioned them. Nevertheless, user operation at the IMAGE beamline has not started yet and we cannot estimate the long-term effects and benefits for the entire operation. Continuous re-evaluations and adaptations of the experiment workflows are and will be necessary to cope with the demands of different experiment environments. As of now, only basic feedback-controlled experiments were conducted and part of the ASTOR project will be extensive use and research of experiment methodologies that make use of our fast experiment feedback loops.

8 Conclusion

Massively parallel multi-core architectures are the predominant features of current processors and accelerators. These architectures permit to scale the compute performance despite the clock frequency stagnation of single-core architectures. The parallel compute paradigm is the backend for the X-ray imaging experiment pipeline developed in this thesis. This pipeline consists of data acquisition, data processing and control phases. A tight integration of the low-level components in workflow-oriented pipelines helped to ensure low latencies and high throughput and for the first time permits the development of flexible, soft real-time on-line experiments. The core contributions of this thesis are the design and evaluation of the low-level components leveraging arbitrary heterogeneous compute and control systems.

High-throughput data acquisition

Any X-ray imaging experiment starts with the acquisition of large volumes of image data. For this task, we designed the flexible low-level UCA detector interface for high throughput purposes. We use a zero-copy approach to keep latencies low and retain the native bandwidth provided by the detector. Our generic detector model supports different kinds of synchronous and blocking readout modes and emulates a mode in software if it is not applicable to the hardware device. On top of the interface we built a generic InfiniBand-based RDMA server that allows us to stream frames to a remote acquisition machine at up to 29 Gbit s^{-1} and covering the data rates of contemporary streaming detectors. We wrapped the RDMA remote client behind the generic UCA interface, thus a user can control a remotely attached detector in the same way as a locally available one.

Heterogeneous computing

After the acquisition, the data is pre-processed and reconstructed on-the-fly. The reconstructed data can give both the beamline operator as well as the experiment control system itself input how to influence the experiment. To cover all aspects of X-ray imaging on parallel compute architectures, we modelled and developed a heterogeneous system architecture for streaming data. Although, the system is designed to support arbitrary heterogeneous system topologies, we focused on single and multi GPU systems as well as clusters with multi GPU nodes. The main scientific contribution is a flexible scheduling and mapping approach that adaptively duplicates and assigns tasks to the available hardware infrastructure. We carried out optimization strategies from low-level inter-node data transfers to high-level algorithmic adaptations and kernel fusion techniques. We used an holistic parallel processing approach and exploited parallelism from the instruction to the cluster level. This approach enables lowest possible latencies and highest acquisition, processing and experiment throughputs.

High-performance reconstruction

Although, the base system architecture is designed for data stream processing, it does not enforce particular applications. Nevertheless, our main goal in this thesis is the integration of fast X-ray image processing such as tomography and laminography for fast feedback experiments. Thus, we implemented the major reconstruction algorithms such as FBP, DFI and ART derivatives to give the user the possibility to choose between fast or high quality reconstructions. Using the optimized system enables tomographic reconstruction throughputs between 200 MB s^{-1} and 3 GB s^{-1} depending on the desired reconstruction quality.

Experiment control system

The user can store the resulting data of the processing as it is or designate it for use as a control variable in the on-going experiment loop. Using the experiment results for control decisions is necessary for novel dynamic experiment types that adapt the acquisition or the experiment setup but also to provide feedback about the data quality to the operator. To use the processed results in an efficient and user-friendly way, we designed a new high-level experiment control system. The main novelty is its complete asynchronous control model wrapped by a high-level coroutine abstraction. These mechanisms enable parallel control to improve the experiment throughput and use of the processing systems.

Rapid prototyping

We designed each aforementioned component that do one thing only and connected these sub systems together on a higher level. From a developer perspective this simplifies usage, extensibility and integration within existing tools and allows for re-usability. To ease the burden of writing GPU code for the end user, we designed a source-to-source translator that parses ordinary Python functions and emits OpenCL kernel code. To test the algorithms, we provide a JIT compilation facility that translates the code on the fly and executes it transparently on the GPU. Our results have shown linear scalability using multiple GPUs as well as a speed up of three orders of magnitude using a tomographic reconstruction task and compared to the native NumPy implementation.

Bibliography

[1] A. Alexandrov, M. F. Ionescu, K. E. Schauser, and C. Scheiman. “LogGP: Incorporating Long Messages into the LogP Model – One Step Closer Towards a Realistic Model for Parallel Computation”. In: *Proceedings of the Seventh Annual ACM Symposium on Parallel Algorithms and Architectures*. SPAA ’95. New York, NY, USA: ACM Press, 1995, pp. 95–105. ISBN: 0-89791-717-0. DOI: 10.1145/215399.215427.

[2] J. Als-Nielsen and D. McMorrow. *Elements of modern X-ray physics*. 2nd ed. John Wiley & Sons, 2011.

[3] G. M. Amdahl. “Validity of the Single Processor Approach to Achieving Large Scale Computing Capabilities”. In: *Proceedings of the April 18-20, 1967, Spring Joint Computer Conference*. AFIPS ’67 (Spring). New York, NY, USA: ACM Press, 1967, pp. 483–485. DOI: 10.1145/1465482.1465560.

[4] S. Arzt, A. Beteva, F. Cipriani, S. Delageniere, F. Felisaz, G. Förstner, E. Gordon, L. Launer, B. Lavault, G. Leonard, T. Mairs, A. McCarthy, J. McCarthy, S. McSweeney, J. Meyer, E. Mitchell, S. Monaco, D. Nurizzo, R. Ravelli, V. Rey, W. Shepard, D. Spruce, O. Svensson, and P. Theveneau. “Automation of macromolecular crystallography beamlines”. In: *Progress in Biophysics and Molecular Biology* 89.2 (2005), pp. 124–152. ISSN: 0079-6107. DOI: 10.1016/j.pbiomolbio.2004.09.003.

[5] K. Asanovic, R. Bodik, B. C. Catanzaro, J. J. Gebis, P. Husbands, K. Keutzer, D. A. Patterson, W. L. Plishker, J. Shalf, S. W. Williams, et al. *The landscape of parallel computing research: A view from Berkeley*. UCB/EECS-2006-183. EECS Department, University of California, Berkeley, 2006.

