Doctoral Dissertation

Processing Multimedia Workloads on Heterogeneous Multicore Architectures

Håkon Kvale Stensland

February 2015

Submitted to the Faculty of Mathematics and Natural Sciences at the University of Oslo in partial fulfilment of the requirements for the degree of Philosophiae Doctor

Series of dissertations submitted to the
Faculty of Mathematics and Natural Sciences, University of Oslo
No. 1601

ISSN 1501-7710

Cover: Hanne Baadsgaard Utigard.
Printed in Norway: AIT Oslo AS.

Produced in co-operation with Akademika Publishing.
The thesis is produced by Akademika Publishing merely in connection with the
thesis defence. Kindly direct all inquiries regarding the thesis to the copyright
holder or the unit which grants the doctorate.

Abstract

Processor architectures have been evolving quickly since the introduction of the central processing unit. For a very long time, one of the important means of increasing performance was to increase the clock frequency. However, in the last decade, processor manufacturers have hit the so-called power wall, with high heat dissipation. To overcome this problem, processors were designed with reduced clock frequencies but with multiple cores and, later, heterogeneous processing elements. This shift introduced a new challenge for programmers: Legacy applications, written without parallelization in mind, gain no benefits from moving to multicore and heterogeneous architectures. Another challenge for the programmers is that heterogeneous architecture designs are very different with respect to caches, memory types, execution unit organization, and so forth and, in the worst case, a programmer must completely rewrite the application to obtain the best performance on the new architecture.

Multimedia workloads, such as video encoding, are often time sensitive and interactive. These workloads differ from traditional batch processing workloads with no real-time requirements. This work investigates how to use modern heterogeneous architectures efficiently to process multimedia workloads. To do so, we investigate both simple and complex workloads on multiple architectures to learn about the properties of these architectures. When programing multimedia workloads, it is very important to know how the algorithms perform on the target architecture. In addition, achieving high performance on heterogeneous architectures is not a trivial task, often requiring detailed knowledge about the architecture. We therefore evaluate several optimizations so we can learn how best to write programs for these architectures and avoid potential pitfalls. We later use the knowledge gained to propose a framework design and language called Parallel Processing Graph (P2G). The P2G framework is designed for multimedia workloads and supports heterogeneous architectures. To demonstrate the feasibility of the framework, we construct a proof-of-concept implementation. Two simple workloads show that we can express multimedia workloads in the system. We also demonstrate the scalability of the designed solution.

Acknowledgements

Working with the PhD has been a long journey, at times, it has been both frustrating and stressful, but it has mostly been lots of fun. I would like to thank my supervisors, Professor Carsten Griwodz and Professor Pål Halvorsen for interesting discussions, their deep insights and valuable feedback over the years.

I would also like to thank my colleagues Håvard Espeland and Paul Beskow who I have shared office with for good collaboration and inspiring discussions on the topic of processing multimedia workloads. During the work with this thesis I have supervised several master students. I would like to thank all of them, the discussions have been very inspiring, and have helped me with this thesis.

The work environment at Simula Research Laboratory and the Media-department has also been excellent. Here, I would like to thank Andreas Petlund, Kristian Evensen, Ragnhild Eg, Preben Olsen and Vamsidhar Gaddam for making Simula a great place to be.

Finally I would like to thank my family, friends and especially my wife Marianne, for being patient and always supporting me no matter what I decide to do.

Contents

List of Figures

List of Tables

Part I

Overview

Chapter 1

Introduction

1.1 Background and Motivation

Multimedia applications are one class of applications that typically follow the trend of increasing processing demands to continuously increase quality and perceived experience. For example, live interactive multimedia services are steadily growing in volume. In this respect, Internet users uploaded over 100 hours of video to YouTube every minute in 2014 [139]. In the future, consumers will demand features such as interactively refined video search, dynamic participation in video conferencing systems, and user-controlled views in live media content. To support these features, we must be able to process compute-intensive workloads such as those required in extracting video features to identify objects, in the calculation of three-dimensional depth information from camera arrays, or in generating Freeview video from multiple camera sources in real time. This adds further magnitudes of processing requirements to already computationally intensive tasks such as the traditional video encoding of high-definition videos.

Over the last decade, we have witnessed two paradigm shifts within modern processor architectures. The first shift was when single-core processors reached their power and frequency limits, forcing chip designers to start focusing on on-chip parallelism. This started with the introduction of IBM's POWER4 dual-core processor [118] and was followed by the introduction of Hyper-Threading Technology [60] on Intel's Pentium 4 processors. Today, dual- and quad-core processors from Intel and AMD are a commodity in desktop and laptop computers. Mobile devices have followed the same trend and several processor designs, such as Nvidia's Tegra 4 mobile system on a chip (SoC) [91], have a quad-core general-purpose processor.

The other paradigm shift was the introduction of heterogeneous processing architectures, such as the Cell Broadband Engine [54] from Sony, Toshiba, and IBM (STI) and the graphics processing units (GPUs) of Nvidia, AMD, and other vendors.

Heterogeneous processing architectures provide more computing power than traditional general-purpose single- and multicore systems. The processing cores have different instruction sets, use several different types of memory, and provide different programming abstractions compared to traditional desktop processors. Heterogeneous processing cores are often designed very differently from general-purpose cores, being more specialized toward solving specific tasks. Because of this, the cores in a heterogeneous architecture can utilize more of the die space on the chip for arithmetic and logic units (ALUs) and use less space for caches and control logics. We are also witnessing the trend of heterogeneous processing in mobile devices, where all SoCs have dedicated processors for audio, video, and imaging. Even the latest generation of Intel x86 processors has a dedicated coprocessor called Quick Sync [61] for video encoding, decoding, and transcoding.

Today, programmers who want to utilize these heterogeneous processing architectures face several challenges. They must write their applications with a very detailed knowledge about the target architecture and, when the target architecture is changed or even upgraded to a new generation by the hardware vendors, applications optimized for one architecture will in some cases require a complete rewrite for utilization or at least efficient execution on the new architecture. We therefore need abstractions and programming concepts that will ease the development of applications for heterogeneous processing architectures. This thesis is meant as a step toward this goal.

1.1.1 Heterogeneous Architectures

Processor architectures have been evolving quickly since the introduction of the central processing unit (CPU) and vendors often release either a new architecture or an updated architecture every year. For a very long time, one of the important means of increasing performance was to increase the clock frequency of the processors. However, in the beginning of the 2000s, this approach started to become problematic [79]. As processors evolved, the chips also shrank, with the manufacturing process packing more and more transistors onto smaller areas. With the introduction of the 90nm process in 2004, several vendors hit the so-called power wall: The transistors, working at very high frequencies, leaked power and, to make them run stably, the voltage had to be increased, resulting in higher heat dissipation.

The CPU vendor's first solution to this challenge was to place multiple cores onto a single chip and AMD was the first to introduce such a processor for consumers, with the Athlon 64 X2 in 2004 [3]. Multi-core processors were nothing new and had previously been provided by multiple vendors, but only in high-end systems, with multiple processor sockets connected to the buses. The introduction of multicore systems was a paradigm shift for the average developer, who now had to parallelize applications to scale performance.

A different approach to continuously scale performance is to add simple cores or more specialized cores that can carry out certain tasks more quickly and more efficiently—hence the term *heterogeneous architectures*. Heterogeneous architectures in a simple form have been around for a long time. One example is from 1985, when Intel's 80386 processor had the option of adding an x87 floating-point coprocessor. This coprocessor was later integrated into the main processor core as an independent pipeline when the 80486 processor was launched in 1989. In Chapter 2, we examine the x86 core in more detail and see that what seems to be a single core is in fact built up of several heterogeneous elements.

Another example of heterogeneous architecture is the use of processor cores with different capabilities and instruction sets. One example of such an architecture is Intel's specialized network processor IXP1200, which was launched in 1999 [56]. This architecture had one general-purpose core to run the operating system and execute the control plane in the network and several specialized cores for packet processing. Another example of such a system in the consumer market is the Sony PlayStation 3, launched in 2006 [55].

The PlayStation 3 featured the Cell Broadband Engine, a heterogeneous processor optimized for floating point operations, which are important in both computer games and multimedia processing. Perhaps the most common heterogeneous architecture in computers today, however, consists of GPUs working together with the CPU. Over the last decade, GPUs have evolved from simple fixed-function pipelines to fully programmable processors. With the launch of CUDA in 2007 [92], Nvidia introduced low-cost high-performance computing to the masses.

However, programming heterogeneous architectures has proven to be a significant challenge. Legacy applications written without parallelization gain no benefit from the move to multicore and heterogeneous architectures. In some cases, legacy applications may even perform worse on modern architectures, because the architectures are focused toward adding more cores at a lower frequency instead of having fewer cores at a high frequency. Another challenge with heterogeneous architectures (e.g., GPUs) is that they can be very different in design in regard to caches, types of memory, execution unit organization, and so forth, and, in the worst case, the programmer must completely rewrite the application to obtain the best performance from a new architecture. Chapters 3 and 4 present our experience with developing multimedia workloads for some of these heterogeneous architectures.

1.1.2 Multimedia Workloads

A multimedia workload is often characterized as being time sensitive and iterative. Video processing is an example that involves a multimedia workload and it is explored later in this thesis. Encoding and decoding video are very computationally intensive operations, where calculations have to be done for all the pixels in a frame. Video encoding is a common operation in video processing; raw data from the imaging sensor are often compressed according to bandwidth or storage requirements defined by the scenario. The most common standard for video coding today is H.264 [131], which was defined by the International Telecommunication Union (ITU) and the International Organization for Standardization (ISO) as a video codec for everything from streaming video to mobile devices to TV broadcasting and high-definition videos stored on Blu-ray disks.

Since H.264 is currently the de facto standard for video coding in the industry, a great deal of research has been performed on how to optimize its encoding and decoding. Most of the devices available have dedicated hardware to either assist the processor or carry out the decoding. The same trend appears for encoding. Dedicated hardware implementations are fast and have low energy requirements. However, they lack the flexibility of software implementation, they are complex and expensive to manufacture, and, when parameters (e.g., the resolution or the frame rate) are changed, the hardware must also be changed in most cases. Multimedia workloads developed in software are more flexible and are easier to update and replace compared to hardware implementations, where the entire chip has to be changed for a different implementation. This thesis therefore focuses on software implementations of multimedia workloads.

The software multimedia workloads investigated in this thesis are independent filters and operations on video data organized in pipelines, often with real-time processing requirements. One example of such a multimedia workload is the real-time panorama stitching pipeline in the Bagadus soccer analysis system [114], which is investigated further in Chapter 4.

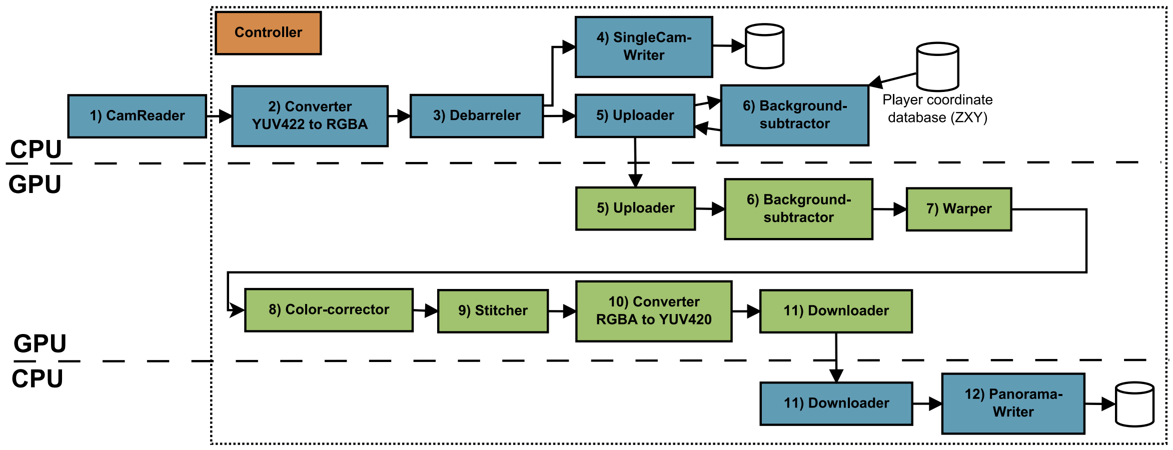


Figure 1.1: The real-time panorama video stitching pipeline in the Bagadus soccer analysis system [114].

Figure 1.1 provides an overview of the Bagadus pipeline. In the Bagadus pipeline, raw video data from multiple video cameras is read over the network in real time. The pipeline comprises several independent steps for converting the data between different formats, optimizing the videos, and finally stitching them together into a panoramic video. Most of the steps in the pipeline have dependencies that have to be satisfied before they can start processing and each step also has different degrees of parallelization potential. To fulfill the system's real-time requirements of 30 frames per second, the pipeline has to deliver an encoded panoramic video frame every 33 milliseconds. Another challenge of the Bagadus pipeline is the scalability of the video data, where adding more cameras, increasing the resolution of existing cameras, or increasing the system's frame rate will increase its computational demands. The addition of more and other more complex steps such as high dynamic range (HDR) rendering in the pipeline will also increase the complexity of the system.

1.2 Problem Statement

When used efficiently, modern heterogeneous architectures provide the processing power required by resource-hungry multimedia workloads. However, the diversity of resources to which developers are exposed makes it very hard to develop programs that are portable and scalable on multiple architectures. Even with new languages such as OpenCL, which are supposed to be a "recompile-only" solution, the applications must be tuned and in many cases hand optimized for the different heterogeneous architectures. In examining this area of computing, we proceeded according to the following problem statement:

How can programmers efficiently develop multimedia workloads for modern heterogeneous multicore architectures?

The problem stated in this thesis focuses on how to use modern heterogeneous architectures to efficiently process multimedia workloads. We want to learn how programmers need to think when writing their applications for these architectures. Many details about the architectures are also undocumented, so we need to learn how the architectures behave when processing multimedia workloads. We approached the problem statement in three steps, presented in Chapters 3, 4, and 5 and briefly formulated as follows:

- Learn about the behavior of heterogeneous architectures by implementing and evaluating prototypes where simple multimedia workloads run on a single heterogeneous architecture.
- Learn how to use multiple heterogeneous architectures to process a complex pipeline with several multimedia workloads running in real time.
- Propose new ways of designing and developing multimedia workloads with a framework for processing multimedia workloads on heterogeneous architectures.

1.3 Limitations

To reduce the scope of this thesis, we limited the number of heterogeneous architectures investigated. We also examine GPUs from only one vendor, Nvidia. The main reason for focusing on only one vendor is the availability of programming tools and documentation when the project started. The multimedia workloads we consider are data intensive and have good parallelization potential.

We focus only on processing multimedia workloads on the constrained resources of a single machine, not those of large-scale distributed systems. Many small devices today—such as smartphones, tablets, laptops, and desktop PCs—can process multimedia workloads, which is also expected by the users of such devices. Our main focus is therefore the efficient utilization of resources on these platforms.

1.4 Research Method

As defined by the Association for Computing Machinery (ACM) Education Board [25] in 1989, the discipline of computer science is divided into three major paradigms. Each of these paradigms has its roots in different areas of science and all can be applied to computing and computer science. The ACM Education Board states that the paradigms are intertwined and that it is irrational to say that any one of the paradigms is fundamental. These three computer science paradigms are as follows.

- *Theory*: This paradigm has its roots in mathematics. It defines objects of study and hypothesizes their interrelations; it then determines whether these relations are true and interprets the results. This paradigm is concerned with the ability to describe and prove relationships among the objects of study.
- *Abstraction*: This paradigm is from the experimental research field and consists of four stages. A scientist forms a hypothesis, constructs a model, makes a prediction before designing an experiment, and collects data. This paradigm is concerned with the ability to use predictions that can be compared with real-world situations.

- *Design*: This paradigm is from the field of engineering and involves building a device or system to solve the given problem. The scientist states the requirements and specifications of a design and the system's implementation. The system is then tested and the previous steps can be repeated if the system does not meet the requirements. This paradigm is concerned with the ability to implement specific instances and use them to perform useful actions.

All three of these paradigms are applied in our field of multimedia systems research, the one to use depending on the problem to be solved. In our case, in which we want to investigate how to use heterogeneous multicore architectures for processing multimedia workloads, we first determined if the *theory* paradigm could be applied. The *theory* paradigm requires a precise definition and modeling of the multimedia software and the heterogeneous multicore hardware we want to use. This is particularly challenging with hardware, since many of the details about the low-level behavior of schedulers and the execution pipeline are known only to the hardware vendors. We therefore rejected this paradigm for this thesis. The next paradigm is the *abstraction* paradigm. This paradigm is often used, typically in combination with simulators. One can model the behavior of both the software and the hardware architecture in a simulator to try to capture the system's behavior. We used this paradigm in one of the initial investigations on using heterogeneous architectures for multimedia workloads. However, the challenge is that simulators are no better than the models used and, as mentioned previously, because of the black boxes in the hardware, many of the details about the architectures are unknown. It is therefore impossible to know how good the simulated behavior is without conducting experiments on real systems and we rejected this theory as well. The last paradigm is the *design* paradigm. With this paradigm, we specified the requirements and built prototype systems. These prototypes were evaluated on real-world systems and, based on the data gathered, the prototype improved over multiple iterations. One of the challenges with this paradigm is that it can take significant effort to implement these prototypes, which in itself is not necessarily scientific work. The fact that many of the details of the hardware are not published by their vendors is a challenge in our investigation of how to program multimedia workloads for modern heterogeneous multicore architectures. We therefore have to make many assumptions and simplifications if we want to simulate exactly how the hardware works. We therefore decided to use the *design* paradigm in this thesis. It requires more engineering work, but we were not limited by inaccurate models as a basis for the simulations.

1.5 Main Contributions

The main research question in Section 1.2 states the challenge on how to efficiently use modern heterogeneous multicore architectures to process multimedia workloads. This problem statement is addressed from both a low-level standpoint, where we carry out experiments with simple multimedia workloads on different heterogeneous multicore architectures, and a more high-level one, where we present, implement, and evaluate a programming model to support the real-time processing of multimedia data. The papers that are part of the contribution of this thesis have been published in a number of peer-reviewed conference proceedings and international journals and are included in full in Chapter 6. Several other papers related to processing multimedia workloads were not included in the thesis to limit its scope. A summary of the main contributions included is as follows.

- We learned about the behavior of heterogeneous architectures with simple multimedia workloads. The first architecture we experimented with was Intel's IXP network processor. This architecture was used for experiments involving network protocol translation [32]. The next architecture was the x86 processor architecture. Here, the first case study involved the efficient implementation of Motion JPEG (MJPEG) encoding [112]. We also conducted case studies on using multiple x86 cores for multi-rate video encoding [34] and running a multi-threaded game server prototype [102]. For the GPU architecture, we conducted case studies on its memory [94], optimization of host-to-device communication [13], and cheat detection [117]. We also revisited the MJPEG workload with both the GPU [112] and Cell architectures. The IXP architecture provided experience in working with a state-of-the-art (at the time) heterogeneous architecture with several special features. With the MJPEG workload on the x86 architecture, we learned the importance of selecting algorithms optimized for the target architecture and the benefits of using the vector unit available on all modern processors. The shared memory model on the x86 revealed the importance of reusing parts of computationally heavy multimedia workloads and of using the optimal number of threads for the number of cores available. On the GPU architecture, we learned about the importance of using the memory architecture correctly and optimizing transfers between the host CPU and the GPU with asymmetrical transfers. We also observed that when offloading multimedia workloads to a GPU, they have to be large enough to compensate for the added latency of data transfers and launching kernels on the GPU. With the MJPEG workload, as on the x86, we learned the importance of optimizing algorithms for the target architecture. On the Cell architecture, we also learned the same lessons as on the x86 and GPUs: It is important to select an algorithm suited for the target architecture and always vectorize your workload when possible. Our MJPEG experiments also suggested that programmers prefer programming models exposed by the GPU compared to models exposed by the vector unit on x86 and the Cell.

- The knowledge obtained from investigating simple multimedia workloads was used to investigate a more complex multimedia workload, that is, part of the Bagadus soccer analysis system, which has three components: a tracker subsystem, an analytics subsystem, and a video subsystem [114]. The complex workload used in these experiments was that of a video subsystem that involved the real-time capture, pre-processing, stitching, and encoding of a panoramic video stream from a soccer stadium [113]. Here, we had to optimize the workload for multiple heterogeneous architectures to run the workload in real time. By implementing the subsystem as a pipeline and optimizing for both the x86 architecture and GPUs, we were able to capture the five 720p streams and stitch them into a panoramic video on a single commodity gaming PC.

- We used our knowledge from processing both simple and complex multimedia workloads on heterogeneous systems and from our evaluation of multicore scheduling mechanisms [115] as follows. We proposed a programming language and framework that exposes the parallelization opportunities of a multimedia workload for a runtime that allows for the efficient execution of a workload on the available heterogeneous hardware [31]. We also developed a prototype of this system running on a single machine with support for multicore x86 processors, together with several simple multimedia workloads running on the system. In addition, we conducted experiments demonstrating the system's usability for multimedia workloads.

There are many opportunities for further work within this field; however, we aim to show the essential challenges of using heterogeneous multicore architectures for processing multimedia workloads and the essential considerations that programmers must make when choosing both the architectures and the algorithms.

1.6 Outline

This thesis is written as a collection of paper and it is therefore organized in two parts. Part I provides an introduction and places the research papers in context and Part II includes the selected full papers.

The first part is organized as follows: In Chapter 2, we introduce the heterogeneous architectures used in our research. We also look at the low-level programming abstractions that are used when programming the different heterogeneous architectures. In Chapter 3, we research several simple multimedia workloads on the heterogeneous architectures and we discuss how to use the architectures for the different multimedia workloads and how to structure the workloads to obtain the best performance from the architectures. This is followed up in Chapter 4 with an investigation of a more complex pipeline, with several components running on different heterogeneous architectures. Based on our results in Chapters 3 and 4, we introduce in Chapter 5 the P2G framework for running multimedia workloads on heterogeneous architectures. We benchmark and evaluate a set of multimedia workloads within the P2G framework. Chapter 6 gives an overview of the research papers and, finally, Chapter 7 provides a conclusion.

Chapter 2

Heterogeneous Computing

In this chapter, we introduce the heterogeneous hardware architectures used for experiments in this thesis. This introduction provides insight into the history of the different architectures; it gives a basic introduction to the architecture by looking at the state-of-the-market products available in each of the architectures.

The architectures we have chosen are very different with respect to the amount of available computational resources—floating point units, arithmetic and logic units (ALUs), and so forth—and how these are connected in the architecture's execution pipeline. The memory types and layouts are also very different, some of the architectures having a shared memory model that hides the memory management from programmers and other architectures having an explicit memory model, which gives programmers full control over the memory. We also examine caches, how they are organized, and whether programmers have any control over how they are used. Next, we look at the buses that connect the resources within the processors and as well as the processors with each other and resources in the machine. Finally, we look at what these heterogeneous architectures expose to programmers. We investigate three examples of what the architectures expose to programmers and how the programming model is used to hide some of the architecture's complexity.

2.1 Hardware Architectures

In this section, we take a more detailed look at some heterogeneous architectures. First, we look at the family of x86 processors, specifically those produced by Intel. Next, we introduce the Intel IXP2400 network processing unit (NPU) with a heterogeneous architecture before looking into the Cell Broadband Engine (CBE). Finally, we take a look at Nvidia graphics processing units (GPUs) used for general-purpose programming.

2.1.1 Intel x86 Processor Architecture

The x86 processor architecture has a long history, dating back to Intel's 8086 central processing unit (CPU) released in 1978 as a fully 16-bit processor. One of the reasons this instruction set succeeded in becoming the dominant instruction set in the mainstream computer market was the fact that IBM selected the 8086 for the original IBM PC. Over the years, the x86 instruction set has undergone many extensions (32 bit in 1985 and

64 bit in 2003) and additions. However, the instruction set has always been backward compatible with previous versions. A modern x86 processor from 2014 is still able to execute 16-bit code compiled for the original 8086. Over the years, several vendors have also been designing and manufacturing CPUs compatible with the x86 instruction set (e.g., Intel, AMD, VIA, and Cyrix). However, today only two remain and, of those, Intel is the dominant vendor.

Originally, x86 was a little-endian, variable instruction length complex instruction set computer (CISC) design. However, over the years, the processors executing the instruction set changed greatly. The introduction of superscalar pipelines, where a CPU with a single instruction stream can dynamically check data dependencies and process multiple instructions per clock cycle, made it possible for the x86 processors to execute more operations in parallel in a single clock cycle (i.e., fetch, decode, execute, memory, and write back). Modern x86 architectures are able to decode x86 instructions into smaller operations called micro-operations (μops). The processors then use out-of-order execution to reorder those μops. This approach combined with a superscalar pipeline enables modern processors to extract parallelism out of the code stream for improved performance. A great deal of responsibility is left to the instruction decoders. It is the decoder's job to make the execution as efficient as possible, as well as all the instructions and μops analyzed by branch predictors. If branches are detected, the processors will use speculative execution to try to prevent miss predicted branches stalling the pipeline. In the same way, a great deal of effort is also put into prefetching data from memory into the different caches, so that as much data as possible is ready in the caches for execution.

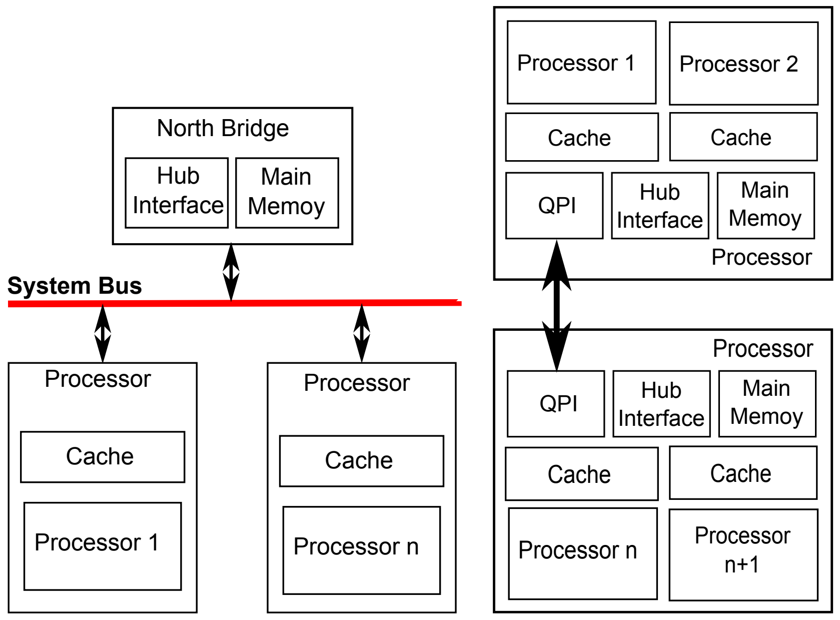
A technique called symmetric multiprocessing (SMP) is used to implement multiple processor cores in a x86 system. Here, two or more identical processors, all of which have full access to the I/O devices, connect to a single shared main memory and are all controlled by a single operating system. Even though the architectures of x86 processors have evolved over the years, the principles behind SMP are still the same, leading to several challenges when trying to scale the numbers of processing cores in an SMP system. One of the challenges is memory. In early SMP systems, all cores shared the same memory controller, but with integration of the memory controller onto the die of processors, the access times to the different parts of memory are not the same. Another challenge is cache coherency. Since all processor cores have the same access to the main memory, all the caches on the processors must be kept up-to-date. If one core changes data, this change must be broadcasted to all the other cores that work with the same data. The first x86 implementations with multiple cores had one processor core per socket and multiple sockets on a motherboard. The first attempts at processing more than one thread simultaneously on a single die used simultaneous multithreading (SMT) on Intel's Pentium 4 processors. Because of the very long pipeline on Pentium 4 processors, several parts of the pipeline were often idle. To use more resources, Intel implemented Hyper-Threading Technology [60], it's version of hardware multithreading. Intel's first true multicore processor, with two separate independent cores on a single die, was the Pentium D, introduced in 2005. Today, most commodity desktop machines and laptops have two or four processor cores on a single die. In the server and workstation space, up to 18 cores are fitted onto a single die.

During work on this thesis, several generations of processor architectures were released by Intel, a list of which can be found in Table 2.1. The processor roadmap used by Intel

Architecture	Codename	Fabrication process	Released
Core	Merom	65 nm	2006
	Perryn	45 nm	2007
Nehalem	Nehalem	45 nm	2008
	Westmere	32 nm	2010
Sandy Bridge	Sandy Bridge	32 nm	2011
	Ivy Bridge	22 nm	2012
Haswell	Haswell	22 nm	2013
	Broadwell	14 nm	2014

Table 2.1: Roadmap of Intel processors used in our experiments.

today was introduced in 2007 and is called the Tick Tock CPU roadmap [58]. The idea of this roadmap is to follow every new architecture, referred to as a "tock," with a shrink in fabrication processes referred to as a "tick." One of the advantages with this strategy is the reduction of risk when moving to a completely new fabrication process, since the architecture is already known, and vice versa when developing a new processor architecture.



(a) Traditional system bus (b) Point-to-point links

Figure 2.1: Comparison of SMP architectures.

Over the period of this thesis, the architecture connecting multiple processors and external devices has also changed and an overview of the two different architectures can be seen in Figure 2.1. During the 1990s and the 2000s, until the Nehalem architecture, Intel used a simple bus called the front-side bus (FSB) to connect multiple CPUs and to connect these to the so-called north bridge (Figure 2.1(a)). The north bridge is also referred to as the memory controller hub. This is where the memory controller is located and also where you would connect external devices that require fast access to main memory. These devices are often connected by a PCI, AGP, and, later, PCI Express bus. Such I/O devices as network and hardware are connected on the "south bridge" via the hub interface, also referred to as the DMI. One of the challenges with the FSB was that it is a shared bus and was quickly becoming a bandwidth bottleneck when multiple CPUs were connected. With the Nehalem architecture, the FSB was replaced by several point-to-point interconnects, called QuickPath Interconnect (QPI) [59] (Figure 2.1(b)), and the memory controller was integrated onto the same die as the processor cores were. To better share the memory controller between multiple cores on the die, a level 3 cache was added. Furthermore, the PCI Express bus used to connect external devices was also added to the same die, thus eliminating the need for a north bridge. With the Sandy Bridge architecture, a GPU was also integrated into the same die as the processor cores were. This GPU also shares the memory controller with the processor cores and uses the same level 3 cache to connect to the memory.

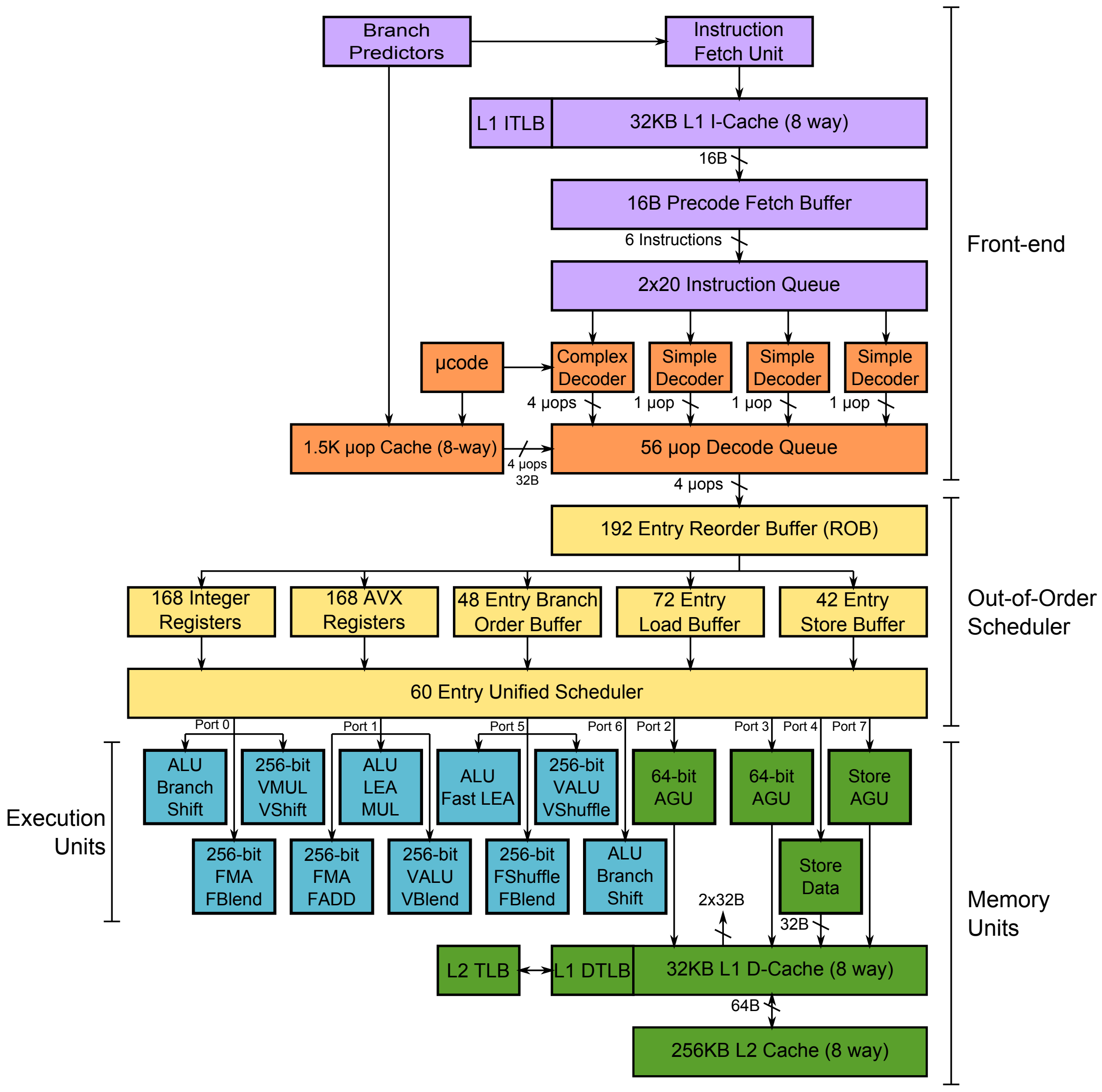


Figure 2.2: Intel Haswell architecture diagram.

The Haswell Architecture

The latest x86 architecture from Intel is called Haswell [104]. It was released in 2013 and, according to the Intel CPU roadmap [58], it was a tock, that is, a new architecture. A detailed overview of a single Haswell core is shown in Figure 2.2.

The first part of the Haswell processor architecture, orange and purple in Figure 2.2, is called the *front end*. This part of the processor is perhaps one of the most complex parts of a modern x86 CPU. The instruction cache is kept coherent with the data cache. Haswell has four decoders to decode the x86 instructions, including three simple decoders that are able to decode one fused μop and one complex decoder that is capable of decoding four fused μops. With μops fusion, the decoders are able to combine x86 instructions that can be executed in parallel (i.e., jump and compare). The front end also includes other important features, such as the branch predictor; however, details of the branch prediction unit are not published by Intel. The next part of the pipeline, in yellow in Figure 2.2, is the *out-of-order scheduler*. Haswell has a 192-instruction out-of-order window. The first part of out-of-order execution is renaming. Here, the renaming allocates resources and maps the source and destination x86 registers onto the underlying physical register files. The idea of out-of-order execution is to optimally reorder the instructions for execution, making sure that as many dependencies as possible are met before the instruction is executed, and, after reordering, to place the μops in a unified scheduler queue ready for execution. Execution is split into two parts: *Execution units*, in blue, and *memory units*, in green in Figure 2.2. In total, the Haswell architecture has eight execution ports, meaning that each cycle the oldest eight non-conflicting μops that are ready for execution are taken out of the unified scheduler and dispatched to the execution ports. Computational μops are sent to port 0, 1, 5, and 6. Three of the four computational execution ports are also capable of executing 256-bit single instruction multiple data (SIMD) instructions. Memory operation μops are sent to ports 2, 3, 4, and 7. The memory hierarchy on the Haswell core is similar to that of earlier Intel processors. For each core, there is a 32-kB level 1 data cache and a 256-kB unified level 2 cache, both private for the core. The level 3 cache, however, is shared with the other cores on the die and with the on-die GPU.

A single Haswell x86 core can therefore have up to two hardware threads that execute up to eight instructions in parallel. This is carried out with techniques such as superscalar pipelines, multithreading, and out-of-order-execution.

Intel Many Integrated Core (Intel MIC) Architecture

Intel MIC is a multicore architecture developed by Intel incorporating work from the now defunct Larrabee GPU architecture [109] and the Single-chip Cloud Computer research project [78]. The first commercial product released in the MIC family had the codename Knights Corner and was later branded Intel Xeon Phi [21].

The Xeon Phi architecture consists of up to 61 simple x86 cores and a basic overview of the architecture is shown in Figure 2.3. The processor architecture used is the P54C [21] architecture, originally used in the Pentium processor from 1993. The cores have been modified with support for the 64-bit x86 instruction set and support for four-way SMT. The process still keeps its original in-order execution pipeline design and the coherent level 2 cache has been extended to 512 kB per core. The main change in the architecture from the original P54C cores lies in the floating-point pipeline. The P54C has a simple

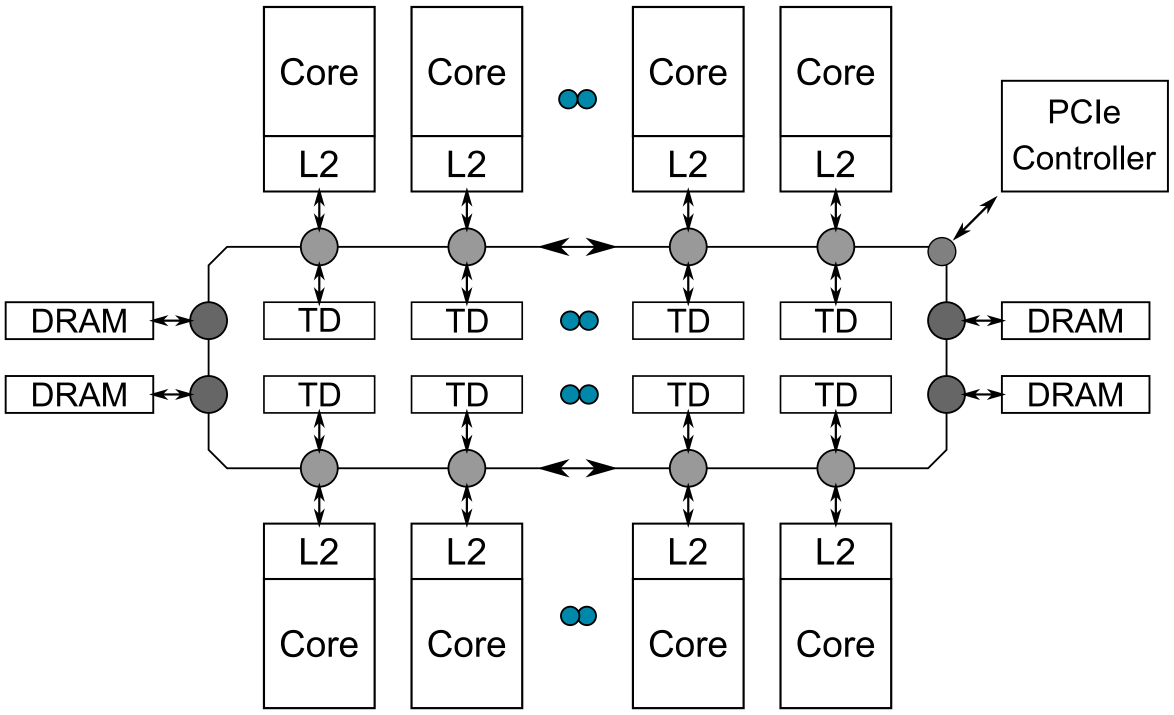


Figure 2.3: Intel Xeon Phi MIC architecture.

x87 floating point unit [21] and the Xeon Phi has a 512-bit SIMD unit in addition to the x87 pipeline. This SIMD unit is able to process 16 independent 32-bit values and eight independent 64-bit values. Additionally, fused multiply and add operations are supported. However, the unit is not compatible with existing MMX/SSE code; so, even though the Xeon Phi cards use x86 cores, they are not backward compatible with the x86 instruction set. All the cores are connected by a 512-bit ring bus and the card has up to 16 GB of on-board GDDR5 memory. The Xeon Phi cards are connected to the host computer through a PCI Express 2.0 interface.

To control the processor, a customized version of Linux is booted on the card when initialized from the host computer. The Xeon Phi supports the offloading of parts of programs from the host processor and, since the card has its own operating system, it can also work as a stand-alone system.

2.1.2 Intel IXP Network Processor

The Intel Internet eXchange Processor (IXP) Architecture [57] is an NPU. An NPU is a specialized processor designed and optimized for efficient packet handling and throughput. A typical NPU features several small processing elements optimized for pipelining and executing data plane tasks and a general-purpose processing core to execute control plane workloads, thus making this a heterogeneous multicore architecture. Intel's first generation of NPUs was called the IXP1200 [56]. This processor features a 32-bit StrongARM general-purpose core and six special-purpose cores called micro engines (μEngines). The NPU used in our research is a second-generation processor from Intel called the IXP2400 [57]. The IXP2400 cards were chosen because it was one of the easiest architectures to program. Intel continued developing their NPUs with a third generation of processors called IXP28XX and, in 2007, sold the entire product line to Netronome.

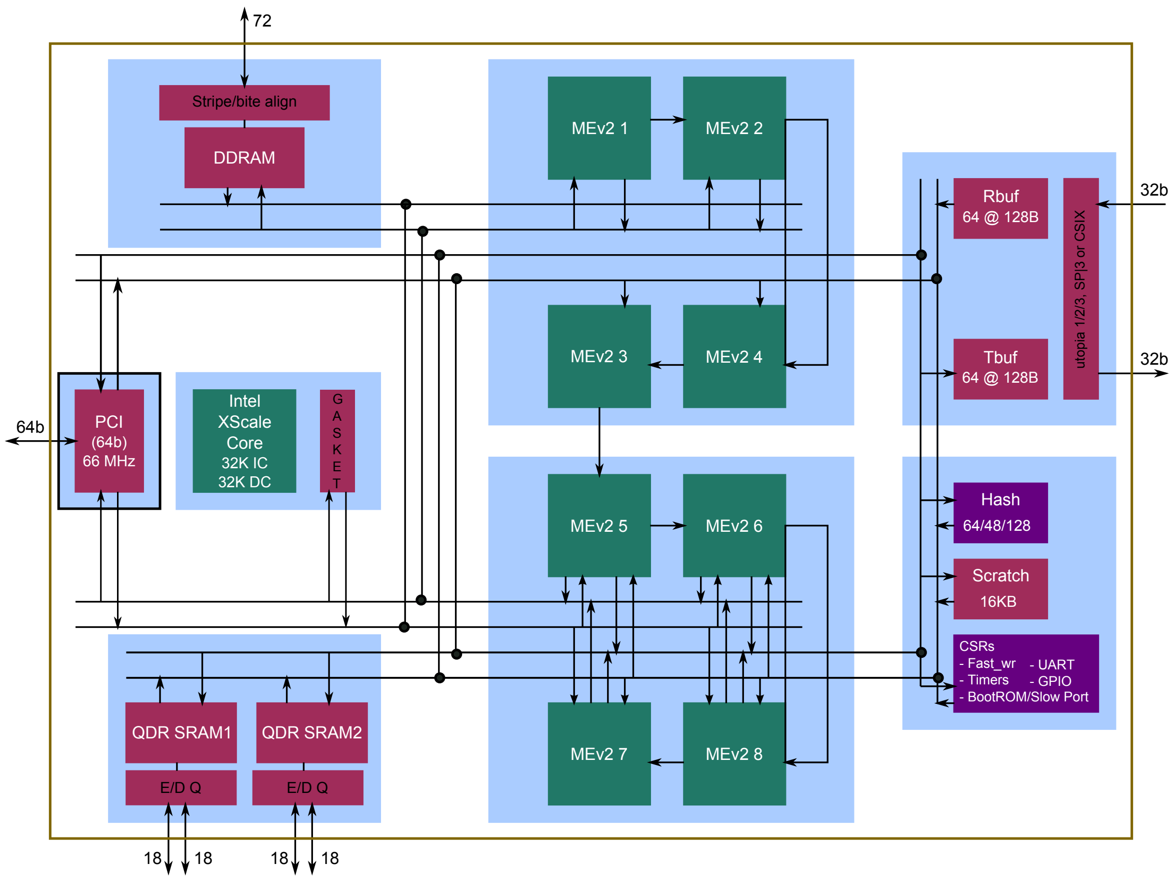


Figure 2.4: Intel IXP2400 architecture diagram.

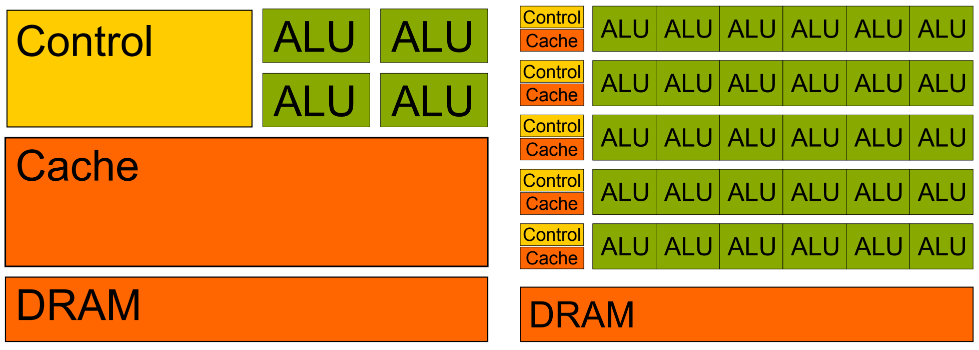
The Intel IXP2400 architecture overview is shown in Figure 2.4. The basic elements on the chip include a 600 MHz XScale processor. The XScale is a general-purpose processor using the ARMv5 instruction set and the general-purpose processor also runs a Linux or VxWorks operating system to control the card. Additionally, the IXP2400 has eight specialized packet μEngines also running at 600 MHz. The μEngine uses a proprietary instruction set that is not compatible with the ARMv5 instruction set on the XScale. Each μEngine is capable of running four threads in hardware and the μEngines are optimized for general packet processing in the data plane (fast path). The XScale is used for the control plane (slow path). The μEngines are grouped in clusters of four cores and, within each cluster, the cores can communicate with neighbor cores via specialized registers. In normal configurations, two μEngines are reserved for low-level network receive and transmit functions using open-source software, leaving only six μEngines available for application usage.

Moreover, the IXP2400 has three kinds of memory, with different bandwidths and access times. The 256 MB of SDRAM is used for the operating system and packet store, the 8 MB of SRAM is used for metadata (e.g., packet headers), and the 16 kB of on-chip scratchpad is used for interprocess communication and synchronization between the cores. The IXP2400 is connected to the host computer and supports direct memory access (DMA) transfers with a 64-bit PCI connector and the card has three physical mini-GBIC connector for gigabit Ethernet.

The software development kit for the IXP2400 cards includes a specialized compiler to program the μEngines. The compiler is a C compiler for the MicroC language. The IXP2400 allows data to be processed at wire speeds with very low latency. The cards can process packets with a very limited protocol stack. This allows the programmer to both update and extract information with very low processing overhead. This makes these NPUs ideal for applications such as deep packet inspection and statistics collection.

2.1.3 Nvidia Graphics Processing Units

A GPU is a specialized and dedicated hardware originally designed to render graphics on a screen. A GPU can be integrated as part of the computer's chipset, as a discrete expansion card, or integrated on the die of the main processor. Originally, the GPU was designed to render three-dimensional (3D) scenes onto a two-dimensional frame of pixels. The first generations of GPUs had a fixed rendering pipeline with very limited flexibility and programmability. Compared to a normal general-purpose processor such as the x86 architecture from Intel, GPUs have a very different architecture. On the CPU, as seen in Figure 2.5, much of the die space is used for control logics, such as out-of-order execution, cache, and branch prediction [69]. A GPU has much less control logic and more ALUs. The GPU is designed to perform the same calculations over a large number of values, which is very similar to vector processors; for example, when rendering a 1080p (Full HD) frame, about 2 million independent pixels are processed in parallel.



(a) CPU (b) GPU

Figure 2.5: Comparison of transistor usage on a CPU and on a GPU.

The term GPU was defined in 1999, when Nvidia launched their GeForce 256 graphics adapter [133]. This was not the first 3D graphics card, but it was the first card to support hardware transform and lightning, meaning that the entire graphics pipeline now ran on the graphics adapter.

