UNIVERSITY OF OSLO
Department of Informatics

Parallel Multimedia Algorithms in P2G

Master's Thesis

Preben N. Olsen

Parallel Multimedia Algorithms in P2G

Preben N. Olsen

Abstract

Multimedia applications have a history of driving the demand for computational resources. In entertainment and digital arts, computer animation is popular, but also compute-intensive.

We see that there is an increasing demand on multimedia processing, yet there is a lack of support for complex multimedia workloads with real-time requirements in current parallel and distributed execution frameworks. As a result of this observation, the P2G framework was initiated.

With inherent support for multimedia, parallel, distributed, and architecture-independent processing, it stands out as unique. However, to verify its programming model, we want to investigate how it is to implement workloads in P2G and its Kernel Language.

In this thesis we evaluate and verify its programming model and investigate how it is to implement multimedia algorithms and expose parallel execution using the P2G framework.

Acknowledgements

I would like to thank my supervisors Håvard Espeland and Pål Halvorsen for their guidance throughout this work. I would also like thank the rest of Team P2G, especially Paul B. Beskow, Carsten Griwod, Håkon K. Stensland, without their initial work, feedback, discussions this would not have been possible.

I would also like to thank everyone at the NDLab for a great conversations, laughs and flame wars.

At last, I would like to thank all of my family and friends for their unconditional support.

Preben N. Olsen,
Oslo, May 23. 2011

Contents

Chapter 1

Introduction

1.1 Background and Motivation

Multimedia Workloads

Multimedia applications have a history of driving the demand for computational resources. In entertainment and digital arts, computer animation is popular, but also compute-intensive. Animated 3D movies and 3D gaming drives even a greater expectation and need for performance. The TV industry's recent focus on 3D is another paradigm that needs more computations on top of traditional video processing. In other domains like scientific applications and medicine, numerous live visual representations of complex models help researchers draw more informed and better conclusions. Many of these compute-intensive visualizations can only be achieved by using parallel systems to accelerate the algorithms.

As development of multimedia services progresses and becomes an integrated part of consumers' life, new and more intelligent solutions emerge. At the same time as multimedia services become more complex, perhaps with on-demand and interactive real-time capabilities, there is a constant push for higher media quality, e.g., High-Definition (HD) video. All together, this put a strain on computational resources when processing or producing multimedia content, which again leads to a demand for more efficient use of such resources.

Since its early beginnings, processor development has made tremendous advancements and current hardware provides the needed resources. Graphical Processing Units (GPUs), multicore, heterogeneous Central Processing Unit (CPU) architectures, and Digital Signal Processors (DSPs) are available for speeding up multimedia workloads.

Architecture Changes

In the mid-sixties, Gordon Moore predicted that the number of transistors on a computer chip would double every other year, a prediction of exponential growth that became Moore's Law. Moore's Law has proved to be very close to the evolution we have witnessed, and is due to a highly competitive market where the CPU industry is in constant need of developing new products and technologies for continuous economic growth.

Until recently, processors manufacturers used the doubling of transistors to manufacture new CPUs with increasing clock frequencies and additional logic, like instruction level parallelism, pipelining and branch prediction. These processor enhancements gave the processors better performance and computing power as more CPU instructions and operations could be executed within the same time frame. Software developers used these performance gains to create more sophisticated solutions without thinking too much about performance issues.

Since then, chip manufacturers discontinued its race for ever increasing clock frequencies and are now focusing on adding multiple symmetric cores on one CPU as another way to increase computing performance. This approach is a natural extension of the traditional Symmetric Multiprocessing (SMP) model. Before multicore architectures, SMP had been enabled by interconnecting multiple unicore CPUs together within one computer. Today, multicore CPUs are common in every new workstation or laptop and are starting to appear in smart phones[1].

Alongside this evolution, the GPU was introduced. The GPU is a specialized coprocessor that is optimized to do common graphic operations to help offload the CPU. Because such graphic operations are inherently parallel, GPUs have evolved to include a great number of cores, e.g., Nvidia's Fermi[2] architecture includes 512 cores. GPUs increase performance by sacrificing generality. However, even though the GPU was intended to be a specialized coprocessors, general-purpose computations found their way onto the it and are now accelerating problems in other domains, we mentioned scientific computing, but also statistics. In addition to symmetric multicore CPUs and GPUs, other asymmetric architectures have been introduced and help accelerate multimedia workloads, we look into these in the next chapter.

Distributed Computing

When there is not enough computing power in one single computer, connecting several computers together and utilizing their collective computing power can yield great performance gains. Workloads working with large data sets or highly compute-intensive tasks can be distributed on large clusters or server farms.

[1]LG's Optimus 2X includes Nvidia's Tegra 2 with a dual-core ARM CPU

[2]"Nvidia's Next Generation CUDA Compute Architecture: Fermi" (Nvidia White Paper, 2009)

Distributed High-Performance Computing (HPC) often have a rigid setup and focus on batch processing. The batch mentality is also common when processing large data sets in a distributed manner, as done with Google's MapReduce [1]. Distributing workloads across the Internet, utilizing large remote server farms, is referred to as cloud computing. Another subset of distributed computing is *grid computing*, it has a less rigid design and are often associate with utilization of heterogeneous architectures.

Even though there are hardware resources and distributed techniques available, writing software for parallel and distributed workloads is not trivial. This development model is very different from the traditional where single threaded applications are loaded by the Operating System (OS) and executed on a local CPU.

Frameworks

Parallel and distributed programming has been receiving a lot of attention recent years and several programming language extension, Application Programming Interfaces (APIs), frameworks and libraries have been proposed and provided to ease development of parallel applications on specialized architectures and distributed systems.

Developing on special parallel hardware requires special, in-depth knowledge of the specific architecture. Acquiring such knowledge is time consuming, and architecture specific code makes the workload incompatible for other architectures. Writing code for distributed computing is also challenging, and explicit distribution of code and data across networks or other databuses creates an additional burden for the parallel developer. To better assist developers and increase productivity of parallel and distributed software development a number of approaches have been developed.

Perhaps the most prominent framework for general-purpose distributed computing today is the mentioned MapReduce [1], which can utilize server farms for processing large data sets. Microsoft Research's Dryad [2] project addresses the troubles of writing efficient parallel and distributed applications that can utilize thousands of general-purpose servers, but also future multicore processors. On heterogeneous architectures, the OpenCL [3] standard has received major industry backing for its effort to unify programming of parallel hardware. Nvidia's Compute Unified Device Architecture (CUDA) [4] framework has been widely adopted for General Purpose computation on Graphics Processing Units (GPGPU).

Parallel and distributed computing provide processing power to do heavy computations, but parallel development can be a daunting task. The mentioned frameworks provide means of easing development on heterogeneous architectures and of distributed execution, but none do both. The distributed execution systems also lack key features for processing real-time multimedia workloads, such as the ability to model loops. The P2G [5] framework provides the tools needed as it lets developers easily express parallel code. It is an ongoing research project being developed at the University of Oslo and only a prototype implementation is available. Today, P2G provides a runtime for parallel execution on multicore CPUs while a distributed execution system is being developed. To express parallelism and data transformations, P2G includes a Kernel Language, which supports programming constructs needed by real-time multimedia workloads. Future implementations of P2G aims at using special heterogeneous architectures for an even greater performance increase.

1.2 Problem Statement

With increasingly compute-intensive multimedia algorithms and little support for complex streaming multimedia workloads in traditional parallel and distributed frameworks, the P2G framework was created. With inherent support for multimedia, parallel, distributed, and architecture-independent processing, it stands out as unique. However, to verify its programming model, we want to investigate how it is to implement such workloads in P2G and its Kernel Language.

In this thesis, we present multimedia processing and how some of these workloads could benefit from timeliness in computation, but also how multimedia streams are different from batch processing. We explain and discuss different existing approaches to parallel programming, what are are the underlying concepts and techniques in parallel and distributed programming, and list some different tools are available (chapter 2). Then, we explore the P2G framework outlining some of P2G's important features and explain how a developer writes parallel software in P2G's Kernel Language (chapter 3).

As mentioned, the P2G framework is an ongoing research project and work-in-progress, our main task is to look at P2G and evaluate and validate some of P2G's ideas so far and make suggestions for potential improvements, with respect to workload development. Examples of such is P2G's ability to model both task and data parallelism and its approach for implementing workloads, the current implementation of the framework, and the expressive power of its Kernel Language, to make sure that we are able to develop multimedia algorithms in P2G.

To achieve a realistic evaluation of P2G and Kernel Language, we try to adapt existing code for Motion JPEG (MJPEG) video encoding and implement Scale-Invariant Feature Transform (SIFT) [6] feature extraction from scratch, with this framework. These are algorithms that represents a subset of well-known known and compute-intensive workloads that the finished P2G framework should be able to support. The experiences we get while using the current prototype implementation of P2G are discussed (chapter 6), pointing out what is good about the P2G approach, but also see if there is any room for improvements in order to assist decision making of the moving target of P2G.

1.3 Contributions

We have created a parallel design for both MJPEG and SIFT using P2G's Kernel Language and its way of expressing parallelism through data dependencies and data transformation. The MJPEG workload's parallel design and P2G implementation have been tested and benchmarked, and have also contributed to a paper which the author of this thesis co-authored [5]. The paper is pending review. Additionally, these ideas and results have been presented and discussed at Eurosys as a poster [7]. The results show that inherent parallelism in MJPEG is possible to exploit using P2G, and that it scales this workload and achieves an speed-up when executing in parallel. Furthermore, the design and partial implementation of a parallel execution of SIFT feature extraction, i.e., a larger, more complicated workload, are provided.

During our work, we got a lot of experience with P2G and Kernel Language, we believe the programming model is simplistic, yet powerful. Kernel Language is good at taking advantage of our algorithms parallelism, its data structures and concepts are familiar and easy to grasp. We did identify some improvement ideas, which are discussed and also added some proposed solutions and additions to the framework.

1.4 Outline

In chapter 2, we give a background and more rigorously explain the motivation for our work. Chapter 3 discusses the P2G framework and its way of expressing parallel workloads in detail, while we in chapter 4 provide the design and implementation of our Motion JPEG video encoding workload using the P2G framework. In chapter 5, a P2G design and implementation of a parallel SIFT algorithm is presented. A discussion of our experiences are found in chapter 6, while chapter 7 summarizes and concludes this thesis.

Chapter 2

Background and Motivation

In Chapter 1, we gave an introduction, explained our initial motivation and problem statement. In this chapter, we introduce needed functionality in languages and frameworks used for real-time multimedia processing. We also provide a much broader overview of the distributed and parallel computing landscape. First, we introduce different types of parallelism and then give the reader insight into established principles and challenges of parallel programming. Then, we look at different types of parallel hardware and architectures, before we discuss different approaches and programming models used for parallel programming. At the end of this chapter, we briefly summarize the tools, highlighting their differences and similarities while also point out essential features.

2.1 Multimedia Processing

Multimedia is a broad field, ranging from interactive computer games to video, to text, and as multimedia content is arbitrary different, so are data types and algorithms. This means that any framework used to write multimedia algorithms must support the same data and programming constructs as any other language normally used for processing multimedia content, i.e., intrinsic data types and sizes, multi-dimensional and custom, user specified data structures, and control flow operations for looping and branching. However, as we list and discuss in the next subsections, there are also other properties of multimedia processing that is not inherently supported by common programming languages.

2.1.1 Real-Time

The use of timeliness in computation and *real-time* execution is important for certain applications. Manacher [8] introduced *timing constraints* for computer systems that are interacting with the external, physical environment. The article presents two basic constraints, *start-times* and *deadlines*. A task conforming with the former should not start executing before the start-time is passed, and the tasks conforming with the latter should not execute after a specified time limit.

Examples of applications where real-time constraits are inherent include industrual robots, and flight and vehicle control, but the time perspective is also previalent in multimedia applications, for example in live video or interactive computer games. In such applications, data sources, e.g., disks and network, and computational resources put restrictions on what result can be provided within a specific time frame. The physical environment for multimedia applications are the consumers of multimedia content, which expects a steady flow of the highest quality.

To provide a better understanding of timeliness in multimedia processing, consider the following scenario of video encoding. A developer wants to provide the best media quality possible for its viewers. However, in live streaming scenarios there are given time limits; one is supposed to produce a number of frames per second. Whenever the computation of a frame exceeds this time limit, the frame is either delayed or dropped, the encoding complexity can be decreased reducing image quality, but in either way, this leads to a poorer experience for the viewers. When encoding the video, one could have two parallel executions perform motion estimation. One execution would perform high-quality estimation providing best quality and experience, but this estimaion is very time consuming. Another indepedent, parallel execution could execute a subset of the former execution's operations and thus be faster. By setting a deadline on the high-quality encoding, the enconding application could fall back on the lower quality execution when the deadline is reached and provide the lower quality video frame. In this scenario, there must be enough computational resources available for both executions, i.e., the low-quality execution does not compete with the high-quality execution over CPU time.

Deadlines can also be useful in scenarios where multimedia streams adapt to available network throughput; a delayed frame can trigger a deadline and handle this event in a graceful manner. Varying network throughput and link quality also demands workloads that are capable of handling significant changes. Frameworks should be flexible enough to reflect and handle such changes.

2.1.2 Stream Programming

Another aspect of multimedia processing is its relation to stream programming. Stream programming is a programming abstraction that helps developers write software that incorporates the notion of streams. William Thies et al. [9] indetify six properties of a streaming application:

- Large streams of data
- Independent stream filters
- A stable computation pattern
- Occasional modification of stream structure
- Occasional out-of-stream communication
- High performance expectations

In multimedia processing, these large data streams can be audio or video flowing through several transformation filters to produce some desired result. *Filter* represents the code segments in a streaming application that transforms the streams of data, these filters communicate with other filters through input and output channels, e.g., First-In-Fist-Out (FIFO) queues.

William Thies et al. [9] also state that traditional programming languages like C or C++ do not provide adequate support for stream programming, and are among other things pointing out that they do not have an intutive way of representing streams or their parallelism and communication patterns. In response to this, the authors propose a new stream language called StreamIT [9], which is more flexible, feature rich and efficient than other streaming languages at that time. As multimedia applications often work on streams of data that is transformed in different and sometimes parallel steps, stream programming is an abstraction that would benefit frameworks for multimedia processing.

2.1.3 Summary

We have argued that multimedia workloads are arbitrary complex, meaning that frameworks and languages used for processing multimedia must support regular general-purpose programming constructs, data types and sizes. In addition to this, it can be beneficial to include support for timeliness as real-time constraints are inherent in different multimedia scenarios. At last, we discussed the nature of stream programming and lack of expressive power in regular programming languages to support this paradigm communication and parallelism. In the next section, we take a look at parallelism in algorithms, identifying the different types of parallelism that should be supported in frameworks or languages processing streaming algorithms.

2.2 Parallelism in Algorithms

An algorithm often consists of different steps to get the desired final result. In a fully *serial* or *sequential* algorithm, every subsequent step is dependent on the current, i.e., the output produced in one step is the input to the next. In contrast, a fully parallel algorithm have no steps that are dependent on any other step. Between these two extremes, there are algorithms where parts of the algorithm is sequential and the other parts are parallel, which also is the case for larger multimedia workloads in general.

What is important to have in mind at this point is that the expected speedup of an partly parallel algorithm is proportionate to its sequential parts, i.e., an algorithm can never be executed faster by using parallelization techniques than its sequential parts as stated by Amdahl [10]. When writing parallel algorithms, one tend to differentiate between *task parallelism*, *data parallelism*. Some also include *pipeline parallelism* [11].