[6] K. Batenburg and J. Sijbers. “DART: A Practical Reconstruction Algorithm for Discrete Tomography”. In: *Image Processing, IEEE Transactions on* 20.9 (Sept. 2011), pp. 2542–2553. ISSN: 1057-7149. DOI: 10.1109/TIP.2011.2131661.

[7] K. Belkhale and P. Banerjee. “A scheduling algorithm for parallelizable dependent tasks”. In: *Parallel Processing Symposium, 1991. Proceedings., Fifth International*. Apr. 1991, pp. 500–506. DOI: 10.1109/IPPS.1991.153827.

[8] E. Bendersky. *pycparser*. https://github.com/eliben/pycparser.

[9] H. Berger. *Automatisieren mit SIMATIC : Controller, Software, Programmierung, Datenkommunikation, Bedienen und Beobachten*. 5th ed. Erlangen: PUBLICIS Publishing, 2012. ISBN: 978-3-89578-386-9.

[10] G. D. Bergland. "Numerical Analysis: A Fast Fourier Transform Algorithm for Real-valued Series". In: *Commun. ACM* 11.10 (Oct. 1968), pp. 703–710. ISSN: 0001-0782. DOI: 10.1145/364096.364118.

[11] Y. Blanter and M. Büttiker. "Shot noise in mesoscopic conductors". In: *Physics Reports* 336.1–2 (2000), pp. 1–166. ISSN: 0370-1573. DOI: http://dx.doi.org/10.1016/S0370-1573(99)00123-4.

[12] A. Boni, A. Pierazzi, and D. Vecchi. "LVDS I/O interface for Gb/s-per-pin operation in 0.35- mu;m CMOS". In: *Solid-State Circuits, IEEE Journal of* 36.4 (Apr. 2001), pp. 706–711. ISSN: 0018-9200. DOI: 10.1109/4.913751.

[13] M. Boyer, J. Meng, and K. Kumaran. "Improving GPU Performance Prediction with Data Transfer Modeling". In: *2013 IEEE International Symposium on Parallel & Distributed Processing, Workshops and Phd Forum* (2013), pp. 1097–1106. DOI: 10.1109/IPDPSW.2013.236.

[14] T. D. Braun, H. J. Siegel, N. Beck, L. L. Bölöni, M. Maheswaran, A. I. Reuther, J. P. Robertson, M. D. Theys, B. Yao, D. Hensgen, and R. F. Freund. "A Comparison of Eleven Static Heuristics for Mapping a Class of Independent Tasks onto Heterogeneous Distributed Computing Systems". In: *Journal of Parallel and Distributed Computing* 61.6 (2001), pp. 810–837. ISSN: 0743-7315. DOI: http://dx.doi.org/10.1006/jpdc.2000.1714.

[15] A. Brogna, M. Balzer, S. Smale, J. Hartmann, D. Bormann, E. Hamann, A. Cecilia, M. Zuber, T. Koenig, A. Zwerger, et al. "A fast embedded readout system for large-area Medipix and Timepix systems". In: *Journal of Instrumentation* 9.05 (2014).

[16] A. Buades, B. Coll, and J.-M. Morel. "A Non-Local Algorithm for Image Denoising". In: *Proceedings of the 2005 IEEE Computer Society Conference on Computer Vision and Pattern Recognition (CVPR'05)*. Vol. 2. Washington, DC, USA: IEEE Computer Society, June 2005, pp. 60–65. ISBN: 0-7695-2372-2. DOI: 10.1109/CVPR.2005.38.

[17] R. Buchty, V. Heuveline, W. Karl, and J.-P. Weiss. "A survey on hardware-aware and heterogeneous computing on multicore processors and accelerators". In: *Concurrency and Computation: Practice and Experience* 24.7 (2012), pp. 663–675. ISSN: 1532-0634. DOI: 10.1002/cpe.1904.

[18] I. Buck, T. Foley, D. Horn, J. Sugerman, K. Fatahalian, M. Houston, and P. Hanrahan. “Brook for GPUs: Stream Computing on Graphics Hardware”. In: *ACM Trans. Graph.* 23.3 (Aug. 2004), pp. 777–786. ISSN: 0730-0301. DOI: 10 . 1145 / 1015706.1015800.

[19] D. Buttlar and J. Farrell. *Pthreads programming: A POSIX standard for better multiprocessing.* ” O’Reilly Media, Inc.”, 1996.

[20] M. Caselle, M. Brosi, S. Chilingaryan, T. Dritschler, N. Hiller, V. Judin, A. Kopmann, A.-S. Müller, J. Raasch, L. Rota, L. Petzold, N. Smale, J. Steinmann, M. Vogelgesang, S. Wuensch, M. Siegel, and M. Weber. “Picosecond Sampling Electronics “Capture” for Terahertz Synchrotron Radiation”. In: *Proceedings of the 3rd International Beam Instrumentation Conference.* IBIC ’14. (to appear). Monterey, USA, 2014.

[21] M. Caselle, S. Chilingaryan, A. Herth, A. Kopmann, U. Stevanovic, M. Vogelgesang, M. Balzer, and M. Weber. “Ultra-Fast Streaming Camera Platform for Scientific Applications”. In: *IEEE Transactions on Nuclear Science* (2012).

[22] B. Catanzaro, M. Garland, and K. Keutzer. “Copperhead: compiling an embedded data parallel language”. In: *Proceedings of the 16th ACM symposium on Principles and practice of parallel programming.* ACM. 2011, pp. 47–56.

[23] R.-C. Chen, D. Dreossi, L. Mancini, R. Menk, L. Rigon, T.-Q. Xiao, and R. Longo. “*PITRE*: software for phase-sensitive X-ray image processing and tomography reconstruction”. In: *Journal of Synchrotron Radiation* 19.5 (Sept. 2012), pp. 836–845. DOI: 10.1107/S0909049512029731.

[24] S. Chilingaryan, M. Balzer, M. Caselle, T. Rolo, T. Dritschler, T. Farago, A. Kopmann, U. Stevanovic, T. van de Kamp, M. Vogelgesang, V. Asadchikov, T. Baumbach, A. Myagotin, S. Tsapko, and W. M. “UFO – Status and Perspectives of Ultrafast X-ray Imaging at ANKA”. In: *Deutsche Tagung für Forschung mit Synchrotronstrahlung, Neutronen und Ionenstrahlen an Großgeräten.* SNI ’14. (to appear). Bonn, Germany, 2014.