The progression of GPUs was mainly driven by the gaming industry in the beginning and the development of graphics cards was closely tied to the development of Microsoft's DirectX application programming interface (API). A new version of the API brought new generations of GPUs. In DirectX 8, programmable vertex shaders were added, which allowed the programmer to control how each vertex in the 3D scene was converted to discrete pixels in the output. DirectX 9 also added support for programmable fragment

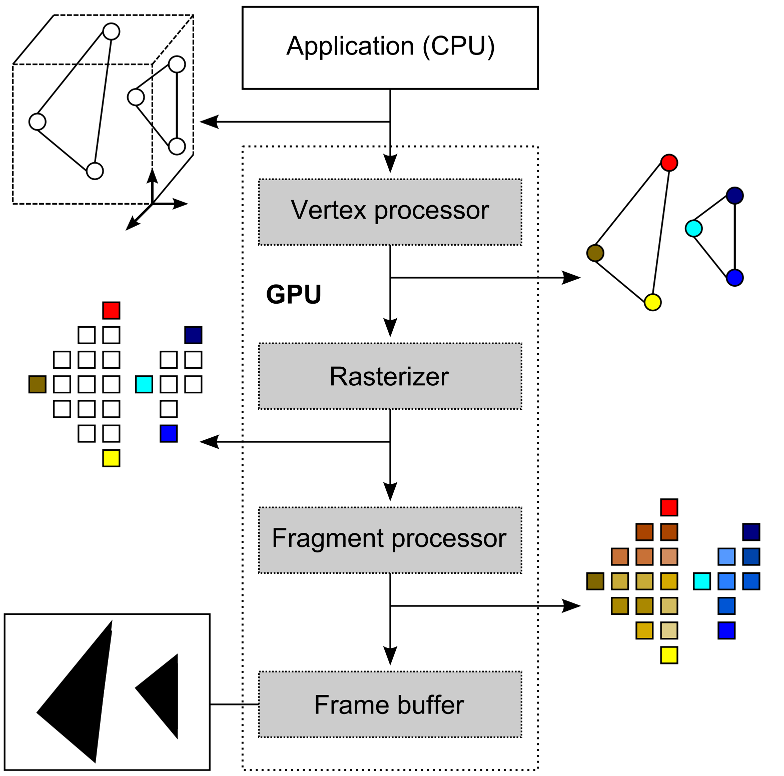


Figure 2.6: A pre-DirectX 10 graphics pipeline [43], with a programmable vertex processor and fragment processor.

shaders. An example of a DirectX 9 pipeline is shown in Figure 2.6. The fragment shaders gave the programmers direct control over pixel lightning, highlighting, translucency, and shadows. It was also possible to use these fragment and vertex shaders for general-purpose computations, given that one's problem could be mapped as a frame. To do so, data had to be stored as textures on the GPU and shader programs were executed on the data; the results had to be stored either in texture memory or in the frame buffer. With DirectX 10, the traditional 3D pipeline was replaced with a unified processing architecture [12]. Gaming was still the main driver for the GPUs, but changes in the GPU architecture made it much more suitable for general-purpose stream processing. It was also with the first DirectX 10 GPUs in 2007 that Nvidia launched CUDA, which made it possible to program GPUs with extensions to the C language [84]. Since the initial release of CUDA, both the underlying hardware and software stack have undergone both minor and major revisions, referred to as compute capability. During work on this thesis, we used three different generations of GPUs from Nvidia, an overview of which is shown in Table 2.2. During the last phase writing of this thesis, Nvidia also released a new-generation GPU called Maxwell; however, we take a more detailed look into how the third generation of CUDA-capable GPUs, codename Kepler, from Nvidia are designed and how their memory architecture works compared to that of a normal CPU.

Architecture	Codename	Compute	Released
Tesla	G80	1.0	2006
	GT200	1.3	2008
Fermi	GF100	2.0	2010
	GF110	2.1	2010
Kepler	GK104	3.0	2012
	GK110	3.5	2013
Maxwell	GM107	5.0	2014

Table 2.2: Roadmap of Nvidia GPUs used in our experiments.

The Kepler GK110 Architecture

The Kepler architecture consists of a number of processing cores clustered together in what is called a streaming multiprocessor (SMX), a shared level 2 cache, memory controllers, and a PCI Express interface to the host machine. The GPU used in the latest revision of the Kepler architecture is called GK110 [90] and this high-end chip has up to 15 active SMX clusters.

The SMX shown in Figure 2.7 on the GK110 GPU contains 256 separate cores. Of these cores, 192 cores have single-precision integer and floating point ALUs and 64 support double precision. The SMX also includes registers, instruction cache, and 64 kB of shared memory and level 1 cache, shared by all the cores on a single SMX that can be partitioned by the programmer. Each SMX also contains 32 load and store units to provide groups of threads access to DRAM in parallel. Finally, the SMX has 16 special function units (SFUs), used to calculate complex mathematical instructions such as cosines and square roots.

Scheduling on the GPU is done at two levels: High-level scheduling on the entire chip is handled by what Nvidia calls a GigaThread engine. In the documentation, it is also referred to as the Grid Management Unit [90]. This unit controls all the groups of threads executing on the GPU and can manage both CPU- and GPU-generated workloads. Low-level scheduling at the SMX level is carried out by a quad warp scheduler. An SMX schedules threads in groups of 32 parallel threads called warps. The scheduler is capable of selecting four warps in parallel and each warp can dispatch two independent instructions per cycle. All the threads in a warp have to execute the same instruction and branching is not supported. If branching should occur in the code, each of the branches must be evaluated for all the running threads.

The memory hierarchy on GPUs is different from that on CPUs. There are different types of memory with different properties and the memory is explicitly managed by the programmer; thus memory usage will often have an impact on performance. Figure 2.8 shows an overview of the memory hierarchy on the Kepler architecture. The first level in the hierarchy is at the thread level. All the cores on the SMX share a total of 65,536 32-bit registers. *Registers* are the fastest memory type on the GPU, with access times of one clock cycle. The challenge with the registers is that their number is limited and if the threads use up all the registers, the overflow data will be stored in what is called local memory. *Local memory* is private to each thread and resides in the DRAM, which is described later. The second level in the memory hierarchy involves shared memory, level 1 cache, and a

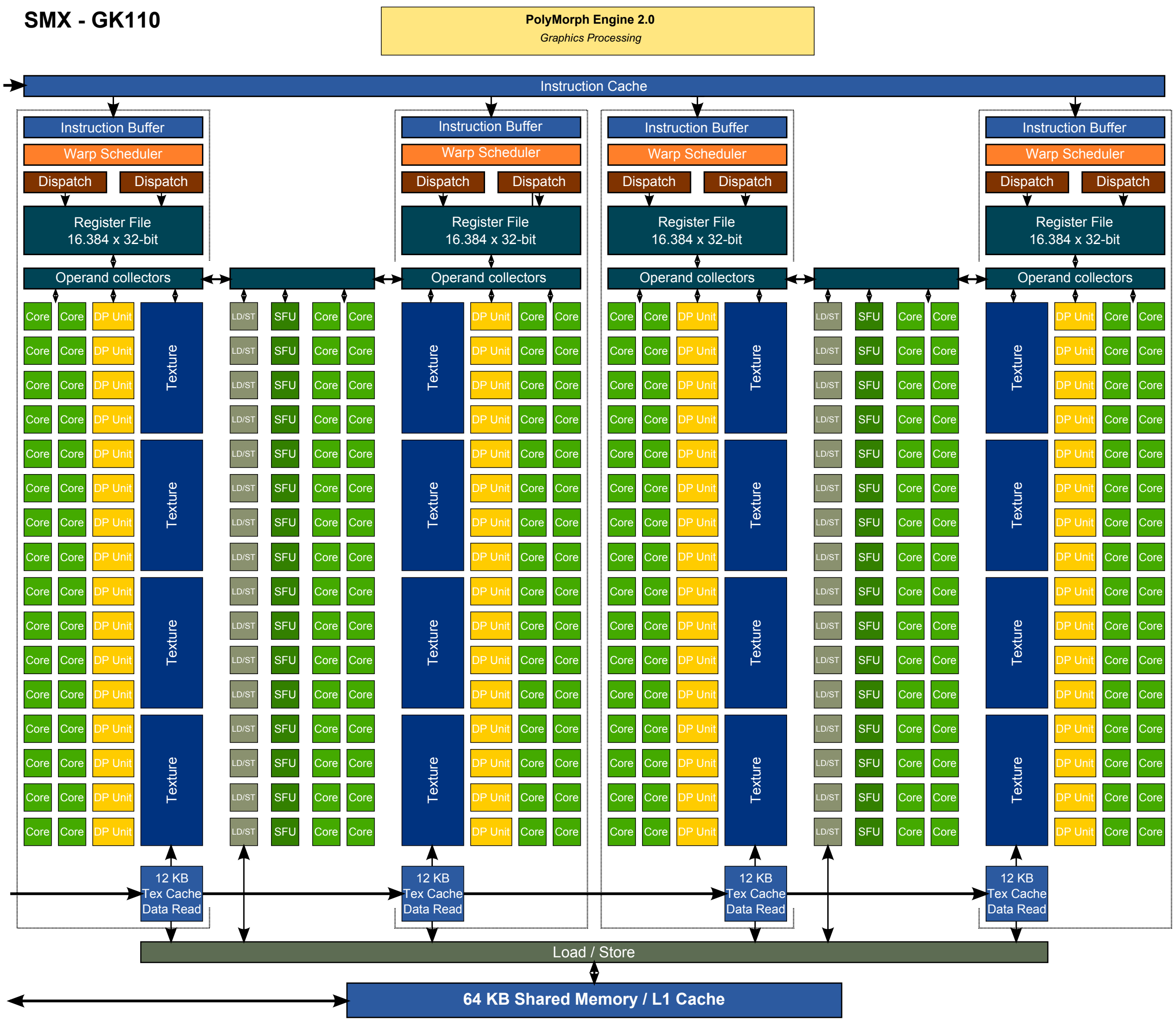


Figure 2.7: Nvidia GK110 SMX architecture [90], slightly modified.

read-only data cache. This level is also on chip within each SMX. The on-chip memory in the SMX can be dynamically partitioned by the programmer between *level 1 cache* and *shared memory*. The access time is also a single clock cycle and the memory is accessible by all threads running on the same SMX. Shared memory provides the programmer fast memory for sharing data and reducing the need for slow off-chip memory access. However, the shared memory is uniformly divided into banks and the throughput is dependent on the data layout. The Kepler architecture also introduced a 48-kB *read-only data cache*, which in previous GPU generations were only accessible by the texture units. This data cache does not have the same bank structure as the shared memory and can support full-speed unaligned memory access patterns. At the third level, the Kepler architecture has 1536 kB of *level 2 cache*. This cache is shared among all SMX units on the GPU and services all load, store, and texture requests to DRAM, enabling more efficient data sharing across the GPU. Programmers are not allowed to explicitly control the level 1 or level 2 cache. The last level in the memory hierarchy is the DRAM. The DRAM is built up of three different memory spaces: global memory, texture memory, and constant memory. With Kepler, the DRAM is off chip and is significantly larger and much slower

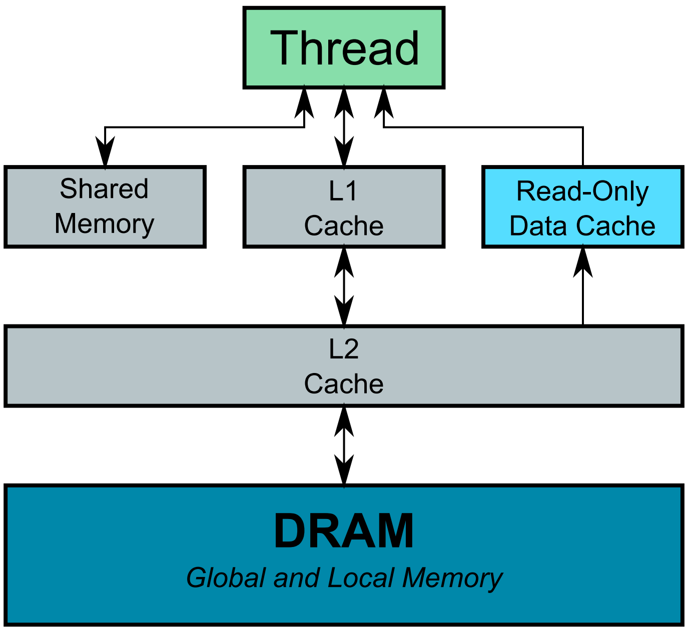


Figure 2.8: The Kepler memory hierarchy.

compared to the higher levels in the hierarchy. The DRAM is often referred to as *global memory*. Global memory can be accessed by all SMX units and can be accessed with 32 , 64-, or 128-byte transactions, given that the access address is aligned with the transaction size. The memory controller on the GPU tries to combine multiple operations into fewer and larger transactions, which is known as memory coalescing. In early GPU generations, memory coalescing was very important; however, with more advanced memory controllers that are not only optimized for graphical workloads, the demands on memory operations have been relaxed.

The two other memory spaces in DRAM are constant and texture memory. Both are read only by the GPU, meaning that they have to be allocated and written to by the host CPU. Both these memory spaces are cached and, depending on the access patterns, this can reduce access time to the data. *Constant memory* is limited to 64 kB in total and *texture memory* is limited to 48 kB per SMX unit. If the data accessed in both constant and texture memory are not in the cache, the GPU must fetch the data from DRAM with the same access time as regular global memory.

The GK110 chip is used in two different product lines: as a dedicated compute coprocessor for general-purpose GPU workloads (Nvidia Tesla) and as a graphics card (Nvidia GeForce and Quadro) for playing computer games and running other 3D applications. The chip itself is comprised of 7.1 billion transistors and has up to 2880 cores and up to 12 GB of on-board DRAM. With a typical power consumption of 250 watts, it can deliver up to 5121 tera floating point operations per second (TFLOPS) of single-precision processing power and 1707 TFLOPS of double-precision processing power.

2.1.4 STI Cell Broadband Engine

The CBE [54] is a heterogeneous multicore processor developed in cooperation by Sony, Toshiba, and IBM. The project was started by Sony in 2000 when they requested a CPU for the new PlayStation 3 game console. The design goal of the CBE was a processor 1000 times faster than the Emotion Engine from the PlayStation 2 game console [7]. Development of the CBE started in 2001 and the first product to utilize the processor was Sony's PlayStation 3 game console. The CBE was later also used in high-performance computing, when IBM launched several blade servers featuring multiple Cell processors. The chip was also used by Toshiba in laptops and HDTVs as an accelerator for offloading media processing, such as in decoding, upscaling, and post-processing.

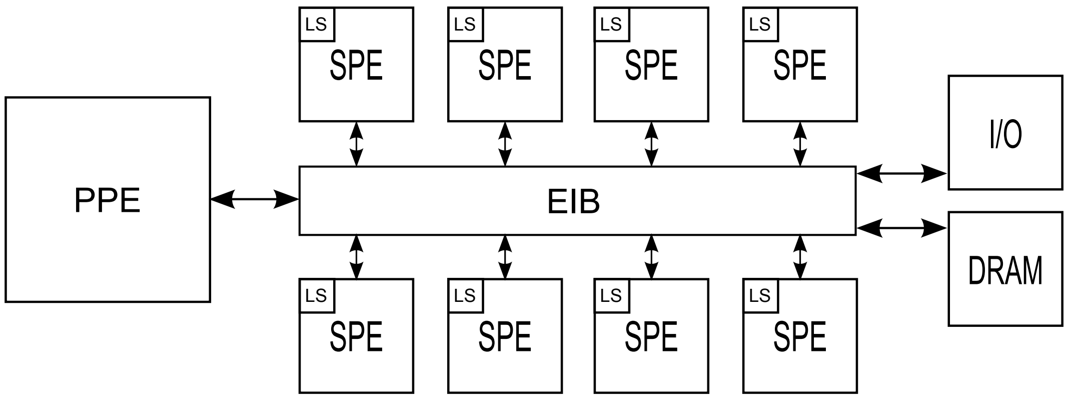


Figure 2.9: CBE architecture.

The CBE consists of several main components: a power processing element (PPE) for general-purpose processing, up to eight synergistic processing elements (SPEs) for high-throughput computations, a FlexIO interface for connecting multiple processors and devices such as network and disk controllers, and, finally, a memory controller. All the components on the chip are connected by the element interconnect bus. An overview of the CBE architecture is shown in Figure 2.9.

The general-purpose processor on the CBE, called the PPE, is based on a standard IBM PowerPC 970 [54]. This processor is from the POWER4 family and implements both the 32- and 64-bit PowerPC instruction set and is capable of executing two threads in parallel. The PPE has two levels of cache (32-kB level 1 data cache, 32-kB level 1 instruction cache, and 512-kB level 2 cache), has a simple branch prediction unit, supports virtual memory, and has a vector unit called the VMX. The vector unit on the PPE is a standard IBM AltiVec SIMD unit and it is capable of processing a 128-bit vector with either four independent 32-bit words, eight 16-bit shorts, or 16 eight-bit bytes. The VMX supports both floating point and integer values.

The specialized computational cores in the CBE are called SPEs and an overview of an SPE is shown in Figure 2.10. A single SPE contains a synergistic processor unit (SPU). The SPU has a large 128-entry 128-bit register for vector processing. The SPU is able to execute two hardware threads in parallel; even though its vector unit has much in common with the VMX unit in the PPE, they do not share the same AltiVec instruction set. The SPE also only supports a 32-bit instruction set. The SPU supports both single- and double-precision floating point values. However they are not fully compatible with the IEEE 754 standard for double precision [54]. Support for “not a number” and infinity have been removed to extend the range and numbers are truncated downward, toward zero. The memory flow controller in the SPE cannot directly access data in the main system memory; it can only access data in a small, 256-kB local storage. The local storage is

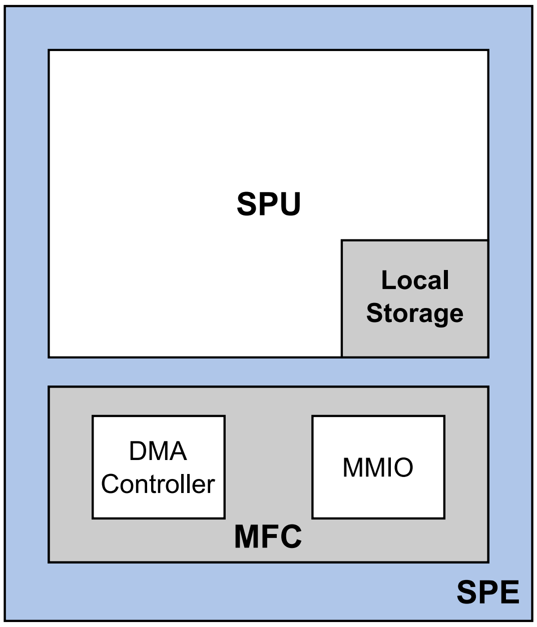


Figure 2.10: Overview of an SPE.

non-coherent and is basically a user-controlled cache, which stores both the code and data that the SPU will process. To copy data between other SPEs, the PPE, or the main memory, the SPEs must use DMA transfers. These transfers can be requested both by the PPE and the SPEs and must be set up explicitly when programming the CBE. The architecture includes hardware support for primitives such as signals, message passing, and atomic updates. There are message queues for communication between the SPEs, PPE, or internally among the SPEs and the queues can be used by either an interrupt handler or polling.

The CBEs used in our research are from the Sony PlayStation 3 game console. The CBE in a PlayStation 3 is clocked at 3.2GHz. One of the SPEs is disabled to improve the yield in the manufacturing process and a second SPE is reserved for a hypervisor, which leaves six SPEs for the user. The system has 256 MB of main memory, where about 40 MB of memory is reserved for the hypervisor. The hypervisor also blocks access to certain parts of the hardware, including the GPU and certain hardware debugging features in the CBE. Early versions of the console also supported booting Linux. In Linux, the CBE can be programmed using the standard GNU Compiler Collection to generate code for both the PPE and SPE.

2.1.5 Other Hardware Architectures

There are also several other interesting hardware architectures from a heterogeneous standpoint. In this section, we mention a couple of other architectures that we have not been able to investigate during work on this thesis and we take a brief look at their main features.

Field-Programmable Gate Array (FPGA)

A FPGA is an integrated circuit that is designed to be configured by users. To configure an FPGA, hardware description language is used. The process is similar to that used when making dedicated application-specific integrated circuits for specific applications. The FPGA contains programmable logic blocks and a reconfigurable interconnect to connect the blocks. After a schematic hardware description language design is completed, an electronic design tool must be used to generate a netlist, which describes how the logic blocks are supposed to behave and how they are connected. Finally, the design must be validated with respect to the placement and timing of the chip.

From a technical standpoint, an FPGA can be used to solve any problem that is computable. One of the challenges with FPGAs is that the programmer has to break down the problems to the logical gate level. Since all the logic gates on an FPGA work in parallel, programmers must also take into account synchronization issues. For complex workloads, another challenge is that only a limited number of logic blocks are available on an FPGA and the blocks cannot be efficiently reconfigured during execution. [73]

Modern FPGAs such as the Xilinx Zynq [135] series have started to combine traditional design with logic blocks and interconnects with an embedded ARM microprocessor. This design is often referred to as a system on a programmable chip. It makes it easier to reconfigure the chip during runtime and programmers can use the programmable part of the chip for parallel parts of their applications. We have not looked into FPGAs. The programming model is very different from that of traditional multicore systems and we do not have any experience with hardware design.

Very Long Instruction Word (VLIW) Architectures

A VLIW architecture is a type of processor architecture that is designed to take advantage of the instruction-level parallelism available in program code. The idea behind VLIW is to enable programs and compilers to explicitly specify which instructions can execute in parallel. The idea is not new, as described in Section 2.1.1. For example, x86 processors use their superscalar architecture to execute independent instructions in different parts of the processor and out-of-order execution reorders instructions to improve efficiency. The drawbacks with both these approaches is that they make the hardware more complex, resulting in larger circuits and higher power consumption. With VLIW, the processor executes operations in parallel based on a fixed schedule generated when the program is compiled. For even better efficiency, hints can be given to the compiler.

The VLIW concept was first invented in the early 1980s [36]. The first implementation of VLIW was Intel's first 64-bit processor, called the i860, released in 1989 [71]. The only VLIW processor architecture produced today is the Itanium architecture from Intel and it is used only in enterprise-class server systems. We have therefore not investigated this architecture further. Graphics processors from AMD also used VLIW [51]; however, these have since been replaced by a RISC SIMD architecture called Graphics Core Next.

NEC SX Vector Supercomputer Architecture

The SX series architecture from NEC involves a dedicated vector processor. The latest iterations of the processor are the SX-ACE and SX-9 processors [142]. Each core contains four vector processing units. Each of the vector processing units has a 16-stage pipeline and 16 vector registers that can store 256 64-bit words. Many of the details about how the cores operate are not public. The theoretical performance of a chip with four cores is 256 GFLOPS and multiple chips can be connected in multinode systems with SMP.

The architecture is highly optimized toward vector processing and, as a result, scalar programs do not scale well on this architecture. The systems are shipped with SUPER-UX, with is a custom UNIX operating system maintained by NEC. One of the main applications for the SX architecture is to run simulations of complex climate models for meteorological use [107]. The SX architecture is not considered in this thesis, since it is only used in high-end enterprise-class servers.

2.1.6 Summary

In this section, we introduced the four heterogeneous architectures that we are going to use for our experiments in this thesis. The architectures are very different but they share some properties. Table 2.3 compares the state-of-the-market products in each of the heterogeneous architectures.

Feature	**IXP**	**x86**	**Cell**	**GPU**
General-purpose cores	1	1–16	1	0 [†]
Vector instruction set	No	256-bit AVX2	128-bit AltiVec	No [♯]
Specialized cores	8	0	8	192–2880
Instruction set	ARMv5/ μC	x86	PowerPC	PTX
SMT (multithreading)	No	Yes	Yes	No
Memory model	Shared	Shared	Exclusive	Exclusive
Cache coherency	Yes	Yes	No [‡]	No
On-chip memory	Yes	No	Yes	Yes
Off-chip memory	Yes	Yes	Yes	Yes
Memory types	SRAM/ DRAM	DRAM	DRAM	DRAM
Branch prediction	Limited	Yes	Limited	Limited
Cache hierarchy	L1	L1–L2–L3	L1–L2	L1–L2
User-controlled cache	No	No	Yes	Yes [♭]
Active development	Yes	Yes	No	Yes

[†] The GPU needs a host CPU to operate, but there is no CPU on the card.

[♯] Not exposed outside the driver.

[‡] Coherence between multiple Cell processors, but not with SPEs.

[♭] Shared memory is a cache; architecture also has caches not controlled by the programmer.

Table 2.3: Comparison of the four heterogeneous architectures.

The very different properties of the architectures makes their utilization challenging for programmers. Architectures such as the x86 use a shared memory model, where the memory management unit on the processor takes care of all the data transport between the cores and caches. However, on architectures such as the Cell and GPUs, which have an exclusive memory model, programmers need to carefully consider the data flow of the program. Even GPUs and the IXP have multiple memory spaces available that are more suited for some operations. The properties of these memory types on the GPU are investigated further in Section 3.3.1. The number of cores in these architectures is also very different. On x86 processors and on the Cell, programmers typically need tens of threads for optimal performance, whereas on a GPU programmer need thousands and perhaps tens of thousands of threads to obtain optimal performance. Finally, all the architectures have different instruction sets, so there will be very little portability of code between the architectures. This is further explored in Chapter 3.

2.2 Hardware Abstractions and Programming Models

In Section 2.1, we described several modern heterogeneous processing architectures used in our research on processing multimedia workloads. If we look at the processing cores, memory, caches, and buses that connect them, these architectures are very different. However, the hardware abstractions and programming model used to expose the hardware to programmers do not change that often.

One example where a hardware abstraction is used to expose the hardware differently to the programmer and to the operating system is SMT. SMT is used both in x86 cores and on the PPE in the CBE. The basic idea behind SMT is that a single processing core is exposed as two or more separate cores to programmers and to the operating system. Another example in which hardware abstractions are hiding the underlying hardware is the GPUs from Nvidia described in Section 2.1.3. The last-generation Kepler GK110 GPU uses the programming model called single instruction, multiple threads (SIMT), which was already used by the first programmable Tesla G80 GPU released in 2006. The underlying hardware between these two GPUs has, however, changed considerably.

2.2.1 SMT

SMT is a technique used to improve the throughput of a superscalar CPU pipeline. When a single thread is running and stalls due to a cache miss or any other high-latency instruction, it leaves parts of the processor idle. With SMT, a single processor core is exposed to the operating system as two or more cores and the hardware tries to efficiently utilize all the resources in the superscalar pipeline, as shown in Figure 2.11.

Intel calls this Hyper-Threading Technology. It was first used in 2002 in the Pentium 4 architecture to expose a single CPU core as two virtual cores. The same technology is also used in the Intel Xeon Phi many-core architecture, whereas a single processor core is exposed as four virtual cores on the Phi. In addition to Intel, IBM is using SMT on the PPE PowerPC processing core in the CBE. SMT does not always improve performance. There are several cases in which it can actually reduce performance for both threads running on the processor core. The basic aim of a general-purpose CPU design is to run a single thread as quickly as possible. The cores are often designed with techniques such as out-of-order execution, superscalar pipelines, branch prediction, and prefetching. SMT is often considered the last resort in filling the pipeline to prevent stalls. When two threads compete for resources, they often take more time to finish than if they did not have to share any resources. On the other hand, in on some architectures, such as

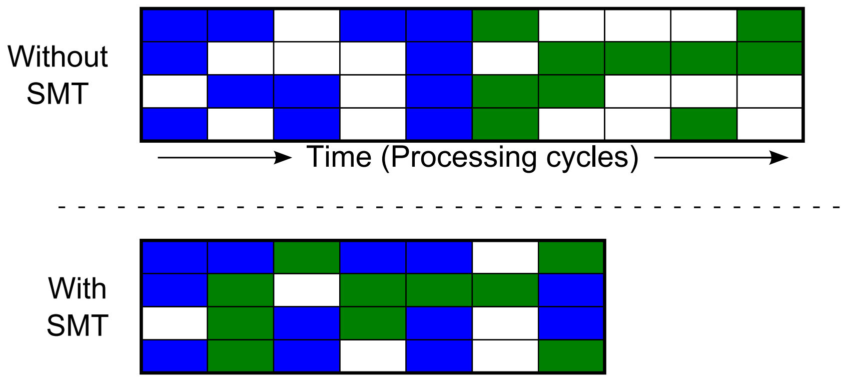


Figure 2.11: A superscalar processor design with and without SMT.

the Xeon Phi many-core architecture, many cores share the memory controller, resulting in greater memory latency. With SMT, the cores are better able to hide some of this memory latency.

2.2.2 SIMD

SIMD, described in Flynn's taxonomy [38] and shown in Figure 2.12, is a technique where multiple processing elements perform the same operation on multiple data points in parallel. SIMD is also often referred to as vector processing, since the multiple data points are often stored in a data vector. The first mainstream use of SIMD was Intel's Pentium processors with MMX extensions, launched in 1996.MMX supports 64-bit long vectors. After Intel released MMX, Motorola and IBM quickly introduced their AltiVec vector extensions for the PowerPC and POWER systems. Since its release, Intel has also improved and extended MMX, first to 128 bits with several versions of Streaming SIMD Extensions and, finally, to 256 bits with two versions of the Advanced Vector Extensions.

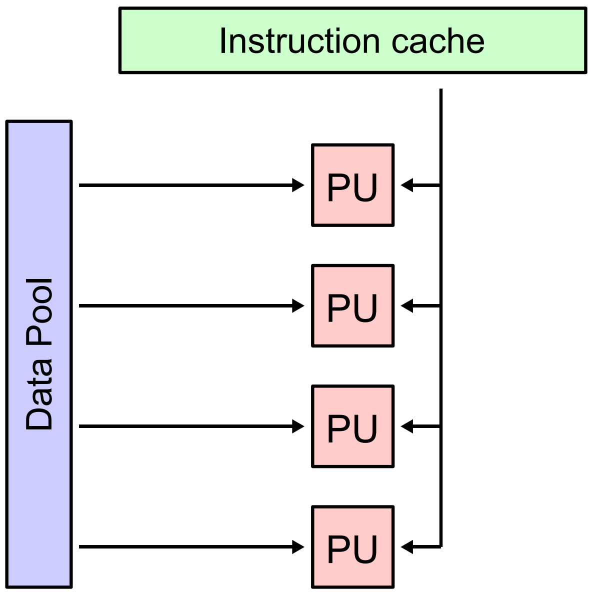


Figure 2.12: SIMD programming model.

Multimedia workloads frequently perform identical operations on large data sets, which is one of the reasons why SIMD was brought to desktop and mobile processors. In the CBE, SIMD is an essential part of the architecture. The PPE has an AltiVec SIMD unit and the SPEs work on a 128-bit vector. Although SIMD instructions became mainstream with the Pentium processor in 1996 and the adoption of the PowerPC for MacOS, their use has been and still is an art. On the Cell, SIMD instructions are used explicitly through vector extensions to C/C++, which allows basic arithmetic operations on vector data types of intrinsic values. This means that the programmer can apply a sequential programming model but needs to adapt the memory layout and algorithms to the use of SIMD vectors and operations. On the x86, programmers also have to explicitly use the SIMD instructions and, even though the compilers are able to auto-vectorize some simple data patterns, these operations generally have to be made manually.

2.2.3 SIMT

The abstraction used when programming GPUs from Nvidia is called SIMT. Nvidia first introduced this model with the CUDA processing framework, released together with the Tesla G80 GPU in 2006. CUDA uses a two-tiered threading model that maps to the architecture. Threads are bundled into groups, which are organized in a grid, as illustrated in Figure 2.13.

The global scheduler on the GPU distributes the groups to available SMXs and all the threads in a group execute on the same SMX. The program that is executed on the GPU

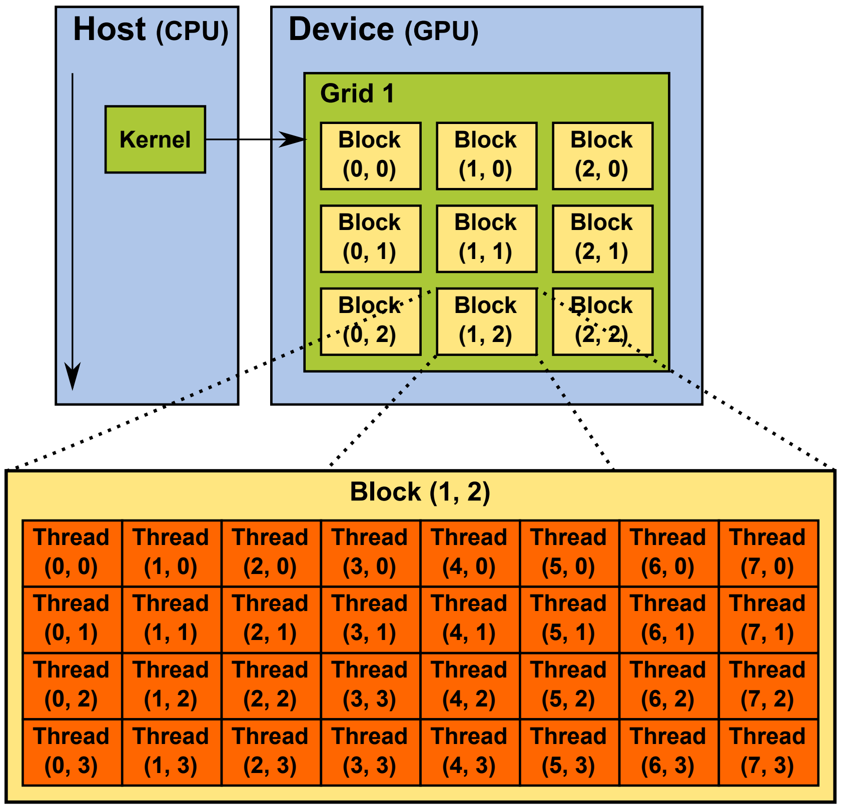


Figure 2.13: Nvidia CUDA programming model.

is called a kernel. It is up to the programmer to choose how the threads are organized. The optimal number of blocks and number of threads per block vary depending on the GPU generation used. The optimal size of these parameters also varies depending on the register space each thread in a block requires. If the register space on an SMX is exhausted, the GPU will use local memory,[1] which is located off chip, resulting in a massive increase in access time. SIMT enables code that uses only well-known intrinsic types that can be massively threaded. Low-level operating system functions schedule these threads in groups called warps (the size of a warp can be hardware specific).The control flow of the threads can diverge as in an arbitrary program. However, this will essentially serialize all the threads in the block, which will impact performance. If none of the threads in the warp diverge, all the threads will execute the same operation. The operation is then performed as a vector operation containing the data of all the threads in the block.

2.2.4 Summary

In this section, we briefly introduced the different programming models used by the architectures in this chapter. SMT is typically used by the architectures with more complex general-purpose cores (i.e., Cell and x86), to try using the execution pipelines more efficiently. For programmers and operating systems, the use of SMT is transparent, which can be challenging, since some applications have shown reduced performance when they have to share resources on the same processor cores.

Both the Cell and x86 architectures use explicit SIMD instructions. This means that the programmer can apply a sequential programming model but needs to adapt (if possible) the algorithms and memory layout to use SIMD vectors and operations. The GPUs from Nvidia use an abstraction called SIMT. Such abstractions enable programmers to write code that uses well-known intrinsic types but which are massively threaded. It is the runtime of the GPU that schedules the threads. In this model, it is possible for the threads to diverge, as in arbitrary programming, even though this will have negative effects on performance. The functionalities provided by SIMD and SIMT are very similar. In SIMD programming, vectors are used explicitly by the programmer, many of whom think in terms of sequential operations on very large operands. In SIMT programming, the programmer can think in terms of threaded operations on intrinsic data types. The SIMT concept has an interesting property: If SIMD is used, the vector width must be known to the programmer. SIMT hides this and the code can be optimized for several vector widths. Even though the functionality is similar, programmers still need to think differently when using these architectures, as demonstrated in our case studies on the different architectures in Chapters 3 and 4.

2.3 Summary

In this chapter, we introduced the heterogeneous hardware architectures and the programming models used to program them. The architectures have very different properties, which makes their utilization challenging for programmers. In the next chapter, we

[1]Local memory in OpenCL is the same as shared memory in CUDA.

look at case studies with simple multimedia workloads running on the different heterogeneous architectures presented in this chapter. These case studies are used to learn how to efficiently use the different architectures.

Chapter 3

Using Heterogeneous Architectures for Simple Tasks

Heterogeneous systems have recently received a lot of attention. They provide more computing power than traditional single-core systems, but their efficient use of available resources is a challenge. On some architectures, the processing cores have different strengths and weaknesses compared to desktop processors. Several different types and sizes of memory are exposed to the developer and limited architectural resources require considerations of data and code granularity.

To learn more about the properties of our heterogeneous architectures, we performed different experiments on the architectures with simple tasks related to multimedia to gain experience. By simple tasks we mean small operations, simple steps, or small parts of a larger complex pipeline. In most of the cases, only the part of the workload running on the heterogeneous architecture was optimized for performance, since we wanted to isolate only this part of the workload. The simple tasks in our investigations ranged from experiments on how to use the different memory spaces on an Nvidia graphics processing unit (GPU) most efficiently to protocol translation on the Intel IXP network processor and offloading parts of Motion JPEG (MJPEG) video encoding pipeline to the single instruction multiple data (SIMD) unit on an x86 processor, the synergistic processing element (SPE) unit on the Cell, or the cores of an Nvidia GPU.

This chapter is organized by the heterogeneous architectures. First, we take a look at the Intel IXP network processor. We then experiment with the Sony–Toshiba–IBM Cell Broadband Engine before we run tests on the x86 processor architectures. Finally, we evaluate the performance of different workloads on GPUs. In all these sections, we take a closer look at each architecture, with one or more case studies. We use these case studies to gain experience on how to efficiently use these architectures for parallel processing. However, not all the workloads have been tested on all architectures.

3.1 Intel IXP Network Processor

The Intel IXP network processor was used in the early stages of this thesis as an architecture that could explore the limits of integrated layer processing [24]. To do this, we used a protocol translation prototype. The IXP card provided early insight into how to program an asymmetric shared memory architecture and experience with video streaming.

In Section 3.1.1, we take a closer look at a case study based on our work with network protocol translation [32,33]. We use an Intel IXP2400 network processor to transparently translate RTP/UDP video streams, which was the popular way of streaming video a decade ago, to HTTP/TCP, which is the de facto solution today.

3.1.1 Case Study: Network Protocol Translation

In this section, we describe our implementation of a dynamic transport protocol translator on an Intel IXP2400 network processor. The IXP architecture was used in the early stages of the research work to gain experience with a heterogeneous architecture. Streaming services are available almost everywhere today. Major newspapers and TV stations provide on-demand and live video content. Video on-demand services are common and even personal media are frequently streamed using services such as YouTube.

Setting

The debate about the best protocols for streaming has been going on for years. Initially, streaming services on the Internet used UDP for data transfer because multimedia applications often have demands for bandwidth, reliability, and jitter that could not be offered by TCP. Today, this approach is hampered by the filters of Internet service providers (ISPs) and by firewalls in access networks and in end systems. ISPs reject UDP because it is not fair to accept it over TCP traffic and many firewalls reject UDP because it is connectionless and requires too much processing power and memory to ensure security. It is therefore fairly common to use HTTP streaming, which delivers streaming media over TCP. The disadvantage is that the end user can experience playback hiccups and quality reduction because of the probing behavior of TCP congestion management, leading to oscillating throughput and slow packet rate recovery. A sender who uses UDP would, in contrast, be able to maintain a constant desired sending rate. Servers are also expected to scale more easily when sending smooth UDP streams and avoid dealing with TCP-related processing.

To explore the benefits of both TCP and UDP, we experiment with a proxy that carries out a transparent protocol translation. This is similar to the proxy caching ISPs employ to reduce their bandwidth and we do, in fact, aim at a combined solution. There are, however, too many different sources of adaptive streaming media for end users to apply proxy caching for all of them. Instead, we aim at live protocol translation in a TCP-friendly manner that achieves high perceived quality for end users. Our prototype proxy is implemented on the Intel IXP2400 network processor and enables the server to use UDP on the server side and TCP on the client side.

Preliminary tests comparing HTTP/TCP video streaming from a web server and RTSP/RTP/UDP streaming from a Komssys video server [45] show that, in case of loss, our solution using a UDP server and a proxy later translating to TCP delivers a smoother stream at the playout rate while the TCP stream oscillates heavily.

Workload: Translating Proxy

An overview of our protocol translating proxy is shown in Figure 3.1. The client sends a GET request, which is translated to RTSP by the proxy. The proxy then generates the

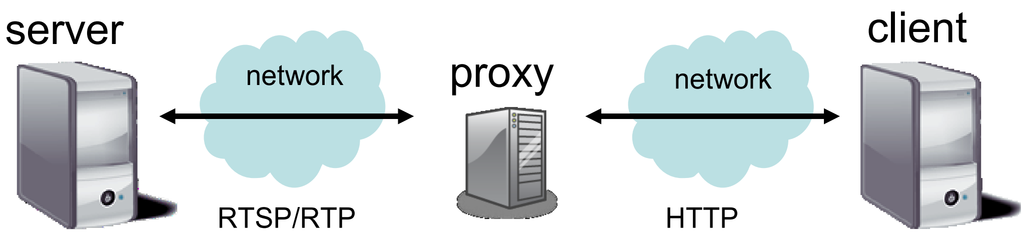


Figure 3.1: Overview of the streaming scenario.

pipeline and starts to push data. The TCP connection between the proxy and client is assumed to be fast enough. If not, the proxy will drop packages before the TCP sequence numbers have arrived. Note that both peers are unaware of each other. The server assumes the client uses UPD and vice versa.

The steps and phases of a streaming session are as follows. The client sets up a HTTP streaming session by initiating a TCP connection to the server; all packets are intercepted by the proxy and modified before being passed on to the streaming server. The proxy also forwards the TCP three-way handshake between the client and server, updating the packet with the server's port. When established, the proxy splits the TCP connection into two separate connections that allow for the individual updating of sequence numbers. The client sends a GET request for a video file. The proxy translates this into a SETUP request and sends it to the streaming server using the TCP port of the client as its proposed RTP/UDP port. If the setup is unsuccessful, the proxy will inform the client and close the connection. Otherwise, the server's response contains the confirmed RTP and RTCP ports assigned to a streaming session. The proxy sends a response with an unknown content length to the client and issues a PLAY command to the server. When received, the server starts streaming the video file, using RTP/UDP. The UDP packets are translated by the proxy as part of the HTTP response, using the source port and address matching the HTTP connection. Because the RTP and UDP headers combined are longer than a standard TCP header, the proxy can avoid the penalty of moving the video data in memory, thus permitting reuse of the same packet by padding the TCP options field with NOPs. When the connection is closed by the client during or after playback, the proxy issues a TEARDOWN request to the server to avoid flooding the network with excess RTP packets.

Implementation

Our prototype is implemented on a programmable network processor using the IXP2400 chipset [57]. The chipset is Intel's second-generation, highly programmable network processor and is designed to handle a wide range of access, edge, and core network applications. A more detailed overview of the architecture is given in Section 2.1.2.

The transport protocol translation operation is shown in Figure 3.2. The protocol translation proxy uses the XScale core and one micro engine (μEngine) application block. In addition, we use two μEngines for the receiving (RX) and the sending (TX) blocks. Incoming packets are classified by the μEngine based on the header. The RTSP and HTTP packets are queued for processing on the XScale core (control path), while the

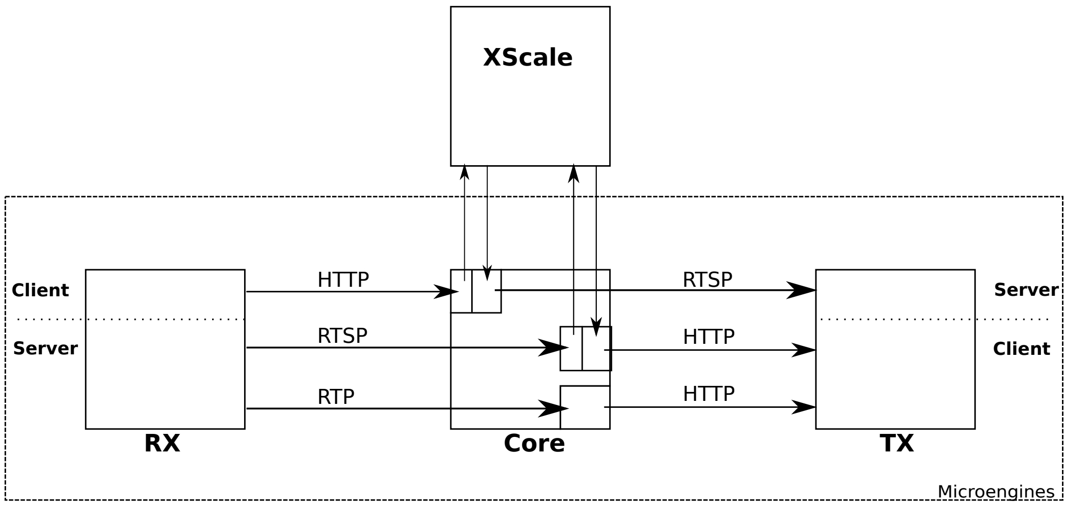


Figure 3.2: Packet flow on the Intel IXP2400.

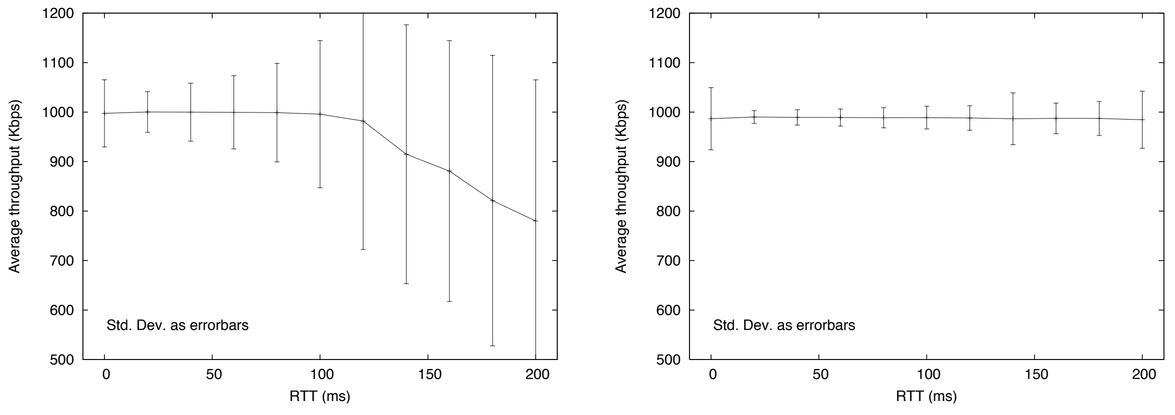
RTP packets are handled on the μEngine (fast path). TCP acknowledgments with a zero payload size are processed on the μEngine for performance reasons.

The main task of the XScale is to set up and maintain streaming sessions, but after initialization, all video data are processed (translated and forwarded) by the μEngines. The proxy supports partial TCP/IP implementation, covering basic features. This is done, to save both time and resources on the proxy.

Experiments

We investigated the performance of our protocol translation proxy compared to plain HTTP streaming in two different settings. In the experiment shown in Figure 3.3, we induced unreliable network behavior between the streaming server and the proxy, while in the second experiment, the unreliable network connected the proxy and the client. We performed several experiments where we examined both the bandwidth and the delay while changing both the link delays (0–200 ms) and the packet drop rate (0–1%). We used a web server and an RTSP video server using RTP streaming, running on a standard Linux machine. Packets belonging to end-to-end HTTP connections made to port 8080 were forwarded by the proxy, whereas packets belonging to sessions initiated by connections made to port 80 were translated. The bandwidth was measured on the client by monitoring the packet stream with tcpdump [121]. We include only the server–proxy loss experiments in this thesis. For more details about the TCP congestion control implementation and the full evaluation, see paper I [32].

The results from the test where we introduced loss and delay between the server and the proxy are shown in Figure 3.3. The plot shows that our proxy that transparently translates from RTP/UDP to TCP achieves a mostly constant rate for the delivered stream. Sending the HTTP stream from the server, on the other hand, shows large performance drops when the loss rate and the link delay increase.



(a) HTTP streaming (b) Protocol translation

Figure 3.3: Achieved bandwidth, varying drop rate and link latency with 1% server–proxy loss.

Discussion

Even though our proxy seems to provide better, more stable bandwidths, there is a trade-off, because instead of retransmitting lost packets (and thus old data if the client does not buffer), the proxy fills the new packet with new, updated data from the server. This means that the client in our prototype does not receive all the data and artifacts may be displayed. On the other hand, in case of live and interactive streaming scenarios, delays due to retransmission may introduce dropped frames and delayed playout. This can cause video artifacts, depending on the codec used. However, this problem can be easily reduced by adding a limited buffer per stream sufficient for one retransmission on the proxy.

One issue in the context of proxies is where and how the proxy should be implemented. For this study, we chose the IXP2400 platform, since we explored the offloading capabilities of such programmable network processors earlier. With such an architecture, the network processor is suited for many similar operations and the host computer could manage the caching and persistent storage of highly popular data served by the proxy itself. However, the idea itself could also be implemented as a user-level proxy application or integrated into the kernel of an intermediate node performing packet forwarding packets at the cost of limited scalability and potentially greater latency.

Both TCP and UDP have their strengths and weaknesses. In this case study, we used a proxy that carried out transparent protocol translation to utilize the strengths of both protocols in a streaming scenario. It enabled the server to use UDP on the server side and TCP on the client side. The server gained scalability by not having to deal with TCP processing. On the client side, the TCP stream was not discarded and passed through firewalls. The experimental results show that our protocol transparent proxy achieved translation and delivers smoother streaming than HTTP streaming.

Summary

For the context of this thesis, we learned that the IXP is a complicated architecture to work with. Writing the network translation proxy requires detailed knowledge about the platform. Another observation is that, when working with a cutting-edge architecture, the quality of the documentation and compilers can be a challenge.

3.1.2 Implications

The IXP2400 is an asymmetric multicore architecture and has processing elements with different capabilities. The XScale core is a general-purpose ARM11 core and the μEngines are specialized cores built for packet processing. This architecture is a shared memory architecture, meaning that all the cores (both XScale and μEngines) have access to the same memory. This is very convenient for the programmer when developing applications, since inter-process communication is very simple. However, there are challenges with shared memory architectures and you might end up with unnecessary transfers to prefetch local caches as a result. In addition, cache coherency protocols can consume resources on the interconnects between the processing elements, preventing efficient performance scaling.

The asymmetric nature of the IXP2400 chip can also be a disadvantage. The specialized cores often require special compilers, application programming interfaces (APIs), and tools to write applications. This was an issue with the IXP2400 chip when we conducted experiments on the platform, where lack of documentation and subpar, buggy compilers can create problems for a programmer.

Writing an application for the IXP2400 also requires the programmer's detailed knowledge about the architecture. One of the main advantages with this architecture is the ability to analyze and manipulate network packets at full line speed (1 Gbps) while adding as little latency as possible to the stream. To achieve this, programmers need to know the different memory types on the board and how to use them. The architecture also has many special features, such as a hardware thread context switch that waits until after memory fetch/store completion. A disadvantage is that one loses almost all portability with an application written for an architecture such as this. When moving to the next generation of hardware, an application might require a complete rewrite.

Revisiting with State-of-the-Market Hardware

When these experiments were conducted in 2006 and 2007, it was not possible to write a protocol translation proxy that could run on a standard Linux desktop PC at the full line speed of 1 Gbps. With recent general-purpose hardware such as the Intel Haswell architecture, it is possible to do this with software executing on the central processing unit (CPU). On the other hand, network processors have also evolved. Netronome, the company that bought the IXP technology from Intel, has continued developing both hardware and software. Their adapters are now capable of processing two 100-Gbps fiber links at line speed [82]. This is not possible on a desktop PC today.