2.2.1 Data Parallelism

Data parallelism is achieved with algorithms that allow its data set to easily be divided into a subset of noncontiguous, discrete values that can be processed independently of each other. The result of each computation are joined together at the end of execution. When this type of execution shares no state, it does not need locking, but it might need synchronization when every data element is joined together. In data parallelism, one task is instantiated multiple times, with each instance working with a subset of the larger data set. The algorithm is characterized as *embarrassingly parallel* if the algorithm scales with every extra data element executed in parallel, and the data set is easily partitioned. In figure 2.1, we represent data parallelism as five equal tasks doing the same operations. Each instance of "Task B" does $\frac{1}{5}$ of the work, and with five instances we see that the time of processing it is reduced.

2.2.2 Task Parallelism

Task parallelism is achieved with algorithms that allow processing of multiple tasks independently of each other. By separating each task and assigning it into a separate processes or thread, it can be executed in parallel without interfering with any other thread. An algorithm usually has a predefined number of tasks, and embarrassingly task-parallel algorithm should scale with each task that is executed in parallel, e.g., a web-server handling incoming requests by dispatching threads from a thread pool, with each thread managing a disjoint and independent set of requests.

In some scenarios, parts of an algorithm is task parallel while other parts are data parallel. To get most out of parallel execution, the developer needs to optimize for both these type of parallelisms. To complicate matters even worse, parts of an algorithm that are executed as a separate task can also benefit from data parallelism. In figure 2.1, we

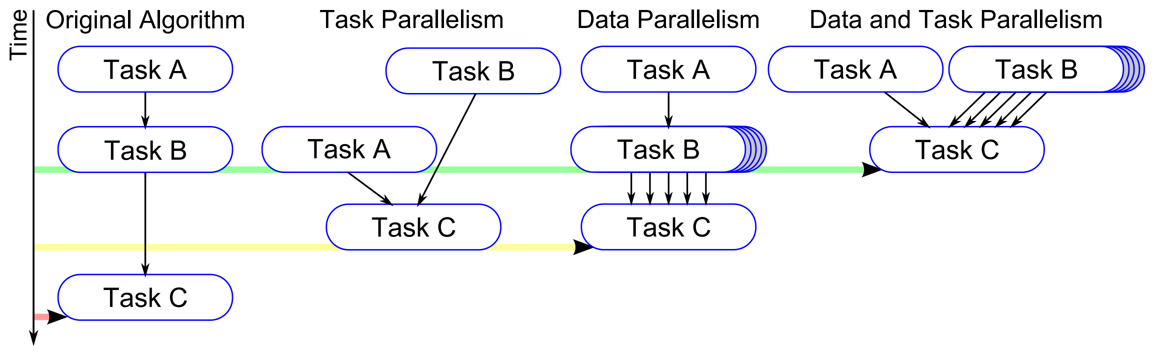

Figure 2.1: Task and Data Parallelism

visualize how using both data parallelism together with task parallelism we end up with the lowest executuion time.

2.2.3 Pipeline Parallelism

Pipeline parallelism *at the software level* is achived by decomposing repeating sequential operations, e.g., operations within a loop. Operations that are, *(A)* independent of each other within the loop, or *(B)* indepedent of finishing the loop, can be isolated and divided into different *stages* and executed in parallel.

In other words, given a loop that iterates n number of times, and where computations in iteration i needs input from previous iteration $i-1$. In this scenario, a data parallel approach does not make sense as a loop is needed, but if stages independent of each other is identified, these can be executed in parallel by assigning input values from $i-1$. A stage is similar to a task, which also can be data parallel. One distinguishes between fine-grained pipelines on instruction level [12] and coarse-grained pipelines on code block level [13].

2.2.4 Summary

As depicted in figure 2.1, algorithms that allow parallel execution can be executed a lot faster if the parallelism (see the green line) is exploited. In this section, we went through and described different types of parallelism, but to take advantage of these types and develop parallel applications, developers need some way of expressing parallelism. Common imperative languages, like C, C++, or Java, have an inherent sequential execution model where the global program state can be altered in any step of execution. To enable sharing of global state in imperative languages, different techniques are applied. The next section explore what abstract techniques are developed for parallel execution.

2.3 Concurrency

Developers who want to gain performance benefits from parallel execution models are often using methods and techniques from concurrent programming. While parallelism is about executing an algorithm faster or more efficiently, concurrent programming provides the abstractions to structure parallel algorithms. In concurrent programming, the usage of threads is prominent. There are different implementations of threads, depending on the type of hardware and execution environment (OS or runtime). We use the term thread for a single executing entity that executes independently from other executing entities, which it shares the same address space and memory. This is contrary to two processes that do not share any memory and arrange Inter-process Communication (IPC) differently. Two parallel processes do not need to be executed on the same computer.

Given the same input, if two or more threads have the opportunity to arbitrary read from and write to the shared data element(s) and if the time they wish to do so, i.e., how they are scheduled, determines the output, the result is unpredictable. This is called an indeterministic dataflow. The places were these reads and writes happen are often called critical sections, and data in these critical sections alter the global state of the execution. This phenomenon is commonly refered to as a race condition. Race conditions are sometimes hard to notice, and debugging them is often a challenging task.

2.3.1 Mutual Exclusion

Mutual exclusion on shared data element(s) can be used to eliminate race conditions by ensuring atomic writes. Mutual exclusion can be implemented as a lock. When a thread wants to change a shared data value, it takes the lock associtated with it, then when another thread wants to change the same value, it tries to take the same lock, but instead of recieving it, the thread is put on a queue waiting for it. The thread waiting is not able to change or read the value until the first instance is done altering and has released the lock. Locking is an intuitive way of solving race conditions, but locking is difficult and has been the source of many frustrations, e.g., process starvation due to deadlocks. There are other implementations of mutual exclusion, including reader-writer locks, semaphores and monitors [14].

In scenarios where the shared data structure only allow coarse granularity of mutual exclusion, it becomes a bottleneck as the number of threads depending on the data scale. Fine granularity helps to scale the application that rely on shared data, but there is a trade-off between locks with fine and coarse granularity as the cost of lock administration must not be greater than the cost of waiting to get the lock. In such cases, spin locks (buys-wait) are more favorable [14]. Lock contention can also be prevented by applying lock-free data structures, e.g., Read-Copy Update (RCU) [15] available in the Linux kernel and implemented with CPU instructions for atomic and conditional memory writes, like x86's `CMPXCHG`[1]. Atomic operations are also possible to achieve with Transactional Memory (TM), both in hardware [16] and software [17].

2.3.2 Explicit Synchronization

As parallel threads or processes execute independently from others, a developer can never assume that the parallel executions are synchronous, even when they are executing the exact same operations. The time that one thread has finished its operations must be assumed to be different from other threads. When the algorithm itself is asynchronous, this is not a problem, but for algorithms that demands synchronous execution it is. For example, in a task parallel algorithm, a thread executing one task might start when another independent thread is finished. To synchronize threads, different techniques have been developed and among them are barriers, condition variables and joining of threads on termination. Both data sharing and synchronization primitives are used for IPC.

2.3.3 Message Passing

Another form of IPC used to control access to shared resources, distribute workloads and results, or synchronize threads and processes in concurrent programming is message-passing. With message-passing, data do not have to be shared between different parallel executions, instead, they send messages. The messages are sent to the threads or processes that control resources or data expressing their needed operations. The executions controlling shared resources then handles incoming requests, making sure that the execution is deterministic. Message-passing to resource controlling threads is analogous to the waiter solution applied to the famous problem of dining philosophers.

Message-passing is used for communication within one computer, but also in distributed environments as it is network transparent. To enable message-passing, there has to be communication channels present in between threads. For intra-computer communication, pipes and mailboxes are examples of such communication channels, and for network communication, Berkeley sockets can be used. When communicating over networks, monitoring and assessing performance is important as the overhead of moving data and processing on other machines must not be greater than processing it locally. Message-passing has been used for a long time in OSes with a microkernel design, e.g., Minix [14], but has lately experienced an upswing and is yet explored in OS research [18] [19], with perhaps Barrelfish [20] being the most prominent example.

[1]"Intel 64 and IA-32 Architectures Software Developer's Manual (2A/2B)", (Intel Manual, 2011)

2.3.4 Summary

The shared resources in a multi-threaded application defines its global, shared state. Sharing a global state might not scale with a huge number of threads, and is perhaps especially troublesome when disseminating such state in network distributed environments. Network communication tends to have higher latency and is thus often slower than internal data buses. When using networks, monitoring performance is important as the overhead of moving data and processing on different machines must not be greater than processing it locally.

In this section, we reviewed the high-level approaches to deal with concurrent executions, and concurrent programming done right ensures data integrity when handling indeterministic dataflows and has for long been the de-facto standard of how to write parallel software. There are developed several libraries that implements the principles of concurrent programming for accelerating development of multi-threaded software, e.g., POSIX Threads, but to be able to execute in parallel, parallel hardware must be provided.

2.4 Parallel Hardware

With algorithms that allow parallel execution and ways of expressing parallel programming, the only thing missing is processors and architectures that enable parallel processing. Michael J. Flynn proposed [21] a classification of CPU architectures. There were four classifications, three of which enables parallel processing.

Single Instruction – Single Data (SISD): A SISD architecture is a scalar unicore processor executing one instruction on a register with only one data element enabling no parallelism at all.

Single Instruction – Multiple Data (SIMD): A SIMD architecture enable execution of one instruction on several data elements. This can be achieved with only one core working on a register packed with multiple values, e.g., the SSE extension to Intel's x86 architecture.

Multiple Instructions – Single Data (MISD): A MISD architecture do different instructions on one data element. This architecture is very uncommon, but a multicore processor can mimic such behaviour.

Multiple Instructions – Multiple Data (MIMD): A MIMD architecture is able to execute multiple (different) instructions on multiple data elements. A multicore processor is a true MIMD architecture.

As we see, a SIMD architecture maps perfectly with data parallelism, while task parallelism benefits from MIMD architectures. Note that a multicore processor can execute the same instructions on different cores as well. Thus, even though a multicore architecture actually is a MIMD architecture, it can mimic a SIMD architecture as multiple cores can execute the same instructions on multiple data. Such processing is often referred to as Single Process – Multiple Data (SPMD). A developer writes such solutions by creating multiple threads that do the exact same operations and having one instance executing on each core with data divided in between them. This way, multicore CPUs can scale data parallel algorithms.

2.4.1 Multicore Processors

One of the first multicore processor for general-purpose use was IBM's POWER4 [22], It was released in October, 2001. One configuration of this processor had two cores, but it also came in a second configuration with four cores. In 2005, Intel and AMD followed with their first multicore processors. Intel's Pentium D was released in April and the Athlon 64 X2 in May. Both the Pentium D and Athlon 64 X2 were dualcore processors. This was the start of new era in processor development, and most new processors now come in multicore configurations. Today, some high-end desktop processor implement six and eight cores, while AMD offers a 12 core Opteron[2]. These multicore architectures are symmetric and homogenous, where every processing core is equal and has the exact same properties.

Different cores in a symmentric multicore processors also usually have equal access to the processors memory space. This means that two threads that share paged virtual memory can access the same memory positions. This is analogous the shared data elements or resources discussed in section 2.3. When these two threads execute in parallel on two different cores and share same memory positions, some of these addresses might be cached for faster access. As some processor caches are functional units local to each core, it means that a processor altering some memory, only does so in their own local cache, and thus outdating the value in the other core's cache. To ensure that this does not happen, cache coherence protocols have been developed.

When a processor has a coherent cache, altered cache lines on one core must somehow invalidate other cache lines containing same memory address on other cores. Cache coherence yields consistency, but cache coherent protocols introuduce latency. In [23], a performance test of Intel's Nehalem multicore architecture with respect to memory access and cache coherence was conducted. It showes that the latency introduced with cache coherence when requesting cached data that had been accessed on by another processing core on the same CPU or on another interconnected discrete CPU. As one would imagine, ensuring coherent cache on different CPUs yields largest latency.

A second solution to concurrent programming is message-passing, as noted in section 2.3. In April 2010, Intel announced the SCC (Single-Chip Cloud Computer)[3]. The SCC is a 48-core processor, but Intel states the chip and its technology is able to scale

[2]"The AMD Opteron 6000 Series Platform" (AMD White Paper, 2010)
[3]"Introducing the Singlechip Cloud Computer" (Intel White Paper, 2010)

up beyond 100 cores. The CPU is based on a P54C processing core, used in the successor of the original Pentium processor. The P54C is smaller and less complex than cores in today's multicore processors, which is on par with [24] proposing many smaller cores than the one used in current multicore processors for a more affordable power envelope.

In regard to message passing, perhaps of special interest are the new functional units on each core and extensions to its ISA. In addition to the traditional cache, each processing core has a new LMB (Local Memory Buffer), MIU (Mesh Interface Unit), LUT (Lookup Table), and a router. Instead of sharing data by having a coherent cache, the LMB (also called MPB for Message Passing Buffer), is intended to be used for message passing between the cores as an alternative way of communicating program state, shared resources or other IPC. This message passing approach is very much in sync with message passing in scalable concurrent programming [20], and it has been shown that the MPB enables low latency message passing [25]. The SCC is a research microprocessor that hopes to aid research on parallel software and is not a commercial product.

The multicore trend is most likely going to continue and provide us with massively parallel many-core processors, and it is therefore important to establish good frameworks and best practices for expressing and executing parallel multimedia workloads so that we can benefit from such advances.

2.4.2 Heterogeneous Architectures

In heterogeneous architectures, processors with different capabilities are joined together. These architectures often have general-purpose capabilities complemented by an additional co-processor with special-purpose capabilities for accelerating specific tasks, e.g., GPUs for processing 3D graphics, DSPs for accelerated functions in the frequency domain or just a Floating-Point Unit (FPU) for doing floating point operation.

In contrast to homogeneous architectures, heterogeneous architectures often do not share address spaces, which results in explicit data transfers from the general-purpose environment and back. Data transfers from one address space to another introduce latency. This means that the CPU execution time must be greater than the time it takes to move input data, execute and move computed results back.

The use of special-purpose processing core(s) for accelerating multimedia give great performance gains, but as we discuss below, the development and execution model are both a bit different for evey architecture. Requiring special knowlegde of the architecture and its programming model creates a natural barrier for adopting the architecture.

GPU: Graphic Processing Units

As we mentioned in section 1.1, GPUs have become an important tool for offloading the CPU in processing graphics. Graphic algorithms are often embarrassingly parallel and are able to scale with a great number of threads executed in parallel, and because of this, GPUs often have many parallel cores. Because the purpose of the GPU is to process graphics, the GPU has a special-purpose design in contrast to the CPU's general-purpose design. GPUs are used for producing multimedia content and support the commonly used datatypes in such algorithms.

GPUs have also been proven to be effective at solving other task and because of this, GPGPU (General-Purpose computation on Graphics Processing Units) have become popular in several computational fields, e.g., in scientific computing. Even though some new CPUs[4] and System-on-Chip (SoC) solutions[5] have a GPU on die, GPUs are often mounted on a discrete card connected to the CPU, for example via a PCI Express bus. Utilizing GPUs to solve general data parallel problems can give great performance gains, but programming GPUs efficiently requires indepth knowledge of the specific architecture [26].