[25] T. H. Cormen, C. Stein, R. L. Rivest, and C. E. Leiserson. *Introduction to Algorithms.* 2nd ed. McGraw-Hill Higher Education, 2001. ISBN: 0070131511.

[26] D. Culler, R. Karp, D. Patterson, A. Sahay, K. E. Schauser, E. Santos, R. Subramonian, and T. von Eicken. “LogP: Towards a Realistic Model of Parallel Computation”. In: *SIGPLAN Not.* 28.7 (June 1993), pp. 1–12. ISSN: 0362-1340. DOI: 10.1145/173284.155333.

[27] L. R. Dalesio, M. Kraimer, and A. Kozubal. “EPICS architecture”. In: *ICALEPCS.* Vol. 91. 1991, pp. 92–15.

[28] A. Danilewsky, J. Wittge, K. Kiefl, D. Allen, P. McNally, J. Garagorri, M. R. Elizalde, T. Baumbach, and B. K. Tanner. “Crack propagation and fracture in silicon wafers under thermal stress”. In: *Journal of Applied Crystallography* 46.4 (Aug. 2013), pp. 849–855. DOI: 10.1107/S0021889813003695.

[29] A. Danowitz, K. Kelley, J. Mao, J. P. Stevenson, and M. Horowitz. “CPU DB: Recording Microprocessor History”. In: *Commun. ACM* 55.4 (Apr. 2012), pp. 55–63. ISSN: 0001-0782. DOI: 10.1145/2133806.2133822.

[30] S. Darbha and D. Agrawal. “SDBS: a task duplication based optimal scheduling algorithm”. In: *Scalable High-Performance Computing Conference, 1994., Proceedings of the.* May 1994, pp. 756–763. DOI: 10.1109/SHPCC.1994.296717.

[31] F. De Carlo and B. Tieman. “High-throughput x-ray microtomography system at the Advanced Photon Source beamline 2-BM”. In: vol. 5535. 2004, pp. 644–651. DOI: 10.1117/12.559223.

[32] J. Dean and S. Ghemawat. “MapReduce: Simplified Data Processing on Large Clusters”. In: *Commun. ACM* 51.1 (Jan. 2008), pp. 107–113. ISSN: 0001-0782. DOI: 10.1145/1327452.1327492.

[33] R. Diestel. *Graph Theory {Graduate Texts in Mathematics; 173}*. Munich, Germany: Springer-Verlag Berlin Heidelberg, 2005.

[34] T. Donath, F. Beckmann, and A. Schreyer. “Automated determination of the center of rotation in tomography data”. In: *J. Opt. Soc. Am. A* 23.5 (May 2006), pp. 1048–1057. DOI: 10.1364/JOSAA.23.001048. URL: http://josaa.osa.org/abstract.cfm?URI=josaa-23-5-1048.

[35] P. C. Donoghue, S. Bengtson, X.-p. Dong, N. J. Gostling, T. Huldtgren, J. A. Cunningham, C. Yin, Z. Yue, F. Peng, and M. Stampanoni. “Synchrotron X-ray tomographic microscopy of fossil embryos”. In: *Nature* 442.7103 (2006), pp. 680–683.

[36] T. Dritschler, M. Vogelgesang, A. Kopmann, and T. Farago. “Using InfiniBand for High-throughput Data Acquisition in a TANGO Environment”. In: *Proceedings of the 10th International Workshop on Personal Computers and Particle Accelerators.* PCAPAC ’14. to appear. 2014.

[37] P. Du, R. Weber, P. Luszczek, S. Tomov, G. Peterson, and J. Dongarra. *From CUDA to OpenCL: Towards a Performance-portable Solution for Multi-platform GPU Programming.* Tech. rep. Electrical Engineering and Computer Science Department, University of Tennessee. LAPACK Working Note 228, 2010.

[38] C. Dubach, P. Cheng, R. Rabbah, D. F. Bacon, and S. J. Fink. “Compiling a High-level Language for GPUs: (via Language Support for Architectures and Compilers)”. In: *SIGPLAN Not.* 47.6 (June 2012), pp. 1–12. ISSN: 0362-1340. DOI: 10.1145/2345156.2254066.

[39] R. Duncan. "A survey of parallel computer architectures". In: *Computer* 23.2 (Feb. 1990), pp. 5–16. ISSN: 0018-9162. DOI: 10.1109/2.44900.

[40] D. Eppstein and Z. Galil. "Parallel Algorithmic Techniques For Combinational Computation". In: *Annual Review of Computer Science* 3.1 (1988), pp. 233–283. DOI: 10.1146/annurev.cs.03.060188.001313.

[41] W. Feng, P. Balaji, C. Baron, L. Bhuyan, and D. Panda. "Performance characterization of a 10-Gigabit Ethernet TOE". In: *High Performance Interconnects, 2005. Proceedings. 13th Symposium on.* Aug. 2005, pp. 58–63. DOI: 10.1109/CONECT.2005.30.

[42] D. Fernandez-Baca. "Allocating Modules to Processors in a Distributed System". In: *IEEE Trans. Softw. Eng.* 15.11 (Nov. 1989), pp. 1427–1436. ISSN: 0098-5589. DOI: 10.1109/32.41334.

[43] M. Flynn. "Some Computer Organizations and Their Effectiveness". In: *Computers, IEEE Transactions on* C-21.9 (Sept. 1972), pp. 948–960. ISSN: 0018-9340. DOI: 10.1109/TC.1972.5009071.

[44] M. Frigo and S. Johnson. "The Design and Implementation of FFTW3". In: *Proceedings of the IEEE* 93.2 (Feb. 2005), pp. 216–231. ISSN: 0018-9219. DOI: 10.1109/JPROC.2004.840301.