Another possibility would be to use another heterogeneous architecture to process the network traffic. Han et al. [47] have shown that a software router implemented on an Nvidia Fermi GPU can forward data at a rate of 39 Gbps, outperforming a CPU-based software router by a factor of four. A GK110 Kepler GPU, which is the latest GPU generation, would do this even faster. However, a GPU implementation will have some of the same challenges as our IXP2400 implementation. Heterogeneous architectures provide great flexibility and performance, but often at the cost of portability.

3.2 x86 Processor Architecture

The x86 processors in modern computers are a symmetric multicore CPU with a shared memory model. In the symmetric multicore processing (SMP) model, the cores should theoretically be identical. However, many architectural details, such as the underlying hardware architecture, are hidden from programmers. We therefore want to learn more about the architecture.

The x86 architecture can have a different degree of connectivity to the other cores. Cores can be on the same chip, in the same package, in different sockets on the same machine, and in some cases even distributed over multiple machines with interconnects such as Numascale [86]. The system memory can also be segmented in such a way that access to other parts of system memory may have to traverse through other cores via the processor interconnect. In our experiments, x86 processors are also often a part of a heterogeneous architecture, with one or more GPUs connected over PCI Express as a coprocessor.

In this section, we take a closer look at three case studies. The first case study presented in Section 3.2.1 is based on our article "Tips, Tricks and Troubles: Optimizing for Cell and GPU" [112]. Even though that research paper was mainly focused on the Cell and GPUs, we also conducted experiments on the efficiency of discrete cosine transformation (DCT) algorithms and the effect of using SIMD on the x86. The second study presented in Section 3.2.2 is based on our work on multi-rate encoding with the VP8 codec [34, 35]. In these experiments, we use the shared memory model of the x86 architecture to reuse parts of the computational heavy analysis stage of the video encoder. In the final study in Section 3.2.3, we investigate the parallel execution of a game server [102], using a thread pool to execute lightweight game server -related tasks running on a multi-socket x86 SMP system.

3.2.1 Case study: Motion JPEG Encoding

We want to learn how to *think* when the multicore system at our disposal is a Cell, x86, or GPU. We aim to understand how to use the resources efficiently and point out tips, tricks, and problems as a small step toward a programming framework and a scheduler that parallelizes the same code efficiently on several architectures. Specifically, we look at effective programming for the workload-intensive yet relatively straightforward MJPEG video encoding. This task consumes many CPU cycles in the sequential DCT, quantization, and compression stages. On single-core systems, it is almost impossible to process a 1080p high-definition (HD) video in real time, so it is reasonable to apply multicore computing in this scenario.

Workload: MJPEG

The MJPEG format is widely used by webcams and other embedded systems. It is similar to video codecs such as Apple ProRes and VC-3, used for video editing and post-processing due to their flexibility and speed— hence the lack of inter-prediction between frames. As shown in Figure 3.4, the encoding process of MJPEG comprises the splitting of video frames into 8x8 macroblocks, each of which must be individually transformed to the frequency domain by forward DCT and quantized before the output

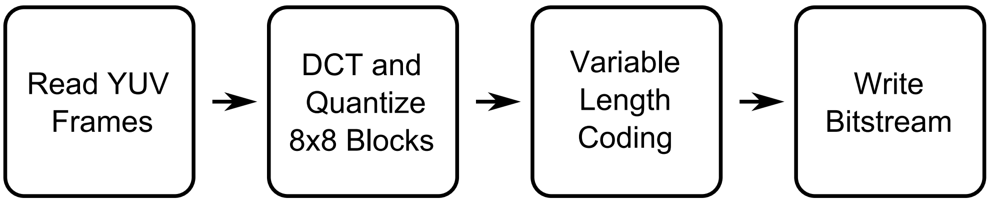


Figure 3.4: Overview of the MJPEG encoding process.

is entropy coded using variable-length coding (VLC). JPEG supports both arithmetic coding and Huffman compression for VLC and our encoder uses predefined Huffman tables for the compression of the DCT coefficients of each macroblock. The VLC step is not context adaptive and macroblocks can thus be independently compressed. The length of the resulting bitstream, however, is probably not a multiple of eight and most such macroblocks must be completely bit-shifted when the final bitstream is created.

The MJPEG format provides many layers of parallelism; starting with the many independent operations in calculating DCTs, the macroblocks can be transformed and quantized in arbitrary order and frames and color components can be encoded separately. In addition, every frame is entropy coded separately. Thus, many frames can be encoded in parallel before the resulting frame output bitstreams are merged. This provides a very fine level of granularity for parallel tasks, providing great flexibility in implementing the encoder.

The forward two-dimensional (2D) DCT function for a macroblock is defined in the JPEG standard for image component $s_{y,x}$ to output DCT coefficients $S_{v,u}$ as

$$S_{v,u} = \frac{1}{4} C_u C_v \sum_{x=0}^{7} \sum_{y=0}^{7} s_{y,x} cos \frac{(2x+1)u\pi}{16} cos \frac{(2y+1)v\pi}{16}$$

where $C_u, C_v = \frac{1}{\sqrt{2}}$ for $u, v = 0$ and $C_u, C_v = 1$ otherwise. The equation can be directly implemented in an MJPEG encoder and is referred to as 2D plain. The algorithm can be sped up considerably by removing redundant calculations. One improved version that we label one dimensional (1D) plain uses two consecutive 1D transformations with a transposition operation in between and after. This avoids symmetries and the 1D transformation can be optimized further. One optimization uses fast DCT, the Arai–Agui–Nakajima [5], or AAN, further refined by Kovac and Ranganathan [72]. Another uses a precomputed 8x8 transformation matrix that is multiplied with the block together with the transposed transformation matrix. The matrix includes the post-scale operation and the full DCT operation can therefore be completed with just two matrix multiplications, as explained by Kabeen and Gent [15]. More algorithms for calculating DCT exist, but they were not covered in our experiments.

x86 Experiments

The first x86 experiments investigated the efficiency of choosing the correct algorithm for the platform. We implemented the different DCT algorithms as scalar single-threaded versions on an Intel Core i5-750 based on the Nehalem microarchitecture. The performance details for encoding HD video were captured using Oprofile and can be seen in Figure 3.5.

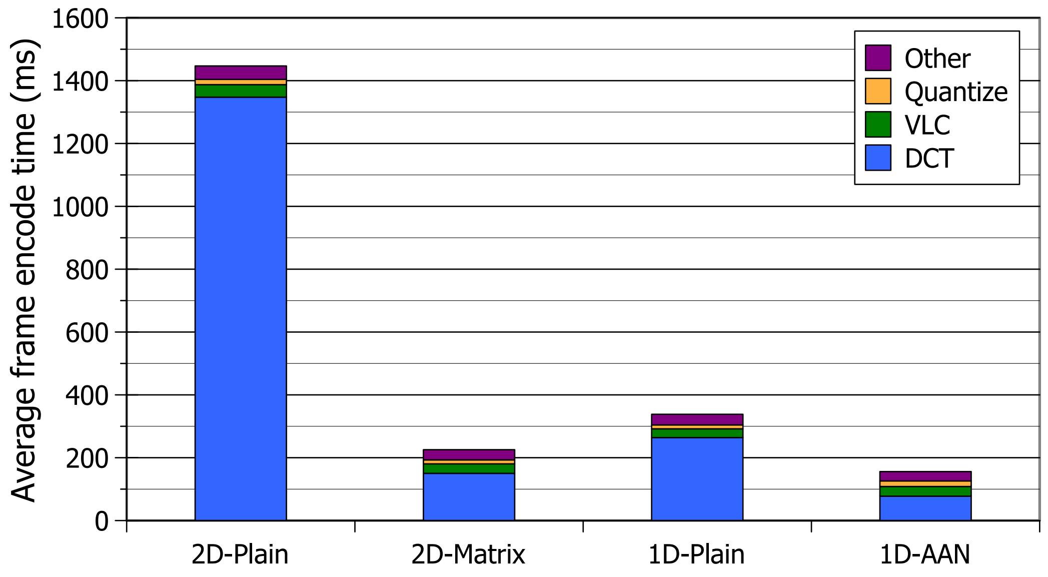


Figure 3.5: MJPEG encoding time on a single-thread x86.

The plot shows that the 1D AAN algorithm using two transposition operations was the fastest in this scenario, with the 2D matrix version second fastest. The average encoding time for a single frame using a 2D-plain arrangement is more than nine times slower than a frame encoded using 1D AAN. For all algorithms, the DCT step consumed the most CPU cycles.

Using a scalar version of the DCT is not the most efficient use of the execution pipeline on a x86 processor. We therefore took the simple DCT algorithm (2D plain) and optimized it with Streaming SIMD Extension (SSE) vector instructions. The experiments were conducted on an Intel Core i7-3720QM processor and the optimized version of the DCT algorithm used the 128-bit SSE 4.2 instruction set. In this experiment, we use the 1080p standard test sequence "tractor" to benchmark the implementation.

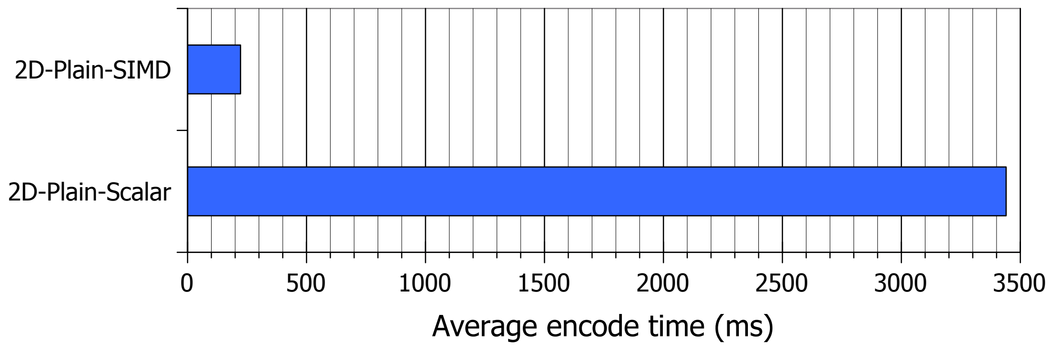


Figure 3.6: Scalar and SIMD versions with a 2D-plain arrangement on an x86.

The results are shown in Figure 3.6. The scalar version of the code that is a straightforward implementation of the 2D-plain algorithm uses around 3500 ms per HD frame. The SIMD optimized version uses only 222 ms. This implementation uses 128-bit SIMD vectors, meaning that we can process four DCT values in parallel, while the scalar version only processes one value at a time.

Summary

These results show that even on the x86 processor architecture, it is important to both choose the right algorithm and optimize the selected algorithm for the platform. However, writing an SIMD version of the x86 code is, as we see later with Cell, a tedious process. Everything must be done by hand and, since not all x86 processors support the same SIMD instructions, you might need multiple versions of the optimization if you want code portability.

3.2.2 Case Study: Multi-Rate Video Encoding with VP8

To learn more about the importance of sharing data between multiple threads and processes when parallelizing multimedia workloads, we investigate how to use the x86 architecture for multi-rate video encoding with the VP8 codec. We use the shared memory architecture of the x86 processors to reuse the computationally expensive analysis step between multiple instances of the VP8 encoder running in different threads.

Setting

The amount of video data available on the Internet is exploding and the number of video streaming services, both live and on demand, is quickly increasing. For example, consider the rapid deployment of publicly available Internet video archives providing a wide range of content such as newscasts, movies, and educational videos. Internet users uploaded 100 hours of video to YouTube every minute in October 2014 [139]. Furthermore, all major (sports) events, such as European soccer leagues, NFL hockey, NBA basketball, and NFL football, are streamed live with only a few seconds' delay to millions of concurrent users over the Internet, supporting a wide range of devices, from mobile phones to HD displays. The number of videos streamed from such services is on the order of tens of billions per month [37, 139] and leading industry experts conjecture that traffic on mobile phone networks will soon be dominated by video content [23].

Adaptive HTTP streaming is frequently used on the Internet and is currently the de facto video delivery solution. For example, Move Networks [81] was one of the first providers of segmented adaptive HTTP streaming, later followed by major actors such as Microsoft [141], DASH [110], Apple [96], and Adobe [2]. In these systems, the bitrate (and thus video quality) can be changed dynamically to match varying bandwidths and CPU resources, providing a large advantage over non-adaptive systems, which are frequently interrupted due to buffer underruns or data loss. The video is thus encoded in multiple bitrates matching different devices and different network conditions.

Today, H.264 is the most frequently used codec. However, an emerging alternative is the simpler VP8, which is very similar to H.264's baseline profile and supposedly well suited for web streaming, with native support in major browsers, royalty-free use, and similar video quality as H.264 [95, 108]. Nevertheless, for both codecs, the challenge in the multi-rate scenario is that each version of the video requires a separate processing instance of the encoding software. This may be a challenge in live scenarios, where all the rates must be delivered in real time, and, in YouTube's case, will require an enormous data center to maintain the upload rate. Thus, the process of encoding videos at multiple levels of quality and data rates consumes both time and resources.

Workload: The VP8 Codec

The VP8 codec [9] was originally developed by On2 Technologies as a successor to VP7 and is a modern codec for storing progressive video. After acquiring On2 Technologies in 2010, Google released VP8 as an open-source *WebM* project, a royalty-free alternative to H.264. The WebM format was later added as a supported format in the upcoming HTML5 standard.

Many of the features in the VP8 codec are heavily influenced by H.264. The VP8 codec has similar functionality as the H.264 baseline profile. One of the differences is that VP8 has an adaptive binary arithmetic coder instead of context-adaptive VLC (CAVLC). However, VP8 is not designed to be an all-purpose codec and it primarily targets web and mobile applications. This is why VP8 has omitted features such as interlacing, scalable coding, slices, and color spaces other than YUV 4:2:0. This reduces encoder and decoder complexity while retaining video quality for the most common use case, that is, making VP8 a good choice for lightweight devices with limited resources.

A VP8 frame is either of the *intra-frame* or the *inter-frame* type, corresponding to I- and P-frames in H.264, but it has no equivalent to B-frames. In addition, VP8 introduces the concept of tagging a frame as *altref* and *golden* frames, which are stored for reference in the decoder. When predicting, blocks may use regions from the immediately previous frame, from the last *golden* frame, or from the last *altref* frame.

The encoding loop of VP8 is very similar to that of H.264. The process consists of an analysis stage, which decides if intra- or inter-prediction will be used, DCT, quantization, dequantization, and inverse DCT (iDCT), followed by an in-loop deblocking filter. The result of the quantization step is entropy coded using a context-adaptive Boolean entropy coder and stored as the output bitstream. The output bitrate of the resulting video is dependent on the prediction parameters in the bitstream and quantization parameters.

Multi-Rate Encoding

The multi-rate encoder is based on the reference VP8 encoder, released as part of the WebM project [9]. Figure 3.7 shows a simplified call graph of the VP8 reference encoder. In this call graph, we can see the flow of the program, how many times a function has been called, and the percentage of execution time spent in different parts of the code. The basic flow of the entire encoder is illustrated in the upper part of Figure 3.8, with an analysis and the encoding part of the pipeline.

The *analysis* part consists of macroblock mode decision and intra/inter-prediction, which corresponds to `vp8_rd_pick_inter_mode` in Figure 3.7. The *encode* part refers to transformation, quantization, dequantization, and inverse transformation, corresponding to the functions `vp8_encode_inter*` and `vp8_encode_intra*` for the various block modes chosen. The *Output* involves entropy coding and writing the output bitstream to a file. This part of the encoder is not shown in the call graph. Profiling of the VP8 encoding process shows that during encoding of the *foreman* test sequence, over 80% of the execution time is spent in the analysis part of the code; that is, if this part can be reused for encoding operations for other rates, resource consumption can be greatly reduced. This

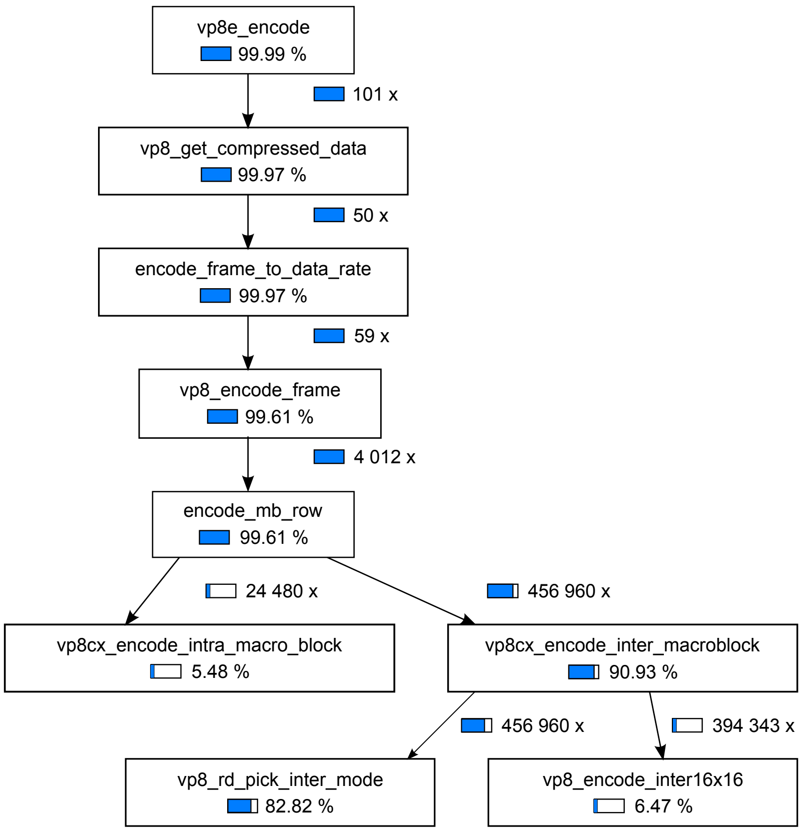


Figure 3.7: Profile of the main parts of the reference VP8 encoder.

can be done because the outputs have identical characteristics except for the bitrate and the main difference between them involves the quantization parameters. Regardless of the target bitrate, the analysis step that includes searching for motion vectors and other prediction parameters can be carried out without considering the target bitrate, trading this for prediction accuracy.

To evaluate this approach, we implemented a VP8 encoder with support for multiple outputs. We reused a single analysis step for several instances of the encoding part, as seen in Figure 3.8. This mitigates the requirements for re-doing the computationally heavy analysis step and at the same time allows the encoding instances to emit different output bitrates by varying the quantization parameters in the encoder step. The encoder starts one thread for each specified bitrate, where each thread corresponds to a separate encoding instance. The instances have identical encoding parameters, such as key frame interval and subpixel accuracy, except for the target bitrate. Since the bitrate varies, each instance must maintain its own state and reconstruction buffers. The threads are synchronized on a frame-by-frame basis, where the main encoding instance analyzes the frame before the analysis computations are made available to the other threads. This process involves macroblock mode decision and intra- and inter-prediction. The non-main encoding instances reuse these computations directly without carrying out the computationally intensive analysis steps. Most notably, `vp8_rd_pick_inter_mode` (Figure 3.7) is only performed by the main encoding instance. Since the VP8 encoder is not written with the intention of running multiple encoding instances in parallel, the encoder goes through

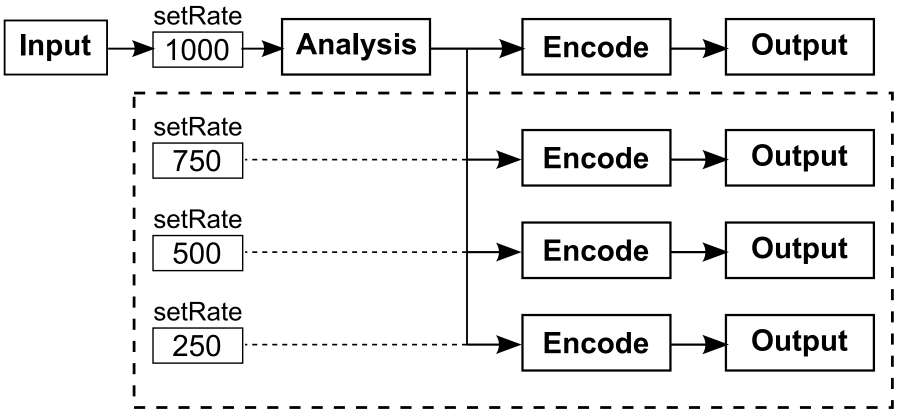


Figure 3.8: Basic flow for the multi-rate VP8 encoder.

significant changes to adapt itself to run multiple instances in parallel.

Experiments

For this case study, we include only one of the experiments that investigated encoding performance at HD resolution. We do not include quality assessment tests, prediction bitrate selection, and the analysis of encoding behavior for different content. These tests are considered out of the scope of this thesis and can be found in paper V [34].

In the HD streaming scenario, we performed experiments using the 1080p resolution test sequence *blue sky* encoded at 1500 kbps, 2000 kbps, 2500 kbps, and 3000 kbps. To measure performance, we used *time* to measure the CPU time consumed. All experiments were run on a four-core Intel Core i5-750 processor based on the Nehalem microarchitecture. This processor does not support SMT.

Figure 3.9 shows the results for the four different output rates. To see if there is a difference for the different *prediction bitrates* chosen when using the multi-rate encoder, we included one test for each prediction bitrate. These results are compared to the combined CPU time used when encoding the videos for the same rates using the reference encoder with both a single thread and multiple threads. The CPU time used in the multi-rate approach needs only 40.5% of the time it takes to encode four sequences using the reference encoder. The multi-rate approach scales further if the number of encoded streams is increased. In addition, the time spent in kernel space is far less in the multi-rate approach compared to the reference encoder and we believe this is a result of reading the source video from disk only once.

Summary

To demonstrate our idea, we implemented a prototype that reuses the most expensive operations based on a performance profile of the encoding pipeline. In particular, our multi-rate encoder reuses the analysis part consisting of macroblock mode decision and intra/inter-prediction. The experimental results indicate that we can encode the different videos at the same rates with approximately the same levels of quality compared to the VP8 reference encoder, while significantly reducing the encoding time.

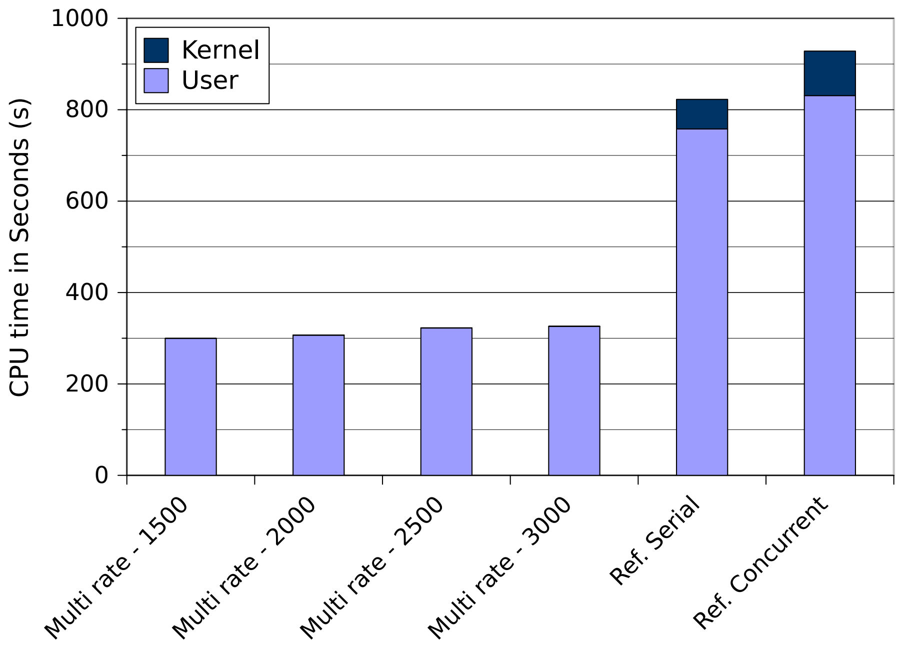


Figure 3.9: CPU time in an HD streaming scenario (blue sky).

We analyzed and performed experiments with Google's VP8 encoder, encoding different types of video at multiple rates for various scenarios. Our main contribution is that we propose a way of reusing decisions from intra- and inter-prediction in the video encoder to avoid computationally expensive steps that are redundant when encoding for multiple target bitrates of the same video object. The method can be used in any video codec, comprising an analysis and encoding step with similar structure as for H.264 and VP8. Furthermore, the method was implemented in the VP8 reference encoder as a case study and the experimental results show that the computational demands are significantly reduced at the same rates and approximately the same quality levels compared to the VP8 reference implementation; that is, for a negligible loss in quality in terms of PSNR, the processing costs can be greatly reduced.

We also learned that the shared memory architecture on the x86 is suited for sharing the data from the computationally expensive steps in the VP8 encoder. Our experiments also show that if we use multiple instances of the reference encoder, the performance is actually better if the workload is executed sequentially instead of concurrently. This is due to greater contention in both the operating system scheduler and on buses, caches, and execution resources on the CPU.

3.2.3 Case Study: Parallel Execution of a Game Server

Many multimedia workloads are massively parallel and, when such workloads are optimized on the x86 architecture, the number of threads used is an important parameter. Too many threads will result in decreased performance due to the context switching overhead in the operating system. With our game server workload, we want to investigate this threshold on the x86 architecture.

Setting

Over the last decade, online multiplayer gaming has experienced amazing growth. Providers of the popular online games must deliver reliable service to thousands of concurrent players, meeting strict processing deadlines for the players to have an acceptable quality of experience.

One major goal for large game providers is to support as many concurrent players in a game world as possible while preserving strict latency requirements for the players to have an acceptable quality of experience. The load distribution in these systems is typically achieved by partitioning game worlds into areas of interest to minimize message passing between players and allow the game world to be divided between servers. Load balancing is usually completely static, where each area has dedicated hardware. This approach is, however, limited by the distribution of players in the game world and the problem is that the distribution of players is heavily skewed, with about 30% of players in 1% of the game area [20]. To handle the most popular areas of the game world without reducing the maximum interaction distance for players, individual spatial partitions cannot be serial. The most CPU-intensive loads for a massively multiplayer online game (MMOG) server are in situations in which the players experience the most "action." Hence, the worst-case scenario for a server is when a large proportion of the players gather in a small area for high-intensity gameplay.

The traditional design of MMOG servers relies on *sharding* for further load distribution when too many players visit the same place simultaneously. Sharding involves making a new copy of an area of a game, where players in different copies are unable to interact. This approach eliminates most requirements for communication between the processes running individual shards. An example of such a design can be found in Chu et al. [22].

The industry is now experimenting with implementations that allow for greater levels of parallelization. One example is Eve Online [30], which avoids *sharding* and allows all players to potentially interact. Large-scale interactions in Eve Online are handled through an optimized database. At the local scale, however, the servers are not parallel and performance is extremely limited when too many players congregate in one area. With a lockless, relaxed atomicity state (LEARS), we take this approach even further and focus on how many players can be handled in a single game world segment. We present a model that allows for better resource utilization of multiprocessor game server systems that should not replace spatial partitioning techniques for work distribution but, rather, complement them to improve on their limitations. Furthermore, a real prototype game is used for evaluation, where captured traces are used to generate server loads. We compare multithreaded and single-threaded implementations to measure the overhead of parallelizing the implementation and to demonstrate the experienced benefits of parallelization. The change in responsiveness of different implementations with increased loads on the server is studied and we discuss how generic elements of this game design impact performance on our chosen implementation platform.

Workload: LEARS Model Game Server

Traditionally, game servers have been implemented much like game clients: based around a main loop that updates every active element in the game. These elements include, for example, player characters, non-player characters, and projectiles. The simulated world has a list of all the active elements in the game and typically calls an update for each element. The simulated time is kept constant throughout each iteration of the loop, so that all the elements obtain updates at the same points in simulated time. Such a point in time is referred to as a *tick*. Using this method, the active element performs all its actions during the tick. Since only one element updates at a time, all actions can be performed directly. The character reads input from the network, performs updates on itself according to the input, and updates other elements with the results of its actions.

LEARS is a game server model with support for lockless, relaxed atomicity state parallel execution. The main concept is to split the game server executable into lightweight threads at the finest possible granularity. Each update of every player character, AI opponent, and projectile runs as an independent work unit.

White et al. [130] describe a model they call a *state-effect pattern*. Based on the observation that changes in a large actor-based simulation are happening *simultaneously*, the model separates read and write operations. Read operations work on a consistent previous state and all write operations are batched and executed to produce the state for the next tick. This means that the ordering of events scheduled to execute in a tick does not need to be considered or enforced. For this design, we additionally remove the requirement of batching write operations, allowing these to happen at any time during the tick. The rationale for this relaxation is found in the way traditional game servers work. In the traditional single-threaded main loop approach, every update is allowed to change any part of the simulation state at any time. In such a scenario, the state at a given time is a combination of values from two different points in time, current and previous, exactly the same situation as in the design presented here.

The second relaxation relates to the atomicity of game state updates. The fine granularity creates a need for significant communication between threads to avoid problematic lock contention. Systems where elements can only update their own state and read any state without locking [1] obviously do not work in all cases. However, game servers are not accurate simulators and, again, depending on the game design, some (internal) errors are acceptable without violating game state consistency.

The end result of our proposed design philosophy is that there is no synchronization in the server under normal running conditions. Since there are cases in which transactions are required, they can be implemented outside the LEARS event handler, running as transactions requiring locking.

Design and Implementation

In our experimental prototype implementation of the LEARS concept, the parallel approach is realized using thread pools and blocking queues. The creation and deletion of threads incur large overheads and context switching is an expensive operation. These overheads constrain a system's design, that is, threads should be kept as long as possible, and the number of threads should not grow unbounded. We use a *thread pool* pattern to work around these constraints and a thread pool executor (the Java `ThreadPoolExecutor` class) to maintain the pool of threads and a queue of tasks. When a thread is available, the executor picks a task from the queue and executes it. The thread pool system itself is not preemptive, so the thread runs each task until it is done. This means that, in contrast to normal threading, each task should be as small as possible, that is, larger units of work

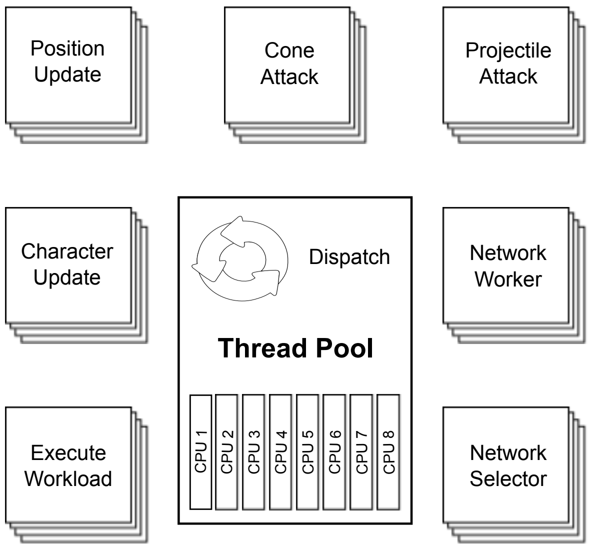


Figure 3.10: Design of a game server.

should be split up into several sub-tasks.

The thread pool is a good way to balance the number of threads when the work is split into extremely small units. When an active element is created in the virtual world, it is scheduled for execution by the thread pool executor and the active element updates its state exactly as in the single-threaded case. Furthermore, our thread pool supports the concept of delayed execution. This means that tasks can be put into the work queue for execution at a specified time in the future. When the task is finished for one time slot, it can reschedule itself for the next slot, delayed by a specified time. This allows active elements to have any lifetime, from one-shot executions to the duration of the program. It also allows different elements to be updated at different rates, depending on the game developer's requirements.

All work is executed by the same thread pool, including the slower input/output (I/O) operations. This is a consistent and clear approach, but it does mean that game updates could be stuck waiting for I/O if not enough threads are available.

The thread pool executor used as described above does not restrict which tasks are executed in parallel. All systems elements must therefore allow any of the other elements to execute concurrently.

To enable fast communication between threads with shared memory (and caches), we use *blocking queues*, using the Java `BlockingQueue` class, which implements queues that are synchronized separately at each end. This means that elements can be removed from and added to the queue simultaneously and, since each of these operations is extremely fast, the probability of blocking is low. Thus, these queues allow information to be passed between active objects. Each active object that can be influenced by others has a blocking queue of messages. During its update, it reads and processes the pending messages from its queue. Messages are processed in the order they were put in the queue. Other active elements put messages in the queue to be processed when they need to change the state of other elements in the game.

Messages in the queues can only contain relative information and not absolute values. This restriction ensures that the change is always based on updated data. For example, if a projectile needs to tell a player character that it took damage, it should only inform the player character about the amount of damage, not the new health total. Since all changes are put in the queue and the entire queue is processed by the same work unit, all updates are based on up-to-date data.

To demonstrate LEARS, we implemented a prototype game containing all the basic elements of a full MMOG, with the exception of persistent states. The system was implemented in Java. This programming language has strong support for multithreading and has well-tested implementations of all the required components. The basic architecture of the game server is described in Figure 3.10. The thread pool size can be configured and will execute the different workloads on the CPU cores. The workloads include the processing of network messages, moving computer-controlled elements (only projectiles in this prototype), checking for collisions and hits, and sending outgoing network messages.

Experiments

In this case study, we only include the resource consumption and thread pool size experiments. We also conducted experiments on client latency. These tests are considered beyond the scope of this thesis, but the experiments can be found in paper VI [102].

To simulate realistic game client behavior, the game was run with five people playing the game with a game update frequency of 10 Hz. The network input to the server from this session was recorded with a timestamp for each message. The recorded game interactions were then played back multiple times in parallel to simulate a large number of clients. To ensure that client performance was not a bottleneck, the simulated clients were distributed among multiple physical machines. Furthermore, since an average client generates 2.6 kbps of network traffic, the 1 Gbps local network interface that was used for the experiments did not limit the performance. The game server was run on a server machine containing four dual-core AMD Opteron 8218 processors with a total of 16 GB of RAM (4 GB of RAM per socket).

We investigated resource consumption when players connected to the game server as shown in Figure 3.11. We present the results for 620 players, since this is the highest number of simultaneous players that the server could handle before a significant degradation in performance. The server was able to keep the update rate smooth, without significant spikes; CPU utilization grew while the clients were logging on and then stabilized at almost full CPU utilization for the rest of the run.

To investigate the effects of the number of threads in the thread pool, we performed an experiment where we kept the number of clients constant while varying the number of threads in the pool. A total of 700 clients were chosen, since this number slightly overloads the server. The number of threads in the pool was increased in increments of two, from two to 256. Figure 3.12 clearly shows that the system utilizes more than four cores efficiently, since the four-thread version shows significantly higher response times. At one thread per core or more, the numbers are relatively stable, with a tendency toward consistently lower response times with more available threads, up to about 40 threads. This could mean that threads are occasionally waiting for I/O operations. Since thread pools are not preemptive, such situations would lead to one core going idle if there were no other available threads. Too many threads, on the other hand, could lead to excessive context switch overhead. The results show that the average slowly increases after about

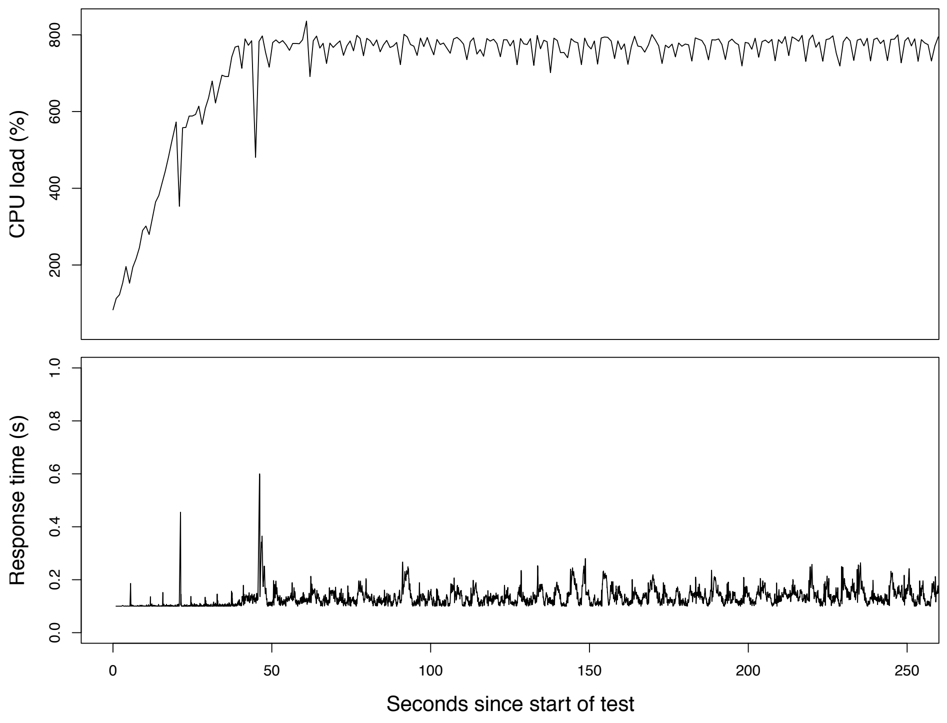


Figure 3.11: CPU load and response time for 620 concurrent clients on a multi-threaded server.

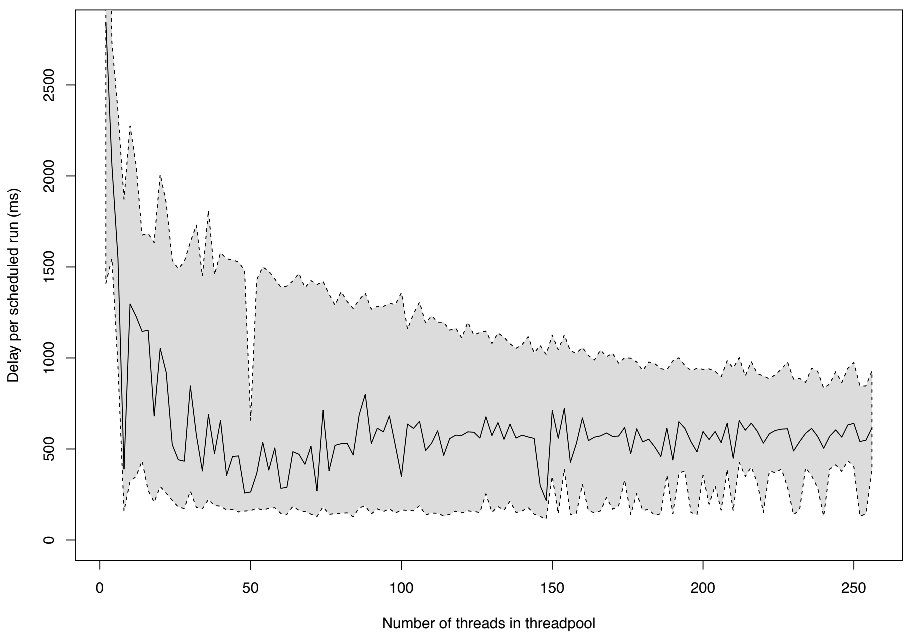


Figure 3.12: Response time for 700 concurrent clients, using various numbers of threads. The shaded area indicates the fifth to 95th percentiles.

50 threads, though the 95th percentile is still decreasing with an increasing number of threads, up to about 100. From then on, the best case worsens again, most likely due to context switching overhead.

Summary

In terms of programming techniques, we have shown that we can improve resource utilization by distributing the load across multiple CPUs in a unified memory multiprocessor system. This distribution is made possible by relaxing constraints on the ordering and atomicity of events. The system scales well, even in the case in which all players must be aware of all other players and their actions. The thread pool system balances load well between the cores and its queue-based nature means that no task is starved unless the entire system lacks resources. Message passing through the blocking queue allows objects to communicate intensively without blocking each other. Running our prototype game, we show that the eight-core server can handle a factor of two more clients before the response time becomes unacceptable.

Our results indicate that it is possible to design an "embarrassingly parallel" game server. We also observe that the implementation is able to handle a quadratic increase in in-server communication when many players interact in a game world hotspot. The experiments also show that if too many threads are added to the thread pool, performance will decrease. This is mainly due to greater contention in the operating system scheduler. We also saw the importance of balancing the number of threads with the number of CPU cores available in the system. If the size of the thread pool is too large, the delays on the game server will increase.

3.2.4 Implications

The x86 architecture is very straightforward for application development. The architecture uses a shared memory model, which means that all the processors available to an operating system are able share the memory and the processor manufacturers have implemented cache coherency protocols to make sure that the data in all the caches are updated. However, this comes at a cost. The traffic generated by these protocols can end up starving the bandwidth on the inter-core communication buses that are used for sharing data and accessing memory.

On asymmetric architectures such as the Cell, IXP, and GPUs, on often has specialized cores that are fast for specific operations. On the x86, all the cores are general purpose. However, they often have specialized functions, such as SSE/AVX units, to carry out fast vector operations, but this requires the applications to be optimized by hand.

Another challenge with the x86 when running an application with many threads is that the threads on the platform are not as lightweight as on the Cell or on GPUs and too many threads executing on too few cores will result, as we saw in Section 3.2.3, in loss of performance due to the context switching overhead. We also saw the same trends with the VP8 encoder in Section 3.2.2. Running Google's reference encoder serially provided better performance than running it concurrently.

The x86 hides a great deal of the architectural details from programmers. This makes the architecture very easy to use, but comes at the cost of performance.

Revisiting with State-of-the-Market Hardware

If we were to revisit these experiments with the latest generation of Haswell x86 processors, we would probably obtain better performance. The advantage for end users with the x86 is its backward compatibility. This means that we can just run the programs, even without recompiling them. This is not an optimal solution for hardware manufacturers. Compatibility with old instructions makes these processors' designs more complex, which increases their power consumption. However, if we want to utilize the more advanced vector instructions and other extensions (i.e., transactional memory support) that are added in new generations of processors, we still have to rewire the applications for support.

3.3 Graphics Processing Units

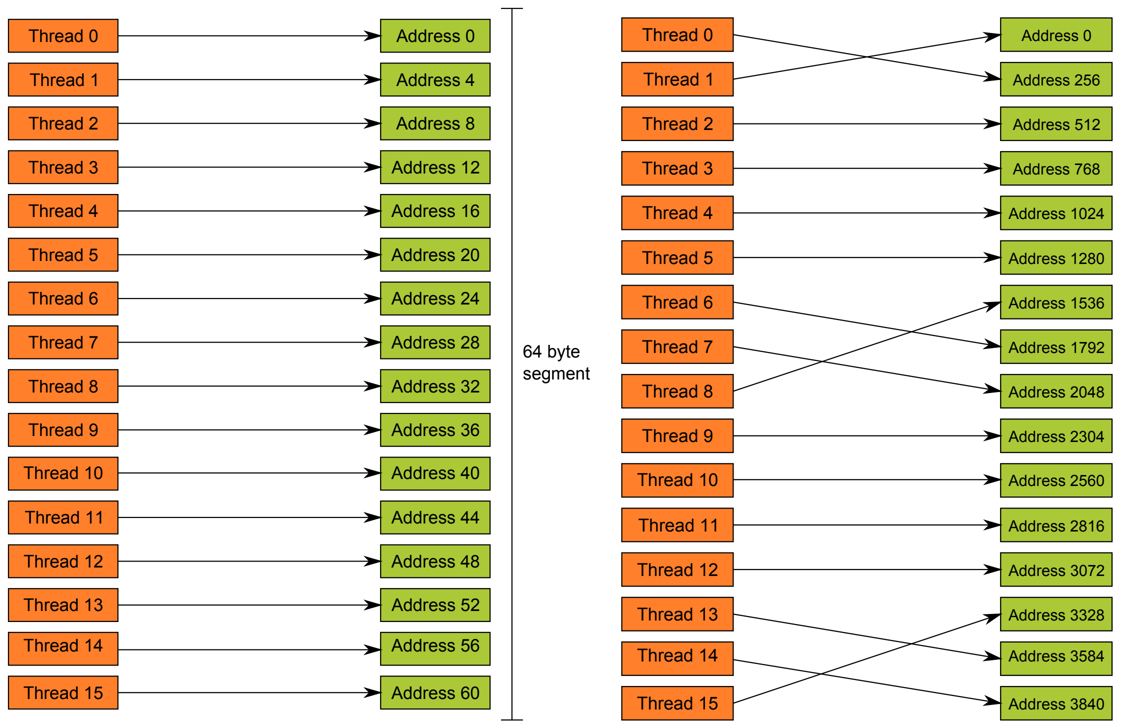
A GPU is an asymmetric coprocessor that is often connected to the CPU with a PCI Express bus. In some cases, it can also be integrated onto the CPU die. The GPU is a highly parallel architecture and, where an x86 processor would require tens of threads to achieve peak performance, a GPU would typically require thousands of threads. A GPU, like the Cell, has an exclusive memory model. The GPU has multiple off-chip and on-chip memory types that a programmers must use correctly to achieve the best performance. The main focus for our experiments on the GPU was first to learn how they performed in different scenarios and, later, how we could offload parallel parts of multimedia workloads.

In the following section, we take a closer look at four case studies. First, in Section 3.3.1, we look at how to use the memory correctly and the implications of not choosing the right memory space. Next, in Section 3.3.2, we look at communication patterns between the host CPU and the GPU, using a multimedia workload. In Section 3.3.3, we use the GPU to detect cheating in a multiplayer game and, finally, in Section 3.3.4, we look at the MJPEG workload that we also touched upon in the Cell and x86 portions of this chapter.

3.3.1 Case Study: GPU Memory Spaces and Access Patterns

To obtain the best possible performance when using a GPU, programmers need to be careful when it comes to resource usage. Registers per thread, occupancy on the GPU, memory placement, and access patterns are properties of a GPU kernel that are important for achieving optimal performance. As part of a master's thesis [94], we conducted experiments with the memory architecture on GPUs released in 2006 and 2008 based on the Nvidia Tesla architecture.

To gain a better understanding of how to optimize memory access, the programmer needs to be aware of how memory instructions are executed by the memory controller on the GPU. This is especially important in the case of global memory, since it is used by every thread and it is the memory space with the highest latency. The threads on an Nvidia GPU are scheduled in groups of 32 threads called warps. To make the scheduling more flexible, the memory transactions from a warp are executed on a half-warp basis. This is due to the design of the shared memory and to ease the handling of memory transactions from threads in a divergent warp. Divergence within a warp means that threads execute different instructions, which can be due to branching in the code or idle threads.



(a) Coalesced access pattern (b) Uncoalesced access pattern

Figure 3.13: Global memory access patterns.

The half-warps executing on the GPU are most efficient when memory access from simultaneously running threads can be combined into a single memory transaction to global memory. This is known as a *coalesced* memory transaction. The half-warp must meet certain requirements to coalesce the memory transaction and these requirements are determined by the GPU's compute capability. The compute capability also affects how the transactions are issued. If the requirements are not met, this is referred to as a *uncoalesced* memory transaction.

An example of a coalesced and an uncoalesced access pattern is illustrated in Figures 3.13(a) and 3.13(b). In this example, coalesced access is achieved by having each thread access a 32-bit word in sequence within a 64-byte segment. The uncoalesced access reads values from different segments, making it impossible for the memory controller to coalesce such access.

Global Memory

Global memory is used most efficiently when all the threads of a half-warp can issue a coalesced memory transaction. The size of a memory transaction that can be executed depends on the compute capability supported by the GPU. A 64- and a 128-byte transaction can be performed on a compute capability of 1.0 and 1.1 GPU, while a compute capability of 1.2 and greater also added support for 32-byte transactions. The transaction size is important, since global memory is considered to be partitioned into segments of 32 bytes, 64 bytes, or 128 bytes.

Constant and Texture Memory

The constant and texture memory spaces are designed for read-only data structures that have elements that reside close in memory. The memory spaces are limited in size, are read only, and are therefore not always suitable for certain applications. Both memory spaces use a caching mechanism in which an 8-kB cache is available for both texture and constant memory on each SM/SMX. If there is a cache miss, a read costs the same as a fetch from global memory, since both memory spaces are subsets of global memory. An advantage of using these read-only memory spaces is that the requirements for optimal performance are not as strict as in global memory. Threads of a warp that read texture addresses that are close together will achieve the best performance; so, mapping the read-only data to fit this alignment is considered a good optimization.

The texture and constant caches differ in the kind of locality for which they are optimized. The constant cache is as fast as reading a register, as long as all the threads in a half-warp read from the same address (data element), and the cost scales linearly with the number of different addresses that are read. The texture cache is a more flexible cache, since it does not require each thread to read the same address for full speed. However, having threads read addresses that are close to each other is recommended, since the cache is optimized for 2D spatial locality used in imaging. Texture memory is normally used for the storage of texture data used for rendering images.

Experiments and Summary

Coalesced memory access can have a large performance impact. However, few quantified results exist and the efficient usage of memory types, alignment, and access patterns remains an art. Weimer et al. [129] experimented with bank conflicts in shared memory but, to shed light on the penalties of inefficient memory type usage, further investigation is needed. We therefore performed experiments that read and wrote data to and from memory with both uncoalesced and coalesced access patterns [94] and used the Nvidia CUDA Visual Profiler to isolate the GPU times for the different kernels.

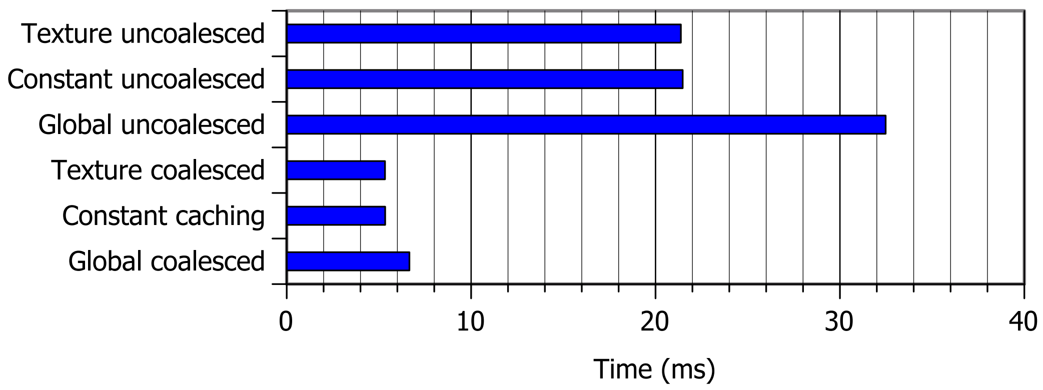


Figure 3.14: Optimization of GPU memory access.