There a two prominent ways of programming GPUs, Nvidia's propriertary approach with CUDA which is discussed in section 2.6.3, and the open and unified approach using OpenCL which is reviewed in section 2.6.2.

Asymmetric Processors

Asymmetric processors have a heterogeneous processing cores implemented on one discrete chip. An example of an asymmetric processor architecture is the Cell Broadband Engine Architecture (Cell BEA) [27] from Sony Computer Entertainment, Toshiba, and IBM. A Cell CPU includes one or more PowerPC Processor Elements (PPE) and one or more Synergistic Processor Elements (SPE).

In case of the Cell BEA, the PPE is a 64-bit processor with a Power ISA for general-purpose processing, and for managing the system and SPEs. An SPE includes a Synergistic Processor Unit (SPU), which has its own ISA with special-purpose vector registers and instructions that are able to do SIMD operations. The SPE has 128 registers which all are 128 bits, this enables operations to be executed on sixteen 8-bit, eight 16-bit, four 32-bit, and two 64-bit values. The SPU has no access to the main memory or to PPE CPU features, and as mentioned, data transfers must be explicitly expressed by the developer. The most famous implementation of the Cell Broadband Engine Architecture is perhaps the processor shipped with Sony's PlayStation 3, usually referred to as the Cell Broadband Engine (Cell BE).

CPUs with asymmentric cores that do specialized operations have been more frequent in embedded applications, like DSPs in Texas Instruments' OMAP3 series or the men-

[4]"AMD Fusion Family of APUs" (AMD White Paper, 2010)
[5]"The Benefits of Multiple CPU Cores in Mobile Devices", (Nvidia White Paper, 2010)

tioned SIMD processors in game consoles, but AMD's Fusion and Intel's Sandy Bridge now bring asymmentric architectures to laptops and desktop computers.

2.4.3 Summary

In this section, we have had a look at parallel hardware, taking a short look at the history and providing examples of the different types of processors available today. Heterogeneous architectures might provide major speedups for certain workloads, but there is a trade-off between specialized high-performance asymmetric and heterogeneous processing, and general-purpose symmetric processing. Asymmetric specialized coprocessors yields a greater performance increase than symmetric given algorithms with different fraction of parallelization potential [28]. However, specialized architectures often result in different programming models, requiring indepth knowledge and most likely unportable source code. General-purpose processors without specialized instructions provides a more unified way of programming and most likely more portable code. In some scenarios, a unified programming model and portable code is more desirable then the acceleration of specialized hardware.

The aforementioned rise of multicore processors, GPGPUs and distributed systems provides us with computational resources, but it has also created a demand for ways of expressing parallel and distributed workloads in an easy manner. The mention abstract techniques for concurrent and parallel programming needs to be implemented to provide tools for developers of parallel workloads.

2.5 Low-Level Concurrent Primitives

At this point, we have discussed, algorithms, parallelism, and parallel hardware, and all of this theory finally comes together when developers finally set out to develop parallel systems. To be able to efficiently develop parallel programs, there are ready-to-use implementations of low-level concurrent primitives a developer can reap benefits from.

2.5.1 POSIX Threads

The POSIX Threads[6] (Pthreads) is a POSIX standard for handling concurrent thread programming. The standard specifies a set of header files accompanied by an API, implemented as a system library, to use when doing thread development in C. The implementations of Pthreads are very different, some have OS kernel support while others are user threads mulitplexed on top of an OS process, with all the implication for portability this may bring.

[6]Linux Programmer Manual: "pthreads" man-page

Developers using Pthreads benefit from functions to create new threads and terminate them. There is also data structures to create different types of locks including read-write locks, spinlocks, and barriers. Further, it specifies other synchronization primitives like condition variables and thread-join at the end of execution.

Writing parallel software with Pthreads lets developers exploit both data parallelism and task parallelism. Pthreads is very expressive and gives the developer detailed control of thread execution. Explicit low-level control of thread execution provides flexibility and is beneficial in some scenarios, but it can also be the source of potential erroneous behaviour and unexpected result, as mentioned earlier, locking is difficult. Pthreads is written in C and is therefore supported on multiple architectures. Its API is sometimes wrapped in other libraries for supporting concurrent programming, e.g., Boost C++ Library[7].

2.5.2 SIMD Instructions

As mentioned, SIMD processing enables multiple data elements to be manipulated by one instruction, e.g., x86 SSE extensions and Cell BEA's SPUs. As mentioned, data parallelism benefits from SIMD processing. SIMD is often targeted for multimedia applications where processing multiple elements at a time instead of one increases the performance.

In the Cell BEA architecture, the cores are asymmetric. There is a general-purpose core and specialized SIMD cores (SPEs). The SIMD cores do not have access to the main memory, so to execute code on the SPEs, a driver needs to provide execution control and access to input data. On top of the driver, an API implements common operations that is needed for exploiting SIMD functionality. All of these operations are done by the general- purpose processor. In the case of x86 and SSE, the SIMD architecture resides along-side the general-purpose instructions and has access to main memory, which make development easier as execution is handled like any other application.

Development is often done in C, C++ or assembler. SIMD operations can be expressed in highly optimized "hand written" functions in assembler, or by higher level libraries that provide SIMD datatypes and more general API functions. The instruction sets and libraries are closely tied to specific CPU achitectures. Depending on the CPU architecture, task parallelism can be achived by having multiple cores executing SIMD operations.

Even though this type of development is a well known model, the developer needs to aquire special knowlegde about the architecture, which can be time consuming. There is, however, OpenCL drivers for the Cell processor as well as the SSE extensions, which provide a unified programming model for both these architectures.

[7]http://www.boost.org/doc/libs/1_46_1/boost/thread/pthread/

2.5.3 MPI: Message Passing Interface

To be able to write portable parallel workloads that use message passing for communication and sharing state, the Message Passing Interface (MPI) [29] was initiated. MPI provides an API with a standardized semantics and protocol which define its behaviour. Messages in MPI can be passed between two parallel threads in a point-to-point fashion and by broadcasting global messages to a pre-defined group of threads. HPC is one major application for MPI, and several implementation are available, supporting several programming langauges with C and Fortran as the most wideley adopted.

MPI message passing is network transparent, and it enables communication between different computers and workload distribution to multiple hosts. All parallelism in MPI is explicit, it gives the developer control of the execution and dataflow. Thus, data and task parallelism is supported, but it demands good usage of primitives for optimal and correct results. All communication channels and nodes must be defined up front of execution, which makes it a rigid design that are not able to adapt to changes in topology, e.g., in case of network failure. Implementations of MPI can provide error handling in case of runtime errors and functionality for timing of exection and profiling tools for monitoring performance, which is important for understanding bottlenecks [30].

Message passing in itself is hardware transparent, datatypes are specified in the comminucation, making conversion for a architecture transparent for the developer and ensures correct binary representation across different architectures. However, the MPI standard does not specify interoperability between different implementations. A MPI library written for one architecture might not be able to work together with a library for another architecture [29].

2.5.4 Concurrency Primitives in Languages

Several programming languages have also implemented support for concurrent programming in their standard libraries. As seen below, many of the different approaches to concurrent programming are implemented.

Erlang is a functional language said to be designed for concurrency, it includes a minimal set of primitives, which also maps to distributed computing [31]. Erlang threads share no memory and communicate only through message passing, employing the renowned *actor model* for concurrency.

Concurrent Haskell is an extension of the Haskell 98 standard [32], which introduced the notion of processes to Haskell, and means of communicating between processes. This was implemented through the *Control.Concurrent* library. Later, Haskell also got support for Software Transactional Memory (STM) [33], which ensures atomic reads and writes. As mutual exclusion with locking or similar primitives, STM can also experience contention [34].

Java includes a *java.util.concurrent* package that has useful utilities for concurrent development. Among them are, thread creation and termination, thread-safe queues, traditional locks and atomic variables for lock-free access.

C# supports thread control with its *System.Threading* namespace. As Java, it supports different types of locking, thread control for shared memory. Also included are primitives for passing messages between different processes, both local and on different machines.

2.5.5 Summary

The low-level concurrency primitives give a developer a lot of control over the execution, it enables every form of parallelism, but low-level concurrent programming is widely recognized as difficult, prone to erroneous behaviour and time consuming. To combat this, new language extensions have been developed to provide a higher level of concurrent programming, in which some are discussed the next section.

2.6 Language Extensions

The detailed and low-level approach to concurrent programming can be needed in certain scenarios, but it is known to be time consuming and often results in large code bases. Language extensions to C and C++ have been developed to lighten the burden of writing concurrent applications. In this chapter we look at OpenMP, OpenCL and CUDA, which are prominent frameworks for parallel execution.

2.6.1 OpenMP

OpenMP [35] approaches parallel programming differently than for example Pthreads. Instead of a complete list of features, it gives the developer *directives* that instruct the compiler to automatically facilitate parallel execution. OpenMP's goal is to provide a simple model for parallel programming that lets developers extend their already sequential applications that relievs them from explicit concurrent programming. OpenMP provide directives for thread creation, distribution of work to threads, data access, and synchronization.

OpenMP is supported in C/C++ and Fortran. In the latter language, the directive is expressed like a regular code comment, while in the former, special pragmas are used. OpenMP includes a runtime that dispatches and terminates threads, and control the parallel execution. It also does lock management and includes a portable timer measuring execution.

An implementation of OpenMP is found in newer versions GNU's GCC[8]. The developer specifies the *-fopenmp* argument compile-time, which tells GCC to look for OpenMP directives and to link with the OpenMP library.

As with the former framework discussed, OpenMP provides both taks and data parallelism. OpenMP is portable to as many architectures as OpenMP compliant compilers support, and OpenMP directives has also been ported to asymmetric architectures like Cell BE [36]. OpenMP might relieve developers from the burden of concurrent programming, but extensive use of OpenMP directives in an application creates implicit barriers and synchronizations points in between, which introduce overhead and scalability issues and requires profiling [37]. Also, note that shared memory concurrency demands cache-coherent processors.

2.6.2 OpenCL

The Open Computing Language (OpenCL) [3] is an open programming standard initiated by Apple in 2008. This standard provides a uniform development environment for general-purpose parallel programming. OpenCL supports and is portable over heterogeneous architectures, including CPUs and GPUs, and it is intended for large servers, workstations and even handheld devices. The development of OpenCL standard is lead by the non-profit Khronos Group and is heavily backed by the computer industry in general. OpenCL targets general-purpose programming, but the hardware it utilizes are often specialized for graphic processing, e.g., GPUs and DSPs, and therefore support typical multimedia workloads.

An OpenCL application executes its workloads on a *host* which has at least one *compute device*. Each compute device has at least one *compute unit*, and one compute unit concists of one or multiple *processing elements*. To set these abstract terms into a real world example, the host could be one computer or server. This computer could have several compute devices, e.g., two similar CPUs and an additional discrete GPU. The two CPUs are two compute units in one compute device, while the GPU is a different compute device with one compute unit. Both the CPUs and the GPU have multiple cores which corresponds to a processing element.

OpenCL has adopted the *kernel* approach from stream programing. A kernel is a code segment written for one *work-item* or multiple work-items joined together in a *workgroup*. Work-items and workgroups are data sets that enable data parallel execution. Kernels are programmed in the OpenCL C Language and are expressed as regular C-functions prefixed with a special kernel keyword. The language supports most of the ISO C99 and it adds features for expressing OpenCL concepts, e.g., work-items, new vector data types that are endian safe and correctly aligned in memory. There are also built-in functions for manipulating vector types and other commonly used operations.

An OpenCL program includes multiple kernels and other functions, it is executed in a *context*. A context defines what devices are available for execution, it has a *command*

[8]"The GNU OpenMP Implementation", (Free Software Foundation, 2006)

queue and the explicitly allocated memory that kernels read and write from. There are four memory address spaces defined in OpenCL, private, local, global and host memory, and data must be explicitly moved between them.

OpenCL is very expressive and lets the developer specify what kind of hardware it is supposed to execute on. Developing in OpenCL is unified, but because the programming model is unified it is also complicated. Workloads written in OpenCL are also not directly portable to different architectures. To be able to execute an OpenCL application on some specialized hardware, there must be a driver that conform with the OpenCL standard and are able to interface the hardware accordingly. The hardware driver schedules, dispatches threads and controls execution of kernels, and is in such a way analogous a runtime.

The OpenCL standard exploits both data parallelism and task parallelism, but with data being the primary model. Task parallelism is only achieved on certain architectures and is expressed by having instances of a kernel or different kernels executing in one work-group independently. OpenCL abstracts same types of architectures well with its programming model and an unified way of programming is very good thing from a developers perspective, but distinct architectures might do processing significantly different, which again can hurt the performance. The framework does not target distributed computing. AMD/ATI has adopted OpenCL as its way of developing GPGPU applications. Nvidia have an OpenCL driver, but they also have their own proprietary development model CUDA.

2.6.3 CUDA

Compute Unified Device Architecture (CUDA) [4] is a software framework developed by Nvidia to allow developers create GPGPU applications that run on CUDA capable GPUs. The CUDA Software Development Kit (SDK) includes drivers for interfacing hardware, software libraries, and a compiler toolchain to convert high-level languages, e.g., CUDA C, to the PTX virtual ISA. There are several ways of programming Nvidia GPUs with CUDA, it supports development in OpenCL, Nvidia's own CUDA C, but also through other language extensions that provide other means of defining *kernels*[9].

CUDA was a forerunner to OpenCL, and as OpenCL, CUDA C kernels are code segments that are executed in parallel and are expressed by prefixing a C-function with a special keyword. The kernel is then called explitcitly with a syntax for defining how many kernel thread that should be created and executed. CUDA allows a thread hierarchy to be created, a multiple of threads is executed in one *block*, and a one, two, or three-dimensional thread block is a *grid*. The number of threads and blocks that are created depends on the data sets and the architecture its executing on. Data parallel operations use the index values associated with each block and thread. Figure 2.2 shows how indices are assigned to blocks and threads.