[45] J. Gabadinho, A. Beteva, M. Guijarro, V. Rey-Bakaikoa, D. Spruce, M. W. Bowler, S. Brockhauser, D. Flot, E. J. Gordon, D. R. Hall, B. Lavault, A. A. McCarthy, J. McCarthy, E. Mitchell, S. Monaco, C. Mueller-Dieckmann, D. Nurizzo, R. B. G. Ravelli, X. Thibault, M. A. Walsh, G. A. Leonard, and S. M. McSweeney. "MxCuBE: a synchrotron beamline control environment customized for macromolecular crystallography experiments". In: *Journal of Synchrotron Radiation* 17.5 (Sept. 2010), pp. 700–707. DOI: 10.1107/S0909049510020005.

[46] B. Gaster, ed. *Heterogeneous computing with OpenCL.* Amsterdam: Elsevier, Morgan Kaufmann, 2012. ISBN: 978-0-12-387766-6; 0-12-387766-0.

[47] M. I. Gordon, W. Thies, and S. Amarasinghe. "Exploiting Coarse-grained Task, Data, and Pipeline Parallelism in Stream Programs". In: *SIGARCH Comput. Archit. News* 34.5 (Oct. 2006), pp. 151–162. ISSN: 0163-5964. DOI: 10.1145/1168919.1168877.

[48] A. Götz, E. Taurel, J. Pons, P. Verdier, J. Chaize, J. Meyer, F. Poncet, G. Heunen, E. Götz, A. Buteau, et al. "TANGO a CORBA based Control System". In: *Proceedings of the 9th International Conference on Accelerator and Large Experimental Physics Control Systems.* ICALEPCS '03. Oct. 2003, pp. 220–222.

[49] D. Gürsoy, F. De Carlo, X. Xiao, and C. Jacobsen. "TomoPy: A framework for the analysis of synchrotron tomographic data". In: *Journal of Synchrotron Radiation* 21 (2014). DOI: 10.1107/S1600577514013939.

[50] J. L. Gustafson. “Reevaluating Amdahl’s Law”. In: *Commun. ACM* 31.5 (May 1988), pp. 532–533. ISSN: 0001-0782. DOI: 10.1145/42411.42415.

[51] L. D. Harris, R. A. Robb, T. S. Yuen, and E. L. Ritman. “Noninvasive Numerical Dissection And Display Of Anatomic Structure Using Computerized X-Ray Tomography”. In: vol. 152. 1978, pp. 10–18. DOI: 10.1117/12.938188.

[52] O. Hensler and K. Rehlich. “DOOCS: A distributed object oriented control system”. In: *Proceedings of XV Workshop on Charged Particle Accelerators, Protvino.* 1996.

[53] C. Hewitt, P. Bishop, and R. Steiger. “A Universal Modular ACTOR Formalism for Artificial Intelligence”. In: *Proceedings of the 3rd International Joint Conference on Artificial Intelligence.* IJCAI’73. San Francisco, CA, USA: Morgan Kaufmann Publishers Inc., 1973, pp. 235–245. URL: http://dl.acm.org/citation.cfm?id=1624775.1624804.

[54] P. Hintjens. *ZeroMQ: Messaging for Many Applications.* ” O’Reilly Media, Inc.”, 2013.

[55] R. W. Hockney. “The Communication Challenge for MPP: Intel Paragon and Meiko CS-2”. In: *Parallel Comput.* 20.3 (Mar. 1994), pp. 389–398. ISSN: 0167-8191. DOI: 10.1016/S0167-8191(06)80021-9.

[56] R. Hofmann, J. Moosmann, and T. Baumbach. “Criticality in single-distance phase retrieval”. In: *Opt. Express* 19.27 (Dec. 2011), pp. 25881–25890. DOI: 10.1364/OE.19.025881.

[57] P. Hölzenspies, G. J. M. Smit, and J. Kuper. “Mapping streaming applications on a reconfigurable MPSoC platform at run-time”. In: *System-on-Chip, 2007 International Symposium on.* Nov. 2007, pp. 1–4. DOI: 10.1109/ISSOC.2007.4427443.

[58] A. Homs, S. Petitdemange, L. Claustre, R. Homs, and A. Kirov. “LIMA: Acquiring data with imaging detectors”. In: *Proceedings of the 13th International Conference on Accelerator & Large Experimental Physics Control Systems.* ICALEPCS ’11. 2011.

[59] A. H. Hormati, M. Samadi, M. Woh, T. Mudge, and S. Mahlke. “Sponge: Portable Stream Programming on Graphics Engines”. In: *SIGARCH Comput. Archit. News* 39 (1 Mar. 2011), pp. 381–392. ISSN: 0163-5964. DOI: http://doi.acm.org/10.1145/1961295.1950409.

[60] J. Howard, S. Dighe, Y. Hoskote, S. Vangal, D. Finan, G. Ruhl, D. Jenkins, H. Wilson, N. Borkar, G. Schrom, F. Pailet, S. Jain, T. Jacob, S. Yada, S. Marella, P. Salihundam, V. Erraguntla, M. Konow, M. Riepen, G. Droege, J. Lindemann, M. Gries, T. Apel, K. Henriss, T. Lund-Larsen, S. Steibl, S. Borkar, V. De, R. Van Der Wijngaart, and T. Mattson. “A 48-Core IA-32 message-passing processor

with DVFS in 45nm CMOS". In: *Solid-State Circuits Conference Digest of Technical Papers (ISSCC), 2010 IEEE International*. Feb. 2010, pp. 108–109. DOI: 10.1109/ISSCC.2010.5434077.

[61] J. Hurwitz and W.-C. Feng. "End-to-end performance of 10-gigabit Ethernet on commodity systems". In: *Micro, IEEE* 24.1 (2004), pp. 10–22.

[62] J. Jaros, B. E. Treeby, and A. P. Rendell. "Use of Multiple GPUs on Shared Memory Multiprocessors for Ultrasound Propagation Simulations". In: *Proceedings of the Tenth Australasian Symposium on Parallel and Distributed Computing - Volume 127*. AusPDC '12. Melbourne, Australia: Australian Computer Society, Inc., 2012, pp. 43–52. ISBN: 978-1-921770-08-1. URL: http://dl.acm.org/citation.cfm?id=2523685.2523691.

[63] M. Juric, I. Rozman, and M. Hericko. "Performance comparison of CORBA and RMI". In: *Information and Software Technology* 42.13 (2000), pp. 915–933. ISSN: 0950-5849. DOI: http://dx.doi.org/10.1016/S0950-5849(00)00128-2.