Figure 3.14 shows that an uncoalesced access pattern increases the latency of the data transfer on the order of four times due to the increased number of memory transactions. Constant memory and texture memory are cached and the performance of their uncoalesced access is improved compared to global memory, but there is still a three-time penalty. Furthermore, the cached memory types support only read-only operations and are restricted in size. When used correctly, the performance of global memory is equal to the performance of the cached memory types. The experiment also shows that correct memory usage is imperative, even when cached memory types are used. It is also important to ensure that the memory access follows the specifications of particular GPUs because the optimal access patterns vary between GPU generations [94].

3.3.2 Case Study: Host–Device Communication Optimization

A very important aspect of using a GPU is moving the data from the host CPU into the GPU as quickly as possible. While doing this, it is also important to always have workloads ready for the GPU and not leave any cores idle. As part of a master's thesis [13], we performed experiments on a CUDA-based H.264 encoder from the National University of Defense Technology, China. The encoder is called *cuve264b* and is a port of their streaming HD H.264 encoder [134] to the CUDA architecture. In this thesis, we investigate the effects of optimizing communications between the host CPU and the GPU. For details about the H.264 video compression standard and for a full evaluation, the reader should consult the master's thesis mentioned [13].

The cuve264b encoder uses slices to help parallelize the encoding process. By dividing each frame into multiple slices, the encoder can encode each slice independently. In a snapshot of the encoder on which we conducted our experiments, the number of slices was hard-coded to 34. This version also only supports the 1080p resolution. However, 720p resolution was added later. All the tests in this experiment were conducted on a GeForce GTX 480 GPU based on the Fermi compute architecture. The CPU used in the experiments was an Intel Core i7-860 based on the Nehalem microarchitecture, with Hyper-Threading Technology (SMT) enabled.

To ensure that video frames from the host CPU are always available to the GPU, readahead was been implemented on the CPU side and makes sure that the encoder has finished reading the frames of uncompressed videos into main memory before the frame is requested by the GPU. This is one of the optimizations implemented to make sure that the cores in the GPU are never idle. By optimizing the code with readahead on the host CPU side, we were able to reduce the encoding time by around 20% [13].

CUDA Streams

Readahead was implemented by using multiple threads on the host CPU to asynchronously perform the IO operations. However, multiple threads on the host cannot perform concurrent operations on the GPU. To enable this, CUDA provides an asynchronous API called CUDA Streams [92]. This API allows us to copy data directly into the GPU's global memory and queue up multiple operations on the GPU. These operations are performed asynchronously to the host threads and the GPU is able to perform some of the operations concurrently. In early-generation GPUs from the Tesla architecture, it was only the transfer of data between the host and device and the computations that could be overlapped. With the next generation of GPUs from the Fermi architecture, the GPU could concurrently execute up to 16 kernels while transferring a single stream. To enable asynchronous memory transfers, pinned memory must be used on the host. This enables the direct memory access (DMA) engine on the GPU to transfer the data without involving the CPU.

Experiments and Summary

To measure the effects, we implemented asynchronous transfers in the cuve264b encoder. We also implemented support for reusing memory, since pinned memory is a limited resource and slow to allocate. As we can see from Figure 3.15, the asynchronous transfers are about 10 times faster than the synchronous version. The results depend on the number of frames that have to be encoded, so, since our test sequences are of limited length, the difference would be greater for long videos, such as feature films and television shows.

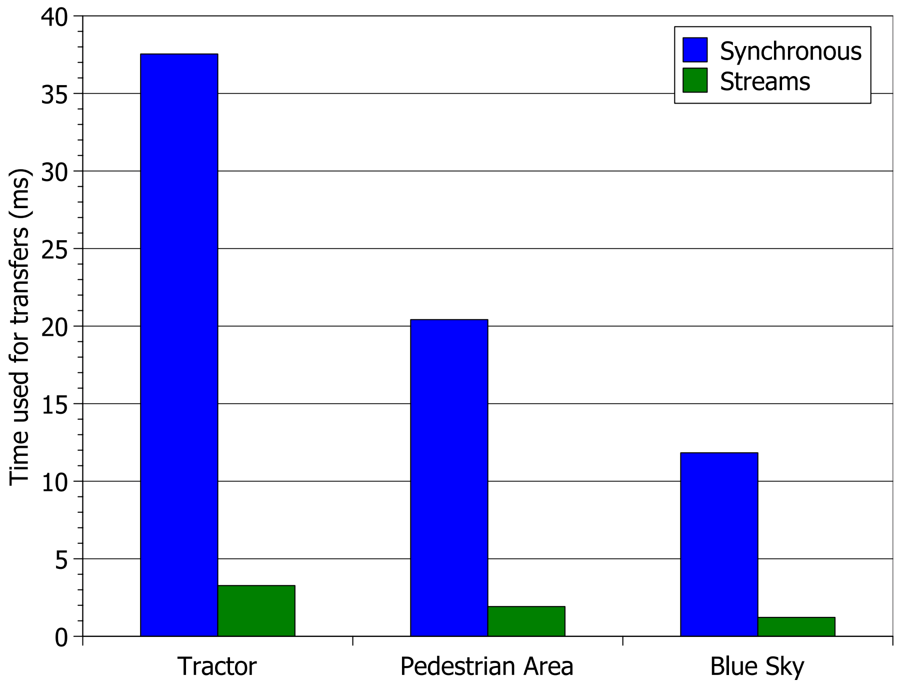


Figure 3.15: Total time spent on transfers to GPU on three different 1080p sample videos.

Our experiments show that using asynchronous transfers such as CUDA Streams to overlap the transfers with computations for large workloads such as H.264 reduces the GPU idle time and consequently also the encoding time. With newer computing devices from the Fermi architecture, the GPU is also able to schedule and execute other kinds of processing tasks concurrently, not only overlap computations from a single kernel and memory transfers.

3.3.3 Case Study: Cheat Detection

When offloading parts of a processing pipeline to a GPU, the data transfer from the host CPU will add latency to the execution. It is therefore important to make sure that the offloaded workload is large enough to overcome this transfer latency. In this section, we use a cheat detection workload to test this effect on a GPU.

Setting

On-line multiplayer gaming has experienced amazing growth over the last decade and has been accompanied by cheating as the most prominent type malicious game player behavior [137]. It is in the best interest of game service providers to eradicate cheating. However, the demand for stable service for resource-intensive games restricts the amount of resources that can be dedicated to cheat detection mechanisms.

Many on-line multiplayer games suffer from excessive cheating in one form or another. However, in many cases, cheating is hard to prove [99]. The only part of a distributed system that a game service provider can trust is the part of the system running on hardware under their control. Any other part of the system can and will most likely be exploited by a cheater.

In-game physics, aimed to increase game realism, has experienced increased popularity in many kinds of games. Most games that have implemented in-game physics use it as a major part of the gameplay experience, some even basing the entire gameplay around physics alone. In-game physics is therefore a very likely part of a game to be exploited. To address this problem, central servers or other trusted entities must ensure consistency in the movements of all the clients in the game. With our approach, the physics engine can be implemented on the server together with the cheat detection mechanisms. This solution frees resources on the game clients; however, it requires more hardware on the server side.

Adding more hardware to a system can increase its performance, but this is only a temporary solution. The hardware used in commercial game server clusters is expensive and the performance gained might only be sufficient for a short period. Because of the physical limitations halting the increase in single-threaded performance in normal CPUs, further performance increase is accomplished by adding more identical processing cores. The modern GPU is a relatively inexpensive example of such a parallel architecture. The process of adding new and faster hardware is now slowly substituted by migrating systems to parallel processors. For this change to be beneficial, serial algorithms must be parallelized.

Our goal in this case study is to use an example cheat detection workload to learn about the overhead of moving the data to the GPU. If we offload too small of a workload to a GPU, the overhead of the data transfer will be greater than the time it takes to perform the calculations on the CPU.

Workload: Cheat Detection

To show the benefits of using a GPU for cheat detection, we created a simple space race game simulation where the spacecrafts must visit virtual positions, also referred to as targets. The clients are placed randomly in the virtual world, giving some clients an advantage, since they might be placed closer to a target. When a target is reached, the clients continue to the next target.

The simulation follows a client–server-based game architecture where all clients send their position updates to the server. This approach is chosen for the same reasons as in consumer market game development: ease of development, total control of client communication, and a centralized control point. Discrete clients are created within the simulation and communication follows the same flow that would be normal in a networked multiplayer game. Furthermore, because we wanted to design our simulation independent of wall-clock time, we used an artificial timeline based on game ticks. A tick is a theoretical time duration specified in the system's configuration.

To allow for reproducible tests, the simulation uses two different modes of operation, named the generation mode and the playback mode. The *generation mode* uses the principles of the game to determine the random placement of a given number of clients in a virtual environment. From these positions, the clients try to reach the closest target. After reaching a target, they continue to the next target, using a thruster to propel themselves. External forces, such as gravity, affect the clients. While this is happening, the server writes each client's location in the virtual environment to several files. These files are used in playback mode. Generation mode also generates movement for cheaters. The numbers of cheaters can be adjusted in generation mode. A cheater behaves in the same manner as an honest client but regularly performs unrealistic motions. *Playback mode* initializes the clients. The client state information is read from the files generated in generation mode and the states are reported to the server. The server samples the state information updates from every client, placing the samples in a sample buffer. The buffer is read by the cheat detection thread when full.

Because all clients in the game were controlled by the computer, rules were needed to determine their behavior in trying to reach a target. To reach their targets, the clients required motion planning. We did not implement any advanced motion planning algorithms. The clients knew the targets that they reached. After a target was reached, the clients continued to the closest unaccomplished target. Client movement was restricted by the physical model. Honest clients did not break the rules of the model, while cheating clients did.

In our simulation, the objects experienced both linear and angular acceleration. There was a constant gravitational pull affecting the objects, much like the gravity on Earth. All the other forces were generated by the objects themselves, using thrusters. Figure 3.16 shows an outline of a game object, with a main rear thruster and bow thrusters. Objects moved forward with the rear thruster and rotated using the bow thrusters. The size and thruster power could be modified by parameters.

The physics engine is one of the main parts of the simulation. The engine is responsible for calculating the sum of all physical forces acting on all objects and updates their positions accordingly. The physics engine is controlled by configuration parameters that allow one to change physical properties quickly, even during runtime. Game objects are registered with the physics engine, so it maintains a pool of objects to manage. Updates of the parameters of an object, such as throttle, are handled by the individual clients. Integration of the time steps from one game tick to the next is carried out by the engine. The physics engine does this by updating every game object in the object pool. The main implementation of the physics engine runs on the CPU and is only used in generation

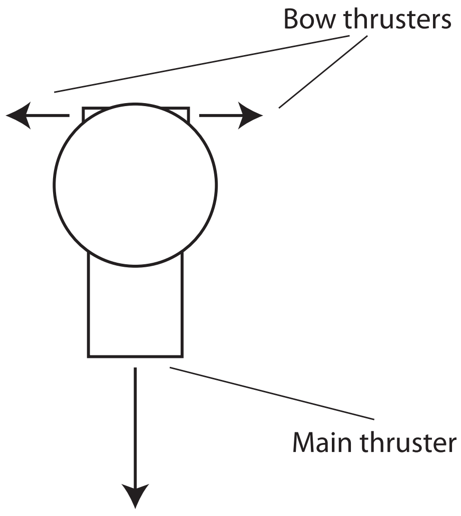


Figure 3.16: Illustration of a game object with bow thrusters in the front and the main thruster in the back.

mode. During playback mode, the cheat detection mechanisms act as a reverse physics engine; they try to determine if the position updates are valid within the current physical model.

The physical model used in this example is a simple model, with only a couple of physical effects. The most basic of these effects is *linear motion*. Basic linear motion is implemented using Newton's second law of motion:

$$\sum \mathbf{F} = m\mathbf{a} \tag{3.1}$$

This law states that the sum of all forces acting on an object is the product of the object's mass and its acceleration. The acceleration is measured by observing the change in speed over a known distance. In our game, two linear forces act on an object. The first is the acceleration applied by the game object's main thruster, as illustrated in Figure 3.16. The second is the vertical gravity, which is constant throughout the entire model. The total linear force is represented by the sum of these two vectors.

The second physical effect is *angular motion*. To allow object rotation in all dimensions, the properties of the objects in the game must be extended. Similar to the linear motion properties of distance, velocity, and acceleration, we have angular motion properties. The equations

$$\Omega = \frac{\mathrm{d}\omega}{\mathrm{d}t} \tag{3.2}$$

$$\omega = \frac{\mathrm{d}\alpha}{\mathrm{d}t} \tag{3.3}$$

where Ω is the angular displacement of an object in radians, ω is its angular velocity in radians per second, and α is its angular acceleration in radians per second squared, show these relations.

Angular motion is applied to the game objects when the bow thrusters illustrated in Figure 3.16 are used to change the course of an object. Support for collisions is implemented in the model. However, due to lack of time, it was not implemented in the cheat detection mechanisms. It was only present in generation mode.

There are different ways to cheat in the simulation. Clients cheat by either temporarily modifying the power of their thrusters or modifying the values of their current state: their position, velocity, and rotation. If a cheating client temporarily increases the capabilities of one of its thrusters, it is able to accelerate faster, perform quicker turns. Cheaters who change their state can position themselves closer to a target or change their rotation to point toward a target. They might also increase or decrease the magnitude of their velocity vector when either dashing for a target or slowing down to avoid passing a target.

Implementation

We implemented two versions of the cheat detection mechanism. One was written for the host CPU, while the other was a CUDA version, written for the GPU device. The cheat detection mechanism on the GPU was implemented with threads. The CPU implementation was not threaded and used a basic looping structure to simulate the same behavior as the CUDA version.

The behavior of the mechanisms is illustrated in Figure 3.17. A single thread works on three consecutive game state samples for a client: Thread one (th1) works on samples s0, s1, and s2, while thread two (th2) works on samples s1, s2, and s3, and so forth. A sample is the state of the client after a tick in the artificial timeline.

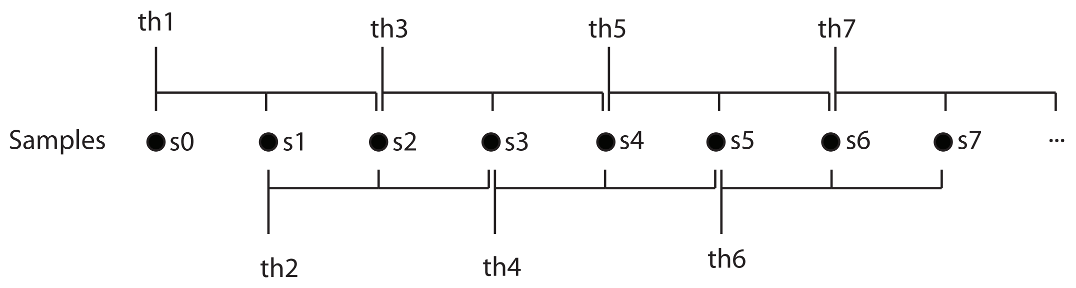


Figure 3.17: Sample reading and execution thread pattern.

A sample contains the movement of each client and a positional vector with three values, x, y, and z, as three-dimensional axes. With three samples, the threads can determine the client's acceleration as a three-dimensional vector. All external forces added by the physical model can now be subtracted by applying the calculations of the physical engine in reverse. The resulting acceleration is the result of the forces the client applied to the game object. If the thrust applied by the client is greater than the maximum thrust allowed by the game, the client is most likely a cheater.

There are two main node types in our simulation: the server and the clients. They exchange data as in real networked games. A packet is either generated by the generation mode or read from a file in playback mode by the clients once for each game tick.

The *server* reads all incoming data from the clients. When a cheater reports erroneous positional data, the cheat detection mechanisms indicate that the player's movement does not follow the rules and restrictions of the game's physical parameters.

Clients act differently depending on the execution mode. During the generation of movement files, clients write their locations and other appropriate data to file. In playback

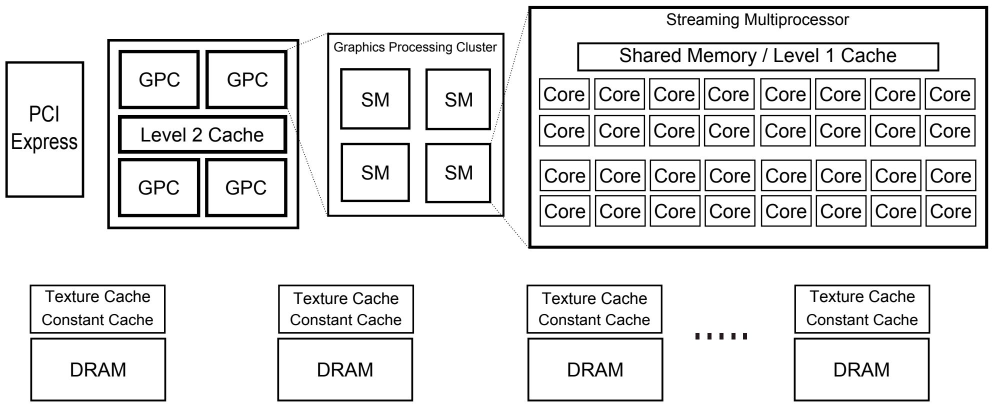


Figure 3.18: Nvidia GF100 compute architecture.

mode, clients read from the generated files and report the data written in generation mode back to the server. In this way, the system allows for reproducible tests as the test data is the same for each test run.

Experiments

In this study, we investigated both the total execution time of the cheat detection system and the total execution time spent on the cheat detection mechanisms. All tests were run on data generated in generation mode over 100 seconds of "game time." The number of clients used in the benchmarks ranges from 10 to 6000. The part of the mechanisms that runs on the GPU in these benchmarks is the reverse physics engine.

The cheat detection mechanism we tested was implemented on the following hardware: The CPU used in the benchmarks was an Intel Core i5-750 processor with 4.0 GB of RAM. The GPU was an Nvidia GeForce GTX 480 with 480 processing cores and 1.5 GB of memory. The chip used in the GeForce GTX 480 is the GF100 GPU, illustrated in Figure 3.18. This GPU is based on the Fermi compute architecture.

Figure 3.19 shows the results of the first benchmark and the total execution time of the cheat detection system. We can observe that, with a low number of clients, the CPU is faster than the GPU, due to the added latency of moving data and code to the GPU. With more than 100 clients in the game, the execution time for the CPU exceeds that of the GPU and the performance gap steadily increases up to 6000 clients, which is the maximum number of tested clients. This is due to the size of the memory on our test machines. When the number of clients increases, the cheat detection processing on the GPU scales much better than on the CPU. When the cheat detection mechanism is processed on the GPU, the CPU is relieved of these tasks and can work on other game-relevant computation.

To determine the offloading effect the GPU has on the CPU, we measured how much of the total execution time was spent on processing cheat detection mechanisms. Figure 3.20 reports the results of the second benchmark. These show that, for a small number of clients, the penalty for transferring data over the PCI Express bus to the GPU is significant, making the CPU more effective for a small number of clients. With more than 50 clients, the GPU implementation spends less time on cheat detection than the CPU implementation does. As the number of clients increases, the time spent on cheat

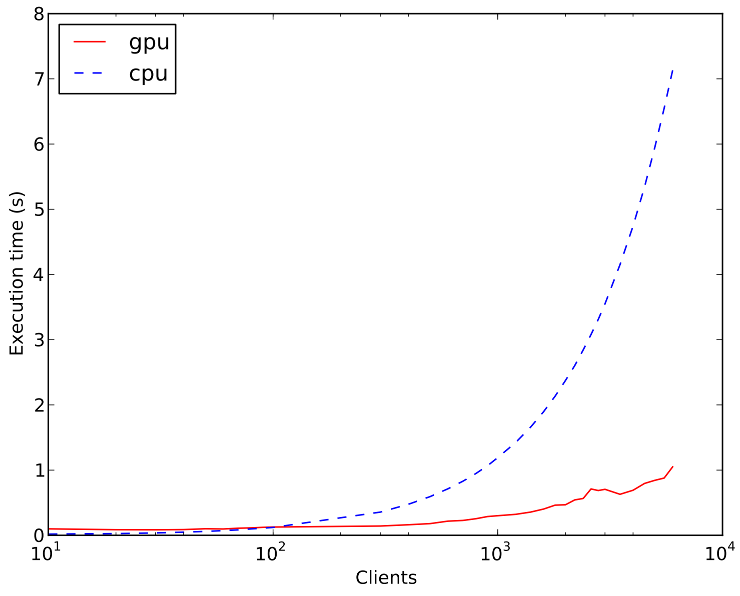


Figure 3.19: Execution time (in seconds) of the cheat detection mechanism on the GPU and the CPU.

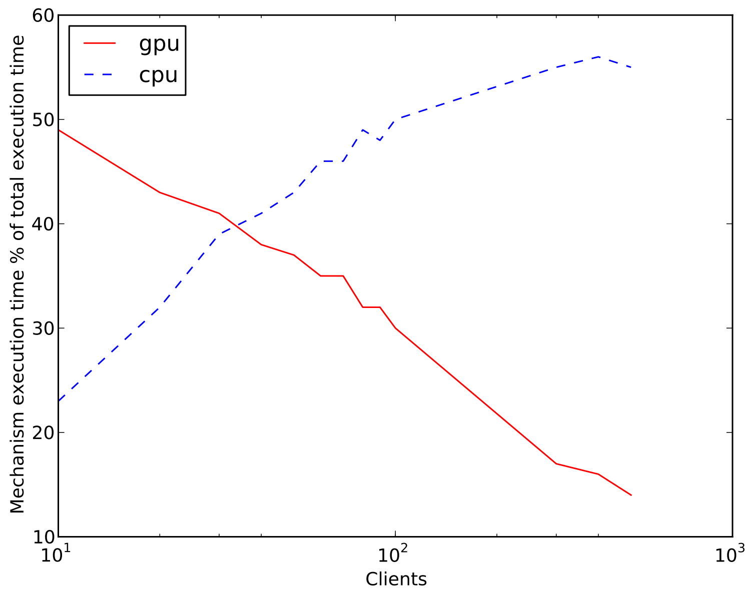


Figure 3.20: Percentage of time spent on cheat detection processing on a host, using the GPU and the CPU.

detection continues to drop to below 15% for the GPU implementation. The CPU version stabilizes around 50%. To improve the performance of the GPU implementation for a low number of clients, it is possible to buffer more samples before executing the mechanisms on the GPU.

Discussion

We have seen how the CPU and GPU implementations of our cheat detection mechanism perform differently when we increase the numbers of clients in the game. The difference between the two is smallest when the number of checks performed on the GPU is small. However, as the number of clients increases, the increase in the execution time of the CPU implementation is much steeper compared to the increase in the GPU implementation. This indicates that the GPU implementation is the more scalable of the two. This is primarily due to the GPU's highly parallel architecture. Physics operations for large numbers of clients are independent of each other. They constitute an embarrassingly parallel workload that maps well to the GPU's multithreaded architecture. Both the CPU and GPU implementations could be further optimized in further work. The CPU implementation could be extended with threading and SIMD operations and the GPU version could be extended with asynchronous transfers, optimized global memory access and the elimination of branching in the compute kernels.

The cheat detection mechanism we implemented for our system is easy to parallelize because the physics computations for clients are independent of each other. Similar systems with workloads that contain operations that can be performed simultaneously by a large number of threads can benefit from using a GPU to offload the processing. When offloading operations to a GPU, it is important to remember that the GPU is most efficient if it has enough data to process. It is also important that the offloaded tasks map well to the GPU's multithreaded architecture.

Summary

Our results show that even a simple physical model can benefit from executing the workload on a GPU. The experiments also clearly show that we need a large workload or a computationally heavy workload to benefit from a GPU. When we benchmarked our cheat detection mechanisms with too few clients, the CPU implementation was faster, because the computational load did not compensate for the latency involved with transferring the data to the GPU. Another advantage of GPU implementation is the offloading effect: While the GPU handles the cheat detection workload, the CPU can perform other tasks.

3.3.4 Case Study: MJPEG Encoding

In our MJPEG case study, we performed several experiments on the GPU architecture and a more detailed overview of the MJPEG workload can be found in Section 3.2.1. As described later for the Cell, several layouts are available for GPUs. However, because of the large number of small cores, it is not feasible to assign one frame to each core. The most time-consuming parts of the MJPEG encoding process, the DCT and quantization steps, are well suited for GPU acceleration. In addition, the VLC step can also be partly adapted.

Our experiments compared 14 different GPU implementations of the MJPEG encoder. This gives a good indication that the architectures are complex to use and that achieving high performance is not trivial. Derived from a sequential codebase, these implementations differ in terms of algorithms used, resource utilization, and coding efficiency. Figure 3.21 shows the performance results of encoding the 1080p tractor video clip in YUV

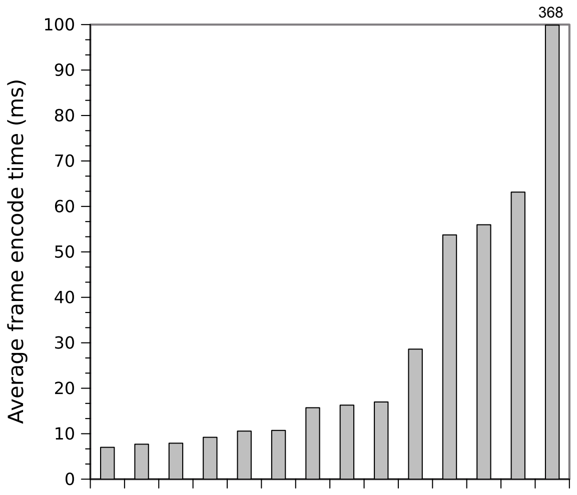


Figure 3.21: Runtime for MJPEG implementations on a GPU (GTX 280).

4:2:0. The difference between the fastest and slowest solutions is 362 ms per frame and the fastest solutions are disk I/O bound. To gain experience in what works and what does not, we examined these solutions. We did not consider coding style, but revisited algorithmic choices, inter-core data communication (memory transfers), and architecture-specific capabilities.

GPU Experiments

A GPU is a dedicated graphics rendering device and modern GPUs have a parallel structure, making them effective for carrying out general-purpose processing. Previously, shaders were used for programming, but specialized languages are now available. In this context, Nvidia released the CUDA framework with a programming language similar to ANSI C. In CUDA, the single instruction, multiple threads (SIMT) abstraction is used to handle thousands of threads.

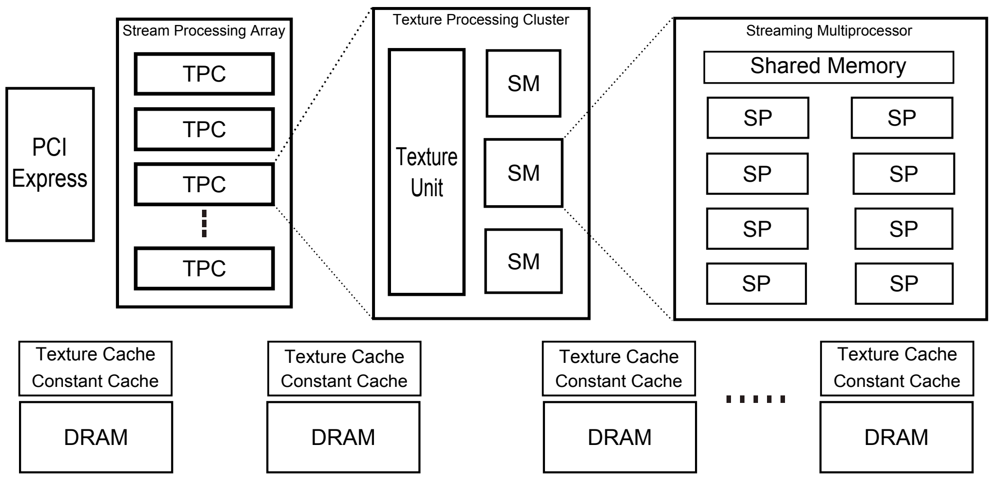


Figure 3.22: Nvidia GT200 architecture.

This case study was carried out with the first generation of programmable GPUs from

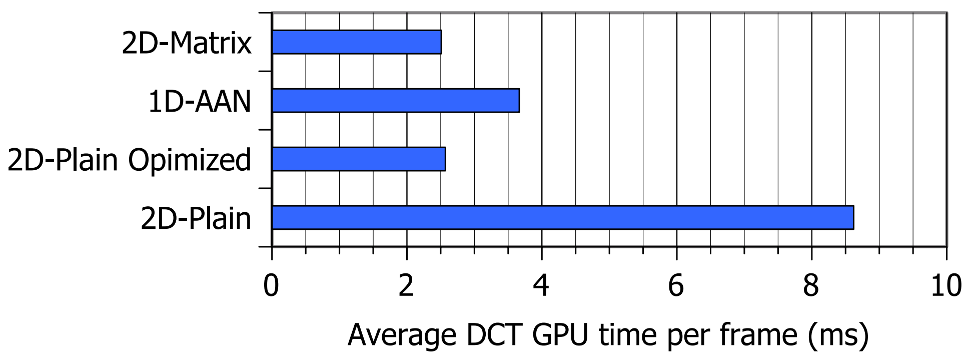


Figure 3.23: DCT performance on a GPU.

Nvidia called Tesla. The chip has the codename GT200, which is the second generation of GPUs in the Tesla architecture, released in 2008. A more detailed overview of the architecture is shown in Figure 3.22. The GT200 chip presents as a highly parallel, multithreaded, multicore processor, connected to the host computer by a PCI Express bus. The GT200 architecture contains 10 texture processing clusters with three streaming multiprocessors (SMs). A single SM contains eight stream processors (called cores in newer GPUs), which are the basic arithmetic and logic units for calculations.

To find out how memory access and other optimizations affect programs such as an MJPEG encoder, we experimented with the same DCT implementations that we used on the x86 architecture in Section 3.2.1. Our baseline DCT algorithm is the *2D-plain* algorithm. The only optimization in this implementation is that the input frames are read into cached texture memory and that the quantization tables are read into cached constant memory. Cached memory spaces improve performance compared to global memory, especially when memory access is uncoalesced. The second implementation, referred to as *2D-plain optimized*, is tuned to run efficiently using principles from the CUDA Best Practices Guide [87]. These optimizations include the use of shared memory as a buffer for pixel values when processing a macroblock, branch avoidance by using Boolean arithmetic, and manual loop unrolling. Our third implementation, the *1D AAN* algorithm, is based on the scalar implementation used on the x86. Every macroblock is processed with eight threads, that is, one thread per row of eight pixels. The input image is stored in cached texture memory and shared memory is used to temporarily store data during processing. Finally, the *2D matrix* DCT uses matrix multiplication, where each matrix element is computed by a thread. The input image is stored in cached texture memory and shared memory is used to store data during calculations.

We know from existing work that to achieve high instruction throughput, branch prevention and the correct use of flow control instructions are important. If threads on the same SM diverge, the paths are serialized, which decreases performance. Loop unrolling is beneficial on GPU kernels and can be done automatically by the compiler using pragma directives. To optimize frame exchange, asynchronous transfers between the host and GPU are used. Transferring data over the PCI Express bus is expensive and asynchronous transfers help us reuse the kernels and hide some of the PCI Express latency by transferring data in the background.

To isolate DCT performance, we used the CUDA Visual Profiler. The profiling results

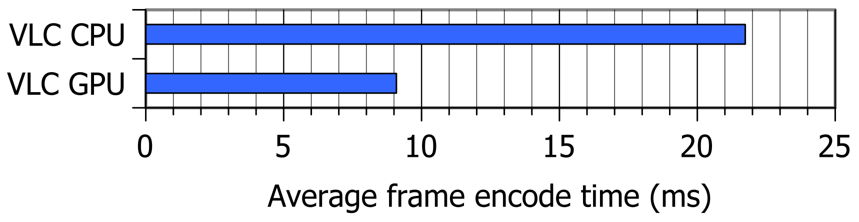


Figure 3.24: Effect of offloading VLC to the GPU.

of the different implementations are shown in Figure 3.23 and we note that the 2D-plain optimized algorithm is faster than the AAN algorithm. The 2D-plain algorithm requires significantly more computations than the others but, by optimizing it to the architecture, we obtain almost as good performance as with the 2D matrix. The AAN algorithm, which does the least number of computations, suffers from the low number of threads per macroblock. A low number of threads per SM can result in stalling, where all the threads are waiting for data from memory, which should be avoided.

This experiment shows that, for architectures with vast computational capabilities, writing a good implementation of an algorithm adapted for the underlying hardware can be as important as the algorithm's theoretical complexity.

The last GPU experiment considers entropy coding on the GPU. VLC can be offloaded to the GPU by assigning a thread to each macroblock in a frame to compress the coefficients and then store the bitstream of each macroblock and its length in global memory. The output of each macroblock's bitstream can then be merged either on the host or by using atomic OR on the GPU. For these experiments, we chose the former, since the host is responsible for the I/O and must traverse the bitstream anyway. Figure 3.24 shows the results of an experiment that compares MJPEG with AAN DCT, with VLC performed on the host and on the GPU, respectively. We doubled the encoding performance when running VLC on the GPU. In this particular case, offloading VLC was faster than running on the host. It is worth noting that by running VLC on the GPU, the entropy coding scales together with the rest of the encoder with the resources available on the GPU. This means than if the encoder runs on a machine with a slower host CPU or a faster GPU, it will still scale.

Discussion

The GPU architecture is different from the x86 architecture used for MJPEG in Section 3.2.1. Some algorithms may be more suited than others. This can clearly be seen in our experiments with DCT, where the AAN algorithm performed best on the x86, but did not achieve the highest throughput on the GPU. This was because of the relatively low number of threads per macroblock for the AAN algorithm, which must perform the 1D DCT operation (one row of pixels within a macroblock) as a single thread. This is only one example of achieving a shorter computation time through increased parallelity at the price of a higher, sub-optimal total number of operations.

Porting the encoder to the GPU in a straightforward manner without significant optimizations for the architecture yields very good offloading performance compared to native x86. This indicates that the GPU is easy to use but, to reap the full potential of the architecture, one must have a deep level of understanding.

Summary

We see that heterogeneous multicore architectures provide the resources required for real-time multimedia processing. However, achieving high performance is not trivial. In general, there are similarities between the architectures, but the way of thinking must be substantially different. The different architectures have different capabilities that must be taken into account, both when choosing a specific algorithm and when making implementation-specific decisions. A great deal of trust is put into the compilers of development frameworks and new languages such as OpenCL, which are supposed to be a "recompile-only" solution. However, to tune performance, the application must still be hand optimized for different versions of the GPUs and x86.

3.3.5 Implications

Working with a GPU is different from working with a x86. Compared to the SIMD model, the SIMT model seems easier for programmers to grasp. With SIMT, programmers have to think in terms of threaded operations on intrinsic data. However, programming a GPU has some challenges, as we saw in the case studies. First, as seen in the memory tests, the GPU has an exclusive memory model and it is important for the programmer to fully understand the access pattern of the GPU kernel to use the correct memory layout and memory space. If the data pattern generates too many uncoalesced memory accesses, it might be an idea to see if it is possible to use one of the cached memory types. In the host–device optimization experiments, we learned that transfers from the host CPU to the GPU over the PCI Express bus can be a slow process and it is very important to try to overlap transfers with computations by using one of the asynchronous APIs available to the programmers. The cheat detection studies show that computational workloads such as physics calculations scale very well on a GPU. However, with too few calculations, the overhead of transferring data from the CPU to the GPU is too large—hence, the importance of efficient transfers between the host CPU and the GPU. Finally, the MJPEG experiments on the GPU show that algorithms that are efficient on a CPU, such as the AAN fast DCT algorithm, might not yield the best performance on a GPU. On the GPU, the naive 2D DCT algorithm with optimizations for the architecture (with both branching and shared memory bank conflicts removed and the correct memory layout) performed better that the AAN fast DCT.

Revisiting with State-of-the-Market Hardware

If we were to revisit these experiments with a state-of-the-market GPU from the Kepler family, all the experiments would have been able to run; however, to obtain optimal performance, we would have to revisit some of the optimizations. The memory controllers on modern GPUs such as the Kepler are more advanced and they will now detect more access patterns, facilitating coalesced memory accesses. Nevertheless, the selection of the different memory spaces, such as shared memory and texture memory, still has to be done by the programmer. Later versions of CUDA also open up the possibility of what they call managed memory [92] between the host CPU and GPU, that is, page faults are used to trigger transfers. The most optimal solution is still to manually use the asynchronous API to transfer the data. Modern GPUs now also have larger caches that, as on the x86 architecture, cannot be managed by the programmer. In many cases, this can also speed up the computations. Another challenge of the Kepler architecture is that, since the number of processors per SMX has been increased, compensating for decreasing the clock frequency requires more active threads to obtain the same performance. The numbers of registers per SMX has not been increased, meaning that each thread has less register space, thus making it easier to run out of space. This is not a fatal problem for the applications, since the threads have a private memory space in global memory. However, it will have negative effects on performance. Using the latest generation of GPUs will therefore, in most cases, provide better performance, since the number of cores has increased many times. However, in some cases, performance can decrease and, in many cases, the applications would have the same efficiency as on the older architectures.

3.4 Cell Broadband Engine

The Cell Broadband Engine is one of the asymmetric architectures we used for experiments with an exclusive memory model. The primary application for the Cell was as the main processor in Sony's PlayStation 3 gaming console, so the processor was designed with multimedia workloads in mind. The focus of the Cell experiments was to try to learn how programmers need to think to efficiently utilize the platform.

In the following section, we take a closer look at a case study based on our paper "Tips, Tricks and Troubles: Optimizing for Cell and GPU" [112]. Our analysis of 14 different MJPEG implementations indicates that there exists great potential for optimizing performance with the Cell architecture, but there are also many pitfalls to avoid.

The Cell Broadband Engine was developed by Sony Computer Entertainment, Toshiba, and IBM. The central components in the Cell are a power processing element (PPE) containing a general-purpose 64-bit PowerPC RISC core and eight specialized synergistic processing elements (SPEs). A more detailed overview of the architecture can be found in Section 2.1.4.

3.4.1 Case Study: MJPEG Encoding

In our MJPEG case study, we conducted several experiments on the Cell architecture. A more detailed overview of the MJPEG workload can be found in Section 3.2.1. As for the GPU, several layouts are available for the Cell. However, because of the small number of more capable cores (SPEs), it is feasible to assign one frame to each core. The most time-consuming parts of the MJPEG encoding process, the DCT and quantization steps, are well suited for Cell acceleration. In addition, the VLC step can be adapted.

We also compared 14 different implementations on the Cell. The results indicate that the Cell is also a complex architecture to use and that achieving high performance is not trivial. Figure 3.25 shows performance results for encoding the 1080p tractor video clip in YUV 4:2:0. The difference between the fastest and slowest solution is 1869 ms and the fastest solutions were disk I/O bound. To gain experience of what works and what does not, we examined these solutions using the same criteria as with the GPU implementations. In general, we found that the Cell architecture has great potential, but

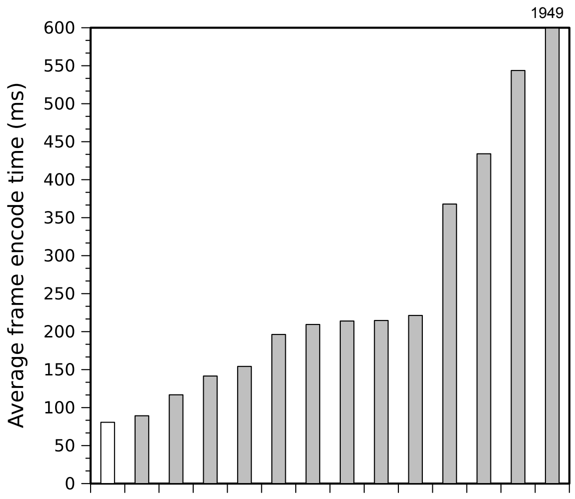


Figure 3.25: Runtime for MJPEG implementations on the Cell on a PlayStation 3 (six SPEs).

also many possible pitfalls, both when choosing specific algorithms and in implementation-specific decisions.

Cell Broadband Engine Experiments

By learning from the design choices of the implementations in Figure 3.25, we designed experiments to investigate how performance improvements are achieved on the Cell. All the experiments encoded HD video (1920x1080, 4:2:0) from raw YUV frames found in the *tractor* video test sequence. However, we used only the first frame of the sequence and encoded it 1000 times in each experiment to overcome the disk I/O limit. This became apparent at the highest level of encoding performance, since we did not have a high-bandwidth video source available. All programs were compiled with the highest level of compiler optimizations using GCC for Cell. The Cell experiments were tested on a QS22 blade server (with eight SPEs; the results in Figure 3.25 were for a PlayStation 3 with six SPEs)

Considering the *embarrassingly parallel* parts of MJPEG video encoding, a number of different layouts are available to map the different steps of the encoding process to the Cell. Because of the amount of work, the DCT and quantization steps should be executed on SPEs, but the entropy coding step can also run in parallel between complete frames. Thus, given that a few frames of encoding delay are acceptable, the approach we consider best is to process full frames on each SPE, with every SPE running DCT and quantization of a full frame. This minimizes synchronization between cores and allows us to perform VLC on the SPEs.

Regardless of the placement of the encoding steps, it is important to avoid idle cores. We resolve this situation by adding a frame queue between the frame reader and the DCT step and another queue between the DCT and VLC steps. Since a frame is processed in full by a single Cell processor, the AAN algorithm is well suited. It can be implemented in a straightforward manner to run on SPEs, with VLC coding on the PPE. We tested the same algorithm optimized with SPE intrinsics for vector processing (SIMD), resulting in

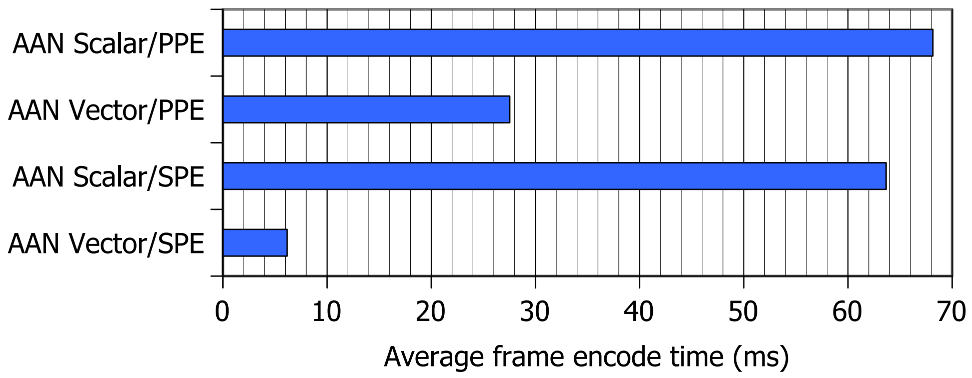


Figure 3.26: Encoding performance on the Cell with different implementations of the AAN algorithm and VLC placement.

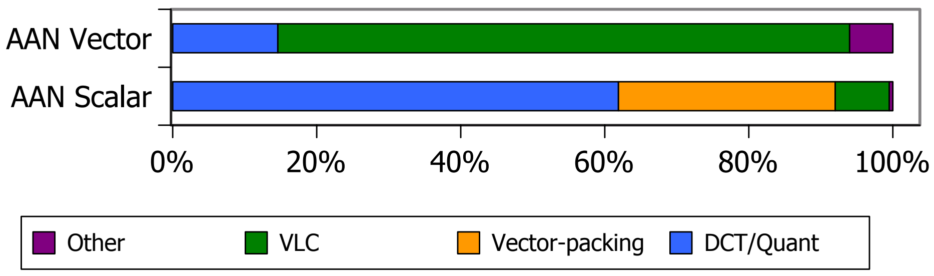


Figure 3.27: SPE utilization using scalar or vector DCT.

double encoding throughput, which can be seen in Figure 3.26 (scalar and vector PPE). Another experiment involved moving the VLC step to the SPEs, offloading the PPE. This approach left the PPE with only the task of reading and writing files to disk, in addition to dispatching jobs to SPEs. To do so, the luma and chroma blocks of the frames had to be transformed and quantized in interleaved order, that is, two rows of luma and a single row of both chroma channels. The results show that the previous encoding speed was limited by the VLC, as shown in Figure 3.26 (scalar and vector SPEs).

To gain some insight into SPE utilization, we collected a trace (using PDTR, part of the IBM software development kit for Cell) showing how much time is spent on the encoding parts. Figure 3.27 shows the SPE utilization when encoding HD frames for the scalar and vector SPEs from Figure 3.26. This distinction is necessary because the compiler does not generate SIMD code, requiring the programmer to hand-code SIMD intrinsics to achieve high throughput. The scalar version uses about four times more SPE time to perform the DCT and quantization steps for a frame than the vector version does and an additional 30% of the total SPE time to pack and unpack scalar data into vectors for SIMD operations. Our vectorized AAN implementation is nearly eight times faster than the scalar version.

With the vector version of DCT and quantization, the VLC coding uses about 80% of each SPE. This can possibly be optimized further, but we did not find time to pursue this.

The Cell experiments demonstrate the necessary level of fine-grained tuning to obtain high performance on this architecture. In particular, correctly implementing an algorithm using vector intrinsics is imperative. Of the 14 implementations for the Cell in Figure 3.25, only one offloaded VLC to the SPEs, but this was the second fastest implementation. The fastest implementation vectorized the DCT and quantization and the vector/SPE implementation in Figure 3.26 is a combination of the two. One reason why only one implementation offloaded the VLC may be that it is unintuitive. An additional communication and shift step is required in parallelizing VLC because the lack of arbitrary bit shifting of large fields on the Cell as well as the GPU prevents a direct port from the sequential codes. Another reason may stem from the dominance of the DCT step in early profiles and the awkward process of later gathering profiling data on multicore systems. The hard part is to know what is best in advance, especially because moving an optimized piece of code from one system to another can be significant work and may even require rewriting the program entirely. It is therefore good practice to structure programs in such a way that parts are loosely coupled. In that way, they can be both replaced and moved to other processors with minimal effort.

When comparing the 14 Cell implementations of the encoder shown in Figure 3.25 to find out what differentiates the fastest from the medium-speed implementations, we found some distinguishing features, the most prominent one being not exploiting the SPE's SIMD capabilities, but also in the areas of memory transfer and job distribution. Uneven workload distribution and lack of proper frame queuing resulted in idle cores. Additionally, some implementations suffered from small, often unconcealed DMA operations that left SPEs in a stalled state, waiting for the memory transfer to complete. It is evident that many pitfalls need to be avoided when writing programs for the Cell architecture and we have only touched upon a few of them. Some of these are obvious, but not all and achieving acceptable performance from a program running on the Cell architecture may require multiple iterations, restructuring, and even rewrites.

Discussion

Heterogeneous architectures such as the Cell provide large amounts of processing power, with encoding throughputs of 480 MB/s on the 1080p tractor video clip. Thus, real-time MJPEG HD encoding may be no problem. However, an analysis of the many implementations of MJPEG available and our additional testing show that it is important to use the right concepts and abstractions and that there may be large differences in the way a programmer must think.

Deciding efficiently the granularity at which data should be partitioned is very hard *a priori*. One approach is to try to design the programs in such a way that the cores are seldom idle or stall. In practice, however, multiple iterations may be necessary to determine the best approach.

Similar to data partitioning, efficient code partitioning is hard to carry out in advance. A rule of thumb is to write modular code to allow the parts to be moved to other cores. In addition, fine granularity is beneficial, since small modules can be merged again and also be executed repeatedly with low overhead. Offloading is by itself advantageous, since resources on the main processor become available for other tasks. It also improves the scalability of the program with new generations of hardware. In our MJPEG implementation on the Cell, we found that offloading DCT/quantization and VLC coding was advantageous in terms of performance, but offloading may not always provide higher throughput.

Summary

The programming models used on the Cell and the GPU require two different ways of thinking in parallel. The approach of the Cell is very similar to multithreaded programming on the x86, with the exception of shared memory. The SPEs are used as regular cores with explicit caches and the vector units on the SPEs require careful data structure consideration to achieve peak performance. The GPU model of programming is much more rigid, with a static grid used for blocks of threads and synchronization only through barriers. This hides the architecture complexity and provides a simpler concept to grasp for programmers. This notion is also strengthened by the better average GPU throughput of the implementations in Figures 3.21 and 3.25. However, to achieve the highest possible performance, the programmer must also understand the nitty-gritty details of the architecture to avoid pitfalls such as warp divergence and uncoalesced memory access.

Heterogeneous multicore architectures such as the Cell and GPUs may provide the resources required for real-time multimedia processing. However, achieving high performance is not trivial and to learn how to think and use resources efficiently, we experimentally evaluated several issues to discover tricks and problems. Generally, there are similarities, but the way of thinking must be substantially different, not only compared to an x86 architecture but also between the Cell and the GPUs. The different architectures have different capabilities that must be taken into account when both choosing a specific algorithm and making implementation-specific decisions.

The encoding throughput achieved on the two architectures was surprisingly similar. Although, the engineering effort to accomplish this throughput was much greater on the Cell, this was mainly due to the tedious process of writing an SIMD version of the encoder.

3.4.2 Implications

We learned that when working with an architecture such as the Cell, which has multiple vector processors (SIMD), it is imperative to vectorize the application, even though this can be a very tedious process. Our experiments have shown that the code on SPEs has to go through vector packing if it is not vectorized. The exclusive memory architecture of the Cell also provides very good performance, but at the cost of complexity.

With the Cell, it is also very important to choose the correct granularity for the architecture. The SPEs have only 256 kB of local storage, for both code and data, and multiple attempts are often required to find an optimal solution. Our MJPEG workload also showed the importance of finding an algorithm that is optimal for the architecture.

It is very important to consider data movement on the chip itself. Since the Cell has an exclusive memory model, the programmer has to consider the DMA transfers to move data between main memory and the local storage on the SPEs. To obtain optimal performance, often one has to implement double-buffering schemes to make sure that one can overlap memory transfers and computations.