[9]"NVIDIA CUDA Programming Guide 3.0", (NVIDIA, 2010)

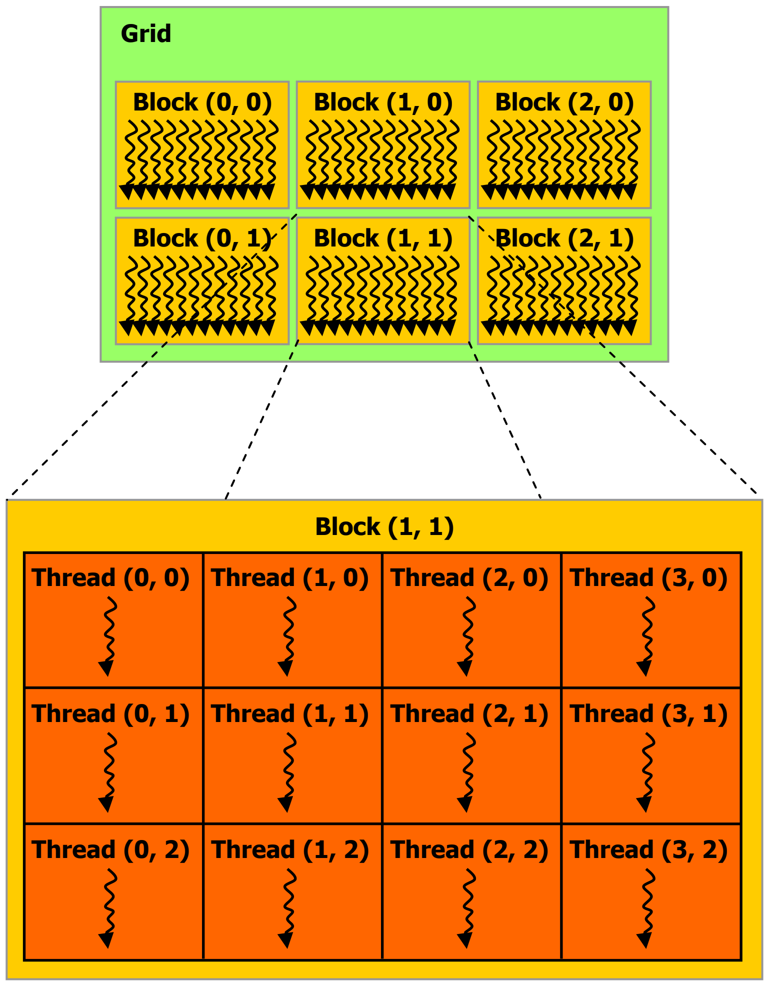

Figure 2.2: CUDA Grid, Blocks and Threads (From CUDA Programming Guide)

In CUDA terminology, the *host* is the computer and its CPU and the *target* is a CUDA capable GPU. The CPU and GPU does not share address space; the two address spaces are referred to as *host memory* and *device memory*, respectively. Memory must be allocated for both address spaces and data explicitly moved between host and target. On new Fermi-based GPUs it is possible to execute concurrent kernels on a target, it does not allow task parallelism, but it does provide more flexibility than previous models.

2.6.4 Summary

OpenMP is a shared-memory API for general-purpose computation for SMP architectures. It expresses parallelism by extending widely adopted languages like C and Fortran, and the OpenMP extensions are supported by numerous compilers and is portable over different architectures. We pointed out some issues with OpenMP, i.e., its reliance on shared memory and cache-coherent processors, and the implicit synchronization points between parallel directives.

CUDA and OpenCL provide a way of utilizing heterogeneous architectures for GPGPU. The differences between OpenCL and CUDA are small, in both models a cross compiler toolchain is used to create executable code that is transferred onto the special hardware by OS drivers. However, by using an open standard, open source toolchain and drivers, development is much more transparent and allows the developer to study its inner workings. Workloads written with CUDA can also only be executed on CUDA capabable Nvidia target devices and this limits its usability. Using these language extensions and frameworks for execution on heterogeneous hardware, yields great speed-ups, but even OpenCL development requires knowledge about the target architecture to get the best performance, even thought it is a unified platform.

Up to this point, we have been focusing on resources available on one computer. The next chapter looks at distributed execution systems, utilzing multiple computers and network connectivity to process workloads in parallel. None of the reviewed language extensions provides means of distributing workloads.

2.7 Distributed Execution Systems

As mentioned in section 1.1, when there is not enough processing capabilities available in one single computer, connecting multiple computers together in a network and exploiting their collective resources can result in large performance gains. *High-Performance Computing (HPC)* has been using such distributed approach for a long time. Workloads are sliced into smaller units which are disseminated to a cluster with hundreds or thousands of computers, where the units are executed in parallel with the end result being joined together at a sink node. HPC clusters can be very expensive and often rely on special high-speed data buses, such as InfiniBand. The type of workloads running on HPC clusters are often static batch processing jobs that often have predefined input parameters and run to completion. Users of HPC clusters are often research centers, unversitites, or other large institutions or organizations with need for solving compute-intensive tasks in scientific computing or for processing large data sets.

Another emerging trend in distrubuted computing is *cloud computing* [38]. In cloud computing, organizations with large datacenters and extreme capacity offer parts of their infrastructure to others that need to execute heavy workloads. This is also sometimes referred to as Infrastructure as a Services (IaaS). Examples of such platforms are Google App Engine[10], Microsoft Windows Azure[11] and Amazon Web Services[12]. These three services differ, but they all provide APIs that lets users develop and execute workloads on large corporate-sized machine clusters.

Cloud computing also introduces elasticity, meaning that computational resources can scale with the workloads that are executed. Infrastructure and resources, such as storage or processing time, can adapt dynamically to the workloads, adding or removing

[10]http://code.google.com/appengine/
[11]http://www.microsoft.com/windowsazure/
[12]http://aws.amazon.com/

resources, either automatically or upon request. This is makes cloud computing framework very flexible in means of utilizing hardware resources, and tolerating faults.

Fault tolerant distributed execution systems ensures reliable execution of workloads [39]. To achieve fault tolerance, the system must first detect the fault, before it can be corrected. As discussed later, in their respective sections, Dryad [2] and MapReduce [1] both have fault tolerance as a intergrated part of their execution system.

Users of cloud computing, usually small businesses or consumers, know little or nothing about where or how their code is executed. These services are more or less black boxes only accessed through the Internet, and to the enduser, the resources seem neverending or unlimited. The name “cloud computing” fits well with the ambiguous nature of this type of computing.

Combining the distributed approach with inexpensive commodity hardware and general-purpose computation on heterogeneous architectures linked together with common, wide spread networking technologies is known as *grid computing*. Grid computing is a subset of distributed HPC with a less rigid design and can be a viable alternative to these clusters when executing computationally intensive tasks. SETI@home[13] and Folding@home[14] are examples of grid computing where Internet users offer their private computing resources, computers and game consoles, to help solve scientific tasks.

In this thesis, we take a look at a small subset of framework used in distributed computing and explain their workings, usage and programming model.

2.7.1 MapReduce

In distributed computing, MapReduce [1] has been getting a lot of attention since its introduction in 2004, it is a programming model for working with large data sets. Each data value is associated with a key, and is built around the two operations *map* and *reduce*. It allows data sets to be distributed and processed in parallel and scale accordingly, e.g., on large clusters of commodity computers.

The data sets are partitioned and, as mentioned, each partition is assigned one key. A developer using MapReduce defines a map operation to execute on one data partition. This map operation is executed in a number of independent processes and every process transforms the data assigned in parallel.

In the reduce step, the partitions that share the same key are executed independently from other keys, which implies that there can be one reduce thread per unique key executed in parallel. The reduce operation is also defined by the developer, it can organize or aggregate data and return either nothing or the reduced result.

These two steps must always be combined in an parallel execution, which makes it a rigid design and inconvenient to model complex algorithms with iterative opera-

[13]http://setiathome.berkeley.edu/
[14]http://folding.stanford.edu/

tions. Several enhancements have been added to this model, e.g., adding an additional *merge*-step [40] and support for Cell BE [41]. Both the map and reduce operations are written in a well-known sequential manner, but it does not allow the flexibility needed in multimedia processing [5].

Google's MapReduce implementation executes on large clusters of networked computers, called *workers*. It consists of a runtime system and software libraries and provides a simple approach to parallel execution. The runtime implicitly executes the operations in parallel. A *master node* reads input data, partitions it to fit the map operation, before it executes the operations in parallel by distributing it to idle workers. When all of the worker threads are finished processing, the results are transferred to reduce threads. Data is disseminated among the different compters via a distributed file system.

The implementation also include fault tolerance that handles worker failures. The master node reschedules map operations that are executed by workers that have become unreachable, finished reduce operations do not need reshceduling. To battle unreasonable slow workers, *backup tasks* of operations that are remaining when the execution is near completion, are scheduled to available workers. This ensures that the workers who have trouble are aided by other workers as the they could be remaining because of technical difficulties. Google's MapReduce implementation reads input and writes output by using the distributed Google File System [42].

2.7.2 Dryad

Dryad [2] takes inspiration from MapReduce, parallel databases and shader languages (Accelerator [43] and Cg [44]) executed on GPUs. The authors believe that developers using these technologies succeed in parallel programming because they are forced to think about the data parallelism in a workload. As MapReduce, there is no need for threads, locks and detailed concurrent programming in Dryad.

Developers of Dryad applications are required to explicitly expose *jobs* data dependencies and data dissemination mechanism through a *communication graph*. The communication graph can be any directed acyclic graph (DAG) in which egdes represents communication channels and verticies represents computation or code segments. The DAG defines the structure of a workload and this logical representation is mapped onto executing cores or machines. This DAG provides a deterministic dataflow, there is no shared global state, thus no need for synchronization, which again makes the workload easier to distribute.

Dryad consists of an execution engine that includes a *job manager*, a *name server*, and available *deamons* that create threads on the job managers command. To get a list of available deamons in a computing cluster, the job manager queries the name server. With the received list, the job manager then distributes jobs by scheduling them to the cluster nodes. In Dryad, data input and output are disseminated via a distributed file system, TCP pipes or shared memory.

The way that a developer expresses a communication graph is through a API in C++. It is written so that it could be implemented alongside legacy source code and libraries. Each vertex in the DAG belongs to a stage, which has a manager object called the *stage manager*. It detects code segements that execute slower than other equal segments and reschedules duplicated vertecies. This functionality is analogous to the backup tasks in MapReduce. Dryad is set up to handle failures and topology changes and because of the deterministic properties of the commuciation graph, failed threads can be rescheduled without any implications.

Comparing Dryad to MapReduce, one see that Dryad allow arbitrary inputs and outputs, where MapReduce only take a single input at its master node and output at its reduce nodes. Contrary to MapReduce, Dryad gives better support for complex workloads, but as MapReduce, it is inable to model iterative algorithms [5]. The authors of [2] specify that Dryad is to be used for coarse-grained data parallel applications, and there is no sign of support for hetergeneous architectures.

2.7.3 Kahn Process Networks

Kahn Process Networks (KPN) [45] model parallel programs by connecting sequential concurrent processes through communication links. The communication links are FIFO channels, with non-blocking write operations, but blocking read operations. The FIFO channels are unbound and thus capable of modeling multimedia streams. KPNs are also deterministic, given the same input, they will always produce the same result. A schematic overview of a KPN applications are represented as a graph with nodes (verticies) and egdes. Each vertex contains the sequential code segment, egdes represent communication FIFOs and are directed. KPNs can model both data parallel and task parallel workloads, Ž. Vrba et al. describe both benefits and limitations of using KPNs as a parallel programming model in Nornir [46].

The Nornir runtime is written in C++ and implements a process scheduler, message transport, deadlock detection and resolution, and accounting. The process scheduler dispatches and multiplexes multiple KP threads on top of a few OS threads, the message transport deals with data dissemination between KPs, while deadlock detection and resolution is needed because of the allowed cycles. Accounting is used to monitor performance and other key information.

Like Dryad's jobs and MapReduce's operations, Nornir's processes are written as sequential code segments and can be modelled as a graph. It, however, stands out with its ability to model cycles which represents loops. The current implementation of Nornir does not allow network distributed processes, but it communication channels are network transparent.

2.7.4 Summary

As we have seen in this chapter, there are frameworks and solutions to choose from when writing parallel and distributed software, with perhaps MapReduce being the most prominent. Other frameworks include IBM's System S with its SPADE [47] programming language, Cosmos [48] and Scope [49] also have an extended language support. Microsoft's Dryad as Cosmos and System S utilize directed graphs for cluster execution. Systems S exploits task parallelism, while MapReduce and Dryad only targets data parallelism.

2.8 Summary of Summaries

MapReduce and Dryad both provide an easy way of utilizing symmetric multicore architecture and distribute workloads, but are targeted towards batch processing, not continous flows of multimedia streams. CUDA and OpenCL provide a way of utilizing heterogeneous architecture, but they are not targeting network distributed workloads. MPI are used for distributed processing, but is most suitable for rigid HPC computation. OpenMP is a language extension that that targets symmetric multicore processors and is not intended for heterogeneous architectures, it also do not offer workload distribution. Other frameworks [47] [48] [49] have also been proposed, but not much is known about these systems, since no open implementations are freely available [5].

We have seen that there are several ways of creating both parallel and distributed workloads, but using low-level concurrency intrinsics is often difficult and time consuming. Using distributed execution systems ease the burden and is step in the rigth direction, but frameworks with real-time multimedia support are missing. The P2G framework was initiated because of this observation and provides such support, we want to evaluate and verify its programming model and investigate how it is to implement workloads using the P2G framework. In the next chapter, we examine P2G, highlighting its functionality and capabilities.

Chapter 3

The P2G Framework

There is an increasing demand on multimedia processing, yet there is a lack of support for complex multimedia workloads with real-time requirements in current parallel and distributed execution frameworks. As a result of this observation, the P2G framework was initiated. P2G is a research project developed at the University of Oslo, and the current implementation includes a runtime library, language, and a compiler. It allows both data and task parallelism of looping complex algorithms, network distribution and is to support multiple and heterogeneous architectures. In contrast to MapReduce, Dryad and HPC, P2G is target towards processing of large continous data streams instead of batch processing. P2G also supports iterative, looping algorithms and deadlines for soft real-time support.

3.1 The Runtime

In the P2G runtime, there are two types of nodes, a *master node* and one or more *execution nodes*. The master node disseminates both code and data to a multiple of execution nodes, which has a number *worker threads* that process data in parallel, very much like MapReduce and Dryad. The runtime features a two level scheduling scheme, a *High-Level Scheduler (HLS)* which task is to partition a dataflow graph and distributes code and data to an execution node where the *Low-Level Scheduler (LLS)* in turn schedules the work locally. An execution node and the LLS have indept knowlegde of its own capabilities, properties and topology. The master node and HLS gathers this information to create a broad, but still detailed overview of the total system. The P2G runtime system also includes a *communication manager*, which handles distribution of workloads to network connected nodes. See figure 3.1 for a visual representation.

The *instrumentation manager* collects information from execution nodes' *instrumentation deamons*. The information disseminated from these deamons are the mentioned capabilities and topology information, which among other information includes computer specification, e.g., architecture and number of cores, its connected coprocessors, such

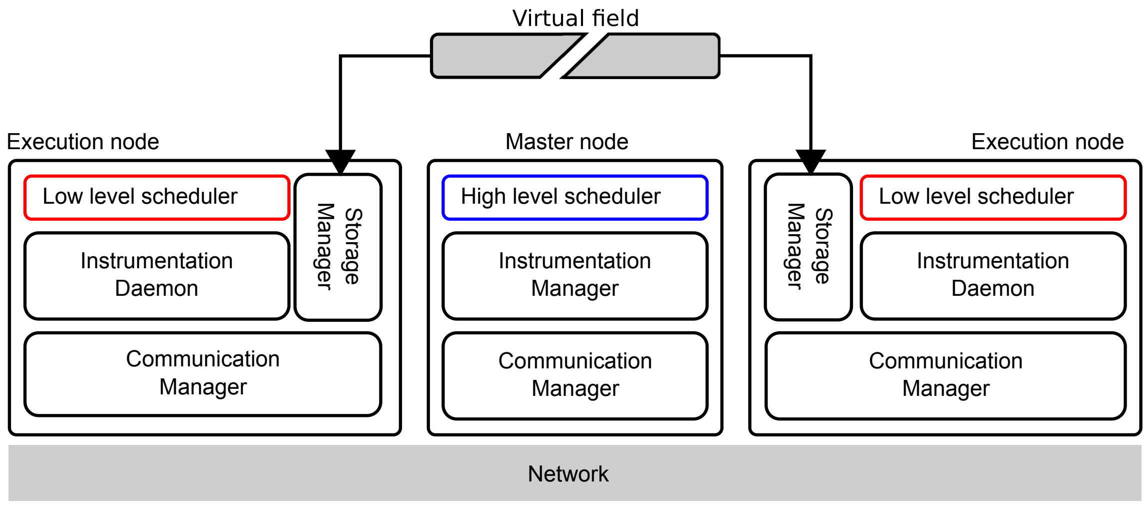

Figure 3.1: P2G Architectureral Overview

as GPUs or DSPs, but also network information. During execution, it reports system-wide CPU and memory utilization along side detailed information about each running kernel, e.g., how long time it takes to process it. The HLS on the master node can use this information to schedule *kernel instances* more intelligently.