[64] A. B. Kahn. "Topological Sorting of Large Networks". In: *Commun. ACM* 5.11 (Nov. 1962), pp. 558–562. ISSN: 0001-0782. DOI: 10.1145/368996.369025.

[65] A. C. Kak and M. Slaney. *Principles of Computerized Tomographic Imaging*. New York, NY, USA: IEEE Press, 1988. ISBN: 9780879421984.

[66] T. van de Kamp, P. Vagovič, T. Baumbach, and A. Riedel. "A biological screw in a beetle's leg". In: *Science* 333.6038 (2011), pp. 52–52.

[67] R. M. Karp. "Reducibility among Combinatorial Problems". In: *Complexity of Computer Computations*. Ed. by R. Miller, J. Thatcher, and J. Bohlinger. The IBM Research Symposia Series. Springer US, 1972, pp. 85–103. ISBN: 978-1-4684-2003-6. DOI: 10.1007/978-1-4684-2001-2_9.

[68] A. A. Khokhar, V. K. Prasanna, M. E. Shaaban, and C.-L. Wang. "Heterogeneous computing: Challenges and opportunities". In: *IEEE Computer* 26.6 (1993), pp. 18–27.

[69] I. Khokhriakov, L. Lottermoser, R. Gehrke, T. Kracht, E. Wintersberger, A. Kopmann, M. Vogelgesang, and F. Beckmann. "Integrated Control System Environment for High-Throughput Tomography". In: *Proceedings of SPIE: Developments in X-Ray Tomography IX*. Vol. 9212. Sept. 2014. DOI: 10.1117/12.2060975.

[70] N. Kim, T. Austin, D. Baauw, T. Mudge, K. Flautner, J. Hu, M. Irwin, M. Kandemir, and V. Narayanan. "Leakage current: Moore's law meets static power". In: *Computer* 36.12 (Dec. 2003), pp. 68–75. ISSN: 0018-9162. DOI: 10.1109/MC.2003.1250885.

[71] A. Klöckner, N. Pinto, Y. Lee, B. Catanzaro, P. Ivanov, and A. Fasih. "PyCUDA and PyOpenCL: A scripting-based approach to GPU run-time code generation". In: *Parallel Computing* 38.3 (2012), pp. 157–174.

[72] P. Klosowski, M. Koennecke, J. Tischler, and R. Osborn. “NeXus: A common format for the exchange of neutron and synchroton data”. In: *Physica B: Condensed Matter* 241 (1997), pp. 151–153. DOI: 10.1016/S0921-4526(97)00865-X.

[73] M. Koop, W. Huang, K. Gopalakrishnan, and D. Panda. “Performance Analysis and Evaluation of PCIe 2.0 and Quad-Data Rate InfiniBand”. In: *High Performance Interconnects, 2008. HOTI ’08. 16th IEEE Symposium on.* Aug. 2008, pp. 85–92. DOI: 10.1109/HOTI.2008.26.

[74] D. Koufaty and D. T. Marr. “Hyperthreading Technology in the NetBurst Microarchitecture”. In: *Micro, IEEE* 23.2 (2003), pp. 56–65.

[75] Y.-K. Kwok and I. Ahmad. “Static Scheduling Algorithms for Allocating Directed Task Graphs to Multiprocessors”. In: *ACM Comput. Surv.* 31.4 (Dec. 1999), pp. 406–471. ISSN: 0360-0300. DOI: 10.1145/344588.344618.

[76] A. Lastovetsky and J. Dongarra. *High performance heterogeneous computing*. Wiley series on parallel and distributed computing. Hoboken, NJ: Wiley, 2009. ISBN: 978-0-470-04039-3.

[77] S. Lee, S.-J. Min, and R. Eigenmann. “OpenMP to GPGPU: A Compiler Framework for Automatic Translation and Optimization”. In: *SIGPLAN Not.* 44.4 (Feb. 2009), pp. 101–110. ISSN: 0362-1340. DOI: 10.1145/1594835.1504194.

[78] V. W. Lee, C. Kim, J. Chhugani, M. Deisher, D. Kim, A. D. Nguyen, N. Satish, M. Smelyanskiy, S. Chennupaty, P. Hammarlund, R. Singhal, and P. Dubey. “Debunking the 100X GPU vs. CPU Myth: An Evaluation of Throughput Computing on CPU and GPU”. In: *SIGARCH Comput. Archit. News* 38.3 (June 2010), pp. 451–460. ISSN: 0163-5964. DOI: 10.1145/1816038.1816021.

[79] R. Levin, E. Cohen, W. Corwin, F. Pollack, and W. Wulf. “Policy/Mechanism Separation in Hydra”. In: *SIGOPS Oper. Syst. Rev.* 9.5 (Nov. 1975), pp. 132–140. ISSN: 0163-5980. DOI: 10.1145/1067629.806531.

[80] E. Lindholm, J. Nickolls, S. Oberman, and J. Montrym. “NVIDIA Tesla: A Unified Graphics and Computing Architecture”. In: *Micro, IEEE* 28.2 (Mar. 2008), pp. 39–55. ISSN: 0272-1732. DOI: 10.1109/MM.2008.31.

[81] X. Llopart, M. Campbell, R. Dinapoli, D. San Segundo, and E. Pernigotti. “Medipix2: a 64-k pixel readout chip with 55-μm square elements working in single photon counting mode”. In: *Nuclear Science, IEEE Transactions on* 49.5 (2002), pp. 2279–2283.

[82] P. Lytaev, A.-C. Hipp, L. Lottermoser, J. Herzen, I. Greving, I. Khokhriakov, S. Meyer-Loges, J. Plewka, J. Burmester, M. Caselle, M. Vogelgesang, S. Chilingaryan, A. Kopmann, and M. Balzer. "Characterization of CCD and CMOS Cameras for Grating-based Phase-contrast Tomography". In: *Proceedings of SPIE: Developments in X-Ray Tomography IX*. Vol. 9212. Sept. 2014. DOI: 10.1117/12.2061389.

[83] K. Mader, F. Marone, C. Hintermüller, G. Mikuljan, A. Isenegger, and M. Stampanoni. "High-throughput full-automatic synchrotron-based tomographic microscopy". In: *Journal of Synchrotron Radiation* 18.2 (Mar. 2011), pp. 117–124. DOI: 10.1107/S0909049510047370.