Revisiting with State-of-the-Market Hardware

Unfortunately, IBM discontinued development of the Cell architecture and the CPU in the PlayStation 4 uses x86 cores together with a GPU.

It is possible to use multicore x86 processors instead of the Cell. These architectures use a shared memory model instead of exclusive memory, so it is simpler for the programmer. The x86 cores also have SIMD units. The closest x86 core compared to the Cell is the Intel Xeon Phi many-core processor, with up to 61 simple x86 cores with a shared memory model and 512-bit vector units per core. The shared memory model makes the programmer's life simpler; however, the SIMD version still has to be written mainly by hand and the AltiVec SIMD code from the Cell is not portable, so a new version must be written for the Xeon Phi. Several of these considerations for the Cell are valid for GPUs; however, the architectures are very different, and both the granularity and numbers of threads need to be different.

3.5 Architecture Comparison

In this chapter, we saw multiple case studies on four heterogeneous multicore architectures. These platforms are in many ways very different; however, they all have heterogeneous processing resources. Our experiments have revealed that, in all these architectures, code placement, code partitioning, and data locality have a huge effect on performance.

The Intel IXP was used at the very start of our investigations. It is a shared memory platform (i.e., all the cores can share memory), with cores with different capabilities. As on the GPUs, the IXP has multiple memory types with different properties, such as SRAM for storing packet headers and DRAM for storing packet payloads. However, the IXP differs greatly from the other three architectures. Since the IXP was built to process network traffic, it has very limited floating point support and is limited to manipulating network traffic. The IXP lives on today as a dedicated network flow processing fast fiber links at line speed for applications such as deep packet inspection. Programming the IXP was also challenging, since compilers and documentation were somewhat lacking; however, when these experiments were conducted in 2007, their performance was impressive and most state-of-the-art desktop computers in 2007 were not able to process multiple 1-Gbps network streams at line speed.

The three remaining architectures—the Cell, x86, and GPUs—are more suited for processing multimedia workloads. They all have optimizations for carrying out fast floating point operations. The GPU and Cell are built for floating point operations and the x86 architecture has been constantly extended with better floating point support since the x87 floating point coprocessor was integrated in the 80486 CPUs and since the first vector unit, called MMX, was added to the Pentium CPUs. One property that differentiates the Cell and GPUs with the x86 architecture is support for a shared memory architecture on the x86. This makes parallel programming much easier for the developer. However, it has also proven to be a challenge when scaling the number of threads that share the memory space: With more cores and threads, more traffic is also required on the CPU interconnect to make sure that no parts of the cache are dirty. Another advantage and challenge with x86 cores compared to the Cell and GPUs is that the cores on the x86 are considered fat cores, meaning that they have many features, such as branch prediction, prefetching from both caches and memory, and multithreading support. This takes up many transistors, making the CPU designs very complex, which is a challenge with regard to power consumption in systems.

The Cell and GPUs have several things in common: They both have an exclusive memory model, where the programmer is responsible for all the allocations. The SIMD programming model on the Cell is the most extreme in this respect, where the programmer must manage the transfer of code and data between main memory and local storage on the SPEs with DMA operations. The local storage is like a user-managed cache and, since each SPE has only 256 kB, it is a very limited resource. On modern GPU architectures, the caches cannot be managed by the programmer. However, the GPU has multiple memory types—on chip, off chip, and cached—which can be used for different parts of the processing. Selecting the correct memory space can sometimes be challenging. Another important factor for both the Cell and the GPU is the efficient transfer of data from main memory on the general-purpose core (the PPE on the Cell and the CPU on the GPU) and into the processing cores. To do this efficiently, asynchronous APIs such as CUDA Streams and double buffering should be used. One important detail of how the Cell and the GPU differ is in the programming model, as we saw in our MJPEG experiments (Sections 3.4.1 and 3.3.4). On the Cell, we use an SIMD model, where the programmer must use vector extensions and adapt the memory layout and algorithms to the use of SIMD vectors and operations. Nvidia uses an abstraction called SIMT on their GPUs. SIMT allows for code that uses only well-known intrinsic types but that can be massively threaded. The functionalities provided by SIMD and SIMT are very similar. However, our experience is that it is much more straightforward to port the program to the GPU and, even without significant optimizations, the GPU architecture yielded very good offloading performance compared to the native x86 architecture. To reap the full potential of the GPU architecture, one must still have a thorough understanding of the architecture, just as with the Cell. The "nail in the coffin" for the Cell architecture is, however, the fact that Sony, Toshiba, and IBM have decided not to continue its development. IBM will instead use GPUs for massively parallel workloads.

With our experiments on simple workloads, we learned that, of the four architectures we experimented with, the GPU and x86 architectures are the most promising. The GPU currently also needs a CPU as a host to run the operating system and manage the data flow. The combination of a x86 processor and a GPU is also a true heterogeneous multicore architecture; that is, the CPU has a few fat cores that are fast with operations that are not very well suited for parallelization and the GPU has many "simple cores" that are very fast at carrying out simple massively parallel operations.

3.6 Summary

In this chapter, we investigated several simple multimedia workloads running on four different heterogeneous architectures. Of the four architectures we evaluated, we are moving forward with the GPU and x86 multicore. The Cell has been discontinued and the IXP network processor is limited to network processing. In the next chapter, we take a closer look at a more complex multimedia workload, with a pipeline with different workloads, executing on a single system in real time, where we have to optimize for both the CPU and the GPU.

Chapter 4

Using Heterogeneous Architectures for Complex Workloads

The previous chapter described our experiments with offloading simple workloads to a heterogeneous architecture. In these experiments, we mainly carried out our optimizations on just the target architecture. Our experiments also showed that good architectural knowledge about the target architecture is essential when optimizing programs. For the x86 architecture, it is important to use threading and to adjust the number of threads to the number of cores available, that is, too many threads will result in reduced performance, and it is also important to use single instruction multiple data (SIMD) units on processors where possible. On graphic processor units (GPUs), we have seen that it is important to use a memory space that is optimized for one's access pattern, as well as to make sure to transfer the data as efficiently as possible to the GPUs and, finally, to try to prevent branching in any code running on the GPU. We decided to focus on x86 processors and GPUs, since the Cell has been discontinued and the IXP network processor is better suited for video processing.

In this chapter, we investigate a more complex workload based on our research on systems for real-time sports analysis [114]. The complex workload is defined in this thesis as a video stitching pipeline, optimized for multiple heterogeneous architectures, which in our case is an x86 processor and a GPU. The workload also has real-time demands, meaning that the pipeline needs to deliver a new video frame every 33 ms to produce video at 30 frames per second (fps).

This chapter is organized as follows: First, in Section 4.1, we introduce our scenario and the non–real-time prototype we implemented. Then, in Section 4.2, we take a closer look at the enhancements and optimizations to make it run in real time on a single machine. The system presented is a large system with many contributors and we focus on architectural optimizations carried out to run the system in real time. We also test how different parameters and heterogeneous architectures affect the performance of the prototype system.

4.1 Bagadus Sports Analysis System

Sports analysis has become a huge industry and a large number of (elite) sports clubs study their athletes' performance, spending great amounts of money. This analysis is conducted either manually or using one of many analytics tools. In soccer, several systems enable trainers and coaches to analyze the gameplay to improve performance. For instance, at Interplay-sports [62], videostreams are manually analyzed and annotated using a soccer ontology classification scheme. ProZone [100] automates some of the manual annotation process with video analysis software. In particular, it quantifies player movement patterns and characteristics such as athlete speed, velocity, and position and has been successfully used, for example, at Old Trafford in Manchester and Reebok Stadium in Bolton [106] in the United Kingdom. Similarly, STATS' SportVU tracking technology [111] uses videocameras to collect the players' positioning data within the playing field in real time. This information is further compiled into player statistics and performance. Camargus [17] provides a very nice video technology infrastructure but lacks other analytics tools. As an alternative to video analysis, which is often inaccurate and resource hungry, both Cairo's VIS.TRACK [16] and ZXY Sport Tracking [146] systems use global positioning and radio-based systems to capture the performance measurements of athletes. Thus, these systems can present player statistics, including speed profiles, accumulated distances, fatigue, fitness graphs, and coverage maps, in many different ways, such as with charts, three-dimensional graphics, and animations.

To improve game analytics, video that replays real-game events has become increasingly important. However, the integration of player statistics systems and video systems still requires a large amount of manual labor. For example, events tagged by coaches or other human expert annotators must be manually extracted from the videos, often requiring hours of work in front of the computer. Furthermore, connecting the player statistics to the video also requires manual work. One recent example is the Muithu system [67], which integrates coach annotations with related video sequences, but the video must be manually transferred and mapped to the game timeline.

As the above examples show, several tools for soccer analysis exist. However, to the best of our knowledge, no system exists that fully integrates all the features stated above. In this respect, earlier we presented [46] and demonstrated [105] a system called Bagadus. This system integrates a camera array video capture system with the ZXY Sport Tracking system for player statistics and a system for human expert annotation. Bagadus allows the game analytics to automatically play back a tagged game event or extract a video of events from the statistical player data, for example, all sprints at a given speed. Using the exact player positions provided by sensors, a trainer can also follow individuals or groups of players, with the videos presented either by using a stitched panorama view or by switching cameras. Our earlier work [46,105] demonstrated the integrated concept but did not address all operations, such as the generation of the panoramic video, in real time. We now present enhancements providing live, real-time analysis and video playback by using algorithms to enhance image quality, parallel processing, and offloading to heterogeneous architectures units such as GPUs. Our prototype was deployed at Alfheim Stadium (Tromsø IL, Norway) and we use a dataset captured at a Norwegian premier league game for our experiments.

4.1.1 Bagadus: The Basic Idea

Interest in sports analysis systems has increased greatly recently and sports analytics are predicted to be a real game changer, that is, "statistics keep changing the way sports are

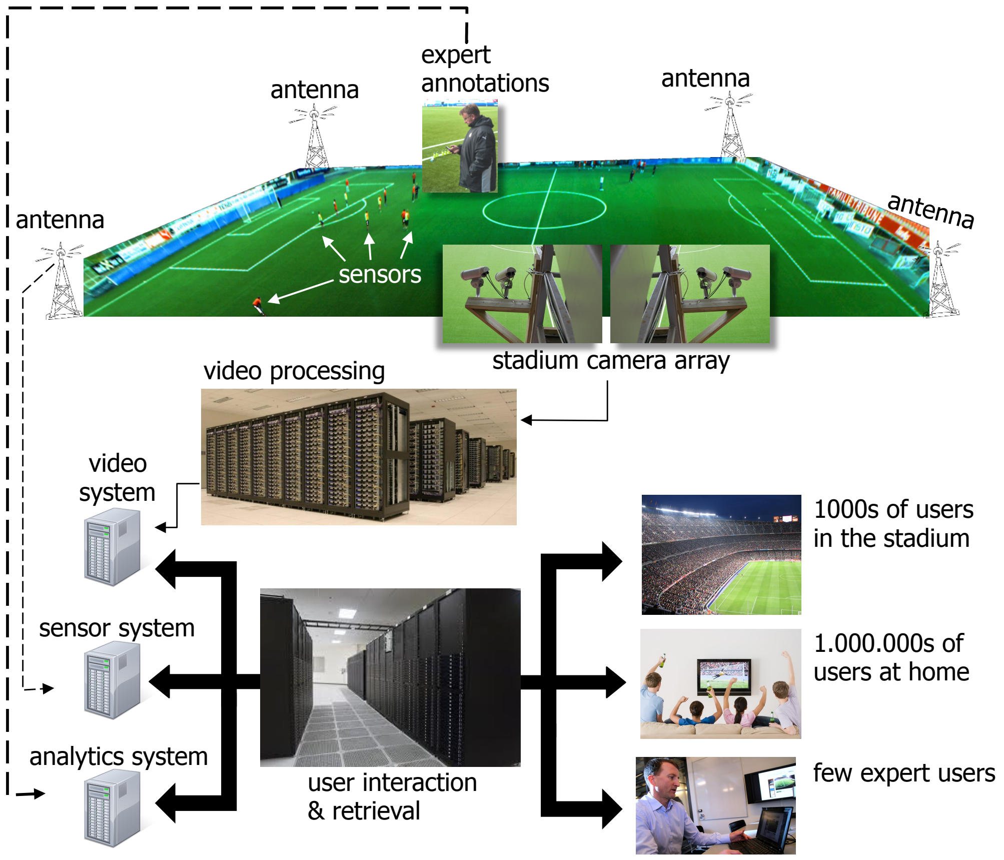


Figure 4.1: Overall sports analysis system architecture.

played—and changing minds in the industry" [29]. As described above, several systems exist, some already providing game statistics, player movements, video highlights, and so forth, since a long time. However, to a large degree, the existing systems are offline systems and require a great deal of manual work to integrate information from various computer systems and expert sport analytics. In this respect, *Bagadus* is a prototype that aims to fully integrate existing systems and enable the real-time presentation of sport events. Our system was built in cooperation with the Tromsø IL soccer club and the ZXY Sport Tracking company. A brief overview of the architecture and interaction of the different components is given in Figure 4.1. The Bagadus system is divided into three different subsystems, all of which are integrated in our soccer analysis application.

The *video* subsystem consists of multiple small shutter-synchronized cameras that record high-resolution video of the soccer field. They cover the full field with sufficient overlap to identify common features necessary for camera calibration and image stitching. Furthermore, the video subsystem supports two different playback options. The first allows playback of video that switches between streams from the different cameras, either by manually selecting a camera or automatically following players based on sensor information. The second option plays back a panorama video stitched from the different camera feeds. The cameras are calibrated in their fixed positions and the captured video is all processed and stored using a capture–debarrel–rotate–stitch–encode–store pipeline. In the offline mode, Bagadus allows a user to zoom in on and mark a player(s) in the retrieved video on the fly, but this feature is not yet supported in the live mode used during the game.

To identify and follow players on the field, we use a *tracking* (sensor) subsystem. Tracking people through camera arrays has been an active research topic since several years. The accuracy of such systems has improved greatly, but there are still errors. Therefore, for stadium sports, an interesting approach is to use sensors on players to capture their exact positions. ZXY Sport Tracking [146] employs such a sensor-based solution to provide player position information. Bagadus uses this position information to track players or groups of players in single camera views, stitched views, or zoomed-in modes.

The third component of Bagadus is an *analytics* subsystem. Coaches have long analyzed games to improve their own team's gameplay and to understand that of their opponents. Traditionally, this was done by taking notes using pen and paper, either during the game or while watching hours of video. Some clubs even hire one person per player to describe the player's performance. To reduce the manual labor, we implemented a subsystem that equips team members with a tablet (or even a mobile phone) with which they could register predefined events quickly with the press of a button or provide textual annotation. In Bagadus, the registered events are stored in an analytics database and can later be extracted automatically and shown along with a video of the event.

The tracking and analytics subsystem does not have the same processing requirements as the video subsystem does and these are therefore not presented in this thesis. More details about the tracking and analytics subsystem in Bagadus can be found in paper VIII [114].

4.1.2 Video Subsystem

To record high-resolution video of the entire soccer field, we installed a camera array using small industry cameras that together cover the entire field. The video subsystem then extracts, processes, and delivers video events based on given time intervals, player positions, and so forth.

There are two versions of the video subsystem: one non–real-time system, which is presented in this section, and one live real-time system optimized with heterogeneous architectures. This system is presented in Section 4.2.

Both video subsystems support two different playback modes. The first mode allows the user to play video from the individual cameras by manually selecting a camera or by automatically following players. The second mode plays back a panoramic video stitched from the four camera feeds. The non–real-time system plays back recorded video stored on disk and, because of the processing times, the video will not be available before the match is finished. The live system, on the other hand, supports playing back video directly from the cameras and the events are available in real time.

Camera Setup

To record high-resolution video of the entire soccer field, we installed a camera array consisting of four Basler industry cameras with a 1/3-inch image sensor supporting 30 fps and a resolution of 1280×960. The cameras were synchronized by an external trigger signal to enable a video stitching process that produces a panoramic video picture. For a minimal installation, the cameras are mounted close to the middle line under the roof covering the spectator area, that is, approximately 10 meters from the side line and 10 meters above the ground. With a 3.5 mm wide-angle lens, each camera covered a field of view of about 68 degrees; that is, all four cameras covered the full field with sufficient overlap to identify common features necessary for camera calibration and stitching (see Figure 4.2).

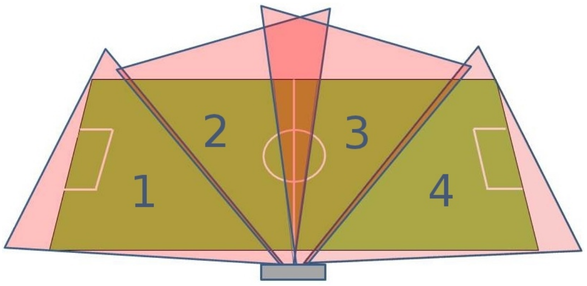


Figure 4.2: Camera setup at Alfheim stadium.

The cameras were managed using our own library, called Northlight, developed for the Verdione project [126], to manage frame synchronization, storage, encoding, and so on. We ran the system on a single computer with an Intel Core i7-3930K at 3.2 GHz, with 16 GB of memory. Northlight integrates the software development kit provided by Basler for the cameras, video encoding using x264, and color space conversion using FFmpeg.

Stitching

Tracking game events over multiple cameras is a nice feature, but in many situations a complete view of the field is desirable. In addition to camera selection functionality, we therefore generated a panoramic picture by combining images from multiple trigger-synchronized cameras. The cameras were calibrated in their fixed positions using a classical chessboard pattern [143], and the stitching operation required a more complex processing pipeline. We used alternative implementations with respect to what to store and process offline, but generally we had to 1) correct the images for lens distortion in the outer parts of the frame due to the fish-eye lens effect, 2) rotate and morph the images into panoramic perspective due to different positions covering different areas of the field, 3) correct the image brightness due to light differences, and 4) stitch the video images into a panoramic image. Figure 4.3 shows the process of combining four warped camera images into a single large panoramic image. The highlighted areas in the figure are the regions of camera overlap.

After the initial steps, the overlapping areas between the frames were used to stitch the four videos into a panoramic picture before storing it to disk. We first tried the open-source solutions given by the computer vision library OpenCV, which are based on the automatic panoramic image stitcher of Brown et al. [14]; that is, we used the auto-stitcher functions using planar, cylindrical, and spherical projections. Our analysis shows that none of the OpenCV implementations are perfect, due to large execution times and varying image quality and resolutions [46]. The fastest algorithm is the spherical projection, but it

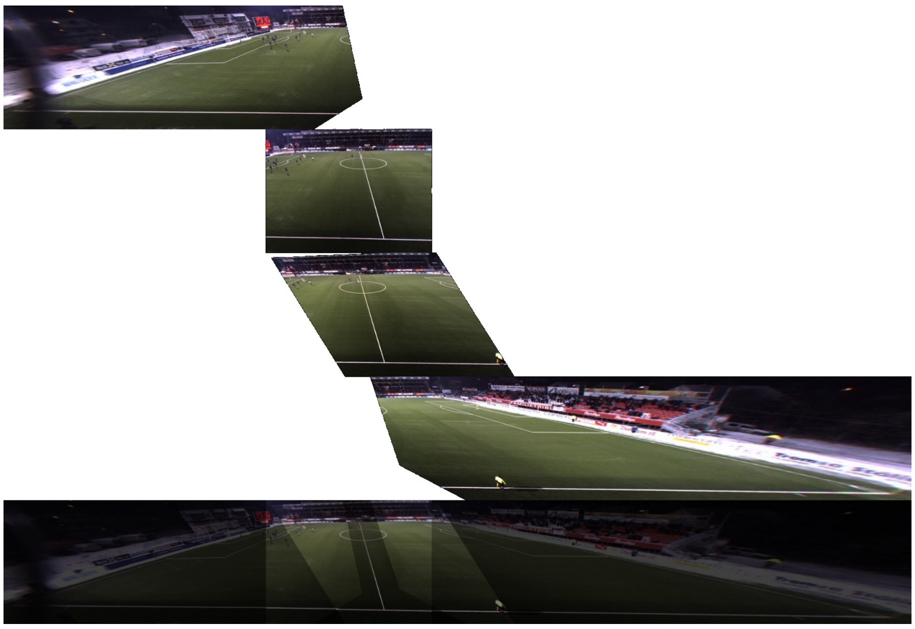


Figure 4.3: The stitching process. Each image from the four different frames is warped and combined into a panorama.

has severe barreling effects and the execution time is 1746 ms per frame, far above our real-time goal. Therefore, a different approach, homography stitching [48], was selected.

Non–Real-Time Processing-Loop Implementation

As a first proof-of-concept prototype [46], we implemented the stitching operation as a single-threaded sequential processing loop, as shown in Figure 4.4, that is, processing one frame per loop iteration.

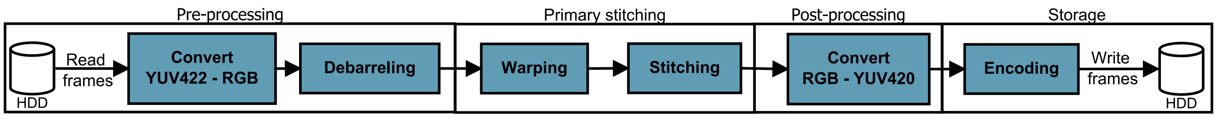


Figure 4.4: The non–real-time Bagadus stitching pipeline.

As seen in the figure, the process consists of four main parts: one pre-processing part, which reads video frames from either disk or the cameras; one part that converts the video from YUV to RGB, which is used by the rest of the pipeline; debarreling, to remove any barrel distortion from the cameras; and primary stitching. This system version used the OpenCV debarreling functions and the primary stitching part used the homography-based stitching algorithm to stitch the four individual camera frames into a 7000×960 panoramic frame. As we can see from Figure 4.5, this last part is the most resource-demanding aspect of the system. After the stitching, post-processing is responsible for converting the video back from RGB to YUV due to the x264 video encoder's lack of support for RGB. The reason for using the RGB color space is that we use OpenCV components, which are written for RGB. The single-threaded loop means that all the steps are performed sequentially for one set of frames before the next set of frames is processed.

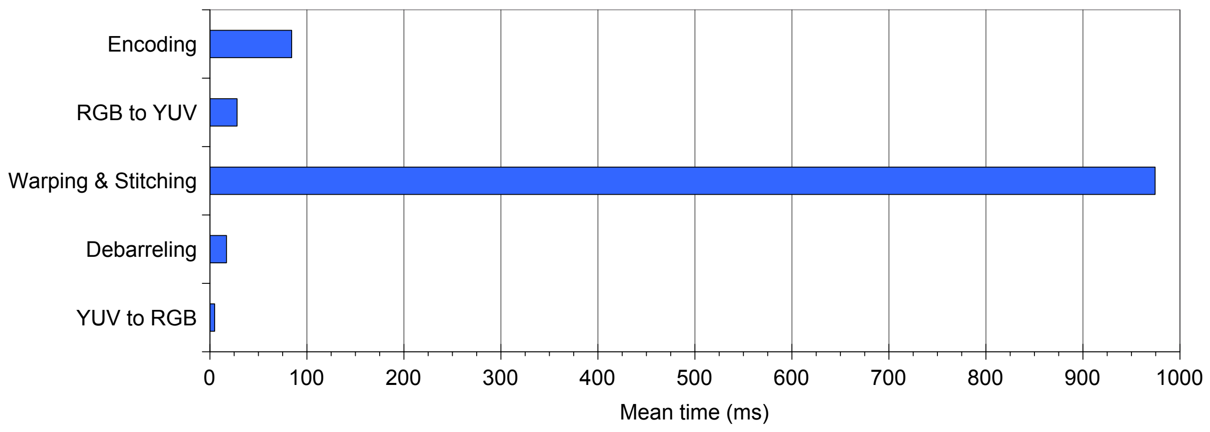


Figure 4.5: Frame processing time in the non–real-time Bagadus stitching pipeline.

The system's performance is presented in Figure 4.5 and the total execution time per panoramic frame exceeds 1100 ms, on average. To meet our 30-fps requirement, our next approach, optimized for heterogeneous architectures and presented in Section 4.2, improves performance by parallelizing and offloading several steps onto a GPU.

4.2 The Real-Time Bagadus Video Pipeline

In this section, we investigate the optimized real-time pipeline shown in Figure 4.6. We analyze the different modules in the pipeline and perform simple benchmark tests to compare the central processing unit (CPU) and GPU performance on the main modules (background subtraction, color correction, stitching, and conversion) running on the GPU. There are two main parts to the real-time pipeline: one part running on the CPU and the other running on a GPU using the CUDA framework. To implement the pipeline's real-time properties, we have to benchmark and load-balance the components with components running on the CPU and others on the GPU. We have to optimize the transfers between the CPU and GPU and try to eliminate any unnecessary transfers.

The experiments in this section were performed on an Intel Core i7-3930K six-core processor with Hyper-Threading enabled, based on the Sandy Bridge-E architecture. The machine had 32 GB of RAM and an Nvidia GeForce GTX 680 GPU based on the Kepler GK104 architecture.

The Controller Module

The single-threaded controller runs on the CPU and is responsible for initializing the pipeline, synchronizing the different modules, handling global errors and dropped frames, and transferring data between the different modules. After initialization, it waits for and receives the next set of frames from the camera reader (CamReader) module (see below). Next, it controls the transfer of data from the output buffers of module N to the input buffers of module $N+1$. This is done primarily through pointer swapping to avoid

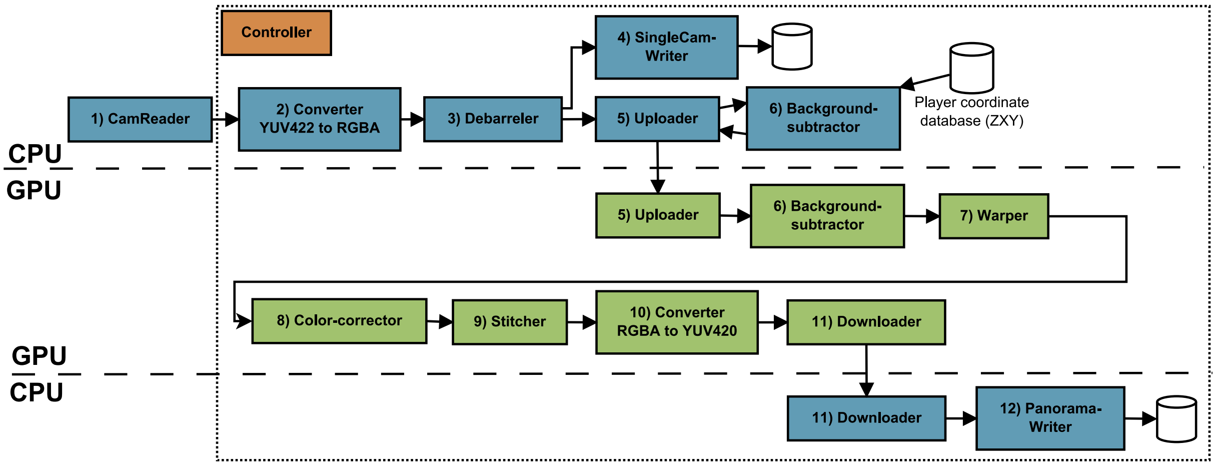


Figure 4.6: The real-time panoramic video stitching pipeline.

unnecessary memory transfers, but with memory copies as an alternative. It then signals all modules to process the new input and waits for them to finish processing. Next, the controller continues looping by waiting for the next set of frames from the reader. Another important task of the controller is to check the execution speed. If an earlier step in the pipeline runs too slowly and one or more frames from the cameras has been lost, the controller will tell the modules in the pipeline to skip the delayed or dropped frame and reuse the previous frame.

The CamReader Module

The CamReader module is responsible for retrieving frames from the Ethernet cameras. It runs on the CPU and consists of one dedicated reader thread per camera. Each of the threads waits for the next frame and then writes the retrieved frame to an output buffer, overwriting the previous frame. The cameras provide a single frame in YUV 4:2:2 format and the CamReader's frame retrieval rate determines the real-time threshold for the rest of the pipeline. As described in Section 4.1.2, camera shutter synchronization is controlled by an external trigger box and, in our current configuration, the cameras deliver a frame rate of 30 fps; that is, the real-time threshold and CamReader processing time are thus 33 ms.

The Converter Module

The CamReader module outputs frames in YUV 4:2:2 format. However, the stitching pipeline requires RGBA internally for processing and the system therefore converts frames from YUV 4:2:2 to RGBA. This process is handled by the Converter module, using *FFm-peg* and *swscale*. The processing time for these conversions on the CPU, as seen later in Figure 4.13, is well below the real-time requirement, so this operation can run as a single thread. Conversion is an embarrassingly parallel operation that can also be carried out efficiently on a GPU; however, the transfer of data to and from the GPU for a single operation would add too much latency to the system.

The Debarreler Module

Due to the wide-angle lenses used in our cameras to capture the entire field, the images delivered suffer from barrel distortion, which needs to be corrected. We found the performance of the existing debarreling implementation in the old stitching pipeline to perform fast enough. The Debarreler module is therefore still based on OpenCV's debarreling function, which is optimized for execution on CPUs with SIMD, using nearest-neighbor interpolation, and executes as a dedicated thread per camera.

The SingleCamWriter Module

In addition to storing the stitched panoramic video, we also want to store the video from the separate cameras. This storage operation is carried out by the SingleCamWriter module, which runs as a dedicated thread per camera. As noted by Halvorsen et al. [46], storing the videos as raw data proves impractical due to the size of the uncompressed raw data. The different CamWriter modules (here SingleCamWriter) therefore encode and compress frames into three-second H.264 files, which has proven to be very efficient. Due to the use of H.264, every SingleCamWriter thread starts by converting from RGBA to YUV 4:2:0, which is the input format required by the x264 encoder. The threads then encode the frames and write the results to disk. There are not many efficient H.264 encoders that can run on GPUs without dedicated hardware encoding blocks; therefore, we did not consider moving this step to the GPU.

The Uploader Module

Due to the great potential of parallelizing the panoramic workload and the high computing power of modern GPUs, large parts of our pipeline run on a GPU. We therefore need to transfer data from the CPU to the GPU, a task performed by the Uploader module. In addition, the Uploader module is also responsible for executing the CPU part of the BackgroundSubtractor module (see Section 4.2). The Uploader module consists of a single CPU thread that first runs the player pixel lookup creation needed by the background subtractor. Next, it transfers the current RGBA frames and the corresponding player pixel maps from the CPU to the GPU. This is done by the use of double buffering and asynchronous transfers (CUDA Streams). We use one stream for each camera and a stream for the pixel maps for the background subtractor.

The BackgroundSubtractor Module

Background subtraction is the process of determining which pixels of a video belong to the foreground and which belong to the background. The BackgroundSubtractor module, running on the GPU, generates a foreground mask (for moving objects such as players) that is later used in the Stitcher module to avoid seams through the players. Our background subtractor can run like traditional systems searching the entire image for foreground objects. However, we can also exploit information gained by the tight integration with the player sensor system. Through the sensor system, we know the player coordinates that can be used to improve both the performance and precision of the module. By first retrieving player coordinates for a frame, we can then create a player pixel lookup map, where we set only the players' pixels, including a safety margin, to one. The creation of these lookup maps is executed on the CPU as part of the Uploader module. The background subtractor on the GPU then uses this lookup map to process only pixels close to a player, which reduces the GPU kernel processing times from 811,793 ms to 327,576 ms, on average, on a GeForce GTX 680. When run in a pipelined fashion, the processing delay caused by the lookup map creation is also eliminated. The sensor system coordinates are retrieved by a dedicated slave thread that continuously polls the sensor system database for new samples.

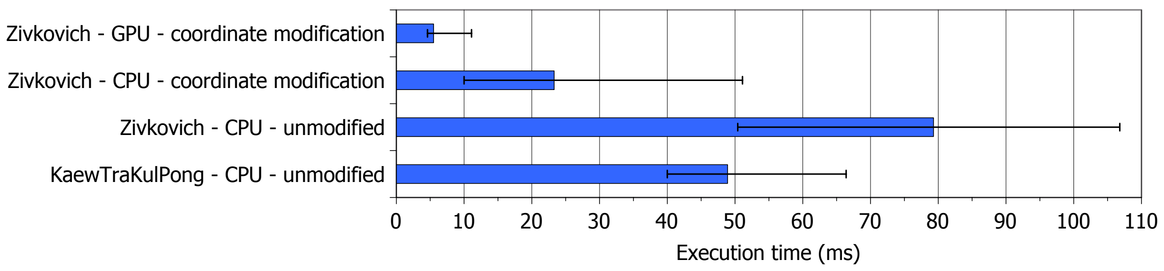
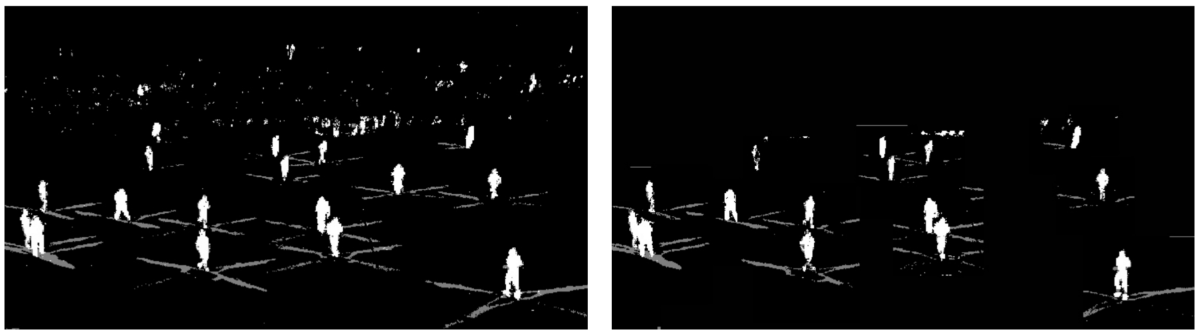


Figure 4.7: Execution time of alternative algorithms for the BackgroundSubtractor module (single camera stream).

Even though we enhanced the background subtraction with sensor data input, there are several implementation alternatives. When determining which algorithm to implement, we evaluated two alternatives: Zivkovic's [144, 145] and Kaewtrakulpong and Bowden's [68]. Even though the CPU implementation was slower (see Figure 4.7), Zivkovic's method provided the best visual results and was therefore selected for further modification. Furthermore, the Zivkovic algorithm proved to be fast enough when modified with input from the sensor system data. The GPU implementation, based on Zivkovic's [98], proved to be even faster and the final performance numbers for a single camera stream are shown in Figure 4.7. A visual comparison of the unmodified Zivkovic implementation and the sensor system-modified version is shown in Figure 4.8, where the sensor coordinate modification reduces noise, as seen in the upper parts of the figures.



(a) Unmodified Zivkovic approach.

(b) Player sensor data modification of Zivkovic's approach.

Figure 4.8: Background subtraction comparison.

The Warper Module

The Warper module is responsible for warping the camera frames to fit the stitched panorama image. By warping we mean twisting, rotating, and skewing the images to fit the common panoramic plane. As we saw from the old pipeline, this is necessary because the stitcher assumes that its input images are perfectly warped and aligned to be stitched to a large panorama. Executing on the GPU, the Warper also warps the foreground masks provided by the BackgroundSubtractor module. This is because the Stitcher module will later use the masks and therefore expects them to fit perfectly to the corresponding warped camera frames. Here, we use the Nvidia Performance Primitives (NPP) library [89] for optimized implementation.

The Color-Corrector Module

When recording frames from several different cameras pointing in different directions, it is nearly impossible to calibrate the cameras to output the exact same colors due to the different lighting conditions. This means that, to generate the best panoramic videos, we need to correct the colors of all the frames to remove disparities. In our panorama pipeline, this is done by the Color corrector module running on the GPU.

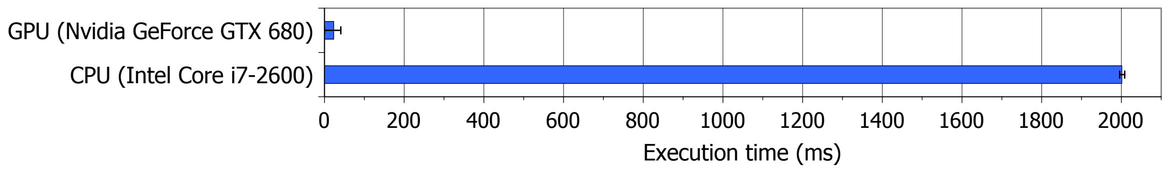
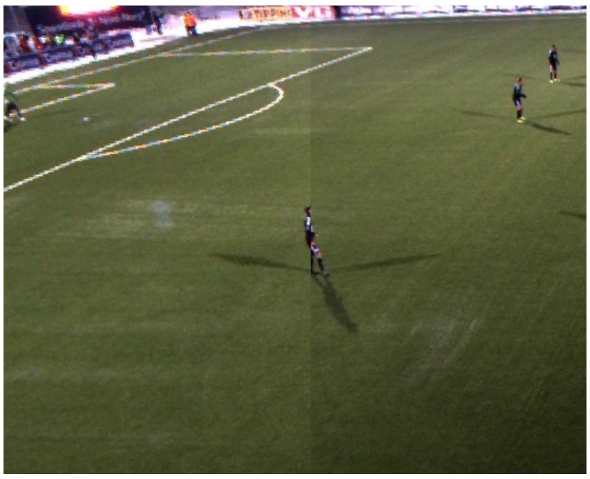


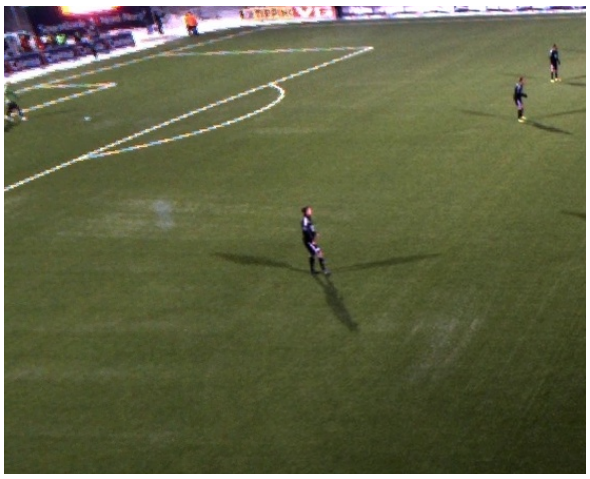
Figure 4.9: Execution time of color correction.

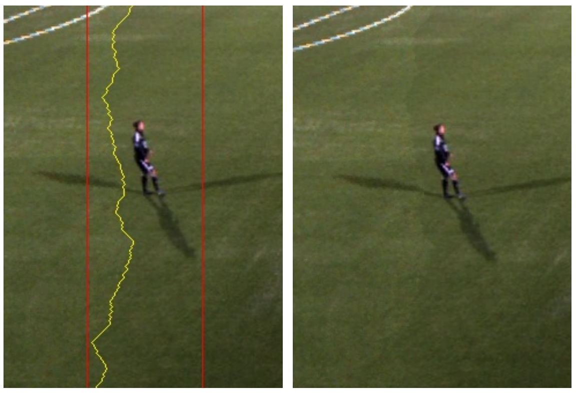
We choose to carry out the color correction after warping the images. The reason for this is that locating the overlapping regions is easier with aligned images and the overlap is also needed when stitching the images together. This algorithm is executed on the GPU, enabling fast color correction within our pipeline. The implementation is based on the algorithm presented by Xiong and Pulli [136], but with minor modifications to optimize for the GPU. We calculate the color differences between the images for every single set of frames delivered from the cameras. We color-corrected each image in sequence, meaning that each image was corrected according to the overlapping frame to the left. The algorithm implemented is easy to parallelize and does not use pixel-to-pixel mapping, which makes it well suited for our scenario. Figure 4.9 compares running the algorithm on the CPU and on a GPU. The CPU version could be further optimized with SIMD; however, the GPU implementation would still be much faster.

The Stitcher Module

As in the old non–real-time pipeline, we use a homography-based stitcher where we simply create seams between the overlapping camera frames and then copy pixels from the images based on these seams. These frames need to follow the same homography, which is why they have to be warped two steps back in the pipeline. In our old pipeline, we used static cuts for seams, which meant that a fixed rectangular area from each frame was copied directly to the output frame. Static cut panoramas are faster but can introduce graphical errors in the seam area, especially when there is movement in the scene, as illustrated in Figure 4.10(a).


(a) The original fixed cut stitching with a straight vertical seam.


(b) The new dynamic stitching with color correction.


(c) Dynamic stitching with **no** color correction. The left image shows the seam search area between the red lines and the seam in yellow. The right image clearly shows the seam going outside the player, but there are still color differences.

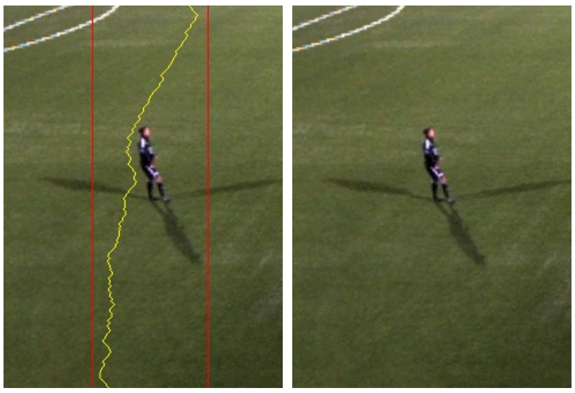

(d) Dynamic stitching **with** color correction. The left image shows the seam search area between the red lines and the seam in yellow. The right image has no visible seam and no color differences.

Figure 4.10: Stitcher comparison, improving the visual quality with dynamic seams and color correction.

To make a seam with a better visual result, we therefore introduced a dynamic cut stitcher instead of the old static cut. The dynamic cut stitcher creates seams by first creating a rectangle of adjustable width over the static seam area. Then, it treats all pixels within the seam area as graph nodes. The graph is directed from the bottom to the top in such a way that each pixel points to the three adjacent pixels above (the left- and right-most pixels only point to the two pixels available). These edges' weights are calculated by using a custom function that compares the absolute color differences between the corresponding pixels in each of the two frames we are trying to stitch. The weight function also checks the foreground masks from the BGS module to see if any player is in the pixel and, if so, it adds a large weight to the node. In effect, both these steps create edges between nodes where the colors differ and the players present have much larger weights. We then run the Dijkstra graph algorithm [28] on the graph to create a minimal cost route from the start of the offset at the bottom of the image to the end at the top. Since our path is directed upward, we can only move up or diagonally from each node and we only obtain one node per horizontal position. By looping through the path, we therefore obtain our new cut offsets by adding the node's horizontal position to the base offset. An illustration of how the final seam looks is shown in Figure 4.10(b), where the seams without and with color correction are shown in Figures 4.10(c) and 4.10(d).

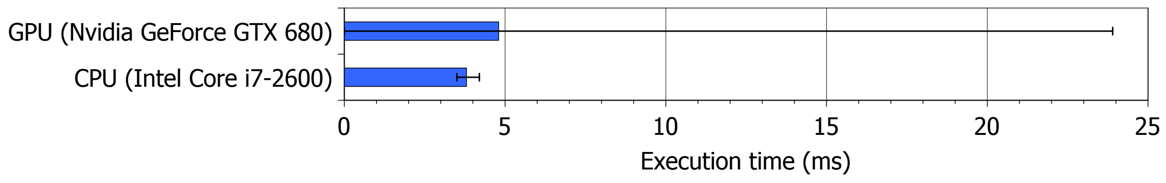


Figure 4.11: Execution time for dynamic stitching.

The timings for the dynamic stitching module are shown in Figure 4.11. The CPU version is currently slightly faster than our GPU version, since this algorithm is more serial than the other image processing algorithms. Searches and branches are also more efficient on traditional CPUs, but further optimization of the CUDA code could improve this GPU performance; however, this is not needed, since we are well within the real-time requirements. The performance difference between the GPU and CPU versions is also not large enough to justify moving the module to the CPU, which would also add delay by transferring the data over the PCI Express bus.

The YuvConverter Module

Before storing the stitched panoramic frames, we need to convert back from RGBA to YUV 4:2:0 for the H.264 encoder, just as in the SingleCamWriter module. However, due to the size of the output panorama, this conversion is not fast enough on the CPU, even with the highly optimized *swscale* library that uses SIMD. This module is therefore implemented on the GPU. For the GPU version, we based the module on a function from Nvidia's NPP [89]. The NPP contains several conversion primitives, but no direct conversion from RGBA to YUV 4:2:0. The GPU-based version therefore first uses the NPP to convert from RGBA to YUV 4:4:4 and we wrote a small CUDA kernel to carry out the final conversion from YUV 4:4:4 to YUV 4:2:0.

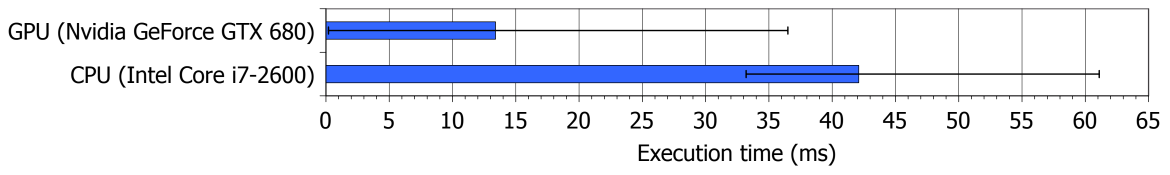


Figure 4.12: Execution time for conversion from RGBA to YUV 4:2:0.

Figure 4.12 compares the performance of the CPU-based implementation with that of the optimized GPU-based version. These results show that the GPU version, even with a two-step conversion, is over twice as fast as the CPU version.

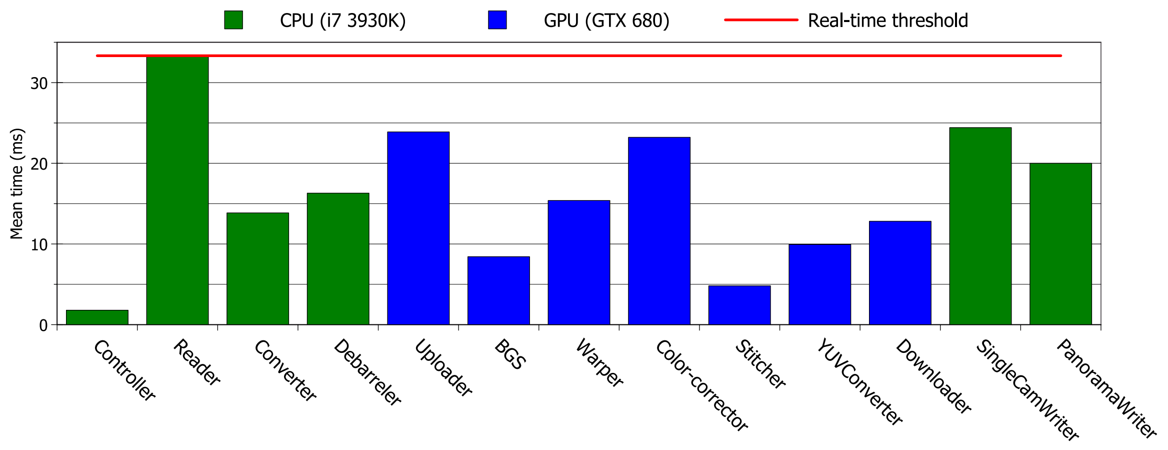


Figure 4.13: Improved real-time pipeline performance: module overview with default setup.

The Downloader Module

Before we can write the stitched panorama frames to disk, we need to transfer them back to the CPU, which is carried out by the Downloader module. It runs as a single CPU thread that synchronously copies a frame to the CPU. We could have implemented the Downloader module as an asynchronous transfer with double buffering, like the Uploader, but since the performance, as shown in Figure 4.13, is very good, this is left as a further optimization.

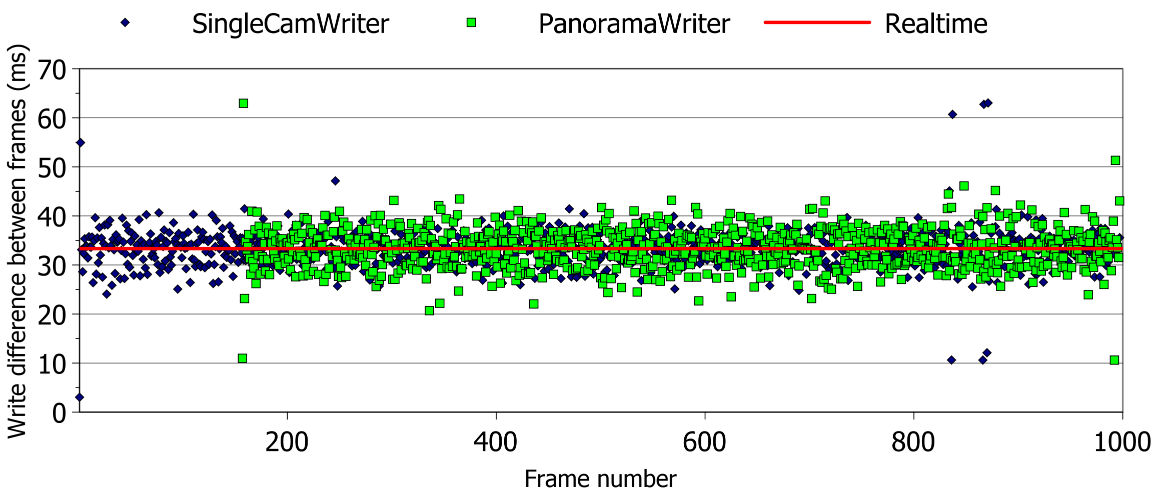
The PanoramaWriter Module

The last module, executing on the CPU, is the Writer module, which writes the panoramic frames to disk. The conversion from RGBA to YUV was already done on the GPU, so the only steps the PanoramaWriter needs to follow are to first encode the input frame to H.264 and then write the result to disk as three-second H.264 video files.