The instrumentation information aggregated in the master node can be used for elastic resource allocation in P2G. Thus, P2G is able to adapt to changes in available processing resources, similar to cloud computing's elasticity, mentioned in section 2.7.

P2G builds on stream programming and use the notion of *kernels*. *Kernels* include independently executing code segments and can be executed in parallel, thus supporting task parallelism. Kernels are written in a *Kernel Language*, which is used to model data dependecies between executing kernel code and data segments, called *fields*. A *runtime dependency analysis* is done continously and kernels that are not waiting on any data are executed implicitly. The sequential code block included in a kernel is written in C++, which is familiar to most developers and allow high-level modeling of data structures, custom data types, but also low-level intrisics like bit-shifting.

P2G resembles both MapReduce and Dryad, but unlike any of the two, it is able to model loops. Supporting iterative loops, branching and deadlines, P2G is designed for processing distributed real-time multimedia workloads, while also taking advantage of specialized coprocessors like GPUs. Influenced by earlier work with KPNs and the Nornir runtime [46], P2G uses a dependecy graphs to derive a (Directed Acylic Graph) DAG. Data flows expressed as a DAGs are beneficial as they are deterministic.

3.1.1 Dependency Graphs

We mentioned that P2G utilize execution graphs to model code segments (kernels), data segments (fields), and the relationship between them. First, the *Static Dependecy Graph* (SDG) present the high level depencies between code segments and data segments. Then, the *Dynamically Created Directed Acyclic dependency Graph* (DC-DAG) describe how the SDG is executed, i.e., the P2G application's dataflow.

The P2G compiler first implicitly create an *intermediate* SDG, which is a one-to-one model of the written code, representing data dependencies between different kernels and fields. Directed egdes from kernel-verticies to field-verticies represent read and write operations, much like KPNs.

The DC-DAG is created from the final SDG during runtime. Even though the SDG is cyclic, a transformation is possible due to the write-once semantics of P2G data fields. Because a kernel is only allowed to store to a field once it can only be executed once.

A side effect of the write-once sematics is that a kernel cannot loop, i.e., it cannot do its operations an arbitrary number of times as it would write over the same position in a data field. However, as noted earlier, being able to do iterative loops are crucial for a lot of algorithms. The way P2G have managed to support loops while at the same time keeping the write-once semantics is by aging data sets. By increasing a field's *age*, P2G is versioning the data, which again allows a kernel to write to the same position in the new version.

3.2 Kernel Language

Many of today's most popular programming languages have adopted the procedural or object-oriented paradigm. In these paradigms, we find the constructs such as variables, functions, objects, message passing, and polymorphism. When programming in procedual or object-oriented languages one often tend to have a sequential chain of thoughts; one operation is done before another, and the application flows from start to finish.

In P2G's Kernel Language, there are defined some new logical constructs to help a application developer easily express a data flow graph (P2G's SDG) with familiar syntax and data structures. The Kernel Language encourages the developer to think in terms of data transformation and express as fine parallel granularity as possible, its programming model is simplistic, without any explicit concurrent programming. The language seeks to aid the developer in describing parallelism in algorithms and problems and create program flow that is easily executed in parallel. However, it is also a modular part of the P2G framework, meaning it is interchangable and can be swapped for another.

With the ability of expressing very fine grained parallelism, one can create kernels that only include a few essential operations and thus be able to execute on different architectures, such as GPUs, but one also hope to join a multiple of fine grained kernels and execute them in parallel on SIMD architectures. Another important focus in the design process of P2G and the Kernel Language is to not be dependent on shared memory and cache-coherence, as discussed below in section 3.2.3.

Listed below are the Kernel Language's additional language constructs:

- Kernel
- Fields
- Index-variables
- Age-variables
- Store- and Fetch-statements
- Kernel Code Block
- Deadlines

Data *fields* have a lot in common with traditional variables, however, they can only be written to once. Like Dryad, P2G can model multiple data inputs and outputs associated with one graph vertex as data is *stored* and *fetched* from fields. As discussed in the next sections, P2G workloads written in Kernel Language are able to exploit both data and task parallelism.

3.2.1 Kernels

A multimedia workload written in Kernel Language consists of *kernels*. Kernels are "ad-hoc" code segments in the sense that they are not explicitly created like threads or called like functions, but as we have mentioned, dispatched and executed implicitly by the LLS whenever its data depedency is fulfilled.

For simplicity, we can think of a kernel as an independent code block that can be threaded. However, a thread cannot be executed without being explicitly created, nor the result from multiple threads not extracted before their execution is finished, and they are joined at a synchronization point. A kernel is executed implicitly when its data dependencies are fulfilled and the produced result is stored to a the data field without having to write code for locking, synchronization or thread management.

Because a kernel does not alter any shared global program state, it can be executed independently of all other code. Further, in the case of data slicing, a kernel is independent of multiple instances of itself, and is analogous to kernels in CUDA and OpenCL.

Below you see an example of a kernel in Kernel Language. The `AddFive`-kernel *fetches* (reads) some input integer from the *global field* `input`, adds five to it the C++ code block, and then *stores* (writes) the result to the second global field, `result`.

```
int32 input;
int32 result;

AddFive:
  local int32 input_, result_;

  fetch input_ = input;

  %{
    // Kernel Code
    result_ = input_ + 5;
  %}

  store result = result_;
```

Listing 3.1: `AddFive`-kernel used as an example

In this example, the kernel name is `AddFive`, which uniquely identifies a kernel, meaning that two kernels can not share the same name. In this kernel we create one *local field* called `result_`[1], which is populated by the *fetch*-operation on the global field. The kernel code block transforms the data, before the *store*-operation finishes the execution. Fields are discussed in further detail in section 3.2.2 and kernel code blocks in section 3.2.6.

Task Parallelism

Below you see an example of two distinct and independent kernels, which is how task parallelism is expressed in Kernel Language. It is important to understand that there are no order of execution in this example, there are no data depedencies in either kernel, which makes the execution order arbitrary.

```
Hello:
  %{
    // Kernel Code
    std::cout << "Hello, ";
  %}

World:
  %{
    // Kernel Code
    std::cout << "World!\n";
  %}
```

Listing 3.2: `Hello` and `World` as examples of task parallelism

[1]During our work we created a convention of post-fixing local fields with an underscore.

3.2.2 Fields

In listing 3.1 we defined two *global* and two *local* fields. We populated the local field `input_` by fetching from the global field `input`, and then we stored the local field `result_` to the global field `result`. The global fields can be accessed by every kernel, while the local fields are only accessible within kernels. Global fields cannot be written to more than once, this is to ensure deterministic output of a parallel execution, no field value is allowed to change. Note that the local fields are created with a `local` keyword in front of the type definition.

We also see that the creation of fields, assignments to them, and their use in arithmetic expression are expressed in the exactly same way as regular variables in common programming languages. The type, in this case, a 32-bit wide integer, comes before the field name. Multi-dimensional fields are also created in the same way as arrays in existing programming languages with square brackets added to the type, e.g., `int32[][] foo`. In this example, we have not defined the size of the field array. A developer can set static sizes of field arrays if the sizes are known up front, but the size of a field array can also be dynamic and grow during execution.

Global fields are virtual, meaning that the whole data set do not necessarily reside on the node that executes the kernel, but they provide an easy and familiar way of expressing *how kernels are connected* in terms of data depedencies and a way of programming that developers are familiar with.

When accessing field arrays in the kernel code block, one cannot access them as one would do with regular arrays. There are special function calls for setting and extracting a field value, these are discussed in section 3.2.6. Note that in some applications, like when representing images, the location and index position of data elements in fields are important, but in other applications, the position is negligible, e.g., one can use a field array as a low-level list structure. The index position of elements in a field is a powerful feature, but computations does not always depend on it.

In the current implementation of P2G and Kernel Language, these familiar data types are available:

- 8-bit Integer: `int8` and `uint8`
- 16-bit Integer: `int16` and `uint16`
- 32-bit Integer: `int32` and `uint32`
- 64-bit Integer: `int64` and `uint64`
- 32-bit Floating Point: `float32` and `float32`
- 64-bit Floating Point: `float64` and `float64`

3.2.3 Store- and Fetch-statements

The *fetch*-statement tells the compiler which data a kernel is dependent on. In the previous code example, our objective is to add five to the input. This means that `AddFive` is executed when there is data ready to be transformed in the `input` field, so before any kernel can fetch an input value, some kernel has to *store* that particular value first. The P2G runtime's data dependency analysis keeps track of kernel dependencies and initiates execution whenever they are fulfilled. In listing 3.3, We build on the previous example and expand our Kernel Language code to also include a new kernel, `Init`, that initializes the global `input` field.

```
int32 input;
int32 result;

Init:
  local int32 input_;

  %{
    input_ = 5;
  %}

  store input = input_;

AddFive:
  local int32 input_;
  local int32 result_;

  fetch input_ = input;

  %{
    // Kernel Code
    result_ = input_ + 5;
  %}

  store result = result_;
```

Listing 3.3: The `Init`-kernel is added to our previous example

As we see, the `Init`-kernel does not have any dependencies and can be executed immediately, with `AddFive` being executed as soon as `Init` has stored to `input` and fulfilled its data dependencies. Because kernels do not execute before its data dependecies are fulfilled, we get an implicit barrier and do not need explicitly do any synchronization. Again, if any of these two kernels are executed twice they also write to the same field twice, which is an illegal operation.

Since kernels might be distributed onto networked nodes or executed on a special device or architecture, the fetch and store operations are logical. This means that what a fetch and store actually do may vary and that one fetch or store operation does not map to one data move or copy operation. Instead, these operations can be mapped to a message passing approach, making fetch and store not rely on shared memory or cache-coherence, but also network transparent. In P2G, a kernel can have an arbitrary number of fetch- and store-statements, meaning a kernel can have as many inputs and outputs as it needs. This is similar to Dryad, but different from MapReduce.

3.2.4 Index-variables

An index-variable is a special type of variable that specifies how a kernel should slice a field array and model data parallelism. Index-variables are ment to replace loops and instead create multiple instaces of a kernel, which can be executed independently. One more time, we expand on our example and mofify the `AddFive`-kernel so that it uses 1-dimensional fields and include an index-variable. The previously added `Init`-kernel has deliberately been left out of this example.

```
int32[4] input;
int32[4] result;

AddFive:
  index x;

  local int32 input_;
  local int32 result_;

  fetch input_ = input[x];

  %{
    // Kernel Code
    result_ = input_ + 5;
  %}

  store result[x] = result_;
```

Listing 3.4: The `AddFive`-kernel with an index

The *index-variable*, `x`, is created with an *index*-type. The fetch statement is evaluated and four independent instances of `AddFive` is created by the runtime during execution, one for each of the four values in the input field. The value of `x` is implicitly set from 0 to 3, slicing the 1D field array into four regular, scalar variables. Note that the store-statement also have to index the 1D `result`-field and that the *extent* of these two fields should match. Kernel instances are assigned numbered indices similar to CUDA and OpenCL threads.

A developer can specify multiple index-variables in a kernel, e.g., fetching from a 3D field array, it is possible to slice every dimension with three index-variables (see listing 3.5). Data slicing expressed with index-variables in Kernel Language, models data parallelism in P2G.

```
int32[2][2][2] data;
int32[][][] result;

Slice3D:
  index x, y, z;

  local int32 data_;
  fetch data_ = data[x][y][z];

  %{
    // Kernel Code ...
  %}

  store result[x][y][z] = data_;
```
Listing 3.5: Example of slicing a 3D field array

Each of the kernel instances created with index-variabels is independent and can be executed in parallel, this is analogous to Single Process – Multiple Data (SPMD), discussed in section 2.4. By doing fine grained parallelism, working with 8, 16, 32 or 64-bit, and doing simple operations, the P2G framework hopes to bundle multiple kernels together and execute them with SIMD instructions in a future implementation.

3.2.5 Aging

Because it is only possible to store to a field position once, the kernel instances in our example would only be executed once. As noted earlier, a second execution would store to the same index and break the write-once semantics. To be able to run the `AddFive`-kernel in a iterative loop, continuing adding five, and create a new result every iteration, we increase the age of the global field `result`, creating an age loop. We specify that a field is allowed to age by adding an *age*-keyword after the field name; see the `results`-field in code listing 3.6. In the current implementation of P2G, a kernel is only allowed to have one age-variable, and an aging field is intiated by storing to the first age ($age = 0$).

By fetching the current age of a field, we create a data depedency, which we self fulfill by storing to the subsequent age, and such we are able to create a loop. Like with the index-variable, we specify a special age-variable and use it in our fetch and store statements. We edit our example to use aging, creating an infinite loop. Note that we specify field age with regular parentheses in between the field name and the square brackets used for indexing.

```
int32[5] results age;

AddFive:
  age a;
  index x;

  local int32 result_;
  fetch result_ = results(a)[x];

  %{
    // Kernel Code
    result_ += 5;
  %}

  store results(a+1)[x] = result_;
```

Listing 3.6: The “AddFive”-kernel now includes an age-variable

To keep a loop from executing infinitly, a developer can specify *finalization*. There are two ways of using finalization, the loop can stop executing either when all of the kernel instances have reached a finalization condition, or when only one of the kernel instances is finished. The chosen finalization modes is specified when creating a global field, by adding `all` or `one` after the age keyword. The last store to an aging global is achieved by using a coditional store (see section 3.2.6), and prefix the store statement with a `final`-keyword.

3.2.6 Kernel Code Block

The kernel code block contains sequential code that work on the data fetched from the global field. The code written in the code block contains regular C++ code. The Kernel Language is also translated into native C++ code, which again can be compiled by any C++ compiler. By having a native code block, a developer is ble to call function in legacy code within a code block, and such is able to reuse written code and reduce the time it takes to write a new parallel version. This is similar to how OpenMP can build on and extend legacy code.

In the kernel code block, when a local field is a singular, scalar variable, either it is sliced into one singular value using indices or a single value is fetched, the field can be used like any regular variable in C++. However, if a local field is an array it must be accessed using special function-calls. The P2G has defined some functions that a developer can use to extract and insert values into a field array.

To read a value from a fetched field array, the developer needs to use a `get()`-function, which returns the value at the provided index. The first argument of this function is a field name, while the rest of the arguments depends on the field dimensions. To get the first element in the 1D field array, `array1D`, one would use `get()` like this get(array1D, 0). With a 2D field array, the call would be get(array2D, 0, 0). The same operations on a normal C or C++ array would be array1D[0] and array2D[0][0], respectively.