[84] A. Maier, H. G. Hofmann, M. Berger, P. Fischer, C. Schwemmer, H. Wu, K. Müller, J. Hornegger, J.-H. Choi, C. Riess, A. Keil, and R. Fahrig. "CONRAD – A software framework for cone-beam imaging in radiology". In: *Medical Physics* 40.11" (2013). DOI: http://dx.doi.org/10.1118/1.4824926.

[85] A. Mangan and R. Whitaker. "Partitioning 3D surface meshes using watershed segmentation". In: *Visualization and Computer Graphics, IEEE Transactions on* 5.4 (Oct. 1999), pp. 308–321. ISSN: 1077-2626. DOI: 10.1109/2945.817348.

[86] W. R. Mark, R. S. Glanville, K. Akeley, and M. J. Kilgard. "Cg: A System for Programming Graphics Hardware in a C-like Language". In: *ACM Trans. Graph.* 22.3 (June 2003), pp. 896–907. ISSN: 0730-0301. DOI: 10.1145/882262.882362.

[87] R. Membarth, F. Hannig, J. Teich, M. Körner, and W. Eckert. "Generating Device-specific GPU Code for Local Operators in Medical Imaging". In: *Proceedings of the 26th IEEE International Parallel & Distributed Processing Symposium (IPDPS)*. IEEE. Shanghai, China, May 2012, pp. 569–581. DOI: 10.1109/IPDPS.2012.59.

[88] G. E. Moore et al. *Cramming more components onto integrated circuits*. 1965.

[89] O. Moreira. *Scheduling Real-Time Streaming Applications onto an Embedded Multiprocessor*. Ed. by H. Corporaal. Embedded Systems ; 24SpringerLink : Bücher. Cham: Springer, 2014. ISBN: 978-331-90124-6-9. DOI: 10.1007/978-3-319-01246-9.

[90] A. L. D. Moura and R. Ierusalimschy. "Revisiting Coroutines". In: *ACM Trans. Program. Lang. Syst.* 31.2 (Feb. 2009), 6:1–6:31. ISSN: 0164-0925. DOI: 10.1145/1462166.1462167.

[91] A. Myagotin, A. Voropaev, L. Helfen, D. Hanschke, and T. Baumbach. "Efficient Volume Reconstruction for Parallel-Beam Computed Laminography by Filtered Backprojection on Multi-Core Clusters". In: *Image Processing, IEEE Transactions on* 22.12 (Dec. 2013), pp. 5348–5361. ISSN: 1057-7149. DOI: 10.1109/TIP.2013.2285600.

[92] L. Neumeyer, B. Robbins, A. Nair, and A. Kesari. "S4: Distributed stream computing platform". In: *Proceedings of the 2010 IEEE International Conference on Data Mining Workshops*. ICDMW '10. Washington, DC, USA: IEEE Computer Society, 2010, pp. 170–177. ISBN: 978-0-7695-4257-7. DOI: 10.1109/ICDMW.2010.172.

[93] J. Nickolls and W. Dally. "The GPU Computing Era". In: *Micro, IEEE* 30.2 (Mar. 2010), pp. 56–69. ISSN: 0272-1732. DOI: 10.1109/MM.2010.41.

[94] NVIDIA. *NVIDIA OpenCL Best Practices Guide*. Tech. rep. NVIDIA, 2009.

[95] T. E. Oliphant. *A Guide to NumPy*. Vol. 1. Trelgol Publishing USA, 2006.

[96] J. Owens, M. Houston, D. Luebke, S. Green, J. Stone, and J. Phillips. "GPU Computing". In: *Proceedings of the IEEE* 96.5 (May 2008), pp. 879–899. ISSN: 0018-9219. DOI: 10.1109/JPROC.2008.917757.

[97] D. Padua. *Encyclopedia of Parallel Computing*. 1st ed. Springer US, 2011. ISBN: 978-0-387-09765-7.

[98] W. J. Palenstijn, K. J. Batenburg, and J. Sijbers. "Performance improvements for iterative electron tomography reconstruction using graphics processing units (GPUs)". In: *Journal of Structural Biology* 176 (Nov. 2011), pp. 250–253. DOI: 10.1016/j.jsb.2011.07.017.

[99] *Parallel Thread Execution ISA: Application Guide*. Version 4.1 (NVIDIA). 2014. URL: http://docs.nvidia.com/cuda/pdf/ptx_isa_4.1.pdf.

[100] G.-L. Park, B. Shirazi, and J. Marquis. "Comparative study of static scheduling with task duplication for distributed systems". In: *Solving Irregularly Structured Problems in Parallel*. Vol. 1253. Lecture Notes in Computer Science. Springer-Verlag Berlin Heidelberg, 1997, pp. 123–134. ISBN: 978-3-540-63138-5. DOI: 10.1007/3-540-63138-0_12.

[101] R. Prasad, C. Dovrolis, M. Murray, and K. Claffy. "Bandwidth estimation: Metrics, Measurement Techniques, and Tools". In: *Network, IEEE* 17.6 (Nov. 2003), pp. 27–35. ISSN: 0890-8044. DOI: 10.1109/MNET.2003.1248658.

[102] *Programming languages – C*. ISO/IEC 9899:1999 (ISO). 1999.

[103] A. Rack, T. Weitkamp, S. B. Trabelsi, P. Modregger, A. Cecilia, T. dos Santos Rolo, T. Rack, D. Haas, R. Simon, R. Heldele, M. Schulz, B. Mayzel, A. Danilewsky, T. Waterstradt, W. Diete, H. Riesemeier, B. Müller, and T. Baumbach. "The micro-imaging station of the TopoTomo beamline at the ANKA synchrotron light source". In: *Nuclear Instruments and Methods in Physics Research Section B: Beam Interactions with Materials and Atoms* 267.11 (2009), pp. 1978–1988. ISSN: 0168-583X. DOI: http://dx.doi.org/10.1016/j.nimb.2009.04.002.

[104] A. Rack, T. Weitkamp, S. Bauer Trabelsi, P. Modregger, A. Cecilia, T. dos Santos Rolo, T. Rack, D. Haas, R. Simon, R. Heldele, et al. "The micro-imaging station of the TopoTomo beamline at the ANKA synchrotron light source". In: *Nuclear Instruments and Methods in Physics Research Section B: Beam Interactions with Materials and Atoms* 267.11 (2009), pp. 1978–1988.