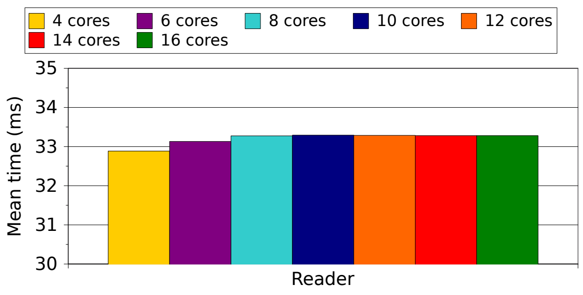
4.2.1 Performance Analysis

To evaluate the performance of our pipeline, we used an off-the-shelf PC with an Intel Core i7-3930K processor and an Nvidia GeForce GTX 680 GPU. We benchmarked each individual component and the pipeline as a whole, capturing, processing, and storing 1000 frames from the cameras.

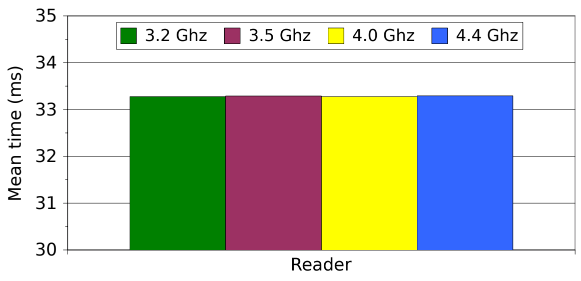
In the non–real-time pipeline [46], the main bottleneck was panorama creation (warping and stitching). This operation alone used *974 ms per frame*. As shown by the performance breakdown into individual components in Figure 4.13, the new pipeline was greatly improved. Note that all the individual components run concurrently in real time on the same set of hardware. All of these, however, add up to times far longer than 33 ms. The reason why the pipeline still runs in real time is because several frames are processed in parallel. Note that all CUDA kernels are executed at the same time on a single GPU, so the performance of the GPU modules is affected by that of the other GPU modules. On earlier GPUs from the Tesla architecture (e.g., the GTX 280), where different CUDA



(a) Pipeline write differences (showing the times for 1000 frames).



(b) Core count scalability.



(c) Core frequency scalability.

Figure 4.14: Inter-departure times of frames when running the entire pipeline. In a real-time scenario, the output rate should follow the input rate (given here by the trigger box) of 30 fps (33 ms).

kernels were serialized, this was not possible. However, the Fermi architecture (GTX 480 and above) introduced concurrent CUDA kernel execution [88]. Thus, since the Controller module schedules the other modules according to an input rate of 30 fps, the resources are sufficient for real-time execution.

For the pipeline to function in real time, the output rate should follow the input rate, that is, deliver all output frames (both for four single cameras and for one panorama) at 30 fps. Thus, to give an idea of how often a frame is written to file, Figure 4.14 shows individual and average frame inter-departure rates. The figures show the time differences between consecutive writes for the generated panorama, as well as for the individual camera streams. Operating system calls, interrupts, and disk accesses most likely cause small spikes in the write times (as seen in the scatter plot in Figure 4.14(a), but as long as the average times are equal to the real-time threshold, the pipeline can be considered to run in real time. As shown in Figures 4.14(b) and 4.14(c), the average frame inter-arrival time (Reader) is equal to the average frame inter-departure time (both SingleCamWriter and PanoramaWriter). This is also the case when testing other CPU frequencies and numbers of available cores. Thus, the pipeline runs in real time.

As stated above and seen in Figure 4.14(a), there is a small latency in the panorama pipeline compared to writing the single cameras immediately. The 33 ms are due to the camera frame rate of 30 fps, meaning that even though a module may finish before the threshold time, the Controller module will make it wait until the next set of frames arrives before signaling it to re-execute.

We added a five-second input buffer to the pipeline, because the sensor system has a latency of at least three seconds before the data are ready for use, and we added a two-second buffer for safety and GPU processing. This means that the end-to-end time from when a picture is recorded by the camera until it is stored on disk is 5.33 seconds per frame, on average.

4.2.2 Discussion

The first non–real-time prototype aimed at full integration at the system level, rather than optimization for performance. However, the challenge with the real-time pipeline has been increased by aiming at running the system in real time on low-cost, off-the-shelf hardware.

The new real-time capability also enables future enhancements with respect to functionality. For example, several systems have already demonstrated their ability to serve available panoramic video to the masses [53, 83] and, by generating the panoramic video live, enables the audience to mark and follow particular players and events. We can also use this information to create video playlists [66] automatically, providing a video summary of extracted events.

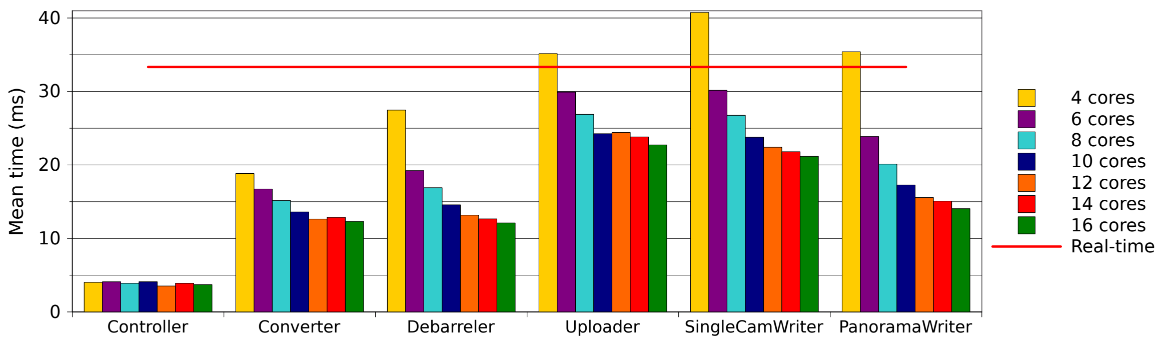


Figure 4.15: Core count scalability.

Due to limited availability of resources during these experiments, we were not able to test our system with more cameras or with higher-resolution cameras. However, to still obtain an impression of the scalability capabilities of our pipeline, we performed several benchmark tests, changing the number of available cores and, the processor clock frequency, and experimented with GPUs from different architectures and with different computing resources.

Figure 4.15[1] shows the results changing the number of available cores that can process the many concurrent threads in the CPU part of the pipeline (Figure 4.14(b) shows that the pipeline is still in real time). As we can observe from the figure, every component

[1]Note that this experiment was run on a machine with more available cores (16), each at a lower clock frequency (2.0 GHz), compared to the machine installed at the stadium, which was used for all the other tests.

runs in real time, using more than four cores and the pipeline as a whole using eight or more cores. Furthermore, the CPU pipeline contains a large but configurable number of threads (86 in the current setup) and, due to the many threads of the embarrassingly parallel workload, the pipeline seems to scale well with the number of available cores.

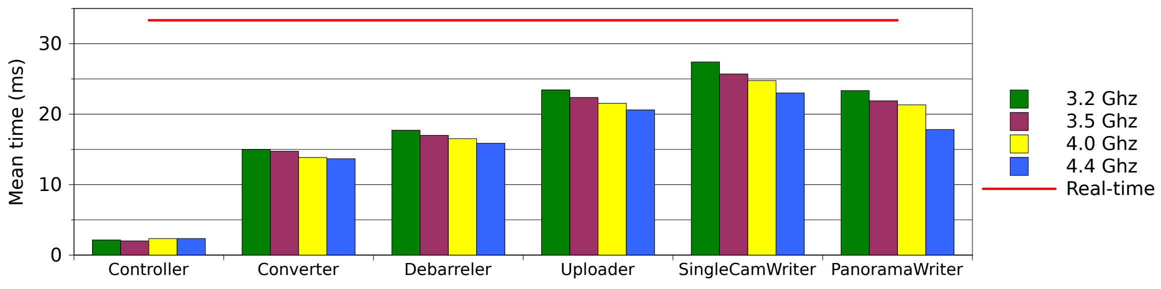


Figure 4.16: CPU frequency scalability.

Similar conclusions can be drawn from Figure 4.16, where the processing time is reduced with a higher processor clock frequency; that is, the pipeline already runs in real time at 3.2 GHz and scaling is almost linear with CPU frequency (Figure 4.14(c) shows that the pipeline is still in real time). The H.264 encoder scales especially well when scaling the CPU frequency.

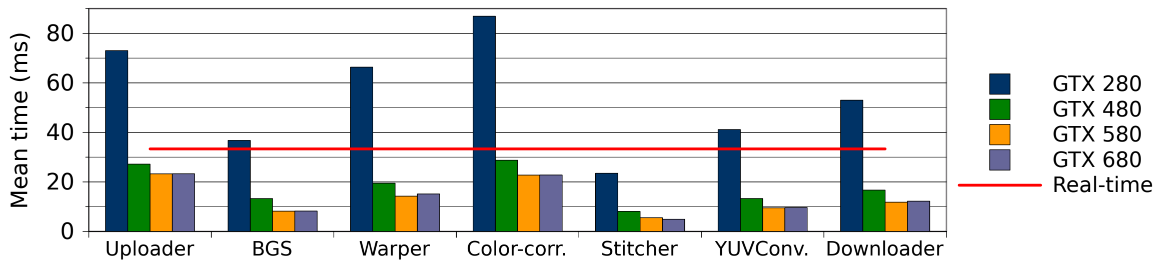


Figure 4.17: GPU comparison.

With respect to the GPU part of the pipeline, Figure 4.17 plots the processing times using different GPUs. The high-end GPUs GTX 480 and above (based on the Fermi architecture) all achieve real-time performance in the current setup. The GTX 280, based on the Tesla architecture, does not support the concurrent CUDA kernel execution introduced with the Fermi architecture [88] and performance is therefore lower than in real time, since the kernels executing on the GPU must be serialized. As expected, more powerful GPUs reduce the processing time. This is shown for the GTX 580 results. The GTX 580 is based on the same Fermi architecture as the GTX 480; however, it has 512 cores versus 480 cores and the cores clocked at a higher frequency. We can also see that the GTX 680 GPU, which is based on the newer Kepler architecture, performs better than the GTX 580 in some cases and has slightly lower performance in other cases. This is due to the fact that we developed and optimized this pipeline for the Fermi architecture, not the Kepler architecture.

Looking at the bandwidth used on the PCI Express bus between the CPU and GPU, we only use a small portion of the pipeline. Our Uploader module takes 737 MB/s and the downloader takes 291 MB/s. The theoretical bandwidth of a 16-lane PCI Express 3.0 link is 16 GB/s. For now, one GPU fulfills our real-time requirement. We did therefore not experiment with multiple GPUs, but the GPU processing power can easily be increased by adding multiple cards. Profiling of the modules running on the GPU showed that the pipeline uses seven kernels running concurrently on the GPU. These seven kernels have an average compute utilization of 14.8% on the latest-generation GPU based on the Kepler GK110 architecture. Thus, based on these results, we believe that our pipeline can be scaled up to both higher numbers of cameras and higher-resolution cameras.

4.3 Summary

Where people earlier used a huge amount of time to analyze games manually, Bagadus is an integrated system that automatically manages the required operations and video synchronization. For example, in online mode, Bagadus receives expert-annotated events from the analytics team and enables immediate playback during a game or a practice session. To enable this feature, we distributed the workload of the video subsystem on both the CPU and GPU. Our experiments show that the pipeline can run in real time on a low-cost six-core machine with a commodity GPU. To achieve this, each component in the pipeline was optimized for the target architecture, both as a standalone component and as a part of a pipeline. We had to carefully consider which of the components we wanted to run on the GPU and the CPU. Some of the components needed input from the network, some needed to write to storage, and some components needed input from other parts of the system, such as the tracking subsystem. This meant that modules such as the camera readers and writers had to run on the CPU.

An important step in the optimizations was the benchmarking of all the separate modules together as a complete pipeline. One of the lessons we learned was that, even though the standalone module is faster on the CPU, we had to keep this part of the pipeline on the GPU, since the next module in the pipeline had to be executed on the GPU. The CPU version of this module could not meet our real-time requirements. Moving only the dynamic stitcher module to the CPU for processing would also add extra latency to the system because of the extra transfer over the PCI Express bus. The same lesson also goes the other way: In some cases (i.e., with the Debarreling module) the workload might be very well suited for offloading to the GPU; however, the next step in the pipeline would require processing on the CPU.

Another lesson learned during our experiments was that the real-time pipeline uses seven kernels running concurrently on the GPU and all seven kernels combined had a average compute utilization of just 14.8% on the GPU. This means that we could potentially share the GPU with other workloads. When programming a complex workload such as the video subsystem of the Bagadus system, we had to have detailed knowledge about both the GPU and CPU architectures in our target system. Several iterations of optimizations are often required to make a complex workload such as the Bagadus video pipeline run in real time. This process involves much trial and error. If we were to change the hardware used for our system, which components to run on the GPU and on the CPU might have to be redetermined.

To make this process easier, we therefore need a framework that is aware of an application's real-time requirements so that, with the help of instrumented runs of the application, the framework can move resources between the GPU and CPU for optimal execution. Ideally, we would just want the programmer to express the maximum level of parallelism in the program code and have the framework generate multiple versions for any heterogeneous architectures that are supported. The framework would also be able to dynamically adapt the pipeline when the hardware resources changed. A processing framework would also be able to utilize the resources on the GPU more efficiently. Our experiments show that the average compute utilization of the GPU when executing all our seven kernels was 14.8%. If the real-time requirements were met, the framework would be able to schedule a greater workload on the GPU.

The Bagadus prototype presented here supports four 1K cameras and runs on a single computer. The next-generation camera setup use five 2K cameras. To support this setup, the Bagadus video pipeline was extended to run on multiple machines. This was implemented after the completion of this thesis. The latest version of the Bagadus pipeline [40,80,97] runs in real time on multiple machines connected with a PCI Express-based interconnect.

In the next chapter, we introduce a programming language and framework designed for the real-time processing of multimedia workloads on heterogeneous architectures. This system is designed to enable complex workloads such as Bagadus to run in real time on distributed systems with heterogeneous architectures without manually tweaking the system.

Chapter 5

The P2G Framework and the Future

In the previous chapter, we saw that complex multimedia workloads with real-time constraints are well suited for heterogeneous architectures. However, we observed that many optimizations were required to make sure that the processing time of all the components was less than the real-time threshold and that the workloads were executed on the optimal architecture for the particular workload or task.

In this chapter, we take a take a closer look at a framework called P2G that we designed for the real-time processing of multimedia workloads on heterogeneous architectures. The goal of this framework is to automate the parallelization of the program code and provide programmers a unified programming abstraction for writing multimedia workloads for heterogeneous architectures. This framework is a work in progress. At the conclusion of this thesis, a simple prototype was running on a single multicore machine with a shared memory architecture. We used two simple multimedia workloads to test the feasibility of our system.

This chapter is organized as follows: First, in Section 5.1, we summarize some of the challenges we observed in previous chapters. In Section 5.2, we present ideas for designing the framework and take a closer look at other frameworks for distributed processing. In Section 5.3, we present related work on other processing frameworks. Next, in Section 5.4, we present the architecture and programming model of our P2G framework and evaluate two simple multimedia workloads with the prototype implementation. Finally, in Section 5.5, we discuss the future vision of our framework.

5.1 Summary of Challenges

In Chapters 3 and 4, we experimented with heterogeneous architectures for simple and complex multimedia workloads. We observed that the architectures can efficiently process the workloads. However, challenges remain, especially for the programmer.

We saw that many of our simple workloads had to be optimized for the architectures to run efficiently. On the x86 and the Cell, single instruction multiple data (SIMD) programming is recommended to process more data per cycle. We also saw the importance, on the x86 architecture, of adapting the number of threads used by the workload to the number of cores in the system. If too many threads are used, performance suffers because of scheduling overhead. If the workload easily scales too many threads, another possibility is to use a graphics processing unit (GPU). When using a GPU, the programmer has to be aware that it takes time to transfer data to and from the GPU, so the workload has to be large enough to compensate for the transfer overhead. There are also several potential pitfalls of not using the memory on the GPU correctly and not optimizing the transfers from the host central processing unit (CPU) to the GPU.

Finally, when programmers want to process more complex multimedia workloads with real-time constraints that execute on multiple heterogeneous architectures, a great deal of profiling, tweaking, and optimization is required to make sure that the right components run on the right architecture.

5.2 Design Ideas for a New Processing Framework

Our work with both simple and complex multimedia workloads led to several observations and challenges with optimizing parts of programs and entire pipelines for different heterogeneous architectures. We want to use some of these ideas when designing our framework for processing multimedia workloads on heterogeneous architectures.

One of the first observations when working with the x86, Cell, and GPUs was that programmers tend to prefer the single instruction, multiple threads (SIMT) abstraction used on GPUs when they have small kernels that execute on the same data over the more rigid SIMD abstraction found on the x86 and the Cell. The main advantage with SIMT is that the programmer can think in terms of scalar code when writing programs. All the architectures we worked with use a C-like programming language, so we want to keep C as the programming language when designing the framework.

Furthermore, moving data between the different architectures is a challenge for many programmers. It is therefore important to design the framework in such a way that it takes care of data transfers. We also want to present the data as arrays to programmers, because this is a familiar data representation when working with multimedia workloads. For scheduling multimedia workloads, we want to use the two-level scheduling that we experimented with on a very simple scheduling simulator [115]. Here, a high-level scheduler (HLS) has global control of all the resources available and a low-level scheduler executes the workloads and time slicing on the different processing cores.

It is important for the framework to track data dependencies in the pipelines. If the dependency tracker is efficient and fine grained, the framework will be able to expose both task parallelism and pipeline parallelism in the programs execution. Dependency tracking is also important for the framework when moving data between different processing cores. To efficiently carry out fine-grained dependency tracking, programmers must write their applications in such way that they express as much parallelism as possible.

5.3 Existing Processing Frameworks

A great deal of research aims at solving challenges with parallel and distributed processing of large quantities of data. Two of the most popular frameworks for distributed processing are Google's MapReduce [27] and Microsoft's Dryad [63]. In addition, we have System S from IBM [41], PigLatin from Yahoo [93], Cosmos [85], Scope [18], and DryadLINQ [140].

MapReduce uses a data-parallel model and is based on keys and values. There are several implementations of MapReduce for multicore processors [103], clusters [4], GPUs [49], and even the Cell Broadband Engine [26]. To process data relations among heterogeneous data more efficiently, which is not supported by the original MapReduce model, Map–Reduce–Merge [138] was introduced. The Oivos project [124] addresses the same issues but, in addition, the system provides a more expressive, declarative programming model. Reducing the layering overhead of the software running on the top of MapReduce is the goal of Cogset [125], where the architecture of the processing is changed to increase performance.

The Dryad, Cosmos, and System S frameworks have several properties in common. All three use directed graphs to model communication between the processing stages and execute them on a cluster. System S also supports cycles in the graphs. However, since all of these systems are closed source, many details are unknown. Compared to the data-parallel MapReduce, which is one of the most cited paradigms for expressing parallel workloads, both Dryad and System S use a task-parallel model.

A limitation of MapReduce, Dryad, and Cosmos is their inability to model interactive algorithms. The rigid semantics of MapReduce does not map well to all types of problems and workloads [138], which in many cases may lead to decreased performance and unnaturally expressed solutions [127]. An alternative frameworks to MapReduce is the Khan process network (KPN). KPNs support arbitrary communication graphs with cycles and are deterministic. However, not many general-purpose KPN implementations exist. Some known implementations include the Sesame project [123], YAPI [70], and the Nornir framework [128]. These frameworks have several benefits, but for application developers the KPN model has challenges. One of the main challenges is that the communication channels between the processes must be specified manually and, in an environment without a shared memory model, distributed deadlock detection must be implemented.

An alternative is a framework based on a process network paradigm, such as StreamIt [44]. Here, we have a language and runtime for the implementation of streaming programs that are described by a graph with computational blocks, called filters, that has a single input and output. The filters can be combined in loops and fork/join patterns but must provide bounds on the number of messages produced and consumed, making a StreamIt graph a synchronous data flow process network [74]. The framework supports multiple machines and processors. However, this must be specified at compilation time.

Processing and developing distributed multimedia applications is more complex than for traditional sequential applications. Multimedia workloads often have strict requirements and deadlines. Iterative processing is also essential for live multimedia workloads, such as Bagadus. Thus, all existing frameworks have shortcomings that are hard to address and the traditional batch processing frameworks simply come up short in our multimedia scenario. In the next section, we describe the design ideas and a basic implementation of our new framework for real-time multimedia processing.

5.4 The P2G Framework

The basic idea behind the P2G framework comes from the observation that most of the frameworks for distributed processing lack support for real-time multimedia workloads. The frameworks often also sacrifice task and data parallelism. With data parallelism, multiple processing cores perform the same operation over multiple disjoint data chunks. Task parallelism uses multiple processing cores to perform different operations in parallel.

Many of the existing processing frameworks optimize for either task or data parallelism, but not both. This means that they can potentially limit the ability to express the parallelism of a given workload. For example, MapReduce and its related approaches provide considerable support for parallelization but restrict runtime processing to data parallelism [39]. Functional languages such as Haskell [52], Erlang [6], and the event-based Specification and Description Language [65], map well to task parallelism. In these languages, programs are expressed as communicating processes either through event distribution or message passing. This makes it challenging to express data parallelism without specifying a fixed number of communication channels.

For multimedia workloads, the Nornir framework improves on some of the shortcomings of the traditional batch processing frameworks, such as Dryad and MapReduce. KPNs are deterministic; each execution of a process network produces the same output given the same input. KPNs also support arbitrary communication graphs (with cycles/iterations), while MapReduce and Dryad restrict application developers to a parallel pipeline structure and directed acyclic graphs (DAGs). However, Nornir is only task parallel and data parallelism must be explicitly added by the programmer. Furthermore, as a framework for heterogeneous multicore architectures, Nornir still has challenges. For example, the message-passing communication channels, with exactly one sender and one receiver, are modeled as infinite first in, first out queues. In real-life heterogeneous architectures, however, queue length is limited by available memory. A heterogeneous and distributed Nornir implementation would therefore require a distributed deadlock detection algorithm. Another issue is the complex programming model. The KPN model requires the application developer to specify the communication channels between the processes manually.

P2G builds on some of the knowledge gained from Nornir and we address the requirements from multimedia workloads, with inherent support for deadlines. A particularly desirable feature of processing multimedia workloads is the automatic combination of task and data parallelism. Intra-frame prediction in H.264 or VP8, for example, introduces many dependencies between the sub-blocks of a frame and, together with other overlapping processing stages, these operations have great potential to benefit from both types of parallelism. Multimedia algorithms are iterative and exhibit many pipeline parallel opportunities. It is hard to exploit them, because an intrinsic knowledge of fine-grained dependencies is required and it is difficult to structure programs in such a way that pipeline parallelism can be used. Thies et al. [122] wrote an analysis tool for finding parallel pipeline opportunities by evaluating memory accesses, assuming stable behavior. They evaluated their system on multimedia algorithms and significantly increased parallelism by utilizing the complex dependencies found. In the P2G framework, application developers model data and task dependencies explicitly, which enables the runtime system to automatically detect and take full advantage of all parallel opportunities without manual intervention.

The main source of non-determinism in the other languages and frameworks lies in the arbitrary order of read and write operations from and to memory. This source of non-deterministic behavior can be removed by using a strict write-once semantics for writing to memory [8]. Several languages take advantage of the concept of single assignment, including Haskell [52] and Erlang [6]. This enables the schedulers to determine when code depending on a memory cell is runnable. This is a key concept that we adopted for P2G. While write-once semantics are well suited for a scheduler's dependency analysis, it is not straightforward to think about multimedia algorithms in the functional terms of Erlang and Haskell. Multimedia algorithms tend to be formulated in terms of iterations of sequential transformation steps. They act on multidimensional data arrays (e.g., the pixels in a picture) and frequently provide very intuitive data partitioning opportunities (e.g., 8*8 pixel macroblocks in a picture). Prominent examples are the computation-heavy MPEG-4 Advanced Video Coding encoding [64] and scale-invariant feature transform [75] pipelines. Both are also examples of algorithms whose subsequent steps provide data decomposition opportunities at different granularities and along different dimensions of input data. Consequently, P2G should allow programmers to think in terms of fields without losing write-once semantics. In this proof-of-concept implementation of P2G, we use multidimensional arrays to implement fields.

The ability to carry out flexible partitioning requires the processing of clearly distinct data units without side effects. The idea in P2G is to use *kernels* as in stream processing [44, 92]. This is the same paradigm used by GPUs and, when we experimented with simple workloads in Chapter 3, the preferred paradigm of programmers. A kernel describes the transformation of multidimensional fields of data. When such a transformation is formulated as a loop of equal steps, the field should instead be partitioned and the kernel instantiated to achieve data-parallel execution. Each of these data partitions and tasks can then be scheduled independently by the schedulers, which analyze dependencies and guarantee a fully deterministic output, independent of order, due to the write-once semantics of fields.

Together these observations determine four basic ideas for the design of P2G:

- The use of *multidimensional fields* as the central concept for storing data in P2G to achieve straightforward implementations of simple and complex multimedia algorithms.
- The use of *kernels* that process slices of fields to achieve data decomposition.
- The use of *write-once semantics* to such fields to achieve deterministic behavior.
- The use of *dependency analysis at runtime* at a granularity finer than entire fields to achieve task decomposition along with data decomposition.

The P2G framework is designed to be language independent; however, for this prototype, we defined a C-like language that captures many of P2G's central concepts. As such, the P2G language was inspired by several other languages. Cray's Chapel [19] language antedates many of P2G's features in a more complete manner. However, P2G adds write-once semantics and support for multimedia workloads. Furthermore, P2G programs consist of interchangeable language elements where the programmer formulates data dependencies with fetch and store statements between implicitly instantiated kernels, (currently) written in C and C++. The biggest deviation from most other modern language designs is that the P2G kernel language makes both message passing and parallelism implicit and allows users to think in terms of sequential data transformations. Furthermore, the P2G concept supports deadlines, which allows for scheduling decisions such as termination, branching, and the use of alternative code paths based on runtime observations.

P2G allows programmers to focus on data transformations in a sequential manner while simultaneously providing enough information to dynamically adapt the data and task parallelization. The fields in P2G look mostly like global multidimensional arrays in C, although their representation in memory may differ. They do not have to be placed contiguously in the memory of a single node; the fields can even be distributed across multiple execution nodes.

5.4.1 Architecture

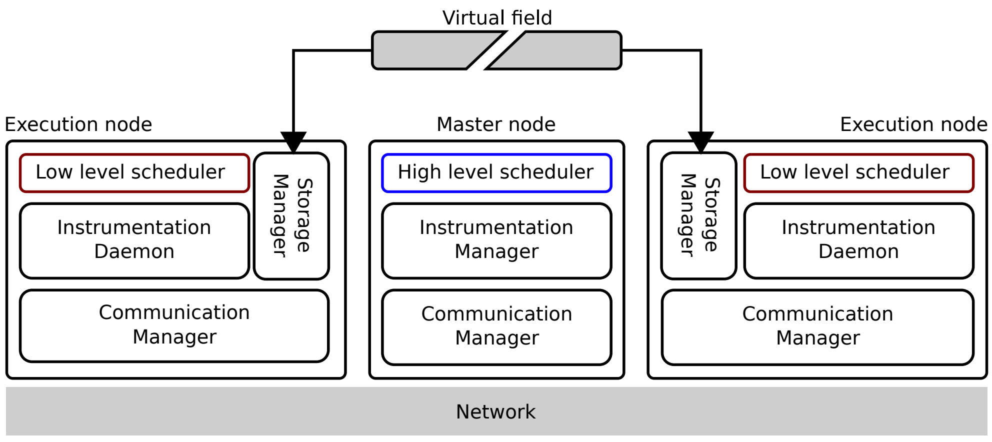
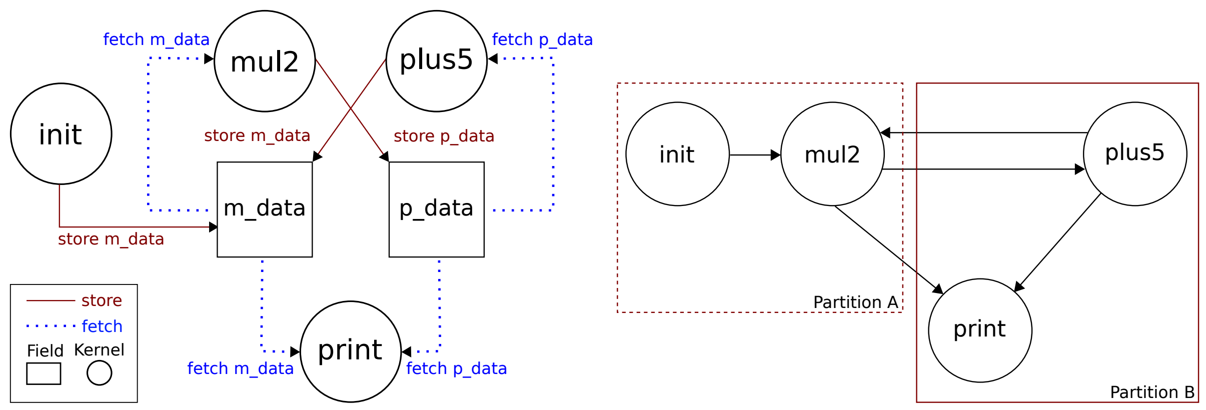


Figure 5.1: Overview of the architecture in the P2G system.

The basic architecture of the P2G framework can be seen in Figure 5.1. The P2G architecture consists of a *master node* and an arbitrary number of *execution nodes*. Each of the execution nodes reports its local topology (a graph of multicore and single-core CPUs and GPUs, connected by various kinds of buses and other networks) to the master node, which combines this information into a global topology of available resources. This global topology can be updated during runtime, as execution nodes are dynamically added and removed to accommodate for changes in the global load.

To maximize throughput, P2G uses the two-level scheduling approach we investigated in 2008 [115]. On the master node, we have an HLS and, on the execution node(s), we use a low-level scheduler (LLS). The HLS can analyze a workload's store and fetch statements, from which it can generate an intermediate implicit static dependency graph. An example of such a graph is shown in Figure 5.2(a), where the edges connecting two kernels through a field can be merged, circumventing the need for a vertex representing the field, which is shown in Figure 5.2(b). From the intermediate graph, the HLS can then derive the final implicit static dependency graph shown in Figure 5.2(b). The HLS will then use a graph partitioning [50] or search-based [42] algorithm to partition the workload into a suitable number of components that can be distributed to and run on the resources available in the topology. Using instrumentation data collected from the nodes executing the workload, we can weight the final graph with this profiling data during runtime. The weighted final graph can then be repartitioned, with the intent of



(a) Intermediate implicit static dependency graph (b) Final implicit static dependency graph

Figure 5.2: Dependency graphs in the P2G system.

improving system throughput, accommodating for changes in the global load or adapting to changes in available resources.

Given a partial workload such as *partition A* in Figure 5.2(b), an LLS at an execution node is responsible for local scheduling decisions. Figure 5.4 shows how the LLS can combine tasks and data to minimize overhead introduced by P2G and take advantage of specialized hardware, such as GPUs and other coprocessors.

The distribution of data, reporting, and other communication patterns in P2G is carried out through an event-based, distributed publish–subscribe model. Dependencies between components in a multimedia workload are deterministically derived from the code and the HLSs' partitioning decisions, resulting in direct communication. As such, the P2G framework relies on a combination of HLS and LLS instrumentation data and the global topology to make the best use of the performance of several heterogeneous processing architectures in a distributed system.

5.4.2 Programming Model

The programming model used in the P2G framework has two main concepts: the *implicit static dependency graph* shown in Figures 5.2(a) and 5.2(b) and the *dynamically created directed acyclic dependency graph* (DC-DAG) illustrated in Figure 5.4. A *kernel language* was also implemented to make it easier for programmers to develop workloads using P2G. An example workload written in the P2G kernel language is shown in Figure 5.5). The initial C++ version of the workload is shown in Figure 5.3.

The P2G version consists of two primary kernels: *mul2* and *plus5*. These two kernels form a pipeline where *mul2* first multiplies a value by two and stores the data, which *plus5* then fetches and increases by five; *mul2* then fetches the data stored by *plus5*; and so on. The *print* kernel runs orthogonally to these two kernels and fetches and writes the data they produced to *cout*. In combination, these three kernels form a cycle. The kernel *init* runs only once and writes initial data for *mul2* to consume. The kernels operate on two one-dimensional, five-element fields. The print kernel writes {*10, 11, 12, 13, 14*}, {*20, 22, 24, 26, 28*} for the first *age* and {*25, 27, 29, 31, 33*}, {*50, 54, 58, 62, 66*} for the second. Since there is no termination condition, this program runs indefinitely.

```
void print( int *in, int number )
{
    for( int i = 0; i < number; ++i ) {
        std::cout << in[i] << "␣";
    }
    std::cout << std::endl;
}

void mul2(int *in, int *out, int number)
{
    for (int i = 0; i < number; ++i)
        out[i] = in[i] * 2;
}

void plus5(int *in, int *out, int number)
{
    for (int i = 0; i < number; ++i)
        out[i] = in[i] + 5;
}

int main()
{
    int m_data[5] = { 10, 11, 12, 13, 14 };
    int p_data[5];

    int number = sizeof(m_data) / sizeof(m_data[0]);

    while( true )
    {
        mul2(m_data, p_data, number);

        print( m_data, number );
        print( p_data, number );

        plus5(p_data, m_data, number);
    }
    return 0;
}
```

Figure 5.3: Initial C++ version of a mul/sum example.

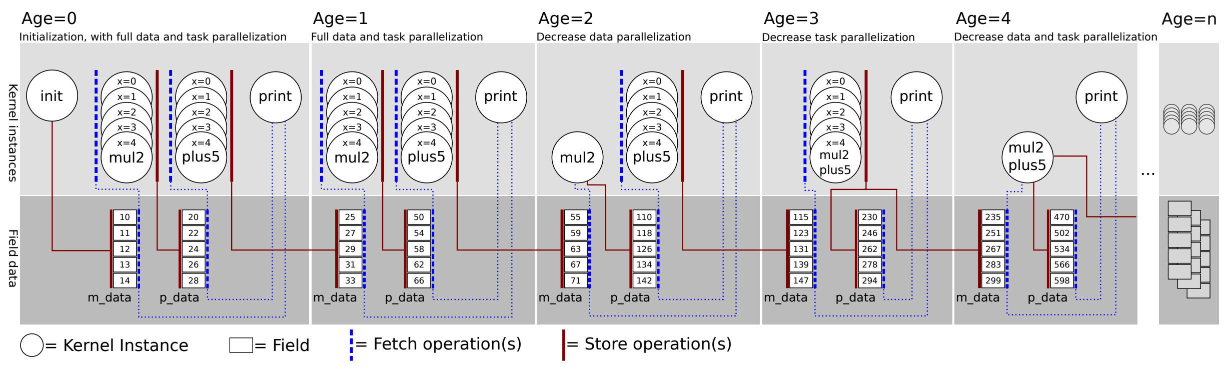

Age=0
Initialization, with full data and task parallelization
Age=1
Full data and task parallelization
Age=2
Decrease data parallelization
Age=3
Decrease task parallelization
Age=4
Decrease data and task parallelization
Age=n
Kernel instances
Field data
init
mul2
plus5
print
x=0
x=1
x=2
x=3
x=4
m_data
p_data
= Kernel Instance
= Field
= Fetch operation(s)
= Store operation(s)

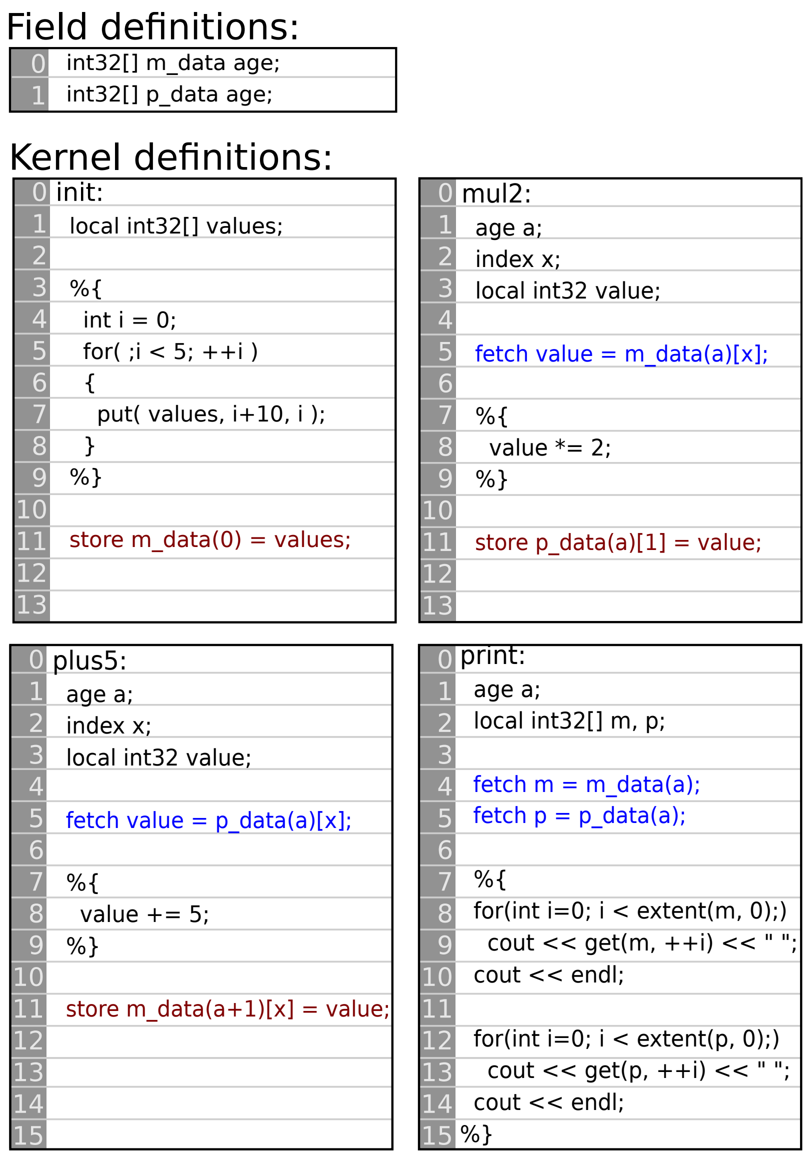
Figure 5.4: Dynamically created directed acyclic dependency graph (DC-DAG).

Dependency Graphs

The intermediate implicit static dependency graph can be seen in Figure 5.2(a) and is extracted from the *fetch* and *store* statements in a kernel. These statements are used by kernels to interact with fields. This intermediate graph can be further refined by merging the edges of kernels linked through a field vertex, resulting in a final implicit static dependency graph, as depicted in Figure 5.2(b). This final graph can serve as input to the HLS, which can use it to determine how best to partition the workload given a global topology. In this proof-of-concept implementation of P2G, dependency analysis is only carried out during runtime.

The graph can be further weighted using instrumentation data during runtime to serve as input for repartitioning. These weighted graphs can also serve as input in static offline analysis. For example, it could be used as input to a simulator to best determine how to initially configure a workload. During runtime, the intermediate implicit static dependency graph is expanded to form a dynamically created directed acyclic dependency graph, as shown in Figure 5.4. This expansion from a cyclic graph to a DAG occurs as a result of our write-once semantics. As such, we can see how P2G is designed to unroll loops without introducing implicit barriers between iterations. We call each such unrolled loop an *Age*. In Figure 5.4, we see how the LLS can then use the DC-DAG to combine both tasks and data to reduce overhead introduced by the P2G framework.

When moving from *Age=1* to *Age=2*, we can see that the LLS made a decision to reduce data parallelity. In P2G, kernels fetch slices of data and, initially, *mul2* was defined to work on each single field entry in parallel, but in *Age=2* the LLS decreased the granularity of the fetch statement to encompass the entire field. The LLS could also split the field in two, leading to two kernel instances of *mul2*, working on disparate sets of the field. Moving from *Age=2* to *Age=3*, we see that the LLS made the decision to decrease task parallelity. This is possible because *mul2* and *plus5* effectively form a pipeline, information that is available from the static graphs. By combining these two tasks, the individual store operations of the tasks are deferred until the data have been fully processed by each task. If the *print* kernel is not present, storage to the intermediate field *m_data* could be circumvented in entirety. In the final step, moving from *Age=3* to *Age=4*, we can see that a decision to decrease both task and data parallelity was made. This effectively renders this single kernel instance into a classical *for loop*, working on each data element of the field, with each task (*mul2*, *plus5*) performed sequentially on the


Field definitions:

```
int32[] m_data age;
int32[] p_data age;
```

Kernel definitions:

```
init:
  local int32[] values;

  %{
    int i = 0;
    for( ;i < 5; ++i )
    {
      put( values, i+10, i );
    }
  %}

  store m_data(0) = values;
```

```
mul2:
  age a;
  index x;
  local int32 value;

  fetch value = m_data(a)[x];

  %{
    value *= 2;
  %}

  store p_data(a)[1] = value;
```

```
plus5:
  age a;
  index x;
  local int32 value;

  fetch value = p_data(a)[x];

  %{
    value += 5;
  %}

  store m_data(a+1)[x] = value;
```

```
print:
  age a;
  local int32[] m, p;

  fetch m = m_data(a);
  fetch p = p_data(a);

  %{
  for(int i=0; i < extent(m, 0);)
    cout << get(m, ++i) << " ";
  cout << endl;

  for(int i=0; i < extent(p, 0);)
    cout << get(p, ++i) << " ";
  cout << endl;
%}
```

Figure 5.5: Kernel and field definitions.

data.

The P2G framework can make these runtime adjustments to data and task parallelism dynamically based on the resources available at the time.

Kernel Language

In the current prototype of the framework, P2G is exposed to the developer through the *kernel language* (BNF grammar of the kernel language can be found in Appendix A). An implementation of a simple workload in kernel language is outlined in Figure 5.5. The language can be replaced easily. However, it exposes the basic functions of the design. The most important parts are the kernel and field definitions, which describe the code and interaction patterns in P2G.

The main purpose of a kernel definition is to describe the required interaction of a kernel instance with an arbitrary number of fields (holding the application data) through the fetch and store statements. A field in P2G serves as an interaction point for kernel definitions, as shown in Figure 5.2(a).

An important aspect of a multimedia workload such as the Bagadus video pipeline is the ability to express deadlines. It is unnecessary to stitch a panorama if the playback has moved past that point in the video. We therefore implemented language support for expressing deadlines. In principle, a deadline gives the application developer the option of defining a global timer. This timer can then be polled and updated from within a kernel definition. Given a condition based on a deadline, a timeout can occur and an alternate code path can be executed. Such an alternate code path is executed by storing to a different field then in the primary path, leading to new dependencies and new behavior. In this proof-of-concept implementation, we did not implement support for timers; however, we are currently reevaluating the concept of timers.

Fields in P2G have several properties, including a type and a dimensionality. An important property mentioned earlier is aging. Aging allows kernels to be iterative while maintaining write-once semantics in such cyclic execution. Aging enables unique storage to the same position in a field several times, as long as the age increases for each store operation, as shown in Figure 5.4). In essence, this adds an extra dimension to the field and makes it possible to accommodate iterative algorithms. It is also important to note that a field is not connected to a single execution node; it can be distributed across multiple execution nodes, as shown in Figure 5.1).

When defining the interaction between kernels and fields, the programmer is encouraged to express the finest possible granularity of kernel definition and, likewise, the most precise slices possible for the kernel within the field. The reason is because it provides the LLS more control over the granularity of task and data decomposition. With instrumentation data, the framework can reduce scheduling overhead by combining several instances of a kernel that process different data or several instances of different kernels that process data in sequence, as shown in Figure 5.4). The scheduler in P2G makes its decisions based on instrumentation data and the implicit static dependency graph.

Runtime

We can now extrapolate the concept of kernel definitions to kernel instances. A kernel instance is the unit of code that is executed during runtime and the number of kernel instances executed in parallel for a given kernel definition will depend on the fetch statements.

A kernel instance works on an arbitrary number of slices of fields, depending on the number of fetch statements in the kernel definition. If we look at the example in Figures 5.4 and 5.5, we can see how the *mul2* kernel, given its *fetch* statement on *m_data* with *age=a* and *index=x*, fetches only a single element of the data. Thus, since the *m_data* field consists of five data elements, this means that P2G can execute a maximum of x kernel instances simultaneously per age, yielding *a*x mul2* kernel instances.

P2G also supports the automatic resizing of fields. This is shown in the kernel definition of the *print* kernel in Figure 5.5. Initially, the extents of *m_data* and *p_data* are not defined. With each iteration of the *for* loop in *init*, the local field *values* is resized locally. This leads to a resizing of the global field *m_data* when *values* is stored to it. These extents are then propagated to the respective fields impacted by this resizing, such as *p_data*.

It is important to note that a kernel instance is only dispatched when all its dependencies are fulfilled, that is, all the data it fetches have been stored to the respective fields and elements. Figures 5.4 and 5.5 also show that *mul2* stores its result to *p_data* with *age=a* and *index=x*. This means that once *mul2* has stored its results to *p_data* with *index=2* and *age=0*, the kernel instance of *plus5* with the fetch statement *fetch(0)[2]* can be dispatched. With our write-one semantics, each kernel instance is only dispatched once.

5.4.3 Prototype

A prototype implementation of the basic concepts of the P2G framework was implemented. The prototype consisted of a compiler for the kernel language and a runtime that could execute P2G programs on the x86 multicore architecture with a shared memory model running a Linux operating system.

Compiler

Programs written for the P2G framework are designed to be platform independent and have native blocks of code written in C or C++. Heterogeneous systems are specifically targeted; however, many of these require a custom compiler for the native blocks, such as Nvidia's NVCC compiler for GPUs with CUDA. For the prototype, instead of generating binaries directly, we decided to compile P2G programs into C++ files, which can be further compiled and linked with native code blocks. This approach provides us with less control of the resulting object code, but we gain both the flexibility and optimizations of the native compilers, resulting in a lightweight P2G compiler.

Runtime

Our prototype implementation of P2G features a basic execution node, with support for multidimensional fields, instrumentation, the implicit resizing of fields, and parallel execution of kernel instances on a single machine, using the implicit dependency graph formed by kernel definitions. Support for deadline expressions was not implemented.

The target architecture for this prototype is a single machine with a shared memory multicore x86 architecture. The system was designed as a push-based system using event subscriptions on field operations. Kernel instances are executed in parallel and produce events on *store* statements, which could require resizing operations. A kernel subscribes to events related to the fields that it depends on, that is, fields referenced by the kernel's *fetch* statements. When such a storage event is detected, the runtime finds all *new* valid combinations of age and index variables that can be processed as a result of the *store* statement and places these in a per-kernel ready queue. This means that the ready queues always contain the maximum number of parallel instances that can be executed at any time, limited only by unfulfilled data dependencies.

The prototype uses a simple LLS that consists of a dependency analyzer and kernel instance dispatcher. The dependency analyzer uses the implicit dependency graph to add new kernel instances to a ready queue, which can later be executed by the worker threads. Dependencies are analyzed in a dedicated thread that handles events emitted from running kernel instances that notifies upon *store* and *resize* operations performed on fields.

When executed, kernel instances are dispatched from the ready queue. They are scheduled in an order that prefers the execution of kernel instances with a lower age value (older kernel instances). This ensures that no runnable kernel instance is starved by others that have no *fetch* statements.

5.4.4 Workloads

To test the initial idea and the prototype, we developed a few simple workloads commonly used in multimedia processing. The P2G kernel language is able to expose both the data and task parallelism of the workloads to the P2G system, so that the runtime is able to adapt the execution of the programs to suit the target architecture.

K-Means Clustering

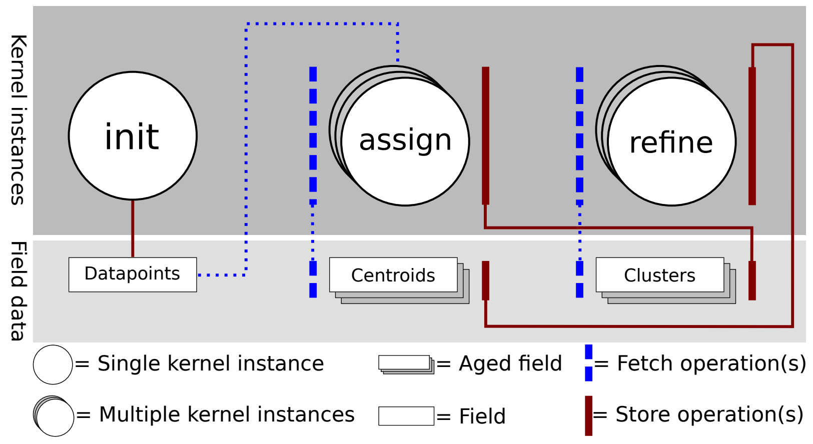


Figure 5.6: Overview of the K-means clustering algorithm.

K-Means clustering is an iterative algorithm for cluster analysis that aims to partition n data points into k clusters, where each data point belongs to the cluster with the nearest mean. The basic structure of the workload is shown in Figure 5.6. Our implementation in P2G consists of an *init* kernel, which generates n data points and *stores* them to the data points field. Next, it randomly selects k of these data points as the initial means and *stores* them to the centroids field. Then, the *assign* kernel *fetches* a slice of data, a single data point per *kernel instance*, the last calculated centroids, and *stores* this data point in the cluster of the closest centroids using a Euclidean distance calculation. Finally, the *refine* kernel *fetches* a cluster, calculates its new mean, and *stores* this information in the centroids field. The kernel definitions of *assign* and *refine* will form a loop that gradually leads to a convergence in centroids, at which point the k-means algorithm is completed.

Motion JPEG

The Motion JPEG (MJPEG) workload is based on the same algorithms and code used to test simple workloads on the x86 architecture, GPUs, and the Cell in Section 3.2.1. The

MJPEG format provides many layers of parallelism and is well suited for illustrating the potential of the framework. We focus on optimizing the discrete cosine transform (DCT) and quantization part, as this is the most computationally intensive part of the codec.

The *read + splitYUV* kernel reads the input video in YUV format and stores the data in three global fields: *yInput*, *uInput*, and *vInput*. In this workload, the three YUV components can be processed independently of each other and this property is exploited by creating three kernels: *yDCT*, *uDCT*, and *vDCT*. In Figure 5.7, we see that the respective DCT kernels are dependent on one of these fields.