Inserting values into a field is done with either a put() or a puts() call. When a developer knows the size of a field and it is statically defined up front, a calling puts() is faster as it does not check if the field needs to be resized. The two functions have the same argument signature, where the first argument is the field name, the second is value to insert. As with the get()-call, the rest of the arguments are dependent on how many dimensions the field array is composed of. To insert the value 3 into the first element of a 2D field array, one would call *put()* like this put(array2D, 3, 0, 0). The same operation with regular arrays would be array2D[0][0] = 3.

```
// Prototype of the get()—function
// Returns value at the indices provided
get(field , x—index , y—index , z—index , ... — index)

// Prototype of put() and puts()—function
put(field , value , x—index , y—index , z—index , ... — index)
puts(field , value , x—index , y—index , z—index , ... — index)

// Prototype of the dimensions()—function
// Return the number of dimensions in an field array
dimensions(field)

// Prototype of the extent()—function
// Returns the extent of a dimension
extent(field , dimension)

// Prototype of the age()—function
// Returns the value of the age—variable
age(variable);

// Prototype of the index()—function
// Returns the value of the index—variable
index(variable);
```

Listing 3.7: Prototypes of functions used on field arrays

In scenarios where the developer does not know how many dimensions a field has, or how many elements long a dimension is, P2G provide two functions that returns such information, dimensions() and extent(), while the index() and age() functions returns the value of of the current index or age variable.

Below you can see how the Init-kernel from the previous example use the extent() and puts() functions to insert 0 (zero) into the local input_ field, which afterwards is stored to the global field result. Note that we do not specify an age variable, but store directly to the initial first age, zero.

```
Init:
  local int32[4] input_;

  %{
    for (int i = 0; i < extent(input_); i++)
      puts(input_, 0, i);
  %}

  store result(0) = input_;
```
Listing 3.8: "Init"-kernel using extent() and puts()

P2G also allow conditional store-statements, which lets the developer store to different fields given a certian condition inside the kernel code block. See example in listing 3.9, where we `return` either true or false, like in a regular C++ function, and use `if`, `then` and `else` keywords to catch and act on the returning value. Conditional store can be used for finalization of age-loops, general branching, or when reaching a deadline. An alternative code-path, storing to a different field, leads to new dependencies and new workload behavior.

```
uint8[][] true;
uint8[][] false;

ConditionalStore:
  local uint8[][] input_;

  if %{
    // Checking some condition
    if (time > deadline) return true;
    else return false;
  %}
  then
    store true = input_;
  else
    store false = input_;
  end
```
Listing 3.9: Example of using conditional stores

I/O Handling

To be able to read and write streams of data, kernels must do I/O handling, e.g., reading and writing from file. In the current version of P2G, this is done inside the kernel code block with standard C or C++ I/O operations, like the *iostream-class* in C++.

What is important to remember is that a P2G kernel is very different from a process, thread or function, i.e., a kernel is executed once and keeps no state from its execution, and it can neither take any start-up and execution parameters. This means that it becomes difficult to model potential workflows that are dependent on keeping I/O state from one execution of a kernel to a future execution of a kernel, as this state must be kept somewhere. Examples of I/O state can be, file descriptors, how much data is read or written, or network connections. In the current version of P2G kernels are not distributed among other networked execution nodes, so handling I/O is restricted to the local machine.

3.2.7 Deadlines

In the introduction, we mentioned that P2G supports deadlines. We discussed how an applications, which interacts with an external, physical environment might need timing constraints, in section 2.1.1. Such timing constraints can specify a start-time for when the execution should start, and a deadline for when the execution should end. The time in between these two points defines the total execution time.

At this point, there is Kernel Language syntax for creating global timers in P2G, which can be initiated, assigned or queried within the kernel code block. Using regular C++ if-statements in conjuction with timers, a developer can express deadlines, e.g., given a `timer t1`, we can test `if (t1+10ms > now)` and return true or false to do a conditional store. Creating alternative code paths based on timing observations is one application for timers, but it can also be used for termination of execution. Currently, there are basic support for expressing deadlines in Kernel Language, but semantics of these expressions require more refinement.

3.3 Current P2G Prototype

This section gives an executive overview of the current P2G prototype. The prototype can execute P2G workloads on multi-core linux machines, and consists of a compiler and a runtime.

3.3.1 Compiler

Programs written for the P2G system are designed to be platform independent. Heterogeneous systems are specifically targeted, but many of these require a custom compiler for the native blocks, such as Nvidia's "nvcc" compiler for the CUDA system and IBM's XL compiler for the Cell Broadband Engine. P2G programs are compiled into C++ files, which can be further compiled and linked with native code blocks, instead of generating binaries directly. This approach gives us less control of the resulting object code, but we gain the flexibility and sophisticated optimization of the native compilers, which also results in a lightweight P2G compiler.

The P2G compiler works also as a compiler driver for the native compiler and produces complete binaries for programs that run directly on the target system.

3.3.2 Runtime

The runtime prototype implements the basic features of a P2G execution node, including multi-dimensional field support, implicit resizing of fields, instrumentation and parallel execution of kernel instances on multiple processors using the implicit dependency graph formed by kernel definitions. However, as mentioned, the prototype runtime does not yet have a full implementation of deadline expressions.

The prototype targets a node with multiple processors. It is 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 may require resize operations.

A kernel subscribes to events related to fields that it depends on, i.e., fields referenced to by the kernels *fetch*-statements. When receiving such a storage event, the runtime finds all *new* valid combinations of age- and index-variables that can be processed as a result of the *store*-statement, and puts these in a per-kernel ready queue. This means that the ready queues contain always the maximum number of parallel instances that can be executed at any time, only limited by unfulfilled data dependencies.

Using the implicit dependency graph, the LLS' dependency analyzer adds new kernel instances to a ready queue, which later can be processed by the worker threads. Dependencies are analyzed in a dedicated thread which handles events emitted from running kernel instances that notifies on *store* and *resize* operations performed on fields.

Kernel instances are executed by worker threads 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 or by groups of kernels that satisfy their own dependencies in age loops.

The runtime is written in C++ and uses the blitz++ library for high-performance multi-dimensional arrays. The source code for the P2G compiler and runtime can be downloaded from http://simula.no/research/mpg/files.

3.4 Summary

The P2G framework gives support for expressing both task and data parallelism by using kernels and data slicing with index-variables. Data fields with familiar regular data types support multimedia data, while aging allow modelling of loops. Thus, it should support the basic requirements of typical multimedia workloads. To verify that these constructs are enough to develop complex multimedia algorithms, we designed and implemented Motion JPEG (MJPEG) video encoding (chapter 4) and Scale-invariant Feature Transform (SIFT) (chapter 5) feature extraction in P2G.

Chapter 4

MJPEG: Motion JPEG

In Chapter 3, we introduced the P2G framework and explained the P2G programming model. In this chapter, we provide a design and implementation of Motion JPEG (MJPEG) video encoding. This workload has elements of parallelism as explained in the section 2.2. The parallelism exposed in the MJPEG workload is easily expressed in Kernel Language and is modelled by using distinct kernels and field array slicing.

4.1 Introduction

This work builds on a previously developed MJPEG video encoder used in the teaching of the course *Programming Heterogeneous Multi-core Architectures (INF5063)* at the University of Oslo. What it does is to read a sequence of YCbCr-frames (explained in section 4.2.1) and encode these frames into a new squence of JPEG-compressed frames. Every frame is indenpendent and compressed individually, with no data dependencies between the frames. The YCbCr frames are read from a file, which means that we have full access to every frame when starting. Having no data dependencies between frames, the intuitive approach would be to process every frame in parallel. However, we are not interested in a batch job, we would rather simulate a live, real-time stream scenario. This means that we want to read one frame at a time, as one would done from a network interface, and instead exploit the parallelism within one frame.

First, we would read a YCbCr-frame and write the three separate color channels (*Y*, *Cb* and *Cr*) to three different buffers. The frames we read are already subsampled in a 4:2:2 relationship. Next, a Discrete Consine Transform (DCT) should be applied, transforming the data into frequency components represented in real numbers, before a quantization step is performed, converting the real numbers into integers. The result of these computations are then entropy coded with Run-Lenght Encoding and Huffman coding before it is written to file. This algorithm has a clear sequential flow, the input is transformed in different steps, and output from one step is used as input in the next step. To read subsequent frames, each one of these steps must be done in a loop.

4.2 Design

In the previous section, we gave a brief overview of the MJPEG algorithm. In this section, we will design a workload and explain design choices with respect to P2G and Kernel Language. As explained in the following subsections, we expose two cases of data parallelism within the enconding of one frame. The first parallelism is possible because the YCbCr-components are independent of each other, and the second data parallelism is possible because the DCT and quantization step is done on 8×8 pixel blocks of data, which are independent of each other. There is more potential parallelism in this workload, but we have chosen to focus DCT and quantization as this is what consumes the vast majority of the CPU time. To create a stream of data we use data aging to model a loop, every new frame read is versioned by storing to an increasing age. From figure 4.1, we see how the DCT kernels are dependent on separate fields.

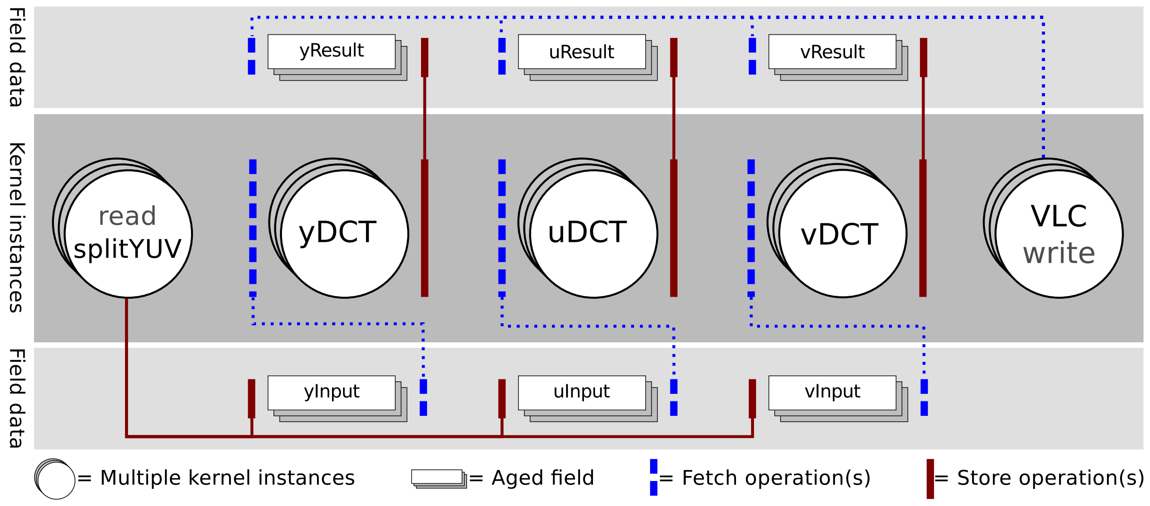

Figure 4.1: Overview of the MJPEG encoding process

4.2.1 Color Components

In the YCbCr-data, the Y-component defines the luma (brightness) of a pixel, a picture with only the Y-component would be a grayscale copy of the image. Cb is a chroma-component and defines how far from the blue color a pixel is, while Cr defines how far from red it is.

The original algorithm process the three color components sequentially. The Y-component is processed first, then the Cb-component is processed, and at last, the Cr-component. However, we mentioned that these components can be processed indepedenty from each other, and therefore also in parallel. The operations transforming the YCbCr-data into JPEG are exactly the same, and can be characterized as data parallelism. However, as we describe in the implementation, we use three distinct kernels for this processing.

This parallelism is coarse and as explained in the next section, finer data parallelism can be exposed.

4.2.2 DCT and Quantization

In addition to the parallelism found in processing every color channel indenpendently, one can also exploit data parallelism in the DCT and quantization step.

The algorithm divides the raw pixel data in each color channels into multiple 8×8 pixel *macroblocks*. Every one of these macroblocks are processed independently without data dependencies, meaning that every macroblock can be processed in parallel. To enable this data parallelism, we need to slice the image into segmented, distinct macroblocks, so that we can instantiate a kernel instance for every macroblock.

Even finer parallel granularity can be found in this step, as DCT in itself can be done in parallel. This can be achieved by using SIMD-instructions to process pixels in parallel. However, it requires a conceptual rewrite of the code that we are using and is not exploited this time.

4.3 Implementation

As mentioned, the code we used for this encoding is derived from a course at the University of Oslo. This code was originally written in C, but instead of editing and adding `extern "C"` to functions and data structures used in the C++ kernel code block, we decided to port the code to C++. We stated that legacy code could be used in a P2G application in section 3.2.6, and by doing so, the development time of a parallel execution can be greatly reduced. What needs to be done in order to achieve parallel execution, is to identify and expose the the parallel parts of the MJPEG encoding application to the P2G framework.

The implementation of MJPEG is composed of six kernels. The `Init`-kernel does nothing but set the file position and frame count to zero and then storing these values to two different fields in their initial, first age. The `Read`-kernel read the frames and create segmented macroblocks. There are three different kernels doing the DCT and quantization, and one `Write`-kernel appending the processed frame to the MJPEG video file.

4.3.1 Read and Create Macroblocks

In the first step of the MJPEG encoding, we read the data into a buffer. The `Read`-kernel is the source of the data stream. However, it needs additional information for reading from the correct file position. We have created a workaround for this to compensate for the lack of *source-kernel* support in P2G.

As mentioned, we mimic a streaming scenario, e.g., when reading from a network socket where there is only one frame available. Since we read from a file and want to finish processing one frame, before reading and starting on the next, we need to store the latest position after every read operation and fetch the same last known file position before initiating a new read operation.

The kernels difference the three color channels it has read. First, we loop over the Y-component, slicing this input data into a number of $8x8$ pixel macroblocks. Then, the same operation is done with both Cb- and Cr-component. The sliced macroblocks are inserted into three different local fields, `yInput`, `uInput`, and `vInput`. How many macroblocks there are created naturally depends on how large the image is and how the three components are sampled, but with a 4:2:2 relationship and a 352×288 pixel image, we get $1,584$ macroblock from the Y-component and 396 macroblocks for both the Cb- and Cr-component. The three mentioned fields can be seen in listing 4.1.

```
int8[1584][64] yInput age;
int8[396][64] uInput age;
int8[396][64] vInput age;
```

Listing 4.1: The global fields that the "read"-kernel stores to

Since the `Read`-kernel reads sequential data from one file, there is only created one instance of the read kernel.

4.3.2 DCT and Quantization

There are three different kernels doing DCT and quantization, we have named them `yDct`, `uDct`, and `vDct`. As we said in the introduction, these kernels are data parallel, but it is a simplification of the real situation. Even though the operations are same in theory, because of the different sampling of the color components, the task of processing the luma (Y) is slightly different than processing the other two chroma components (Cb and Cr). The difference of processing lies in the number of macroblock and the quantization tables used. The number of macroblock can be dynamic because the index `x` will only create as many instances as there are macroblocks. The quantization table needs to be specified within the kernel[1]. That said, our implementation uses two different tables, but the content of the quantization table for Cb and Cr components are actually equal. This means that these two components are true data parallel and can be joined in one three-dimensional field and indexed by two indices, e.g., `x` and `z`. See code listing 8.1 in the appendix on how this is implemented.