[105] O. Ritchie and K. Thompson. "The UNIX Time-Sharing System". In: *Bell System Technical Journal, The* 57.6 (July 1978), pp. 1905–1929. ISSN: 0005-8580. DOI: 10.1002/j.1538-7305.1978.tb02136.x.

[106] W. C. Röntgen. "Über eine neue Art von Strahlen". In: *Annalen der Physik* 300.1 (1898), pp. 1–11.

[107] B. Ryder. "Constructing the Call Graph of a Program". In: *Software Engineering, IEEE Transactions on* SE-5.3 (May 1979), pp. 216–226. ISSN: 0098-5589. DOI: 10.1109/TSE.1979.234183.

[108] T. dos Santos Rolo, A. Ershov, T. van de Kamp, and T. Baumbach. "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.

[109] R. Sedgewick and K. D. Wayne. *Algorithms.* 4th ed. Addison-Wesley Professional, 2011. ISBN: 9780321573513.

[110] L. Seiler, D. Carmean, E. Sprangle, T. Forsyth, M. Abrash, P. Dubey, S. Junkins, A. Lake, J. Sugerman, R. Cavin, R. Espasa, E. Grochowski, T. Juan, and P. Hanrahan. "Larrabee: A Many-core x86 Architecture for Visual Computing". In: *ACM Trans. Graph.* 27.3 (Aug. 2008), 18:1–18:15. ISSN: 0730-0301. DOI: 10.1145/1360612.1360617.

[111] A. Shkarin, S. Chilingaryan, A. Kopmann, and M. Vogelgesang. "UfoART: An Open Source GPU-accelerated Framework for Flexible Algebraic Reconstruction in X-ray CT". In: *8th Workshop on Tomography and Applications- Discrete Tomography and Image Reconstruction.* to appear. 2014.

[112] R. Shkarin, S. Chilingaryan, A. Kopmann, and M. Vogelgesang. "GPU-optimized Direct Fourier Method for On-line Tomography". In: *8th Workshop on Tomography and Applications- Discrete Tomography and Image Reconstruction.* to appear. 2014.

[113] M. Snir, S. Otto, S. Huss-Lederman, D. Walker, and J. Dongarra. *MPI – The Complete Reference, Volume 1: The MPI Core.* 2nd ed. Cambridge, MA, USA: MIT Press, 1998. ISBN: 0262692155.

[114] *Specifications of the Camera Link Interface Standard for Digital Cameras and Frame Grabbers.* Version 1.1 (Automated Imaging Association). 2004.

[115] P. K. Spiegel. "The first clinical X-ray made in America–100 years." In: *AJR. American journal of roentgenology* 164.1 (1995), pp. 241–243.

[116] H. Stark, J. Woods, I. Paul, and R. Hingorani. "Direct Fourier reconstruction in computer tomography". In: *Acoustics, Speech and Signal Processing, IEEE Transactions on* 29.2 (Apr. 1981), pp. 237–245. ISSN: 0096-3518. DOI: 10.1109/TASSP.1981.1163528.

[117] U. Stevanovic, M. Caselle, S. Chilingaryan, A. Herth, A. Kopmann, M. Vogelgesang, M. Balzer, and M. Weber. "High-Speed Camera with Embedded FPGA Processing". In: *Proceedings of the 2012 Conference on Design and Architectures for Signal and Image Processing*. DASIP '12. Oct. 2012.

[118] R. Stotzka, V. Hartmann, T. Jejkal, M. Sutter, J. van Wezel, M. Hardt, A. Garcia, R. Kupsch, and S. Bourov. "Perspective of the Large Scale Data Facility (LSDF) supporting nuclear fusion applications". In: *Parallel, Distributed and Network-Based Processing (PDP), 2011 19th Euromicro International Conference on*. IEEE. 2011, pp. 373–379.

[119] J. Subhlok, J. M. Stichnoth, D. R. O'hallaron, and T. Gross. "Exploiting task and data parallelism on a multicomputer". In: *ACM SIGPLAN Notices*. Vol. 28. 7. New York, NY, USA: ACM Press, 1993, pp. 13–22.

[120] H. Sutter. "The free lunch is over: A fundamental turn toward concurrency in software". In: *Dr. Dobb's Journal* 30.3 (2005), pp. 202–210.

[121] *The OpenCL Specification*. Version 1.2, Revision 19 (Khronos). 2012.

[122] *The TANGO Control System Manual*. Version 8.1 (ESRF). 2013. URL: http://www.esrf.eu/computing/cs/tango/tango_doc/kernel_doc/ds_prog/tango.html.

[123] W. Thies, M. Karczmarek, and S. Amarasinghe. "StreamIt: A Language for Streaming Applications". In: *International Conference on Compiler Construction*. Grenoble, France, Apr. 2002.

[124] M. O. Tokhi, M. A. Hossain, and M. H. Shaheed. *Parallel computing for real time signal processing and control*. Advanced textbooks in control and signal processing. London: Springer, 2003. ISBN: 1-85233-599-8.

[125] K. Uesugi, A. Takeuchi, and Y. Suzuki. "High-definition high-throughput microtomography at SPring-8". In: *Journal of Physics: Conference Series* 186.1 (2009). DOI: 10.1088/1742-6596/186/1/012050.

[126] J. Ullman. "NP-complete scheduling problems". In: *Journal of Computer and System Sciences* 10.3 (1975), pp. 384–393. ISSN: 0022-0000. DOI: http://dx.doi.org/10.1016/S0022-0000(75)80008-0.

[127] A. Van Deursen and P. Klint. "Little languages: Little maintenance?" In: *Journal of software maintenance* 10.2 (1998), pp. 75–92.

[128] A. Van Deursen, P. Klint, and J. Visser. “Domain-Specific Languages: An Annotated Bibliography.” In: *Sigplan Notices* 35.6 (2000), pp. 26–36.

[129] S. Vinoski. “CORBA: Integrating diverse applications within distributed heterogeneous environments”. In: *Communications Magazine, IEEE* 35.2 (1997), pp. 46–55.