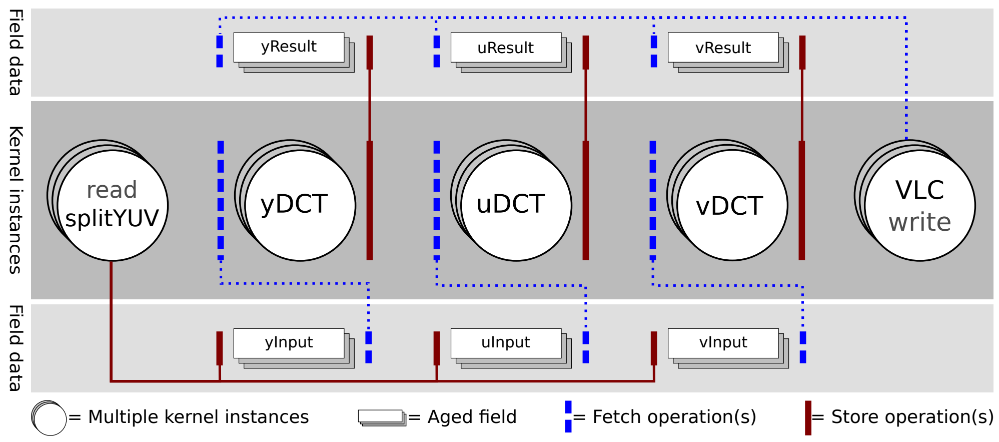


Figure 5.7: Overview of the MJPEG encoding process

The MJPEG encoding process splits the video frames into 8*8 macroblocks. The CIF resolution of 352*288 pixels per frame used in our tests will generate 1584 macroblocks of Y (luminance) data, each with 64-pixel values. The 4:2:2 chroma sub-sampling yields 396 kernel instances from both the U and V (chroma) data. Each of these kernel instances stores the discrete cosine transformed macroblock into the global result fields *yResult*, *uResult*, and *vResult*. Finally, the *VLC + write* kernel stores the MJPEG bitstream to disk.

5.4.5 Evaluation

We ran experiments with the K-means and MJPEG workloads, as described in Section 5.4.4. Each test was run on the following x86 multicore architectures, with the number of worker threads ranging from one to eight:

- One Intel Core i7-860 based on the Nehalem architecture running at 2.8 GHz, with four cores and Hyper-Threading (simultaneous multithreading) enabled.
- Four AMD Opteron 8218 processors based on the K8 architecture running at 2.6 GHz, with two cores per processor, for a total of eight cores.

In addition, we performed micro-benchmark tests for both workloads. These summarize the number of kernel instances dispatched per kernel definition, dispatch overhead, and time spent in the kernel code.

K-Means Clustering

The K-means workload is run with K=100, using a randomly generated data set with 2000 data points. The K-means algorithm does not run until convergence and we defined a breakpoint after 10 iterations. Without any breakpoint, the algorithm's convergence is undefined and, as such, we introduce this condition to ensure that we achieve relatively stable running times.

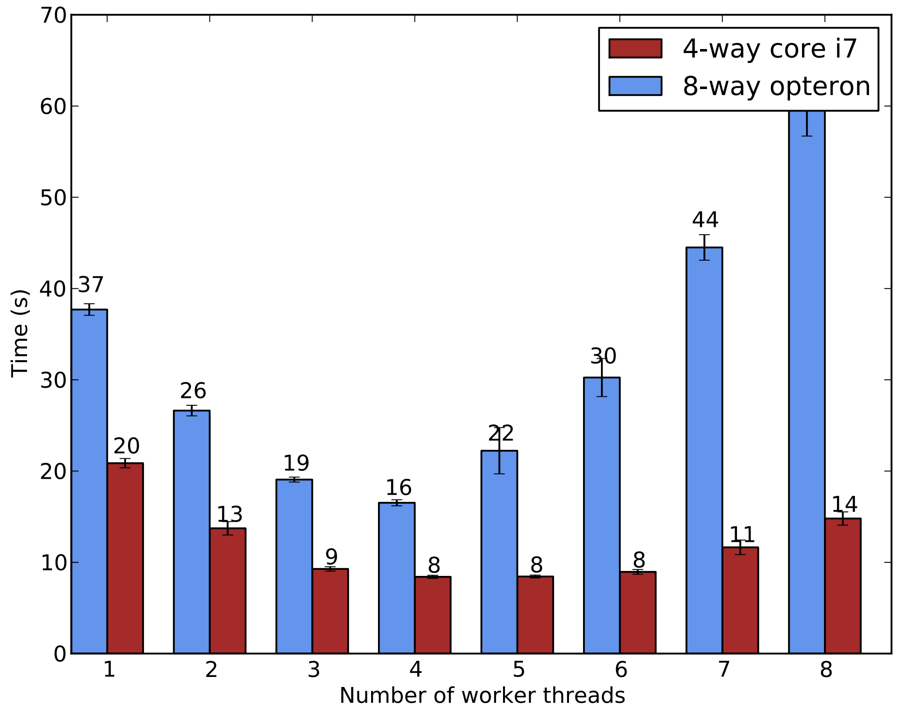


Figure 5.8: Workload execution time for K-means.

As seen in Figure 5.8, the K-means workload scales well to four worker threads. After this, the execution time increases with the number of worker threads. This can be explained by the fine granularity of the *assign* kernel definition. This leads to the serial dependency analyzer becoming a bottleneck in the system. As discussed in Section 5.4.2, this condition could be alleviated by decreasing the granularity of data parallelism, in effect leading to each kernel instance of *assign* working on larger slices of data. By doing so, we would increase the ratio of time spent in kernel code compared to dispatch time and reduce the workload of the dependency analyzer.

The two different test machines behave somewhat differently, in that the Opteron suffers more than the Core i7 when the dependency analyzer saturates a core. The Core i7 is able to increase the frequency of a single core to mitigate serial bottlenecks and the memory architectures of Intel processors are generally more efficient than AMD processors. We think this is why the Core i7 suffers less under the limitations dictated by Amdahl's law.

Kernel	Instances	Dispatch Time	Kernel Time
init	1	58.00 μs	9,829.00 μs
assign	2,024,251	4.07 μs	6.95 μs
refine	1000	3.21 μs	92.91 μs
print	11	1.09 μs	379.36 μs

Table 5.1: Micro-benchmarks of K-means in P2G.

MJPEG

The MJPEG workload is run on 50 frames of the standard *Foreman* test sequence encoded in *CIF* resolution.

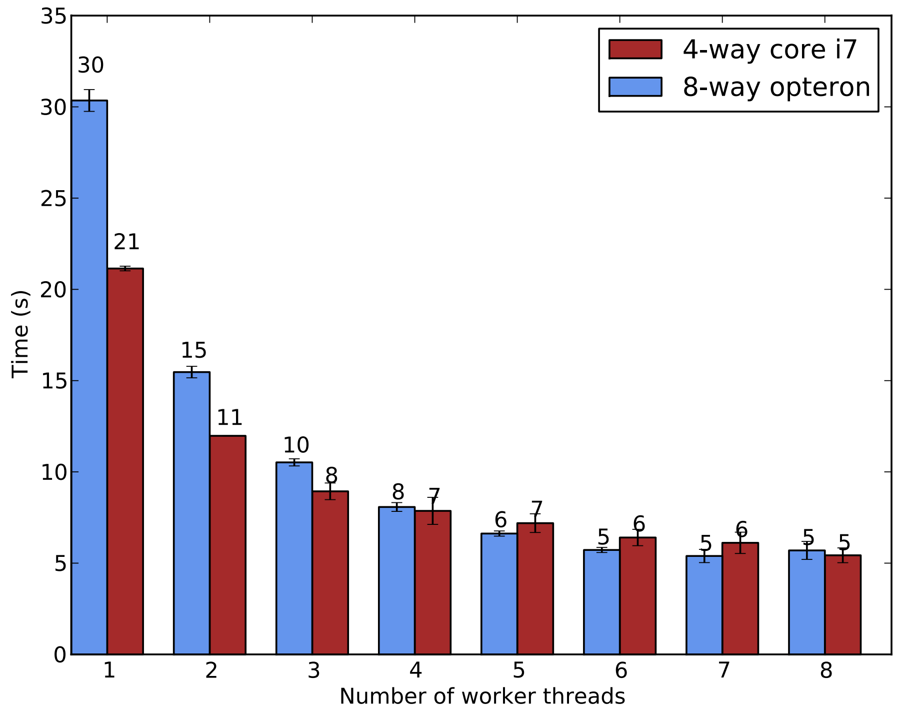


Figure 5.9: Workload execution times for MJPEG.

As we can observe in Figure 5.9, P2G is able to scale close to linearly with its available resources. In P2G, the dependency analyzer of the LLS runs in a dedicated thread. This affects the execution time when moving from seven to eight worker threads, where the eighth thread shares resources with the dependency analyzer.

A native single-threaded version of the MJPEG encoder on which the P2G version is based has a running time of 30 seconds on the Opteron machine and 19 seconds on the Core i7 machine.

From the micro-benchmarks in Table 5.2, we can see that time spent in kernel code is considerably greater compared to the dispatch overhead for the kernel definitions. The

Kernel	Instances	Dispatch Time	Kernel Time
init	1	69.00 μs	18.00 μs
read/splityuv	51	35.50 μs	1,641.57 μs
yDCT	80,784	3.07 μs	170.30 μs
uDCT	20,196	3.14 μs	170.24 μs
vDCT	20,196	3.15 μs	170.58 μs
VLC/write	51	3.09 μs	2,160.71 μs

Table 5.2: Micro-benchmarks of MJPEG encoding in P2G.

dispatch time includes the allocation or reallocation of fields as part of the timing operation. As a result, *init* and *read/splitYUV* have considerably higher dispatch times then the **DCT* operations. We can also see that the majority of the CPU time is spent in the kernel instances of *yDCT, uDCT*, and *vDCT*, which is the most computationally intensive part of this workload. This indicates that decreasing the data and task granularity, as discussed in Section 5.4.2, has little impact on system throughput. This is because the majority of the time is already spent in kernel code.

5.4.6 Summary

With the P2G framework, we proposed a new flexible system for the automatic, parallel real-time processing of multimedia workloads. We encourage the programmer to specify parallelism in as fine a granularity as possible along the axes of task and data decomposition. Using our kernel language, this decomposition is expressed through kernel definitions and fetch and store statements on fields.

Given a workload that uses our kernel language and is compiled for execution in P2G, this workload can be partitioned by the HLS in a P2G master node, which then distributes the partitions to the P2G execution nodes, which will execute the tasks. Execution nodes can consist of heterogeneous resources. The LLS at the execution nodes will adapt the workload to run optimally, using the resources available. Feedback from the instrumentation module at the execution node can lead to workload repartitioning.

We implemented a prototype execution node capable of running on a shared memory multicore architecture. The results from our experiments running on this prototype execution node show the potential of our ideas. However, many features—such as support for agglomeration, distribution, timers, heterogeneous architectures, and so on—have yet to be added.

5.5 The Future

Our prototype implementation of P2G is a proof-of-concept implementation of the P2G execution node with support for shared memory x86 multicore architecture. However, several features have yet to be implemented in the execution node, two of these being timers and agglomeration. Timers are important for enforcing soft deadlines in multimedia workloads and agglomeration is important to adapt the level of parallelism to the heterogeneous architecture we want to use. A kernel running on a GPU would require many more threads than a kernel running on the CPU. This issue is apparent in the K-means workload evaluated in Section 5.4.5.

The master node has also not yet been implemented. One of the reasons for targeting the x86 multicore architecture in our prototype of the framework is the shared memory property. When we add more machines or heterogeneous architectures with an exclusive memory model, we need a distributed name service to manage message passing between machines and processing nodes.

One of the next steps for the P2G framework would be to add support for using GPUs on the same machine as the x86 multicore architecture. This would not require any master node, since the GPUs on a single machine are managed by the host CPU.

Chapter 6

Papers and Author's Contributions

6.1 Overview of Research Papers

The research conducted during the PhD period focused on system support for multimedia. My papers addressed a large set of challenges, ranging from scheduling mechanisms for heterogeneous architectures to the optimization of video codecs. This research was a cooperative effort and the contribution of this thesis is its investigation of homogenous and heterogeneous computing platforms and infrastructures, an understanding of the developer's ability to exploit their performance, and the development of improvements for processing multimedia workloads. Nine of our papers were chosen to document this research effort and are included as the main contribution of this thesis. Five of these papers [32,34,102,112,117] look at processing simple workloads with heterogeneous architectures. Two of the papers [113,114] investigate a complex multimedia workload. The knowledge gained in these two Chapters is used in the previous chapter in the design of the P2G framework [31,115].

Although the other papers [35,46,76,77,116], posters, and demonstrations [10,11,33, 101,105,120,132] we wrote are related to multimedia systems, we limited the thesis to those nine papers that were central in forming our understanding of multicore programming and the development of P2G from the developer's perspective. These papers are presented chronologically in the following.

6.2 Paper I: Transparent Protocol Translation for Streaming

Abstract The transport of streaming media data over TCP is hindered by TCP's probing behavior, which results in the rapid reduction and slow recovery of packet rates. On the other hand, UDP has been criticized for being unfair to TCP connections and it is therefore often blocked out of access networks. In this paper, we try to benefit from a combined approach using a proxy that transparently translates the transport protocol. We translate HTTP requests by the client transparently into RTSP requests and translate the corresponding RTP/UDP/AVP stream into the corresponding HTTP response. This enables the server to use UDP on the server side and TCP on the client side. This is beneficial for the server side, which scales to a higher load when it does not have to deal with

TCP. On the client side, streaming over TCP has the advantage that connections can be established from the client side and data streams can pass through firewalls. Preliminary tests demonstrate that our protocol translation delivers a smoother stream compared to HTTP streaming, where the TCP bandwidth oscillates heavily.

Lessons learned This paper is written as a network research paper, using the Intel IXP network processor to translate a network protocol in real time. However, the experience gained while working with the IXP architecture gave us experience in working with a heterogeneous architecture with a shared memory model. The heterogeneous elements on the IXP also have different instruction sets and compilers, which we can also find in other heterogeneous architectures.

Author's contributions Stensland contributed to the evaluation of the experiments. Espeland and Lunde carried out the design and implementation of the proxy and the experimental setup. The paper was written in collaboration with all the other authors.

Published in Proceedings of the 15th ACM International Conference on Multimedia (MM '07), ACM, 2007.

DOI 10.1145/1291233.1291407

6.3 Paper II: Evaluation of Multi-Core Scheduling Mechanisms for Heterogeneous Processing Architectures

Abstract General-purpose central processing units (CPUs) with multiple cores are established products and new heterogeneous technology such as the Cell Broadband Engine and general-purpose graphic processing units (GPUs) bring an even higher degree of true multiprocessing to the market. However, the means for utilizing the processing power are immature. Current tools typically assume that the exclusive use of these resources is sufficient, but this assumption will soon be invalid because of interest in using their processing power for general-purpose tasks. Among the applications that can benefit from such technology is transcoding support for distributed media applications, where remote participants join and leave dynamically. Transcoding consists of several clearly separated processing operations that consume a great deal of resources, such that individual processing units are unable to handle all the operations of a session of arbitrary size. The individual operations can then be distributed over several processing units and data must be moved between them according to the dependencies between operations. Many multiprocessor scheduling approaches exist but, to the best of our knowledge, the challenge remains to find mechanisms that can schedule dynamic workloads of communicating operations while taking both the processing and communication requirements into account. For such applications, we believe that feasible scheduling can be performed at two levels, that is, divided into the task of placing a job onto a processing unit and the task of multitasking time slices within a single processing unit. We implemented some simple

high-level scheduling mechanisms and simulated a videoconferencing scenario running on topologies inspired by existing systems from Intel, AMD, IBM, and Nvidia. Our results show the importance of using an efficient high-level scheduler.

Lessons learned In this paper, we designed a simple simulator that simulated the high-level scheduling aspect of a multimedia workload on different multicore architectures. We showed that a two-level scheduling approach, where the high level scheduler places jobs onto a processing core and the low-level scheduler takes on the job of time slicing within a single processing unit. Our results show the importance of using an efficient high-level scheduler. Lessons learned from this paper were later used when designing the P2G framework.

Author's contributions Stensland contributed significantly to the design of the event-driven simulator. He also designed the workloads for the simulator and performed the experiments for the paper. Stensland also contributed to writing the paper.

Published in The 18th International Workshop on Network and Operating Systems Support for Digital Audio and Video (NOSSDAV '08), ACM, 2008.

DOI 10.1145/1496046.1496054

6.4 Paper III: Tips, Tricks and Troubles: Optimizing for Cell and GPU

Abstract When used efficiently, modern multicore architectures, such as the Cell and GPUs, provide the processing power required by resource-hungry multimedia workloads. However, the diversity of resources exposed to the programmer intrinsically requires very different mindsets to efficiently utilize these resources—not only compared to an x86 architecture, but also between the Cell and the GPUs. In this context, our analysis of 14 different Motion JPEG (MJPEG) implementations indicate great potential for optimizing performance, but there are also many pitfalls to avoid. By experimentally evaluating algorithmic choices and inter-core data communication (memory transfers) and architecture-specific capabilities, such as instruction sets, we present tips, tricks, and problems with respect to the efficient utilization of available resources.

Lessons learned In the third paper, we analyze 14 different implementations of a multimedia workload from a graduate level course we teach on two different heterogeneous architectures. We learned that heterogeneous architectures such as the Cell and GPUs are suitable for processing real-time multimedia workloads such as MJPEG video encoding. However, it is not trivial for programmers to achieve high performance on either of these architectures. There are similarities between the Cell and GPUs, but the way programmers need to think is substantially different, not only compared to the x86 architecture but also between the Cell and the GPU. The Cell uses a single instruction multiple data (SIMD) programming model, which seems harder to grasp compared to the

SIMT abstraction used by the GPUs. All three architectures have different architectural capabilities that must be taken into account when choosing the algorithms to use.

Author's contributions Stensland contributed significantly to the design, implementation, and evaluation of this work. Together with Espeland, he designed the experiments and evaluated the results. The paper was written in collaboration with the other authors.

Published in The 20th International Workshop on Network and Operating Systems Support for Digital Audio and Video (NOSSDAV 2010), ACM, 2010.

DOI 10.1145/1806565.1806585

6.5 Paper IV: Cheat Detection Processing: A GPU versus CPU Comparison

Abstract In modern online multiplayer games, game providers have been struggling to keep up with the many different types of cheating. Cheat detection is a task that requires great computational resources. Advances made within the field of heterogeneous computing architectures, such as in GPUs, have given developers easier access to considerably more computational resources, enabling a new approach to resolving this issue.

In this paper, we developed a small game simulator that includes a customizable physics engine and a cheat detection mechanism that checks the physical model used by the game. To make sure that the mechanisms are fair to all players, they are executed on the server side of the game system. We investigate the advantages of implementing physics cheat detection mechanisms on a GPU using the Nvidia CUDA framework and we compare the GPU implementation of the cheat detection mechanism with a CPU implementation. The results obtained from the simulations show that offloading the cheat detection mechanisms to the GPU reduces the time spent on cheat detection, enabling the servers to support larger numbers of clients.

Lessons learned In the fourth paper, we used a cheat detection mechanism implemented on a GPU to learn about the effect of offloading workload from the CPU. The results shows that, even with a simple physical model, the GPU is able to outperform the CPU. However, we also observed that, with a low number of clients, the CPU implementation is faster that the GPU implementation. This is due to the latency cost of transferring the workload from CPU memory to the GPU for processing.

Author's contributions Stensland designed the experiments for the paper. He also used a Fermi generation GPU for the paper's experiments and analyzed their data. The prototype was designed and implemented as part of Myrseth's master's thesis, for which Stensland was the main supervisor. The paper text was mainly written by Stensland, with contributions from the other authors.

Published in Workshop on Network and Systems Support for Games (NetGames 2010), ACM/IEEE, 2010.

DOI 10.1109/NETGAMES.2010.5679527

6.6 Paper V: Reducing Processing Demands for Multi-Rate Video Encoding: Implementation and Evaluation

Abstract Segmented adaptive HTTP streaming has become the de facto standard for video delivery over the Internet for its ability to scale video quality to available network resources. Each video is encoded at multiple levels of quality, that is, running the expensive encoding process for each quality layer. However, these operations consume a great deal of both time and resources and, in this paper, the authors propose a system for reusing redundant steps in a video encoder to improve the multilayer encoding pipeline. The idea is to have multiple outputs for each of the target bitrates and quality levels, where the intermediate processing steps share and reuse the computationally heavy analysis. A prototype has been implemented using the VP8 reference encoder and experimental results show that, for both low- and high-resolution videos, the proposed method can significantly reduce processing demands and time when encoding the different quality layers.

Lessons learned The fifth paper looks at the possibility of reusing the computationally intensive analysis part of a video encoder. We learned that the shared memory architecture in x86 multicore processors greatly facilitates the sharing of data between multiple threads. When running the reference VP8 encoder on four videos, we can see that serial encoding performs better that running four instances concurrently. This shows the negative effects of running too many threads on too few cores: The scheduling overhead in the operating system increases and contention arises between the execution resources on the CPU.

Author's contributions The multi-rate prototype was designed and implemented as part of Finstad's master's thesis, of which Stensland was the main supervisor. Espeland contributed with the basic idea of multi-rate encoding. The experiments were evaluated by Stensland and Espeland. The text was also mainly written by Espeland and Stensland, with input from the other authors.

Published in International Journal of Multimedia Data Engineering and Management (IJMDEM), Volume 3, Issue 2, IGI Global, 2012.

DOI 10.4018/JMDEM.2012040101

6.7 Paper VI: LEARS: A Lockless, Relaxed-Atomicity State Model for Parallel Execution of a Game Server Partition

Abstract Supporting thousands of interacting players in a virtual world poses huge processing challenges. Work that addresses the challenge utilizes a variety of spatial partitioning algorithms to distribute the load. If, however, a large number of players need to interact tightly across an area of the game world, spatial partitioning cannot subdivide this area without incurring massive communication costs, latency, or inconsistency. Scaling such areas to the largest number of players possible is a major challenge of game engines. Deviating from earlier thinking, we apply parallelism on multicore architectures to increase scalability. In this paper, we evaluate the design and implementation of our game server architecture, called a lockless, relaxed-atomicity state, or LEARS, which allows for lock-free parallel processing of a single spatial partition by considering every game cycle an atomic tick. Our prototype is evaluated using traces from live game sessions where we measure the server response time for all objects that require timely updates. We also measure how the response time for the multithreaded implementation varies with the number of threads used. Our results show that the challenge of scaling up a game server can be an embarrassingly parallel problem.

Lessons learned The sixth paper describes an architecture to scale a game server workload. We showed that resource utilization can be improved by distributing the load over the shared memory architecture of x86 multicore CPUs. However, we learned the importance of balancing the number of threads executing on the physical CPU. If too many threads are executed concurrently, the performance will start to degrade because of the increased context switching overhead. This is the opposite case from that of a GPU, where thousands of threads are required for good performance.

Author's contributions Stensland contributed to the discussion and evaluation of the game server with respect to scaling on the x86 multicore architecture. He also contributed with architectural knowledge about the evaluation system. Stensland also contributed to writing the paper.

Published in Proceedings of the International Workshop on Scheduling and Resource Management for Parallel and Distributed Systems (SRMPDS)—The 2012 International Conference on Parallel Processing Workshops, IEEE, 2012.

DOI 10.1109/ICPPW.2012.55

6.8 Paper VII: P2G: A Framework for Distributed Real-Time Processing of Multimedia Data

Abstract The computational demands of multimedia data processing are steadily increasing as consumers call for progressively more complex and intelligent multimedia services. New multicore hardware architectures provide the required resources, but writing parallel, distributed applications remains a labor-intensive task compared to their sequential counterpart. For this reason, Google and Microsoft implemented their respective processing frameworks, MapReduce and Dryad, since they allow the developer to think sequentially yet benefit from parallel and distributed execution. An inherent limitation in the design of these processing frameworks is their inability to express arbitrarily complex workloads. The dependency graphs of the frameworks are often limited to directed acyclic graphs or even predetermined stages. This is particularly problematic for video encoding and other algorithms that depend on iterative execution. With the Nornir runtime system for parallel programs, which is a Kahn process network implementation, we addressed and resolved several of these limitations. However, it is more difficult to use than other frameworks are due to its complex programming model. In this paper, we build on the knowledge gained from Nornir and present a new framework, called P2G, designed specifically for developing and processing distributed real-time multimedia data. P2G supports arbitrarily complex dependency graphs with cycles, branches, and deadlines and provides both data parallelism and task parallelism. The framework is implemented to scale transparently with available (heterogeneous) resources, a concept familiar from the cloud computing paradigm. We implemented a (interchangeable) P2G model to ease development. In this paper, we present a proof-of-concept implementation of a P2G execution node and some experimental examples using complex workloads, such as MJPEG and K-means clustering. The results show that the P2G system is a feasible approach to multimedia processing.

Lessons learned In the seventh paper, we present a prototype and framework for the distributed real-time processing of multimedia workloads. We also implemented and evaluated two simple multimedia workloads to verify that multimedia workloads such as K-means and MJPEG can be expressed in the framework. Our experiments shows that our prototype is able to scale performance with the available resources in the system, as long as there is a large enough workload per instance.

Author's contributions Stensland contributed to the design and implementation of the workloads used to benchmark the framework and to the ideas behind the P2G framework. Espeland and Beskow designed, implemented, and micro-benchmarked the framework. The paper and discussions were written in collaboration with all of the authors.

Published in Proceedings of the International Workshop on Scheduling and Resource Management for Parallel and Distributed Systems (SRMPDS)—The 2011 International Conference on Parallel Processing Workshops, IEEE, 2011.

DOI 10.1109/ICPPW.2011.22

6.9 Paper VIII: Bagadus: An Integrated Real-Time System for Soccer Analytics

Abstract The importance of winning has increased the role of performance analysis in the sports industry and underscores how statistics and technology keep changing the way sports are played. Thus, this is a growing area of interest, both from a computer system view, to manage the technical challenges, and from a sports performance view, to aid the development of athletes. In this respect, Bagadus is a *real-time* prototype of a sports analytics application using soccer as a case study. Bagadus integrates a sensor system, a soccer analytics annotation system, and a video processing system using a videocamera array. A prototype was recently installed at Alfheim Stadium in Norway and, in this paper, we describe how the system can be used in real time to play back events. The system supports both stitched panoramic video and camera switching modes and creates video summaries based on queries to the sensor system. Moreover, we evaluate the system from a systems point of view, benchmarking different approaches, algorithms, and trade-offs, and show how the system runs in real time.

Lessons learned The eighth paper describes the integration of the three different subsystems in the Bagadus sports analysis system. The paper focuses on the optimization of the video system, where we optimize the video pipeline for both an x86 multicore and a GPU with the goal of running the system in real time on a single machine. The experiments shows that we are able to process video from four cameras, stitch the video to a panorama, and use video processing algorithms to enhance the quality of this panoramic video.

Author's contributions Stensland contributed to the design and evaluation of the real-time video pipeline presented in this paper. He also analyzed the experimental results from the GK110 GPU presented in this paper and provided insight into the heterogeneous architecture used. Stensland was also a supervisor for all the master's students involved in this project. The paper was mainly written by Stensland, with contributions from the other authors.

Published in ACM Transactions on Multimedia Computing, Communications, and Applications (TOMM), Volume 10, Issue 1s, ACM, 2014.

DOI 10.1145/2541011

6.10 Paper IX: Processing Panorama Video in Real-Time

Abstract There are many scenarios in which a high-resolution, wide field of view video is useful. Such panoramic video may be generated using camera arrays, where the feeds from multiple cameras pointing at different parts of the captured area are stitched together. However, processing the different steps of a panoramic video pipeline in real time is

challenging due to the high data rates and the stringent timeliness requirements. In our research, we use panoramic video in a sports analysis system called Bagadus. This system was deployed at Alfheim stadium in Tromsø, Norway, and, due to live usage, the video events had to be generated in real time. In this paper, we describe our real-time panoramic system built using a low-cost CCD HD videocamera array. We describe how we implemented different components and evaluated alternatives. The performance results from experiments run on commodity hardware with and without coprocessors, such as GPUs, demonstrate that the entire pipeline is able to run in real time.

Lessons learned The ninth paper undertakes a more detailed analysis of the video system in the Bagadus sports analysis system. The paper focuses on how we achieved real-time performance by optimizing the video pipeline for both CPU and GPU execution. Finally, we evaluated the performance of the complete video pipeline on different heterogeneous architectures and machine setups. We learned that a system such as Bagadus, with real-time requirements, requires a GPU that can execute workloads concurrently. The GPU utilization for a system such as Bagadus is also fairly low and the GPU can be shared with other workloads.

Author's contributions Stensland contributed to the design and evaluation of the real-time video pipeline presented in this paper. He also provided insight into the different heterogeneous architectures and the machine setup used to evaluate the video pipeline. Stensland was also a supervisor for all the master's students involved in this project. The paper was mainly written by Stensland, with contributions from the other authors.

Published in International Journal of Semantic Computing (IJSC), Volume 8, Issue 2, World Scientific, 2014.

DOI 10.1142/S1793351X14400054

6.11 Supervised Master's Students

Student: Håvard Espeland

Title: Investigation of parallel programming on heterogeneous multiprocessors

Summary: This thesis investigates different parallelization strategies and performance for real-world problems using two heterogeneous architectures, the Intel IXP2400 architecture and the Cell Broadband Engine. The tests show promising throughput for some applications and the thesis proposes a scheme for offloading computationally intensive parts of an application.

Student: Alexander Ottesen

Title: Efficient parallelization techniques for applications running on GPUs using the CUDA framework

Summary: This thesis investigates the GPU architecture and processing capabilities of the first generation of Nvidia GPUs with support for the CUDA framework. We investigate how CUDA applications can share the GPU resource and see what challenges are connected with concurrent applications executing on the GPU.

Student: Magne Eimot

Title: Offloading an encrypted user space file system on GPUs

Summary: Modern computers often have powerful GPUs and an important use of this technology is to assist the main CPU with computationally intensive tasks. We investigate the challenges of using GPUs to offload the encryption operations from the main CPU.

Student: Martin Øinæs Myrseth

Title: Cheat detection in on-line multi-player games using graphics processing units

Summary: This thesis investigates the benefits of using GPUs for cheat detection mechanisms. We develop a framework for a game simulator that includes a simple customizable physical engine and a cheat detection mechanism. The results shows that, in addition to being faster, the GPU mechanism allows the CPU to perform other game-relevant tasks while the mechanism is executing.

Student: Espen Angell Kristiansen

Title: Dynamic adaption and distribution of binaries to heterogeneous architectures

Summary: Real-time multimedia workloads require extensive processing power. Here, we develop the foundation for network distribution in P2G and suggest a viable solution for the execution of workloads on heterogeneous multicore architectures.

Student: Dag Haavi Finstad

Title: Multi-rate VP8 video encoding

Summary: This thesis addresses the resource consumption issues of encoding multiple videos by proposing a method for reusing redundant steps in a video encoder, emitting multiple outputs with various bitrates and quality levels.

Student: Magnus Funder Halldal

Title: Exploring computational capabilities of GPUs using H.264 prediction algorithms

Summary: We explore the processing power of two generations of GPUs by implementing H.264 prediction algorithms. We implement motion vector search and motion vector prediction on the GPU.

Student: Kristoffer Egil Bonarjee

Title: Investigating host–device communication in a GPU-based H.264 encoder

Summary: This thesis investigates the performance pitfalls of an H.264 encoder written for GPUs. More specifically, we look into the interaction between the host CPU and the GPU. We do not focus on optimizing the GPU code but, rather, on how the execution and communication are handled by the CPU. Given the large amount of manual labor required to optimize the GPU code, it is easy to neglect the CPU part of accelerated applications.

Student: Simen Sægrov

Title: Bagadus: Next generation sports analysis and multimedia platform using camera array and sensor network

Summary: Bagadus, a system that integrates a sensor system, soccer analytics, and video processing with OpenCV on a camera array, is presented. A proof-of-concept prototype is implemented based on the system installed at Alfheim stadium in Norway.

Student: Espen Oldeide Helgedagsrud

Title: Efficient implementation and processing of a real-time panorama video pipeline with emphasis on dynamic stitching

Summary: In this thesis, the Bagadus system is rewritten with real-time processing of the video as one of the main goals. This thesis focuses on the implementation of dynamic stitching and offloading this operation to a GPU.

Student: Marius Tennøe

Title: Efficient implementation and processing of a real-time panorama video pipeline with emphasis on background subtraction

Summary: In this thesis, the Bagadus system is rewritten with real-time processing of the video as one of the main goals. This thesis focuses on different implementations of background subtraction and offloading those algorithms to a GPU.

Student: Mikkel Næss

Title: Efficient implementation and processing of a real-time panorama video pipeline with emphasis on color correction

Summary: In this thesis, the Bagadus system is rewritten with real-time processing of the video as one of the main goals. This thesis focuses on the implementation of the color correction module and offloading this operation to a GPU.

Student: Ragnar Langseth

Title: Implementation of a distributed real-time video panorama pipeline for creating high quality virtual views

Summary: The Bagadus video pipeline with an updated camera array is redesigned with distributed processing in mind. Features such as HDR and the demosaicing of raw Bayer data from the new cameras are added to the GPU pipeline in Bagadus. A virtual camera is also extracted from the panoramic video.

Student: Vegard Aalbu

Title: MovieCutter: A system to make personalized video summaries from archived video content

Summary: This thesis investigates how adaptive streaming can be used to create a montage from events in a video archive. With metadata such as subtitles and chapters, users can search and generate customized video playlists from movies.

Student: Sigurd Ljødal

Title: Implementation of a real-time distributed video processing pipeline

Summary: The Bagadus video pipeline with an updated camera array is redesigned with distributed processing in mind. To do so, we use a PCI Express-based interconnect from Dolphin Interconnect Solutions and a low-level application programming interface for message passing called SISCI. We also investigate different techniques of moving data as efficiently as possible from cameras connected to a capture machine to GPU memory on the processing machine.

Student: Martin Alexander Wilhelmsen

Title: Real-time interactive cloud applications

Summary: This thesis investigates commodity hardware H.264 encoders and uses the NVENC hardware encoder found on modern GPUs from Nvidia to offload encoding in the Bagadus pipeline. We also implement support for streaming in Quake III to test the feasibility of using the hardware encoder in cloud gaming.

6.12 Other Publications

Several other papers were published in conferences during the PhD period. We did not include all the papers to limit the scope of this thesis. Instead, we provide a short summary of their contributions.

Disk input/output (I/O) We worked on I/O performance optimizations by improving file tree traversal performance by scheduling in user space [76, 77]. The technique proposed in these papers orders directory tree requests by logical block order on the physical disk. This optimization significantly improves the performance of file tree traversal operations; however, this is not possible in kernel space, since too few I/O operations are issued at a time for the scheduler to react efficiently. With a dirty file system, we were able to obtain up to four times the performance compared to that of a normal file tree traversal. This work clearly demonstrates the advantage of having several levels of schedulers and it can be adapted to the scheduling approach used by the P2G framework, where a high-level scheduler issues the workload to the different processing cores and low-level schedulers on the different processing cores carry out time slicing.

Fault Tolerant Routing We implemented dynamic fault-tolerant routing in an scalable coherent interconnect (SCI) network [116]. We implemented support for dynamic fault tolerance in an SCI network on hardware produced by Dolphin Interconnect Solutions. By dynamic fault tolerance, we mean that the interconnection network reroutes affected packets around a fault in the network, while the rest of the network remains fully functional. Our implementation focuses on a two-dimensional torus topology and is compatible with the existing hardware and software stack. The routing algorithm is tested in clusters with real hardware and our tests show that distributed databases such as MySQL are able to run uninterrupted while the network reacts to faults.

Chapter 7

Conclusion

Processing multimedia workloads with heterogeneous architectures is not a trivial task. The abstractions to program these architectures can be different and programmers often have to manually tune and optimize their applications on multiple heterogeneous architectures to achieve the desired performance. In many cases, these optimizations are not portable; for example, if the hardware is changed, the applications have to be optimized or even rewritten for the new architecture. Several languages and processing frameworks exist; however, they are typically designed to support batch processing and making them support real-time multimedia workloads is not a trivial task. Our main research question, stated in Section 1.2, states that we want to investigate how to develop and process multimedia workloads for modern heterogeneous multicore architectures. In this thesis, we addressed this issue from a low-level standpoint, learning about the behavior of heterogeneous architectures with simple multimedia workloads and how to use multiple heterogeneous architectures for a complex pipeline with several multimedia workloads. We also addressed the problem statement from a high-level standpoint, where we presented the design and evaluated the prototype of P2G, a framework for processing multimedia workloads on heterogeneous architectures.

7.1 Summary

In this thesis, we looked at heterogeneous multicore architectures and their ability to process multimedia workloads. First, we selected four different architectures. To learn more about their behavior, we first conducted several case studies with simple multimedia workloads, where we only performed optimizations for one architecture. The first architecture we experimented with was the Intel IXP network processor. This architecture was used for experiments involving network protocol translation [32]. The next architecture was the x86 processor architecture. The first case study was the efficient implementation of Motion JPEG (MJPEG) encoding [112]. We also conducted case studies on using multiple x86 cores for multi-rate video encoding [34] and running a multithreaded game server prototype [102]. For the GPU architecture, we carried out case studies on the memory architecture of GPUs [94], optimization of host–device communication [13], and cheat detection [117], and we also revisited the MJPEG workload with both a GPU [112] and Cell architecture.

The knowledge obtained from investigating the simple multimedia workloads was used to investigate a more complex multimedia workload, namely, the video processing pipeline of the Bagadus soccer analysis system [114]. We optimized the workload for multiple heterogeneous architectures to achieve the real-time capture, pre-processing, stitching, and encoding of a panoramic videostream from the soccer stadium on a single commodity gaming computer [113].

Using the knowledge gained from processing both simple and complex multimedia workloads on heterogeneous systems and also from our evaluation of multi-core scheduling mechanisms [115], we proposed a programming language and framework that exposes the parallelization opportunities of multimedia workloads for a runtime that allows for efficient execution on the available heterogeneous hardware [31]. A running prototype of this system, called P2G, running on a single machine with x86 multicore processors was developed together with two simple multimedia workloads. These workloads were used as proof of concept to show that we can express and run multimedia workloads in our framework.

7.2 Concluding Remarks

In the beginning of this thesis, we asked how a programmer efficiently develops multimedia workloads for modern heterogeneous multicore architectures. To answer this question, we further decomposed the question into three steps.

To learn about the behavior of heterogeneous architectures, we selected four architectures and looked at case studies with simple multimedia workloads optimized for only one architecture. From our evaluations, we observed that, for all the architectures, it is important to select algorithms that are suited to the architecture. To obtain optimal performance, especially on the Cell and x86, programmers must use architecture-specific vector extensions and rewrite their programs to use single instruction multiple data (SIMD) intrinsics. On the x86, we also noted the importance of balancing the number of threads used by the workloads to the available number of cores in the system. Too many threads executing on too few cores results in decreased performance due to contention and context switching overhead. On the GPU architecture, we noted the importance of using the correct memory space and also the importance of efficiently moving the data to and from the GPU. If the workload that is offloaded to the GPU is too small, performance can decrease compared to that of a CPU implementation. Our MJPEG experiments also suggest that programmers prefer the single instruction, multiple threads (SIMT) programming model exposed by the GPU compared to the SIMD model exposed by the Cell and the vector unit on the x86.

Next, we selected the two most promising heterogeneous architectures from our evaluation of simple multimedia workloads and evaluated a complex multimedia workload. One of the main requirements for this workload was for it to run in real time. We accomplished this by optimizing the workload for both heterogeneous architectures. The complex workload was the video subsystem of the Bagadus sports analysis system. Here, we learned that, in addition to making every module run separately in real time, we also had to make sure all the modules were running together in real time, as a pipeline. This required a great deal of manual tuning to decide which parts of the pipeline had to run on the CPU and which parts could be offloaded to a GPU. One of our observations was that the GPU's overall utilization was fairly low (only 14.8%). This finding, together with the manual labor required to optimize the pipeline to run in real time, highlighted the importance of having a processing framework supporting real-time multimedia workloads to fully utilize available resources and ease the development of future cross-platform systems.

Using all the knowledge gained from our studies of simple and complex workloads, we designed and proposed a framework for processing real-time multimedia workloads on heterogeneous architectures, called P2G. P2G allows the programmer to use a programming model similar to the SIMT model used in CUDA and to express as much parallelism as possible in the code. The framework will then use the workload's data, task, and pipeline parallelism to optimize the granularity of the programs, either at compilation time or at runtime. The framework is designed to support distributed processing and uses a two-level scheduling approach. A high-level scheduler maps the workloads to processing nodes and a low-level scheduler manages the time slices of the available processing cores in a node. The fundamental ideas of the P2G framework were implemented in a prototype of the framework running on an x86 multicore architecture. To test the prototype implementation, we used two simple multimedia workloads. Developing workloads in the P2G Kernel Language is effective compared to working with low-level abstractions such as SIMD intrinsics and threads. Kernel Language provides good abstractions and helps developers to express both data parallelism and task parallelism.

Even though we did not implement all the concepts in the P2G framework, we showed that we are able to express multimedia workloads in the P2G programming model and to scale performance when more processing resources are added to the system. However, a great deal of work remains before the processing efficiencies of multimedia workloads in the P2G framework are anywhere near what can be achieved for workloads written natively for the architecture. We believe we have made significant contributions in expressing workloads and designing a framework that supports some of the different heterogeneous architectures available today.

7.3 Future Work

Several areas have potential for further work and we highlight a few potential next steps.

- An interesting heterogeneous architecture that is not explored in this thesis is the Intel Xeon Phi many-core processor [21]. This coprocessor uses many simple x86 cores on a shared memory architecture with a 512-bit vector unit per core. We did not test any simple multimedia workloads on this architecture. It would therefore be very interesting to see how it performs compared to a GPU, as well as investigate how programmers have to think to program multimedia workloads for this architecture.

- The P2G prototype presented in this thesis runs on a single machine with x86 multicore processors and a shared memory architecture. An interesting research opportunity could be to rewrite and extend P2G to take advantage of GPUs using the Nvidia CUDA framework or the Intel Xeon Phi many-core architecture.

- In the current version of P2G, only the execution node is implemented. Another potential area for further work is to extend P2G to run on multiple machines. This is crucial for working with large data sets. The distribution of the framework would also introduce a new level of complexity in the scheduling and the design of efficient two-level schedulers presents several interesting research opportunities.

- The video pipeline of the Bagadus sports analysis system can run on a single x86 multicore machine with a high-end Nvidia GPU. If the P2G framework is extended with support for GPUs, it would be interesting to port parts of the Bagadus video pipeline to P2G. The feedback-based scheduling using instrumentation in P2G and the support for real-time workloads should be able to adapt and distribute the Bagadus video workload automatically on the CPU and GPU and make it run in real time, given sufficient processing resources in the system without the manual tuning required today.

It is also possible for new hardware to be developed that will open up interesting research topics within this field.

Bibliography

[1] A. Abdelkhalek and A. Bilas. Parallelization and performance of interactive multiplayer game servers. In *Proceedings of the IEEE International Parallel and Distributed Processing Symposium - IPDPS 2004*, page 72, 2004.

[2] Adobe. HTTP Dynamic Streaming. `http://www.adobe.com/mena_en/products/hds-dynamic-streaming.html` [Online. Last accessed: July 2014], 2010.

[3] Advanced Micro Devices. AMD Demonstrates Worlds First X86 Dual-Core Processor. `http://www.amd.com/us/press-releases/Pages/Press_Release_89872.aspx` [Online. Last accessed: January 2014], 2004.

[4] Apache. Hadoop. `http://hadoop.apache.org` [Online. Last accessed: July 2010], 2010.

[5] Y. Arai, T. Agui, and M. Nakajima. A Fast DCT-SQ Scheme for Images. *Transactions of IEICE*, 71(11):1095–1097, 1988.

[6] J. Armstrong. A history of Erlang. In *Proceedings of the 3rd ACM SIGPLAN Conference on History of Programming Languages*, pages 6:1–6:26, 2007.

[7] Ars Technica. A technical overview of the Emotion Engine. `http://arstechnica.com/gadgets/2000/02/ee/` [Online. Last accessed: July 2014], 2000.

[8] R. S. N. Arvind, R. S. Nikhil, and K. Pingali. I-structures: Data structures for parallel computing. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 11(4):598–632, 1989.

[9] J. Bankoski, J. Koleszar, L. Quillio, J. Salonen, P. Wilkins, and Y. Xu. VP8 Data Format and Decoding Guide. RFC 6386 (Informational), November 2011.

[10] P. B. Beskow, H. Espeland, H. K. Stensland, P. N. Olsen, S. Kristoffersen, E. A. Kristiansen, C. Griwodz, and P. Halvorsen. Distributed Real-Time Processing of Multimedia Data with the P2G Framework, 2011. Poster presented at ACM Eurosys 2011.

[11] P. B. Beskow, H. K. Stensland, H. Espeland, E. A. Kristiansen, P. N. Olsen, S. Kristoffersen, C. Griwodz, and P. Halvorsen. Processing of multimedia data using the P2G framework. In *Proceedings of the 19th ACM International Conference on Multimedia - MM '11*, pages 819–820. ACM, 2011.

[12] D. Blythe. The Direct3D 10 system. *ACM Transactions on Graphics*, 25(3):724, 2006.

[13] K. E. Bonarjee. *Investigating host-device communication in a GPU-based H.264 encoder*. Master thesis, University of Oslo, 2012.

[14] M. Brown and D. G. Lowe. Automatic Panoramic Image Stitching using Invariant Features. *International Journal of Computer Vision*, 74(1):59–73, August 2007.

[15] K. Cabeen and P. Gent. Image Compression and the discrete Cosine Transform. In *Math 45*. College of the Redwoods, 1998.

[16] Cairos Technologies. VIS.TRACK. `http://www.cairos.com/unternehmen/vistrack.php` [Online. Last accessed: October 2013], 2013.

[17] Camargus. Premium Stadium Video Technology Inrastructure. `http://www.camargus.com/` [Online. Last accessed: October 2013].

[18] R. Chaiken, B. Jenkins, P.-Å. Larson, B. Ramsey, D. Shakib, S. Weaver, and J. Zhou. SCOPE: easy and efficient parallel processing of massive data sets. *Proceedings of the VLDB Endowment*, 1(2):1265–1276, August 2008.

[19] B. L. Chamberlain, D. Callahan, and H. P. Zima. Parallel Programmability and the Chapel Language. *International Journal of High Performance Computing Applications*, 23(3), 2007.

[20] K. Chen and C. Lei. Network game design: hints and implications of player interaction. In *Proceedings of the 5th ACM SIGCOMM Workshop on Network and System Support for Games NetGames '06*, 2006.

[21] G. Chrysos. Intel Xeon Phi coprocessor (codename Knights Corner). In *Proceedings of the 24th Hot Chips Symposium - HotChip 2012*.

[22] H. S. Chu. Building a simple yet powerful MMO game architecture. `http://www.ibm.com/developerworks/architecture/library/ar-powerup1/` [Online. Last accessed: June 2010], 2008.

[23] Cisco Systems Inc. Visual Networking Index. `http://www.cisco.com/c/en/us/solutions/service-provider/visual-networking-index-vni/index.html` [Online. Last accessed: June 2010], 2010.

[24] D. D. Clark and D. L. Tennenhouse. Architectural considerations for a new generation of protocols. *ACM SIGCOMM Computer Communication Review*, 20(4):200–208, 1990.

[25] D. E. Comer, D. Gries, M. C. Mulder, A. Tucker, A. J. Turner, and P. R. Young. Computing as a discipline. *Communications of the ACM*, 32(1):9–23, 1989.

[26] M. de Kruijf and K. Sankaralingam. MapReduce for the Cell BE Architecture. *University of Wisconsin Computer Sciences Technical Report CS-TR-2007*, 1625, 2007.

[27] J. Dean and S. Ghemawat. MapReduce: simplified data processing on large clusters. *Communications of the ACM*, 51(1):107–113, 2004.

[28] E. W. Dijkstra. A note on two problems in connexion with graphs. *Numerische mathematik*, 1(1):269–271, 1959.

[29] P. Dizikes. Sports analytics: a real game-changer. `http://web.mit.edu/newsoffice/2013/sloan-sports-analytics-conference-2013-0304.html` [Online. Last accessed: October 2013], 2013.

[30] B. Drain. EVE Evolved: EVE Online's server model. `http://massively.joystiq.com/2008/09/28/eve-evolved-eve-onlines-server-model/` [Online. Last accessed: June 2010], 2008.

[31] H. Espeland, P. B. Beskow, H. K. Stensland, P. N. Olsen, S. Kristoffersen, C. Griwodz, and P. Halvorsen. P2G: A Framework for Distributed Real-Time Processing of Multimedia Data. In *Proceedings of the 40th International Conference on Parallel Processing Workshops - ICPPW 2011*, pages 416–426. IEEE, 2011.

[32] H. Espeland, C. H. Lunde, H. K. Stensland, C. Griwodz, and P. Halvorsen. Transparent protocol translation for streaming. In *Proceedings of the 15th ACM International Conference on Multimedia - MM '07*, pages 771–774. ACM, 2007.

[33] H. Espeland, C. H. Lunde, H. K. Stensland, C. Griwodz, and P. Halvorsen. Transparent protocol translation and load balancing on a network processor in a media streaming scenario. In *Proceedings of the 18th International Workshop on Network and Operating Systems Support for Digital Audio and Video - NOSSDAV '08*, pages 129–130. ACM, 2008.

[34] H. Espeland, H. K. Stensland, D. H. Finstad, and P. Halvorsen. Reducing Processing Demands for Multi-Rate Video Encoding. *International Journal of Multimedia Data Engineering and Management*, 3(2):1–19, 2012.

[35] D. H. Finstad, H. K. Stensland, H. Espeland, and P. Halvorsen. Improved Multi-Rate Video Encoding. In *Proceedings of the IEEE International Symposium on Multimedia - ISM 2011*, pages 293–300, 2011.

[36] J. A. Fisher. Very Long Instruction Word architectures and the ELI-512. In *Proceedings of the 10th annual International Symposium on Computer Architecture - ISCA '83*, pages 140–150. ACM, 1983.

[37] S. L. Flosi. comScore Releases April 2010 U.S. Online Video Rankings, 2010. Press Release.