[1]See the `yQuantTbl` in the `dctQuantize2d` function call

```
yDct:
  age a;
  index x;

  local int8[] yInput_;
  local int16[64] yResult_;

  fetch yInput_ = yInput(a)[x];

  %{
    uint32_t blockSize = extent(yInput_, 0);
    uint8_t *Y = new uint8_t[blockSize];
    int16_t *yDct = new int16_t[blockSize];

    for (uint32_t i = 0; i < blockSize; i++)
      Y[i] = get(yInput_, i);

    // dct and quantization
    Workloads::dctQuantize2d(Y, 8, 8, yDct, yQuantTbl);

    for (uint32_t i = 0; i < blockSize; i++)
      puts(yResult_, yDct[i], i);

    delete Y;
    delete yDct;
  %}

  store yResult(a)[x] = yResult_;
```

Listing 4.2: The `yDct`-kernel which calls a C++ function

We can see that to use functions outside P2G, one must copy data between field arrays and C++ buffers, which are then passed as arguments. When the the results are copied into the local fields, the result is finally stored to a global field. The total number of independent kernel instances created to do DCT and quantization is $1,584 + (2 \times 396) = 2,376$.

4.3.3 Writing Results

The `Write`-kernel is an I/O-bound source-kernel, it fetches the computed result and writes it to a file MJPEG file. Before it is able to write the result, the kernel must "stitch" the macroblocks together by copying the blocks into a continous bitstream buffer. When this is done, the output is written to file by calling a function passing a pointer to the data as an argument.

As with the `Read`-kernel, there is only created one instance per frame of this kernel. There is a stronger incentive to only have one instance of this kernel, as this kernel not only need to share an I/O resource, but also need to synchronize writing to it.

4-way Intel Core i7	
CPU-name	Intel Core i7 860 2,8 GHz
Physical cores	4
Logical threads	8
Microarchitecture	Nehalem (Intel)
8-way AMD Opteron	
CPU-name	AMD Opteron 8218 2,6 GHz
Physical cores	8
Logical threads	8
Microarchitecture	Santa Rosa (AMD)

Table 4.1: Overview of test machines

Kernel	Instances	Dispatch Time	Kernel Time
Init	1	69.00 μs	18.00 μs
Read	51	35.50 μs	1641.57 μs
yDct	80784	3.07 μs	170.30 μs
uDct	20196	3.14 μs	170.24 μs
vDct	20196	3.15 μs	170.58 μs
Write	51	3.09 μs	2160.71 μs

Table 4.2: Micro-benchmark of MJPEG encoding in P2G

4.4 Results

We tested the MJPEG encoding on two different machines with different multicore architectures, one Intel Core i7 and one AMD Opteron architecture. More detailed information about the processors can be found in table 4.1. For our test, we encoded 50 frames of the standard CIF-sized *Foreman* sequence.

To see if P2G was able to scale the design of our Motion JPEG algorithm, we tested with 1 up to 8 worker threads. As we see from figure 4.2, the MJPEG design scales with the number of parallel worker threads. The total execution time is measured in seconds and the height of the bars show the how much time is used (lower is better). The thread count increases along the vertical axis for each worker thread. For every thread count, 10 tests were executed.

In table 4.2 we see micro-benchmarks provided by P2G's instrumentation manager. These benchmarks provide information about how long time it takes to dispatch a kernel, how long it takes to execute the kernel's code block and how many instances were created and executed of that kernel.

What is important to notice from these kernel benchmarks, is that the time spent in the kernel code is larger than the time used to dispatch the kernel, and that the majority of time is spent in the compute-intensive DCT part of the algorithm.

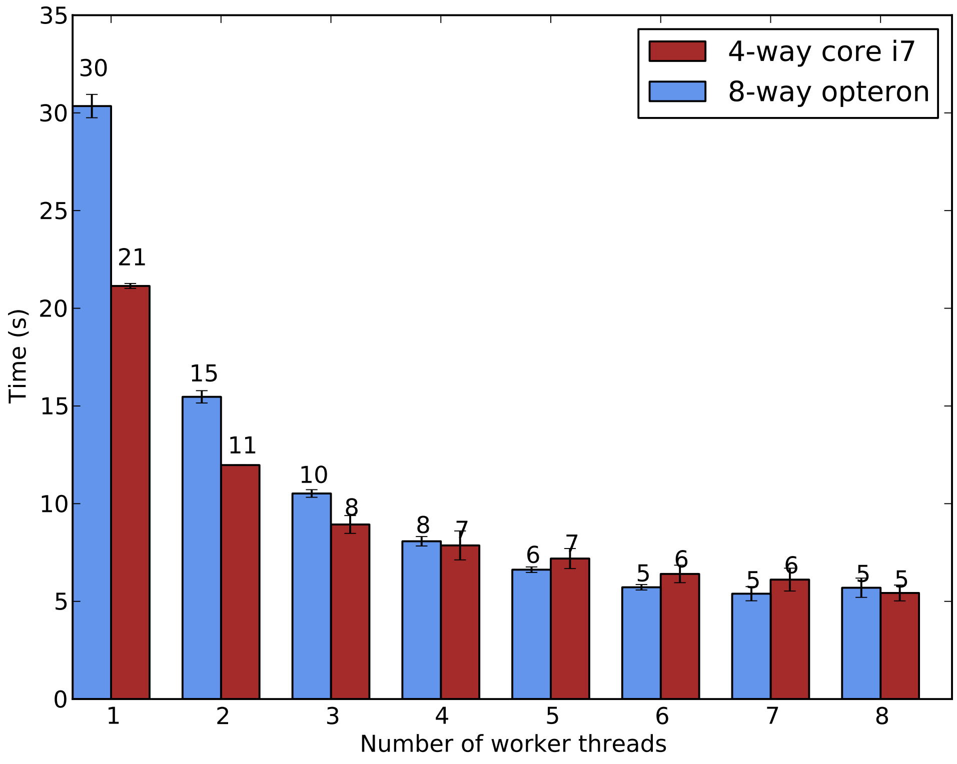

Figure 4.2: Workload execution time for Motion JPEG

4.5 Lessons Learned

We mentioned that we ported the code from C to C++, but when we did so, we wrote it using C++ classes without thinking about how objects instantiated from these classes would transfer their state from one kernel iteration to the next, or even across different kernels. The lesson to be learned from this is that legacy code written to be single threaded might share state between different parts of the program, e.g., the output file can be a global variable opened in the initialization phase of the program, but not used until the end of the program when writing the results. When reusing code in P2G, parts of the application that share state must be identified and refactored to avoid such sharing of state.

Both reading input and writing output to file created some extra challenges. Originally, the reading was done frame by frame, but because the implementation was single threaded and used a global state, it had a different usage of file descriptors. We could not share two file descriptors (YCbCr-input and MJPEG-output) between kernel iterations, we had to open and close the file descriptors for every new read and write operation. We described how we managed to do so in section 4.3.1 and 4.3.3, but recognize the challenges with keeping I/O state in an age-loop.

In section 6.4.1, we discuss the problem of keeping I/O-state in P2G. However, dispite the issues with (object- and) I/O-state, we managed to implement MJPEG relatively easy and shorten the development time.

4.6 Summary

We have provided a design and implementation of a Motion JPEG video encoder in this chapter. We have seen that P2G and Kernel Language can easily reuse legacy code by calling functions within a kernel's native code block. The tests also show that P2G is able to scale our parallel design. In the next chapter we present another compute-intensive workload, in which we do not call any library functions.

Chapter 5

SIFT: Scale-Invariant Feature Transform

In Chapter 4, we looked at MJPEG video encoding and an implementation of this workload using P2G. In this chapter, we describe the Scale-Invariant Feature Transform (SIFT) [6] algorithm and provide a thorough analysis of its pipeline, a parallel design and a partial implementation in P2G.

SIFT is an algorithm developed for detecting *image features* and extracting invariant *image descriptors* from these features. Generally, a *local* image feature is a part of an image, which is different than its surroundings and often are associated with a sudden change in the image, e.g., at an edge [50]. A feature can be a specific point in an image, an edge or a small image patch (region). To create an image descriptor from an image feature, calculations on image data in a region around the local feature are analyzed and utilized to describe the local feature. *Global* image features can also be used to describe image content, an example of such is a color histogram. In SIFT, an image feature is first called a potential *interest point* and then an image *keypoint*.

Image descriptors are used for describing interesting parts of an image, and by extracting image descriptors from one picture and searching through previously extracted descriptors, one can match equal descriptors and recognize the same features. For best results, it is important that the feature descriptors are *invariant*, meaning that image noise, scale, rotation, and other image distortions do not affect the descriptor, i.e., one wants to extract same features from the same scenes, but from different viewpoints, scales and perhaps different heterogeneous cameras.

Extraction and matching of image descriptors are useful in several image applications, e.g., object-recognizion, robotic vision and panorama image stitching. In video streams, it can be for video search, object-tracking, perhaps in 3D applications with stereoscopic video, and 3D scene and model creation. 3D applications are perhaps particulary interesting as it can be used for creating virtual viewpoints and agumented reality.

Compared to other algorithms, SIFT has been proven to be very good [51]. Speeded Up Robust Features (SURF) [52], published in 2006 claims better performance than SIFT, both in terms of time of computation time and robustness against image transformations. It is scale- and rotation invariant and similar to SIFT, it approximates certain operations to achieve a speed-up in computation.

Johannes Bauer et al. [53], evaluated implementations of SURF and SIFT against each other. The result showed that SURF is close to SIFT in descriptor quality, but produced less descriptors. Depending on the application of descriptors, this can be good as it reduces complexity of matching, but also problematic when the application relies on large descriptor sets. However, SURF proved to be a lot faster than SIFT in terms of correct matches per second, which is due to SIFTs more compute-intensive algorithm.

Being very compute-intensive it can also be time-consuming, making real-time feature extraction in video very difficult on a single computer. Because of this, GPU [54] and Field Gate Programmable Arrays (FPGA) [55] implementations are proposed, giving an 8-10x and 250x speed-up compared to single-threaded CPU implementations. Qi Zhang et al. [56] manage to process real-time HD video on a simulated symmetric 64 core machine with their parallel version of the algorithm.

The SIFT algorithm is heavily cited, and several modifications and enhancements are based on it, e.g., Affine-SIFT (ASIFT) [57] yields more and better affine invariance descriptors as it is simulating image views by varying parameters for camera axis orientation. Principal Components Analysis (PCA) is applied to the normalized gradient patch instead of using SIFT's smoothed and weighted histograms in PCA-SIFT [58], which claimes to yield more distinctive and robust image descriptors, but also increase matching accuracy and speed.

SIFT is well-known and frequently used algorithm, and in addition to this, it is complex and compute-intensive. Implementing SIFT in P2G with Kernel Language would be a good measure for the P2G framework's ability to express such multimedia workloads. When creating the MJPEG workload, we utilized already written code and called library functions from external code. However, when creating this workload, we write everything from scratch with hope of exposing as fine parallel granularity as possible.

5.1 The SIFT Algorithm

In this section, we describe the original SIFT algorithm and analyze data dependecies to identify indepedent computations that can be used in a parallel design. The parallel design in P2G is presented in the next section.

There are four main steps in the SIFT algorithm. First, the input image is read and used to create a *scale-space* and detect *extrema*. Then, the *potential* (image) interest points (extrema) are interpolated and located in subpixel accuracy, while *unstable* interest points are discarded. In the third step, one or multiple *orientations* are assigned to the interpolated location. The last step creates a *keypoint descriptor*, which is computed based on image data, transformed relative to the orientation(s), scale and location. The first two steps of SIFT detects and localize interest points, while the two last steps creates the invariant keypoint descriptor from the detected keypoints. This implies that good descriptors relies on good indetification of images features. This algorithm has a clear sequential pipeline, output from one step is input in the next, but there are still parallel aspects within each step that can be exposed and expressed in P2G. A more detailed analysis of the algorithm is described below.

5.1.1 Building Scale-space and Detecting Extrema

The scale-space is built to detect stable features points across all scales, and is created by convolving the input image with Gaussian functions that have an increasing scale σ (standard deviation). Convolving an image with a Gaussian function has an blurring or smoothing effect on the image and is often used to reduce pixel noise.

In figure 5.1, we see that the scale-space is divided into multiple *octaves*. Each of these octaves represents a doubling of the scale (σ). Each octave holds a predefined number of images (denoted as "s"), where each of these images are blurred with an increasing scaling step of $k = 2^{1/s}$. SIFT produces these images by incrementally convolving the input image with the Gaussians. One can think of each octave as a 3D cube with image width and height coordinates on the y and x axes, and blurred images stacked on the third axis (σ). By first doubling the size of the input image, one is able to increase the number of sample points and make better use of the input data. Doubling the image size creates four times as many stable keypoints [6].

Instead of actually doubling the scale σ, SIFT initiates a new octave by subsampling an image in the previous octave and then incrementally blur the octave relative to this. The subsampling is done by taking every other pixel, and creating an *image pyramid* like this do not reduce the accuracy, but it reduces computation as the data set is shrunk. However, this also implies a data dependency between octaves, which is highlighted with blue and read images in the Gaussian pyramid in figure 5.1.

Subtracting pixel values in the neighboring blurred images within an octave, results in a Difference of Gaussians (DoG) scale-space, see figure 5.1. The DoG is a close approximation of the scale-normalized Laplacian of Gaussian (LoG), but a DoG is more efficient to compute. The scale-normalized LoG together with extrema detection, produce true scale invariant and stable image features [6]. To be able to subtract pixel values in an image, its adjacent scales must be available, but this is the only data dependency.

According to several papers [54] [55] [56], creating the DoG scale-space is one of the most compute-intensive steps and where they benefit most from optimizations. We recognize data parallelism in this step, by having the scale step for each image in an octave, we can calculate every scale up front and convolve every image in parallel. We also noted that every pixel is independent of each other during subtraction of pixel values, meaning that each pixel subtraction can be executed in parallel given its neighboring scale is available for reading and enough computing resources.

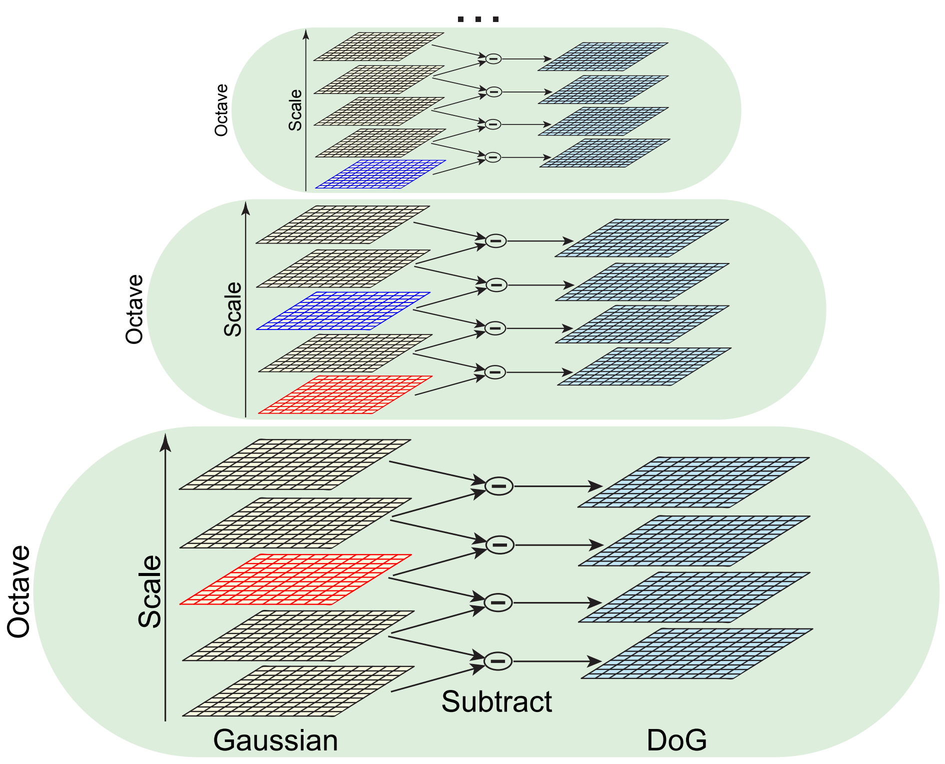

Figure 5.1: Scale-space pyramid with data dependencies between red and blue images (Modified figure from Lowe's paper [6])

Gaussian Smoohting

So far we have discussed the highlevel parallel execution of the scale-space creation, but there are finer granularity of parallel execution when creating the scale-space. As mentioned, by not having a data dependency between two adjacent images in an octave, we are able to blur each image in parallel.