[130] M. Vogelgesang, S. Chilingaryan, T. d. S. Rolo, and A. Kopmann. “UFO: A Scalable GPU-based Image Processing Framework for On-line Monitoring”. In: *Proceedings of The 14th IEEE Conference on High Performance Computing and Communication & The 9th IEEE International Conference on Embedded Software and Systems (HPCC-ICESS)*. HPCC ’12. Liverpool, UK: IEEE Computer Society, June 2012, pp. 824–829. ISBN: 978-1-4673-2164-8. DOI: 10.1109/HPCC.2012.116.

[131] M. Vogelgesang, T. Faragó, T. Rolo, A. Kopmann, and T. Baumbach. “When Hardware and Software Work in Concert”. In: *Proceedings of the 14th International Conference on Accelerator & Large Experimental Physics Control Systems*. ICALEPCS ’13. 2013.

[132] E. Walker, R. Floyd, and P. Neves. “Asynchronous remote operation execution in distributed systems”. In: *Distributed Computing Systems, 1990. Proceedings., 10th International Conference on*. May 1990, pp. 253–259. DOI: 10.1109/ICDCS.1990.89292.

[133] Y. Wang, F. De Carlo, D. C. Mancini, I. McNulty, B. Tieman, J. Bresnahan, I. Foster, J. Insley, P. Lane, G. von Laszewski, et al. “A high-throughput x-ray microtomography system at the Advanced Photon Source”. In: *Review of Scientific Instruments* 72.4 (2001), pp. 2062–2068.

[134] M. Warkus. *The Official GNOME2 Developer’s Guide*. No Starch Press, Feb. 2004.

[135] B. van Werkhoven, J. Maassen, F. Seinstra, and H. Bal. “Performance models for CPU-GPU data transfers”. In: *14th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing*. CCGRID. IEEE/ACM. 2014, pp. 1–10.

[136] S. Wienke, P. Springer, C. Terboven, and D. Mey. “OpenACC – First Experiences with Real-World Applications”. In: *Euro-Par 2012 Parallel Processing*. Ed. by C. Kaklamanis, T. Papatheodorou, and P. Spirakis. Vol. 7484. Lecture Notes in Computer Science. Munich, Germany: Springer-Verlag Berlin Heidelberg, 2012, pp. 859–870. ISBN: 978-3-642-32819-0. DOI: 10.1007/978-3-642-32820-6_85.

[137] W. A. Wulf and S. A. McKee. “Hitting the Memory Wall: Implications of the Obvious”. In: *SIGARCH Comput. Archit. News* 23.1 (Mar. 1995), pp. 20–24. ISSN: 0163-5964. DOI: 10.1145/216585.216588.

[138] Y. Zhang and F. Mueller. “GStream: A General-Purpose Data Streaming Framework on GPU Clusters”. In: *Parallel Processing, International Conference on* (2011), pp. 245–254. ISSN: 0190-3918. DOI: 10.1109/ICPP.2011.22.

List of acronyms

10GbE	10 Gigabit Ethernet
ANKA	Ångströmquelle Karlsruhe
ALU	Arithmetic Logic Unit
API	Application Programming Interface
APS	Advanced Photon Source
AST	Abstract Syntax Tree
ASTOR	Arthropod Structure revealed by ultra-fast Tomography and Online Reconstruction
ART	Algebraic Reconstruction Technique
CCD	Charge-coupled Device
CL	CameraLink
CMOS	Complementary Metal-oxide Semiconductor
CORBA	Common Object Request Broker Architecture
CPU	Central Processing Unit
CT	Computed Tomography
CUDA	Compute Unified Device Architecture
CU	Compute Unit
DAG	Directed Acyclic Graph
DBS	Duplication Based Scheduling
DDR	Double Data Rate
DFI	Direct Fourier Inversion
DMA	Direct Memory Access
DOOCS	Distributed Object-Oriented Control System
DSL	Domain Specific Language
ESRF	European Synchrotron Radiation Facility
EPICS	Experimental Physics and Industrial Control System
FBP	Filtered backprojection
FIFO	First In First Out
FFT	Fast Fourier Transform
FSM	Finite State Machine
FPGA	Field Programmable Gate Array
FPU	Floating Point Unit
GCN	Graphics Core Next
GIL	Global Interpreter Lock
GPU	Graphics Processing Unit
GUI	Graphical User Interface
HCA	Host Channel Adapter
HDF5	Hierarchical Data Format 5
I/O	input/output
ILP	Instruction-level Parallelism

IPE	Institute for Data Processing and Electronics
JIT	just in time
JSON	JavaScript Object Notation
LIMA	Library for Image Acquisition
LSDF	Large Scale Data Facility
LDS	Local Data Storage
MIC	Many Integrated Core
MIMD	Multiple Instruction Multiple Data
MISD	Multiple Instruction Single Data
MPI	Message Passing Interface
NLM	Non-local means
OpenCL	Open Compute Language
OpenMP	Open Multi-Processing
PE	Processing Element
PCIe	Peripheral Component Interconnect eXtended
PRAM	Parallel Random Access Machine
PTX	Parallel Thread Execution ISA
QDR	Quad Data Rate
RDMA	Remote Direct Memory Access
RPC	Remote Procedure Call
SCC	Single-chip Cloud Computer
SCMOS	scientific CMOS
SDK	Software Development Kit
SIMD	Single Instruction Multiple Data
SISD	Single Instruction Single Data
SPMD	Single Program Multiple Data
SFU	Special Function Unit
SM	Streaming Multiprocessor
SMT	Simultaneous Multithreading
SMP	Symmetric Multiprocessing
SP	Streaming Processor
SPMD	Single Program Multiple Data
SGPR	Scalar General Purpose Register
SQL	Structured Query Language
SRμCT	synchrotron radiation-based micro CT
SSE	Streaming SIMD Extensions
TANGO	Taco Next Generation Objects
TCP/IP	Transmission Control Protocol/Internet Protocol
TLP	thread-level parallelism
TPC	Texture/Processor Cluster
VLIW	Very Long Instruction Word
VGPR	Vector General Purpose Register
UCA	Unified Camera Abstraction
UFO	Ultra Fast X-ray Imaging of Scientific Processes with On-line Assessment and Data-driven Process Control
UML	Unified Modeling Language