[38] M. J. Flynn. Some Computer Organizations and Their Effectiveness. *IEEE Transactions on Computers*, C-21(9):948–960, September 1972.

[39] I. Foster. *Designing and Building Parallel Programs: Concepts and Tools for Parallel Software Engineering.* Addison-Wesley, 1995.

[40] V. R. Gaddam, R. Langseth, H. K. Stensland, P. Gurdjos, V. Charvillat, C. Griwodz, D. Johansen, and P. Halvorsen. Be your own cameraman: real-time support for zooming and panning into stored and live panoramic video. In *Proceedings of the 5th Annual ACM Conference on Multimedia Systems - MMSys 2014*, pages 168–171. ACM, 2014.

[41] B. Gedik, H. Andrade, K. Wu, P. S. Yu, and M. Doo. SPADE: the system s declarative stream processing engine. In *Proceedings of the ACM SIGMOD International Conference on Management of Data - SIGMOD 2008*, pages 1123–1134. ACM, 2008.

[42] F. Glover. Tabu search Part II. *ORSA journal on Computing*, 2(1):4–32, 1990.

[43] N. Goodnight and G. Humphreys. Computation on Programmable Graphics Hardware. *IEEE Computer Graphics and Applications*, 25(5):12–15, 2005.

[44] M. I. Gordon, W. Thies, and S. Amarasinghe. Exploiting coarse-grained task, data, and pipeline parallelism in stream programs. In *Proceedings of the 12th International Conference on Architectural Support for Programming Languages and Operating Systems - ASPLOS-XII*, pages 151–162. ACM, 2006.

[45] C. Griwodz and M. Zink. KOM(S) Streaming System. `http://komssys.sourceforge.net/html/index.html` [Online. Last accessed: September 2014], 2001.

[46] P. Halvorsen, S. Sægrov, A. Mortensen, D. K. C. Kristensen, A. Eichhorn, M. Stenhaug, S. Dahl, H. K. Stensland, V. R. Gaddam, C. Griwodz, and D. Johansen. BAGADUS: An Integrated System for Arena Sports Analytics A Soccer Case Study. In *Proceedings of the 4th Annual ACM Conference on Multimedia Systems - MMSys 2013*, pages 48–59, 2013.

[47] S. Han, K. Jang, K. Park, and S. Moon. Packetshader: a gpu-accelerated software router. *ACM SIGCOMM Computer Communication Review*, 41(4):195–206, 2011.

[48] R. I. Hartley and A. Zisserman. *Multiple View Geometry in Computer Vision.* Cambridge University Press, second edition, 2004.

[49] B. He, W. Fang, Q. Luo, N. K. Govindaraju, and T. Wang. Mars: a MapReduce framework on graphics processors. In *Proceedings of the 17th International Conference on Parallel Architectures and Compilation Techniques*, pages 260–269. ACM, 2008.

[50] B. Hendrickson and T. G. Kolda. Graph partitioning models for parallel computing. *ORSA Journal on Computing*, 26(12):1519–1534, 2000.

[51] M. Houston. Anatomy of AMD's TeraScale Graphics Engine, 2008. Slides of a talk given at SIGGRAPH 2008.

[52] P. H. J. Hughes, S. P. Jones, and P. Wadler. A history of Haskell: being lazy with class. In *Proceedings of the 3rd ACM SIGPLAN Conference on History of Programming Languages*, pages 12:1–12:55, 2007.

[53] K. Huguenin, A. Kermarrec, K. Kloudas, and F. Taiani. Content and Geographical Locality in User-Generated Content Sharing Systems. In *Proceedings of the 22nd International Workshop on Network and Operating Systems Support for Digital Audio and Video - NOSSDAV 2012*, pages 77–82, 2012.

[54] IBM. Cell Broadband Engine Architecture. Technical report, 2007.

[55] IBM, Sony, and Toshiba. *Cell Broadband Engine Programming Handbook*. IBM, 2008.

[56] Intel Corporation. Intel IXP1200 Network Processor Family: Hardware Reference Manual. Technical report, 2001.

[57] Intel Corporation. Intel IXP2400 Network Processor Family: Hardware Reference Manual. Technical report, 2003.

[58] Intel Corporation. Tick Tock Model. `http://www.intel.com/content/www/us/en/silicon-innovations/intel-tick-tock-model-general.html` [Online. Last accessed: February 2014], 2007.

[59] Intel Corporation. An Introduction to the Intel QuickPath Interconnect. Technical report, 2009.

[60] Intel Corporation. Intel Hyper-Threading Technology. `http://www.intel.com/content/www/us/en/architecture-and-technology/hyper-threading/hyper-threading-technology.html` [Online. Last accessed: January 2012], 2012.

[61] Intel Corporation. Intel Quick Sync Video. `http://www.intel.com/content/www/us/en/architecture-and-technology/quick-sync-video/quick-sync-video-general.html` [Online. Last accessed: January 2014], 2013.

[62] Interplay Sports. The Ultimate Video Analysis and Scouting Software. `http://www.interplay-sports.com/` [Online. Last accessed: October 2013], 2013.

[63] M. Isard, M. Budiu, Y. Yu, A. Birrell, and D. Fetterly. Dryad: distributed data-parallel programs from sequential building blocks. In *ACM SIGOPS Operating Systems Review*, volume 41, pages 59–72. ACM, 2007.

[64] ISO/IEC 14496-10:2003. *Information Technology - Coding of audio-visual objects - Part 10: Advanced Video Coding*. ISO/IEC, 2003.

[65] ITU-T Z.100. *Specification and Description Language (SDL)*. ITU, 2007.

[66] D. Johansen, T. Endestad, H. Riiser, C. Griwidz, P. Halvorsen, H. Johansen, T. Aarflot, J. Hurley, Å. Kvalnes, C. Gurrin, S. Zav, B. Olstad, and E. Aaberg. DAVVI: a prototype for the next generation multimedia entertainment platform. In *Proceedings of the 17th ACM International Conference on Multimedia - MM '09*, pages 989–990. ACM, 2009.

[67] D. Johansen, M. Stenhaug, R. B. A. Hansen, A. Christensen, and P. M. Høgmo. Muithu: Smaller Footprint, Potentially Larger Imprint. In *Proceedings of the 7th IEEE International Conference on Digital Information Management - ICDIM 2012*, pages 205–214, 2012.

[68] P. Kaewtrakulpong and R. Bowden. An Improved Adaptive Background Mixture Model for Realtime Tracking with Shadow Detection. In *Proceedings of the 2nd European Workshop on Advanced Video Based Surveillance Systems - AVBS '01*, pages 135–144, 2001.

[69] D. B. Kirk and W. H. Wen-mei. *Programming massively parallel processors: a hands-on approach*. Newnes, 2012.

[70] E. A. Kock, G. Essink, W. J. M. Smits, and P. Wolf. YAPI: Application modeling for signal processing systems. In *Proceeding of the 37th Design Automation Conference - DAC'2000*, pages 402–405. ACM, 2000.

[71] L. Kohn and N. Margulis. Introducing the Intel i860 64-bit microprocessor. *IEEE Micro*, 9(4):15–30, 1989.

[72] M. Kovac and N. Ranganathan. JAGUAR: A Fully Pipelined VLSI Architecture for JPEG Image Compression Standard. *Proceedings of the IEEE*, 83(2):247–258, 1995.

[73] I. Kuon, R. Tessier, and J. Rose. FPGA Architecture: Survey and Challenges. *Foundations and Trends in Electronic Design Automation*, 2(2):135–253, 2007.

[74] E. A. Lee and T. M. Parks. Dataflow process networks. *Proceedings of the IEEE*, 83(5):773–801, 1995.

[75] D. G. Lowe. Distinctive Image Features from Scale-Invariant Keypoints. *International Journal of Computer Vision*, 60(2):91–110, 2004.

[76] C. H. Lunde, H. Espeland, H. K. Stensland, and P. Halvorsen. Improving file tree traversal performance by scheduling I/O operations in user space. In *Proceedings of the 28th IEEE International Performance Computing and Communications Conference - IPCCC 2009*, pages 145–152. IEEE, 2009.

[77] C. H. Lunde, H. Espeland, H. K. Stensland, A. Petlund, and P. Halvorsen. Improving disk I/O performance on Linux. In *UpTimes - Proceedings of Linux-Kongress and OpenSolaris Developer Conference*, 2009.

[78] T. G. Mattson, R. Van der Wijngaart, and M. Frumkin. Programming the Intel 80-core network-on-a-chip Terascale Processor. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis - SC 2008*, pages 1–11. IEEE, 2008.

[79] C. Mims. Why CPUs Aren't Getting Any Faster. `http://www.technologyreview.com/view/421186/why-cpus-arent-getting-any-faster/` [Online. Last accessed: July 2014], 2010.

[80] A. Mortensen, V. R. Gaddam, H. K. Stensland, C. Griwodz, D. Johansen, and P. Halvorsen. Automatic event extraction and video summaries from soccer games. In *Proceedings of the 5th Annual ACM Conference on Multimedia Systems - MMSys 2014*, pages 176–179. ACM, 2014.

[81] Move Networks. Internet Television: Challenges and Opportunities. Technical report, Move Networks, Inc., 2008.

[82] Netronome. FlowNICs. `http://www.netronome.com/product/flownics/` [Online. Last accessed: July 2014], 2014.

[83] O. A. Niamut, R. Kaiser, G. Kienast, A. Kochale, J. Spille, O. Schreer, J. R. Hidalgo, J. Macq, and B. Shirley. Towards A Format-agnostic Approach for Production, Delivery and Rendering of Immersive Media. In *Proceedings of the 4th Annual ACM Conference on Multimedia Systems - MMSys 2013*, pages 249–260, 2013.

[84] J. Nickolls and W. J. Dally. The GPU Computing Era. *IEEE Micro*, 30(2):56–69, 2010.

[85] C. Nicolaou. An architecture for real-time multimedia communication systems. *IEEE Journal on Selected Areas in Communications*, 8(3):391–400, 1990.

[86] Numascale. NumaConnect - the Technology. `https://numascale.com/` [Online. Last accessed: September 2014], 2012.

[87] Nvidia. CUDA C Best Practices Guide. `http://docs.nvidia.com/pdf/CUDA_C_Best_Practices_Guide.pdf` [Online. Last accessed: September 2014], 2009.

[88] Nvidia. Nvidia's Next Generation CUDA Compute Architecture: Fermi. Technical report, 2010.

[89] Nvidia. NVIDIA Performance Primitives. `https://developer.nvidia.com/npp` [Online. Last accessed: March 2012], 2011.

[90] Nvidia. Nvidia's Next Generation CUDA Compute Architecture: Kepler GK110. Technical report, 2012.

[91] Nvidia. Tegra 4 Processors. `http://www.nvidia.com/object/tegra-4-processor.html` [Online. Last accessed: December 2013], 2013.

[92] Nvidia. CUDA C Programming Guide, version 6.0. `http://docs.nvidia.com/pdf/CUDA_C_Programming_Guide.pdf` [Online. Last accessed: September 2014], 2014.

[93] C. Olston, B. Reed, U. Srivastava, R. Kumar, and A. Tomkins. Pig latin: a not-so-foreign language for data processing. In *Proceedings of the ACM SIGMOD International Conference on Management of Data - SIGMOD 2008*, pages 1099–1110. ACM, 2008.

[94] A. Ottesen. Efficient parallelisation techniques for applications running on GPUs using the CUDA framework. Master thesis, University of Oslo, Norway, 2009.

[95] J. Ozer. First Look: H.264 and VP8 Compared. http://www.streamingmedia.com/articles/editorial/featured-articles/first-look-h.264-and-vp8-compared-67266.aspx [Online. Last accessed: June 2010], 2010.

[96] R. Pantos, J. Batson, D. Biderman, B. May, and A. Tseng. HTTP Live Streaming. http://tools.ietf.org/html/draft-pantos-http-live-streaming-04 [Online. Last accessed: October 2013], 2010.

[97] S. A. Pettersen, P. Halvorsen, D. Johansen, H. Johansen, V. Berg-Johansen, V. R. Gaddam, A. Mortensen, R. Langseth, C. Griwodz, and H. K. Stensland. Soccer video and player position dataset. In *Proceedings of the 5th Annual ACM Conference on Multimedia Systems - MMSys 2014*, pages 18–23. ACM, 2014.

[98] V. Pham, P. Vo, and V. T. Hung. GPU implementation of extended gaussian mixture model for background subtraction. In *Proceedings of the IEEE RIVF International Conference on Computing and Communication Technologies, Research, Innovation, and Vision for the Future - RIVF 2010*, pages 1–4. IEEE, 2010.

[99] M. Pritchard. How to Hurt the Hackers: The Scoop on Internet Cheating and How You Can Combat It. http://www.gamasutra.com/view/feature/3149/how_to_hurt_the_hackers_the_scoop_.php [Online. Last accessed: May 2009], 2000.

[100] Prozone. Prozone Sports – Introducing Prozone Performance Analysis Products. http://www.prozonesports.com/subsector/football/ [Online. Last accessed: October 2014].

[101] K. Raaen, H. Espeland, H. K. Stensland, A. Petlund, P. Halvorsen, and C. Griwodz. A demonstration of a lockless, relaxed atomicity state parallel game server (LEARS). In *Proceedings of the 10th Annual Workshop on Network and Systems Support for Games - NetGames 2011*, pages 1–3. IEEE, 2011.

[102] K. Raaen, H. Espeland, H. K. Stensland, A. Petlund, P. Halvorsen, and C. Griwodz. LEARS: A Lockless, Relaxed-Atomicity State Model for Parallel Execution of a Game Server Partition. In *Proceedings of the 41st International Conference on Parallel Processing Workshops - ICPPW 2012*, pages 382–389. IEEE, 2012.

[103] C. Ranger, R. Raghuraman, A. Penmetsa, G. Bradski, and C. Kozyrakis. Evaluating MapReduce for Multi-core and Multiprocessor Systems. In *Proceesing of the IEEE 13th International Symposium on High Performance Computer Architecture - HPCA 2007*, pages 13–24. IEEE, 2007.

[104] Real World Technologies. Intels Haswell CPU Microarchitecture. http://www.realworldtech.com/haswell-cpu/ [Online. Last accessed: February 2014], 2012.

[105] S. Sægrov, A. Eichhorn, J. Emerslund, H. K. Stensland, C. Griwodz, D. Johansen, and P. Halvorsen. BAGADUS: An Integrated System for Soccer Analysis (Demo). In *Proceedings of the ACM/IEEE International Conference on Distributed Smart Cameras - ICDSC 2012*, 2012.

[106] V. D. Salvo, A. Collins, B. McNeill, and M. Cardinale. Validation of Prozone: A new video-based performance analysis system. *International Journal of Performance Analysis in Sport (serial online)*, 6(1):108–119, 2006.

[107] T. Sato. The earth simulator: Roles and impacts. *Nuclear Physics B - Proceedings Supplements*, 129-130:102–108, 2004.

[108] P. Seeling, F. H. P. Fitzek, G. Ertli, A. Pulipaka, and M. Reisslein. Video network traffic and quality comparison of VP8 and H.264 SVC. In *Proceedings of the 3rd Workshop on Mobile Video Delivery - MoViD 2010*, pages 33–38, 2010.

[109] L. Seiler, R. Cavin, R. Espasa, E. Grochowski, T. Juan, P. Hanrahan, D. Carmean, E. Sprangle, T. Forsyth, M. Abrash, P. Dubey, S. Junkins, A. Lake, and J. Sugerman. Larrabee: a many-core x86 architecture for visual computing. *ACM Transactions on Graphics*, 27(3):18:1–18:15, 2008.

[110] Iraj Sodagar. The MPEG-DASH Standard for Multimedia Streaming Over the Internet. *IEEE MultiMedia*, 18(4):62–67, 2011.

[111] Stats Technology. STATS — SportVU — Football/Soccer. `http://www.sportvu.com/football.asp` [Online. Last accessed: October 2013], 2013.

[112] H. K. Stensland, H. Espeland, C. Griwodz, and P. Halvorsen. Tips, tricks and troubles: optimizing for Cell and GPU. In *Proceedings of the 20th International Workshop on Network and Operating Systems Support for Digital Audio and Video - NOSSDAV '10*, pages 75–80. ACM, 2010.

[113] H. K. Stensland, V. R. Gaddam, M. Tennøe, E. Helgedagsrud, M. Næss, H. K. Alstad, C. Griwodz, P. Halvorsen, and D. Johansen. Processing Panorama Video in Real-Time. *International Journal of Semantic Computing (IJSC)*, 8(2):209–227, 2014.

[114] H. K. Stensland, V. R. Gaddam, M. Tennøe, E. Helgedagsrud, M. Næss, H. K. Alstad, A. Mortensen, R. Langseth, S. Ljødal, Ø. Landsverk, C. Griwodz, P. Halvorsen, M. Stenhaug, and D. Johansen. Bagadus: An integrated real-time system for soccer analytics. *ACM Transactions on Multimedia Computing, Communications, and Applications (TOMM)*, 10(1s):1–21, 2014.

[115] H. K. Stensland, C. Griwodz, and P. Halvorsen. Evaluation of multi-core scheduling mechanisms for heterogeneous processing architectures. In *Proceedings of the 18th International Workshop on Network and Operating Systems Support for Digital Audio and Video - NOSSDAV '08*, pages 33–38. ACM, 2008.

[116] H. K. Stensland, O. Lysne, R. Nordstrøm, and H. Kohmann. Making an SCI fabric dynamically fault tolerant. In *Proceedings of the IEEE International Symposium on Parallel and Distributed Processing - IPDPS 2008*, pages 1–8. IEEE, 2008.

[117] H. K. Stensland, M. Ø. Myrseth, C. Griwodz, and P. Halvorsen. Cheat detection processing: A GPU versus CPU comparison. In *Proceedings of the 9th IEEE Annual Workshop on Network and Systems Support for Games - NetGames 2010*, pages 1–6. IEEE, 2010.

[118] J. M. Tendler, J. S. Dodson, J. S. Fields, H. Le, and B. Sinharoy. POWER4 system microarchitecture. *IBM Journal of Research and Development*, 46(1):5–25, 2002.

[119] M. Tennøe, E. O. Helgedagsrud, M. Næss, H. K. Alstad, H. K. Stensland, V. R. Gaddam, D. Johansen, C. Griwodz, and P. Halvorsen. Efficient Implementation and Processing of a Real-Time Panorama Video Pipeline. In *Proceedings of the IEEE International Symposium on Multimedia - ISM 2013*, pages 76–83. IEEE, 2013.

[120] M. Tennøe, E. O. Helgedagsrud, M. Næss, H. K. Alstad, H. K. Stensland, P. Halvorsen, and C. Griwidz. Realtime Panorama Video Processing Using NVIDIA GPUs, 2013. Poster presented at Nvidia GPU Technology Conference 2013.

[121] The TCPdump and libpcap project. `http://www.tcpdump.org/` [Online. Last accessed: May 2013], 2013.

[122] W. Thies, V. Chandrasekhar, and S. Amarasinghe. A Practical Approach to Exploiting Coarse-Grained Pipeline Parallelism in C Programs. In *Proceedings of the 40th Annual IEEE/ACM International Symposium on Microarchitecture*, MICRO 40, pages 356–369. IEEE, 2007.

[123] M. Thompson and A. Pimentel. Towards Multi-application Workload Modeling in Sesame for System-Level Design Space Exploration. In *Embedded Computer Systems: Architectures, Modeling, and Simulation*, volume 4599 of *Lecture Notes in Computer Science*, pages 222–232. Springer Berlin / Heidelberg, 2007.

[124] S. V. Valvåg and D. Johansen. Oivos: Simple and Efficient Distributed Data Processing. In *Proceedings of the 10th IEEE International Conference on High Performance Computing and Communications - HPCC'08*, pages 113–122, 2008.

[125] S. V. Valvåg and D. Johansen. Cogset: A Unified Engine for Reliable Storage and Parallel Processing. In *Proceeding of the 6th IFIP International Conference on Network and Parallel Computing - NPC'08*, pages 174–181, 2009.

[126] Verdione Project. Verdione - Technology for mixed-reality stages. `http://verdione.org/` [Online. Last accessed: October 2014], 2011.

[127] Z. Vrba, P. Halvorsen, C. Griwodz, and P. B. Beskow. Kahn Process Networks are a Flexible Alternative to MapReduce. *Proceeding of the IEEE International Conference on High Performance Computing and Communications - HPCC 2009*, pages 154–162, 2009.

[128] Z. Vrba, P. Halvorsen, C. Griwodz, P. B. Beskow, and D. Johansen. The Nornir Runtime System for Parallel Programs Using Kahn Process Networks. In *Proceedings of the 6th IFIP International Conference on Network and Parallel Computing - ICNPC 2009*, pages 1–8. IEEE, 2009.

[129] W. Weimer, M. Boyer, and K. Skadron. Automated Dynamic Analysis of CUDA Programs. In *Proceedings of the 3rd Workshop on Software Tools for MultiCore Systems - STMCS 2008*, 2008.

[130] W. White, A. Demers, C. Koch, J. Gehrke, and R. Rajagopalan. Scaling games to epic proportions. In *Proceedings of the IEEE International Conference on Management of Data - SIGMOD 2007*, pages 31–42, 2007.

[131] T. Wiegand, G.J. Sullivan, G. Bjontegaard, and A. Luthra. Overview of the H.264/AVC video coding standard. *IEEE Transactions on Circuits and Systems for Video Technology*, 13(7):560–576, 2003.

[132] M. A. Wilhelmsen, H. K. Stensland, V. R. Gaddam, P. Halvorsen, and C. Griwodz. Performance and Application of the NVIDIA NVENC H.264 Encoder, 2014. Poster presented at Nvidia GPU Technology Conference 2014.

[133] E. Wu and Y. Liu. Emerging technology about GPGPU. In *Proceedings of the IEEE Asia Pacific Conference on Circuits and Systems - APCCAS 2008*, pages 618–622. IEEE, 2008.

[134] N. Wu, M. Wen, W. Wu, J. Ren, H. Su, C. Xun, and C. Zhang. Streaming HD H.264 encoder on programmable processors. In *Proceedings of the 17th ACM International Conference on Multimedia - MM '09*, pages 371–380. ACM, 2009.

[135] Xilinx. Zynq-7000 All Programmable SoC First Generation Architecture. Technical report, 2013.

[136] Y. Xiong and K. Pulli. Color correction for mobile panorama imaging. In *Proceedings of the First International Conference on Internet Multimedia Computing and Service - ICIMCS '09*, pages 219–226, 2009.

[137] J. Yan and B. Randell. A systematic classification of cheating in online games. In *Proceedings of 4th ACM SIGCOMM workshop on Network and system Support for Games - NetGames '05*, pages 1–9, 2005.

[138] H. Yang, A. Dasdan, R. Hsiao, and D. Parker. Map-reduce-merge: simplified relational data processing on large clusters. In *Proceedings of the ACM SIGMOD International Conference on Management of Data - SIGMOD 2007*, pages 1029–1040. ACM, 2007.

[139] YouTube. Statistics - YouTube. `https://www.youtube.com/yt/press/statistics.html` [Online. Last accessed: October 2014], 2014.

[140] Y. Yu, M. Isard, D. Fetterly, M. Budiu, U. Erlingsson, P. Gunda, and J. Currey. DryadLINQ: A System for General-Purpose Distributed Data-Parallel Computing Using a High-Level Language, 2008.

[141] A. Zambelli. Smooth Streaming Technical Overview. `http://www.iis.net/learn/media/on-demand-smooth-streaming/smooth-streaming-technical-overview` [Online. Last accessed: October 2014], 2009.

[142] T. Zeiser, G. Hager, and G. Wellein. The world's fastest CPU and SMP node: Some performance results from the NEC SX-9. In *Proceedings of the IEEE International Parallel and Distributed Processing Symposium - IPDPS 2009*, pages 1–8. IEEE, 2009.

[143] Z. Zhang. Flexible camera calibration by viewing a plane from unknown orientations. In *Proceeding of the IEEE International Conference on Computer Vision - ICCV 1999*, pages 666–673, 1999.

[144] Z. Zivkovic. Improved adaptive Gaussian mixture model for background subtraction. In *Proceedings of the 17th IEEE International Conference on Pattern Recognition - ICPR 2004*, pages 28 – 31 Vol.2, 2004.

[145] Z. Zivkovic and F. van der Heijden. Efficient adaptive density estimation per image pixel for the task of background subtraction. *Pattern Recognition Letters*, 27(7):773–780, 2006.

[146] ZXY. ZXY Sport Tracking. `http://www.zxy.no/` [Online. Last accessed: January 2014], 2013.

Part II

Research Papers

Paper I: Transparent Protocol Translation for Streaming

Title: Transparent Protocol Translation for Streaming [32].

Authors: H. Espeland, C. H. Lunde, H. K. Stensland, C. Griwodz, and P. Halvorsen.

Published: Proceedings of the 15th International Multimedia Conference (MM), ACM, 2007.

Paper II: Evaluation of Multi-Core Scheduling Mechanisms for Heterogeneous Processing Architectures

Title: Evaluation of Multi-Core Scheduling Mechanisms for Heterogeneous Processing Architectures [115].

Authors: H. K. Stensland, C. Griwodz, and P. Halvorsen.

Published: The 18th International Workshop on Network and Operating Systems Support for Digital Audio and Video (NOSSDAV), ACM, 2008.

Paper III: Tips, Tricks and Troubles: Optimizing for Cell and GPU

Title: Tips, Tricks and Troubles: Optimizing for Cell and GPU [112].

Authors: H. K. Stensland, H. Espeland, C. Griwodz, and P. Halvorsen.

Published: The 20th International Workshop on Network and Operating Systems Support for Digital Audio and Video (NOSSDAV), ACM, 2010.

Paper IV: Cheat Detection Processing: A GPU versus CPU Comparison

Title: Cheat Detection Processing: A GPU versus CPU Comparison [117].

Authors: H. K. Stensland, M. Ø. Myrseth, C. Griwodz, P. Halvorsen.

Published: Workshop on Network and Systems Support for Games (NetGames 2010), ACM/IEEE, 2010.

Paper V: Reducing Processing Demands for Multi-Rate Video Encoding: Implementation and Evaluation

Title: Reducing Processing Demands for Multi-Rate Video Encoding: Implementation and Evaluation [34].

Authors: H. Espeland, H. K. Stensland, D. H. Finstad, P. Halvorsen.

Published: International Journal of Multimedia Data Engineering and Management (IJMDEM), Volume 3, Issue 2, IGI Global, 2012.

Paper VI: LEARS: A Lockless, Relaxed-Atomicity State Model for Parallel Execution of a Game Server Partition

Title: LEARS: A Lockless, Relaxed-Atomicity State Model for Parallel Execution of a Game Server Partition [102].

Authors: K. Raaen, H. Espeland, H. Stensland, A. Petlund, P. Halvorsen, and C. Griwodz.

Published: Proceedings of the International Workshop on Scheduling and Resource Management for Parallel and Distributed Systems (SRMPDS) - The 2012 International Conference on Parallel Processing Workshops, IEEE, 2012.

Paper VII: P2G: A Framework for Distributed Real-Time Processing of Multimedia Data

Title: P2G: A Framework for Distributed Real-Time Processing of Multimedia Data [31].

Authors: H. Espeland, P. B. Beskow, H. K. Stensland, P. N. Olsen, S. B. Kristoffersen, C. Griwodz, and P. Halvorsen.

Published: Proceedings of the International Workshop on Scheduling and Resource Management for Parallel and Distributed Systems (SRMPDS) - The 2011 International Conference on Parallel Processing Workshops, IEEE, 2011.

Paper VIII: Bagadus: An Integrated Real-Time System for Soccer Analytics

Title: Bagadus: An Integrated Real-Time System for Soccer Analytics [114].

Authors: H. K. Stensland, V. R. Gaddam, M. Tennøe, E. Helgedagsrud, M. Næss, H. K. Alstad, A. Mortensen, R. Langseth, S. Ljødal, Ø. Landsverk, M. Stenshaug, P. Halvorsen, C. Griwodz and D. Johansen.

Published: ACM Transactions on Multimedia Computing, Communications and Applications (TOMM), Volume 10, Issue 1s, ACM, 2014.

Paper IX: Processing Panorama Video in Real-Time

Title: Processing Panorama Video in Real-Time [113].

Authors: H. K. Stensland, V. R. Gaddam, M. Tennøe, E. Helgedagsrud, M. Næss, H. K. Alstad, C. Griwodz, P. Halvorsen and D. Johansen.

Published: International Journal of Semantic Computing (IJSC), Volume 8, Issue 2, World Scientific, 2014.

Posters and live demonstrations

Transparent protocol translation and load balancing on a network processor in a media streaming scenario

Title: Transparent protocol translation and load balancing on a network processor in a media streaming scenario [33].

Authors: H. Espeland, C. Lunde, H. K. Stensland, C. Griwodz, and P. Halvorsen.

Published: Proceedings of the 18th International Workshop on Network and Operating Systems Support for Digital Audio and Video - NOSSDAV '08.

Summary: A live demonstration of the proxy was presented at NOSSDAV.

Processing of Multimedia Data using the P2G Framework

Title: Processing of Multimedia Data using the P2G Framework [11].

Authors: P. B. Beskow, H. K. Stensland, H. Espeland, E. A. Kristiansen, P. N. Olsen, S. B. Kristoffersen, C. Griwodz, and P. Halvorsen.

Published: Proceedings of the 19th ACM international conference on Multimedia (MM), ACM, 2011.

Summary: A live demonstration of the P2G framework encoding video and dynamically adapting to the available resources was presented at ACM Multimedia.

Distributed Real-Time Processing of Multimedia Data with the P2G Framework

Title: Distributed Real-Time Processing of Multimedia Data with the P2G Framework [10].

Authors: P. B. Beskow, H. Espeland, H. K. Stensland, P. N. Olsen, S. B. Kristoffersen, E. A. Kristiansen, C. Griwodz, and P. Halvorsen.

Published: EuroSys 2011 (Poster Session), ACM, 2011.

Summary: Ideas about the P2G prototype, with early experimental results were presented at a poster session on the EuroSys conference.

A Demonstration Of a Lockless, Relaxed Atomicity State Parallel Game Server (LEARS)

Title: A Demonstration Of a Lockless, Relaxed Atomicity State Parallel Game Server (LEARS) [101].

Authors: K. Raaen, H. Espeland, H. K. Stensland, A. Petlund, P. Halvorsen, and C. Griwodz.

Published: Workshop on Network and Systems Support for Games (NetGames), IEEE / ACM, 2011.

Summary: A live demonstration of the LEARS lockless game server was presented at NetGames.

BAGADUS: An Integrated System for Soccer Analysis

Title: BAGADUS: An Integrated System for Soccer Analysis [105].

Authors: S. Sægrov, A. Eichhorn, J. Emerslund, H. K. Stensland, C. Griwodz, and P. Halvorsen

Published: Proceedings of the International Conference on Distributed Smart Cameras (ICDSC), ACM /IEEE, 2012

Summary: The offline version of the Bagadus Soccer Analysis system demonstrated at ICDSC.

Realtime Panorama Video Processing Using NVIDIA GPUs

Title: Realtime Panorama Video Processing Using NVIDIA GPUs [120].

Authors: M. Tennøe, E. O. Helgedagsrud, M. Næss, H. K. Alstad, H. K. Stensland, P. Halvorsen, and C. Griwodz.

Published: Nvidia GPU Technology Conference, 2013.

Summary: Poster about the real-time Bagadus panorama pipeline presented at GTC.

Performance and Application of the NVIDIA NVENC H.264 Encoder

Title: Performance and Application of the NVIDIA NVENC H.264 Encoder [132].

Authors: M. A. Wilhelmsen, H. K. Stensland, V. R. Gaddam, P. Halvorsen, and C. Griwodz

Published: Nvidia GPU Technology Conference, 2014.

Summary: Poster about using hardware encoder for delivery of interactive video streams presented at GTC.

Other research papers

Efficient Implementation and Processing of a Real-time Panorama Video Pipeline

Title: Efficient Implementation and Processing of a Real-time Panorama Video Pipeline [119].

Authors: M. Tennøe, E. O. Helgedagsrud, M. Næss, H. K. Alstad, H. K. Stensland, V. R. Gaddam, D. Johansen, P. Halvorsen, and C. Griwodz.

Published: Proceedings of the International Symposium on Multimedia (ISM), IEEE, 2013.

Abstract: High resolution, wide field of view video generated from multiple camera feeds has many use cases. However, processing the different steps of a panorama video pipeline in real-time is challenging due to the high data rates and the stringent requirements of timeliness. We use panorama video in a sport analysis system where video events must be generated in real-time. In this respect, we present a system for real-time panorama video generation from an array of low-cost CCD HD video cameras. We describe how we have implemented different components and evaluated alternatives. We also present performance results with and without co- processors like graphics processing units (GPUs), and we evaluate each individual component and show how the entire pipeline is able to run in real-time on commodity hardware.

Bagadus: An Integrated System for Arena Sports Analytics A Soccer Case Study

Title: Bagadus: An Integrated System for Arena Sports Analytics A Soccer Case Study [46].

Authors: P. Halvorsen, S. Sægrov, A. Mortensen, D. K. C. Kristensen, A. Eichhorn, M. Stenhaug, S. Dahl, H. K. Stensland, V. R. Gaddam, C. Griwodz, and D. Johansen

Published: Proceedings of the 4th annual ACM conference on Multimedia Systems (MMSYS), ACM, 2013.

Abstract: Sports analytics is a growing area of interest, both from a computer system view to manage the technical challenges and from a sport performance view to aid the development of athletes. In this paper, we present Bagadus, a prototype of a sports analytics application using soccer as a case study. Bagadus integrates a sensor system, a soccer analytics annotations system and a video processing system using a video camera array. A prototype is currently installed at Alfheim Stadium in Norway, and in this paper, we describe how the system can follow and zoom in on particular player(s). Next, the system will playout events from the games using stitched panorama video or camera switching mode and create video summaries based on queries to the sensor system. Further- more, we evaluate the system from a systems point of view, benchmarking different approaches, algorithms and trade-offs.

Improved Multi-Rate Video Encoding

Title: Improved Multi-Rate Video Encoding [35].

Authors: D. H. Finstad, H. K. Stensland, H. Espeland, and P. Halvorsen

Published: Proceedings of the International Symposium on Multimedia (ISM), IEEE, 2011.

Abstract: Adaptive HTTP streaming is frequently used for both live and on-Demand video delivery over the Internet. Adaptiveness is often achieved by encoding the video stream in multiple qualities (and thus bitrates), and then transparently switching between the qualities according to the bandwidth fluctuations and the amount of resources available for decoding the video content on the end device. For this kind of video delivery over the Internet, H.264 is currently the most used codec, but VP8 is an emerging open-source codec expected to compete with H.264 in the streaming scenario. The challenge is that, when encoding video for adaptive video streaming, both VP8 and H.264 run once for each quality layer, i.e., consuming both time and resources, especially important in a live video delivery scenario. In this paper, we address the resource consumption issues by proposing a method for reusing redundant steps in a video encoder, emitting multiple outputs with varying bitrates and qualities. It shares and reuses the computational heavy analysis step, notably macro-block mode decision, intra prediction and inter prediction between the instances, and outputs video in several rates. The method has been implemented in the VP8 reference encoder, and experimental results show that we can encode the different quality layers at the same rates and qualities compared to the VP8 reference encoder, while reducing the encoding time significantly.

Improving File Tree Traversal Performance by Scheduling I/O Operations in User space

Title: Improving File Tree Traversal Performance by Scheduling I/O Operations in User space [76].

Authors: C. H. Lunde, H. Espeland, H. K. Stensland, and P. Halvorsen.

Published: Proceedings of the 28th IEEE International Performance Computing and Communications Conference (IPCCC), IEEE, 2009.

Abstract: Current in-kernel disk schedulers provide efficient means to optimize the order (and minimize disk seeks) of issued, in-queue I/O requests. However, they fail to optimize sequential multi-file operations, like traversing a large file tree, because only requests from one file are available in the scheduling queue at a time. We have therefore investigated a user-level, I/O request sorting approach to reduce inter-file disk arm movements. This is achieved by allowing applications to utilize the placement of inodes and disk blocks to make a one sweep schedule for all file I/Os requested by a process, i.e., data placement information is read first before issuing the low-level I/O requests to the storage system. Our experiments with a modified version of tar show reduced disk arm movements and large performance improvements.

Improving Disk I/O Performance on Linux

Title: Improving Disk I/O Performance on Linux [77].

Authors: C. H. Lunde, H. Espeland, H. K. Stensland, A. Petlund, and P. Halvorsen.

Published: UpTimes - Proceedings of Linux-Kongress and OpenSolaris Developer Conference, GUUG, 2009.

Abstract: The existing Linux disk schedulers are in general efficient, but we have identified two scenarios where we have observed a non-optimal behavior. The first is when an application requires a fixed bandwidth, and the second is when an operation performs a file tree traversal. In this paper, we address both these scenarios and propose solutions which both increase performance.

Making an SCI Fabric Dynamically Fault Tolerant

Title: Making an SCI Fabric Dynamically Fault Tolerant [116].

Authors: H. K. Stensland, O. Lysne, R. Nordstrœm, and H. Kohmann.

Published: Proceedings of the IEEE International Symposium on Parallel and Distributed Processing (IPDPS), 2008.

Abstract: In this paper we present a method for dynamic fault tolerant routing for SCI networks implemented on Dolphin Interconnect Solutions hardware. By dynamic fault tolerance, we mean that the interconnection network reroutes affected packets around a fault, while the rest of the network is fully functional. To the best of our knowledge this is the first reported case of dynamic fault tolerant routing available on commercial off the shelf interconnection network technology without duplicating hardware resources. The development is focused around a 2-D torus topology, and is compatible with the existing hardware, and software stack. We look into the existing mechanisms for routing in SCI. We describe how to make the nodes that detect the faulty component do routing decisions, and what changes are needed in the existing routing to enable support for local rerouting. The new routing algorithm is tested on clusters with real hardware. Our tests show that distributed databases like MySQL can run uninterruptedly while the network reacts to faults. The solution is now part of Dolphin Interconnect Solutions SCI driver, and hardware development to further decrease the reaction time is underway.

Appendix A

BNF Grammar of the P2G Kernel Language

```
%token <n>          IDENTIFIER
%token <n>          INTEGER
%token <n>          VECTOR
%token <n>          INTRINSIC
%token <s>          NATIVE
%token TYPE
%token INDEX
%token LOCAL
%token AGE
%token LAST
%token SIZE
%token ORDERED
%token INT
%token INCR
%token WRAP
%token FETCH
%token STORE
%token DEF
%token NEXT
%token TOP_HEADER
%token TOP_CODE
%token HEADER
%token CODE
%token BIND
%token IF
%token THEN
%token ELSE
%token END
%token DOTDOT
%token EQ
%token LE
%token GE
%token HOUR
%token MIN
%token SEC
%token US
%token MS
```

```
%token FINAL
%token FINALIZE
%token FINALIZE_ON_ALL
%token FINALIZE_ON_ONE
%token TIMER
%token SET
%token NOW
%token DEADLINE

%right '='
%left  '<' '>' GE LE EQ
%left  DOTDOT ':'
%left  '+' '-'
%left  '*' '/' '%'
%left  UNARY_MINUS_PRECEDENCE

%%

start: statements
     ;

statements: statements statement
          |
          ;

statement: field_definition
         | timer_definition
         | kernel_definition
         | TOP_HEADER ':' NATIVE /* top of generated header file */
         | TOP_CODE   ':' NATIVE /* top of generated code file */
         | HEADER     ':' NATIVE /* after field definitions in generated
             header file */
         | CODE       ':' NATIVE /* after field declarations in generated
             code file */
         ;

/**
 * % Each field is a virtual multi-dimensional array of data.
 * % It is virtual because the kernel program forbids to write to
 * % the same field in more than one unique kernel, which does
 * % naturally not make sense in reality.
 * % The data cannot be stored contiguously in memory, either,
 * % because parallel architecture will encourage pipelining of
 * % data through several kernels before writing it to a
 * % contiguous location in memory (if ever).
 * % In the process of mapping, many of these fields will be
 * % identified with each other or be optimized out.
 * % A field should not be considered a bunch of memory cells,
 * % but a bunch of memory cells at the time of a kernel's run.
 * % Example field definitions:
 * %   float128 f;
 * %   int32[4] array;
 * %   vector3<float64>[3] three_coordinates, three_more, and_three_more;
 * %   vector8 complex128[][8][][][256] weird_stuff;
 */
```

```
field_definition: datatype define_global_field_names field_attributes ';'
                | intrinsic_datatype error ';'
                ;

datatype: intrinsic_datatype field_dimensions
        | VECTOR intrinsic_datatype field_dimensions
        ;

intrinsic_datatype: INTRINSIC
                  | TYPE type_name
                  ;
type_name: IDENTIFIER
         ;

field_dimensions: field_dimensions '[' INTEGER ']'
                | field_dimensions '[' ']'
                | /* empty */
                ;
field_attributes: field_attributes AGE '(' FINALIZE_ON_ALL ')'
                | field_attributes AGE '(' FINALIZE_ON_ONE ')'
                | field_attributes AGE
                | field_attributes ORDERED
                | /* empty */
                ;
define_global_field_names: define_global_field_names ','
   define_global_field_name
                         | define_global_field_name
                         ;

timer_definition: TIMER def_timer_names timer_attributes ';'
                ;
def_timer_names: def_timer_names ',' define_timer_name
               | define_timer_name
               ;
timer_attributes: timer_attributes AGE
                | /* empty */
                ;

define_local_field_names: define_local_field_names ','
   define_local_field_name
                        | define_local_field_name
                        ;

kernel_definition: kernel_name kernel_native_attributes ':' kernel_head
   NATIVE kernel_tail kernel_deadline
                 | kernel_name kernel_native_attributes ':' kernel_head
                    kernel_tail kernel_deadline
                 | kernel_name kernel_native_attributes ':' kernel_head IF
                    NATIVE THEN kernel_tail END opt_semicolon
                    kernel_deadline
                 | kernel_name kernel_native_attributes ':' kernel_head IF
                    NATIVE THEN kernel_tail ELSE kernel_tail END
                    opt_semicolon kernel_deadline
                 ;
```

```
opt_semicolon: ';'
             | /* empty */
             ;

/**
 * Native attributes will be used to give the dependency
 * analyser hints about special kernels.
 * In principle, a kernel can be placed anywhere, but if
 * the native code requires user input, disk access and so on,
 * it should be expressed by a list of identifiers
 * (commas strictly optional and without semantic meaning).
 */
kernel_native_attributes: BIND kernel_native_attrlist
                        | '%' error
                        | /* empty */
                        ;
kernel_native_attrlist: kernel_native_attrlist IDENTIFIER
                      | IDENTIFIER
                      ;
kernel_head: kernel_head fetch_statement
           | kernel_head index_var_declaration
           | kernel_head field_size_expression
           | kernel_head DEF error ';'
           | /* empty */
           ;
kernel_tail: kernel_tail store_statement
           | kernel_tail next_statement
           | kernel_tail timer_set_statement
           | /* empty */
           ;

kernel_deadline: kernel_deadline deadline_statement
           | /* empty */
           ;

field_size_expression: SIZE field_size '=' field_size_expr ';'
                     ;
field_size_expr: field_size_expr '*' field_size_expr
               | field_size_expr '/' field_size_expr
               | field_size_expr '%' field_size_expr
               | field_size_expr '+' field_size_expr
               | field_size_expr '-' field_size_expr
               | '(' field_size_expr ')'
               | INTEGER
               | '-' INTEGER %prec UNARY_MINUS_PRECEDENCE
               | '+' INTEGER %prec UNARY_MINUS_PRECEDENCE
               | field_size
                 ;

field_size: use_field_name field_ages '.' INTEGER
          ;

field_ages: field_ages '.' AGE '(' age_expr ')'
          |
          ;
```

```
index_var_declaration: INDEX index_var_int_or_not index_var_wrap_or_not
    index_var_increment index_vars ';'
                        | LOCAL  index_var_int_or_not index_var_wrap_or_not
                          index_var_increment index_vars ';'
                        | INT  index_var_wrap_or_not index_var_increment
                          index_vars ';'
                        | AGE    index_vars ';'
                        | LOCAL datatype define_local_field_names ';'
                        | INDEX intrinsic_datatype error ';'
                        | LOCAL intrinsic_datatype error ';'
                        | AGE intrinsic_datatype error ';'
                        ;
index_var_int_or_not: INT
                      | /* empty */
                      ;
index_var_wrap_or_not: WRAP
                       | /* empty */
                       ;
index_var_increment: INCR '(' INTEGER ')'
                     | /* empty */
                     ;
index_vars: index_vars ',' index_name
          | index_name
          ;
fetch_statement: FETCH use_field_name index_use '=' use_field_name
    index_use ';'
               ;
store_statement: store_final_prefix STORE use_field_name index_use '='
    use_field_name index_use ';'
               ;
store_final_prefix: FINAL
                  |
                         ;
finalize_statement: FINALIZE use_field_name index_use ';'
                  ;
next_statement: NEXT index_name ';'
              ;
timer_set_statement: SET use_timer_name age_list '=' timer_expr ';'
                   ;

deadline_statement: DEADLINE '(' deadline_condition ')' deadline_actions
    END opt_semicolon
                  ;
deadline_condition: timer_expr expr_cond_cmp timer_expr
                  ;
deadline_actions: deadline_actions store_statement
                | deadline_actions finalize_statement
                | /* empty */
                      ;

/**
 * % Expression lists allow fetch and store operations with indices
 * % that are computed statically when dependencies are evaluated.
 * % It would be nice if they could be evaluated statically, but that
 * % would require constant field sizes and we don't want to limit
 * % ourselves to that.
 * % As a very simple example consider the following kernel head:
 * %   int64[10] a,b;
 * %   def x;
 * %   fetch i=a(x);
 * %   fetch j=b(9-x);
 * % This initiates 10 kernels, because the fields are C-indexed and
 * % both have 10 fields.
 * % The fetch statements
 * %   fetch i=a(x);
 * %   fetch j=b(10-x);
 * % on the other hand, would initiate only 9 kernels, because the
 * % case x==0 is outside the range of field b.
 * %
 * % However, it is also possible to use larger blocks of memory
 * % in a single kernel. Consider:
 * %   float64[32][32] field;
 * %   fetch v = field[x:8][y:8];
 * % This avoids that 32x32=1024 kernels are started. Instead,
 * % it starts 16 kernels, where v is interpreted as double v[8][8].
 * % However, if you want to start 1024 kernel and make all possible
 * % 8x8 sub-blocks available to the native code, you have to
 * % do the following:
 * %   float64[32][32] field;
 * %   fetch v00 = field[x][y];
 * %   fetch v01 = field[x][y+1];
 * %   ...
 * %   fetch v77 = field[x+7][y+7];
 * % If people want this, we can create a better syntax for this later.
 * % It might make sense, for example to parallelize motion vector
 * % search at a super-fine granularity and let the dependency
 * % analysis take care of efficiency.
 */
index_use: expr_list
         ;
expr_list: expr_list '[' exprs ']'
         | expr_list '[' exprs '|' expr_cond_a ']'
         | expr_list '[' ']'
         | age_list
         ;

age_list: age_list '(' age_expr ')'
        | age_list '.' AGE '(' age_expr ')'
        | /* empty */
        ;

age_expr: age_expr '*' age_expr
        | age_expr '/' age_expr
        | age_expr '%' age_expr
        | age_expr '+' age_expr
        | age_expr '-' age_expr
        | INTEGER
        | '-' INTEGER %prec UNARY_MINUS_PRECEDENCE
        | '+' INTEGER %prec UNARY_MINUS_PRECEDENCE
        | LAST
        | index_name
        ;
```

```
exprs: exprs ',' expr
     | expr
     ;

/** Note: Originally I used BNF for explicit precedence.
 *  Changed to bison extensions %right and %left to
 *  declare precedence to make the code more compact.
 */
expr: expr DOTDOT expr
    | expr ':' INTEGER
    | expr '*' expr
    | expr '/' expr
    | expr '%' expr
    | expr '+' expr
    | expr '-' expr
    | '(' expr ')'
    | INTEGER
    | '-' INTEGER %prec UNARY_MINUS_PRECEDENCE /* %prec overrides
       precedence of %left */
    | '+' INTEGER %prec UNARY_MINUS_PRECEDENCE
    | index_name
    ;

expr_cond_a: expr_cond_a ',' expr_cond_b
           | expr_cond_b
           ;
expr_cond_b: expr expr_cond_cmp expr
           ;
expr_cond_cmp: '<'
             | '>'
             | EQ
             | LE
             | GE
             ;

timer_expr: timer_expr '*' timer_expr
          | timer_expr '/' timer_expr
          | timer_expr '%' timer_expr
          | timer_expr '+' timer_expr
          | timer_expr '-' timer_expr
          | INTEGER timer_unit
          | '-' INTEGER timer_unit %prec UNARY_MINUS_PRECEDENCE
          | '+' INTEGER timer_unit %prec UNARY_MINUS_PRECEDENCE
          | NOW
          | use_timer_name age_list
            ;
timer_unit: HOUR
          | MIN
          | SEC
          | MS
          | US
          ;
```

```
/**
 * % Kernel names have their own namespace.
 */
kernel_name: IDENTIFIER
           ;

define_global_field_name: field_name
                        ;

define_local_field_name: field_name
                       ;

use_field_name: field_name
              ;

define_timer_name: IDENTIFIER
                 ;
use_timer_name: IDENTIFIER
              ;

/**
 * % Field names have their own namespace.
 */
field_name: IDENTIFIER
          ;

/**
 * % The only meaning of an index name is as a placeholder
 * % for one dimension needed for accessing a cell in a field.
 * % It is similar to a variable in a C for-loop, but the
 * % body can't change it. An index name can be used for
 * % computations in several dimensions (e.g. required for
 * % transposing matrices). Using a name index in a fetch
 * % operation is similar to the creation of an implicit
 * % foreach loop. One kernel should be started for every
 * % possible combination of indices. It is impossible to
 * % prevent this from happening without programming a special
 * % kernel that cuts a sub-field out of a larger field.
 * % It is expected that the evaluation of the dependency
 * % graph makes it possible to transform such mapping
 * % operations into no-ops.
 * %
 * % Index names have their own namespace.
 */
index_name: IDENTIFIER
          ;
```