SIFT uses a 2D Gaussian function to convolve images with $(G(x, y, \sigma))$. However, using two 1D Gaussian kernels gives the same end result and is more efficient to compute. By using two 1D Gaussians we can also slice the image data set and exploit data parallelism by processing individual rows idenpendently before procssesing every column indendpendently. In this scenario, the column blur is depedent on the result from the row blur, which means there has to be a synchronization point where the results are joined together. In addition to processing every row and column independently, it is possible to calculate every pixel independently. The smoothing operation only needs read access to the pixels within the boundaries of the Gaussian kernel.

Extrema Detection

In figure 5.2, we see that extrema are detected by comparing every pixel with its 26 pixel neighbors in a 3D cube. Every pixel in an image is tested against its nearest neighbors in the current image, but also against every neighboring pixel in the previous and next scale. Testing is done by checking if a pixel has a greater or lesser value than every other neighboring pixel.

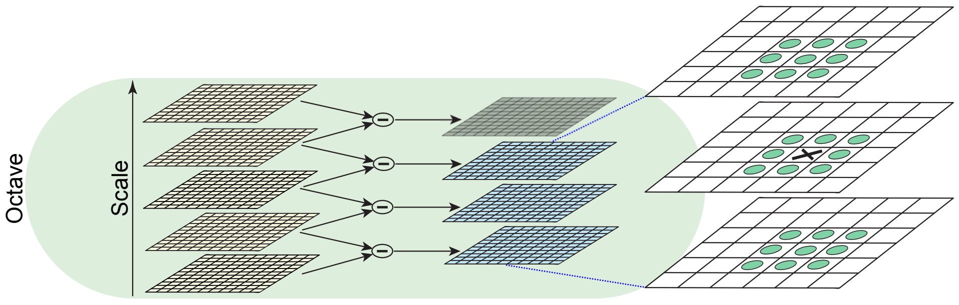

Figure 5.2: Extrema detection (Modified figure from Lowe's paper [6])

From figure 5.2 we also clearly see the data depedencies between the three neighboring images, every image processed needs to access the previous and next scale. This means that processing one octave would result in s number of independent threads, but that the previous and next scale must be available for reading before these executions can start.

In the DoG computation, we identified data parallelism by subtracting every pixel in parallel as long as the corresponding pixel in neighboring scale was available for reading. The same principle applies to extrema detection, but in this case, every detect operation is dependent on reading 26 neighboring pixels instead of just one pixel. Another minor difference between extrema detection and DoG creation is that the number of possible threads is lower, which is because we do not check border pixels (as they do not have a full set of neighbors).

5.1.2 Interpolating Extrema and Eliminating Keypoints

The extrema found are then used as input for subpixel localization of the true extrema. SIFT interpolates in all three dimension, image height (x), width (y), in addition to the scale (σ). The subpixel localization siginificantly improves matching and stability of keypoints [6].

As depicted in figure 5.3, this subpixel localization is applied to detect the true extremum of the keypoint and results in a pixel offset. The *red pixels* in the grid are the extrema detected, while the *green points* are where the true extrema lies. In cases where the interpolation shows that a keypoint is closer to another point in any of the three dimensions, i.e., one of the offsets are larger than 0.5, the keypoint is redefined and interpolation is done on this keypoint instead. A new keypoint is marked with *blue* in figure 5.3.

Every extremum found in the previous step are independent of each other, but the derivatives used in subpixel interpolation are approximated by calculating the difference of the neighboring pixels. However, this image data is available, which means

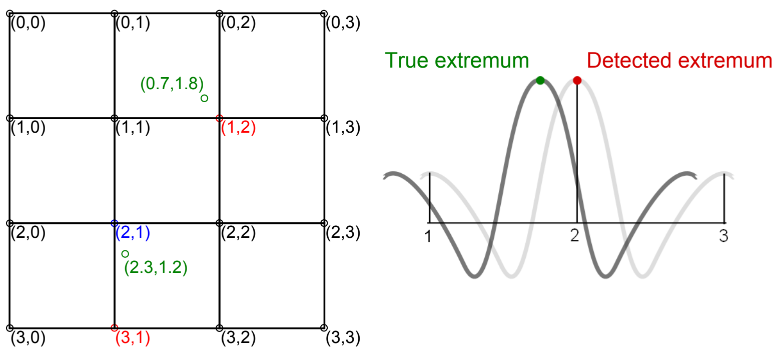

Figure 5.3: Example of subpixel location of extrema in a pixel grid

that every subpixel interpolation can be executed in parallel promptly after a extremum is localized. The interpolated keypoints are then tested for *low contrast* and *egde responses*, eliminating the keypoints that are not robust against change in any of them.

5.1.3 Assigning Orientation

The keypoints that are not discarded are then assigned orientations (θ). The interpolated scale for each keypoint found in the previous step is used to choose the closest Guassian blured image, from which every sample points orientation ($\theta(x, y)$) and gradient magnitude ($m(x, y)$) is calculated by using pixel differences. Lowe [6] also states that these two values should be precomputed for each pixel in each image so that they also can be used in the next step, creating the descriptor.

Examining this step, we found two cases of task parallelism and one case of data parallelism. The calculations of orientation and magnitude are task parallel, and can be processed independently. Every pixel can also be processed in parallel as long as the neighboring pixels are available for reading. This results in a thread count that is $2 \times pixels \times s$. However, these calculation are only depedent on the blurred image data, meaning that calculating orientation and magnitude is task parallel with extrema detection, and subpixel location and keypoint elimination.

When the orientations and magnitudes are computed, an orientation histogram can be created from a region around every keypoint. The peaking bin in the historgram is the dominant orientation. The dominant orientation and other bins that are within 80% of the peaking bin are selected to represent keypoints, implied that there can be created one keypoint per peaking orientation. Only about 15% of the keypoints are created with multiple orientations [6]. Before using the orientations in the histogram, they are weighted with their magnitude gradient and a Gaussian kernel.

5.1.4 Creating SIFT Descriptor

Now that we have an image location (x, y), scale (σ) and a dominant orientation for our keypoints, the local, region-based SIFT descriptor can be created. This descriptor is also created with a historgram of gradient magnitudes and orientations sampled in a 16×16 array around the keypoints, however these orientations and the keypoint coordinates are rotated relative to the keypoint's previously calculated dominant orientation. This is to make the descritor orientation invariant.

The orientations are then used to create 16 historgrams in a 4×4 subregion, each with eight orientation bins representing the magnitude for each orientation entry. The orientations are distributed among the bins, and as with the previous orientation assigment step, a the orientations are wheighted with a Gaussian kernel. The wheighting prevents significant changes in the descriptor with smaller position changes of the image patch. The 16 histograms holds 8 bins, which gives (16×8) 128 values. To make the descriptor more robust, we need to normalize to unit length and reduce the influence of large gradient magnitudes.

5.1.5 Summary

Scale-Space Extrema Detection: The algorithm starts by upsampling the original input picture and Gaussian smooth an *octave*. Then, the algorithm subsamples a smoothed picture to initiate the next octave and this continues to the complete Gaussian pyramid is created. Based on the Gaussian pyramid, a Difference of Gaussian (DoG) pyramid is created by subtracting pixel values in adjacent Gaussian smoothed images. Then at last, every pixel in the DoG is iterated over to search for extrema.

Keypoint Localization: For all the detected extrema in the DoG, derivatives are approximated and used to interpolate the localization of the true extrema. Then, unstable features are discarded based on their contrast and edge responses.

Orientation Assignment: In this step, the pixels orientations and gradient magnitude are computed. Then, magnitudes and orientations in a region around the keypoint are calculated and used to create a orientation histogram. In this histogram, the bin with the peaking orientation and other bins that are within 80% of the peaking orientation are selected and use to create keypoints.

Keypoint Descriptor: To create a descriptor, we again use the orientations and magnitudes in a region around the keypoint. To keep the descriptor orientation invariant, we rotate coordinates and orientations relatively to the dominant orientation found in the previous step. We merge the orientations from a 16×16 matrix into sixteen 8-bin histograms, which results in 128 values.

5.2 Design

Hao Feng et al. [56] describes a parallel SIFT algorithm for multi-core systems. They redesign a sequential algortihm and use OpenMP for expressing parallel code sections, see their parallel pseudo-code in listing 5.1.

```
for all octaves
{
    List keypoint_list;

    for all scales
    {
      ConvolveImageGaussParallel();
      BuildDoGParallel();

      // Detect Keypoint
      #pragma omp parallel for
      for all pixels p in Image
      {
        if(IsKeypoint(p))
        #pragma omp critical
        keypoint_list.add(p);
      }
    }

    #pragma omp parallel for
    for all pixels kp in keypoint_list
    {
      ExtractFeature(kp);
    }
    DownSampleImageParallel();
}
```

Listing 5.1: Pseudo-code for parallel SIFT in [56]

This algorithm is brief and nonspecific, but we see that each octave is built sequentially, due to the data dependency between octaves. We see that convolving an image and creating the DoG is done in parallel, but these two operations are not task parallel, i.e., the output from the Gaussian convolution is used as input when building the DoG. Further, keypoint localization is executed in parallel, with the keypoint list being the shared state between all the threads, potentially creating a bottleneck. The last part extracts features from keypoints in parallel before it downsamples the image second to last in the Gaussian pyramid and loops. The next sections discusses how we can create a SIF pipeline with respect to P2G's functionality and Kernel Languge's expressiveness.

5.2.1 Building Scale-space and Detecting Extrema

To start with, we need some way of expressing a loop that can create the image pyramid, because of its data dependecies. This is normally done with ageing in P2G. Next, we need some way of creating the Gaussian and DoG pyramids. Two-dimensional field arrays can be used to store image data, but we need a two extra dimensions for modelling octaves and intervals (an image set) within an octave.

As mentioned, the most intuitive way would be create a 4D field using the predefined datatypes in Kernel Language to represent image data. First index would point to the octave, second index to the interval, while the third and fourth index position points to the width and height coordinates of the image. The problem with this solutions is that P2G does not allow different array sizes within a field array. Said in other words, with this solution each octave can not contain different images sizes. For a better understanding, we have examplified how to create such *jagged arrays* in C++ in listing 5.2. Notice how height and width differ in octave 0 and 1 (first index position).

```
pyramid = new char***[octaves];

pyramid[0] = new char**[intervals];
pyramid[0][0] = new char*[height];
pyramid[0][0][0] = new char[width];

pyramid[1] = new char**[intervals];
pyramid[1][0] = new char*[(height/2)];
pyramid[1][0][0] = new char[(width/2)];
```

Listing 5.2: Example of jagged arrays in C++

To circumvent this problem, one can create a 4D field array where all image sizes are equal (the largest image would set the size for every other image). Smaller images are stored in this field array, but the unused pixel values are padded and marked invalid. This structure can also be complemented with another field, mapping meta data on the actual image sizes.

The second solutions would be to age the data field and let subsequent ages represent a new octave as aging a field allows the field to change array size. This is well aligned with our initial idea of using aging for creating a loop iterating over octaves. In this case, the age becomes the fourth dimension, but this implies that in a stream of images, every new image read must be stored in the current age plus the number of octaves. For example, the first image would occupy age 0 to 5, a new, second image would then have to be stored into age 6 using the next six adjacent ages (6-12) for its octaves.

We noted that a field could only have one age dimension in section 3.2.5. Since the current version of P2G do not support jagged field arrays and we use aging to create a dimension which allows us to have such jagged arrays, it could be beneficial to have a second age dimension to create an outer loop, e.g., for reading a sequence of images as described above.

Gaussian Smoohting

We mentioned that we wanted to exploit data parallelism when convolving an octave with Gaussian kernels. To enable this and not convolve in a loop, we would create a data parallel kernel, creating as many instances that there are images in a octave.

We also noted that we are able to use two equal 1D Gaussian kernels and process image rows independently, before processing columns independently. To achieve this, we could create two blurring kernels where the second kernel is depedent on input from the first kernel. However, when convolving, there is a concern for how to fetch a range of data, i.e., every data point a Gaussian kernel is dependent on. There is also a problem of how to handle the edge cases, where data dependecies go beyond the image size and the field array's boundaries.

Compute DoG

We stated that computing the DoG is done by subtracting pixel values from two neighboring Gaussian smoothed images in the same octave. As seen in figure 5.1, a Guassian pyramid with five scales in each octave, we end up with four DoG images. Every image processes is dependent on the subsequent images, which contains the values to substract. However, there is no data dependecies between the octaves, with respect to the DoG, each octave can be processed independently.

We recognize similar problem in this step as we did when creating the Gaussian pyramid. We would like to create as fine granularity as possible, i.e., create as many kernel instances as there are pixels, but we need access to additional data. In this case, it is the neighboring scale. One could use two fetch-statements, one fetching index `x` while the second fetches index `x+1`, but there is currently no support for such arithmetic expressions together with index-variables. To solves this, one can create one kernel instance per octave that iterates over every image in the kernel code block, but this yields a very coarse parallelism.

Extrema Detection

As seen in listing 5.1, parallel threads adding to a shared keypoint list alters the global state of the applications and the execution is prone to race conditions. Hao Feng et al. solves this by using the OpenMP *critical* directive. As we have stated earlier, using concurrent programming ensures data integrity and deterministic behaviour, but resource contention can introduce a bottleneck in scenarios where multiple threads needs access to a single shared resource.

However, there is a problem with storing the extremas found in this step in P2G as well. The number of extrema found is not given, the only thing we know is that the number of extrema is significantly lower than the pixel count. There is also no *set* data structure in Kernel Language to handle scenarios where one needs to shrink a data set used to exploit data parallelism, i.e., a data set sliced with an index-variable.

In the current implementation of P2G, we have found two different workarounds, which can solves this issue. First, one could save results in a sparse field, analogous to a sparse matrix, and insert invalid values to field positions where no extrema is found. The subsequent kernel fetching this field using data slicing would create kernel instances for the invalid elements and would have to handle these gracefully in some way. One could also create a special kernel for looping over every element, filtering out the valid values from the field and dynamically growing a new dense field array, which then could be used in a data parallel execution.

A second solution would be to fetch a whole octave and loop over every image. This would not give any parallelism in detection, but would allow dynamically storing extrema by using the `put()`-operation on a single field. The subsequent kernel would fetch from this field, creating one instance for every extrema found.

Both these methods actually utilize a field array as a list structure, where the position of an element is of no importance, it is just another element (of many) in a list. Note that this is not the intended use of a field array.

5.2.2 Interpolating Extrema and Eliminating Keypoints

The number of extrema found in the previous step dictates how many kernel instances we can create in this step. In Lowe [6], a 233×189 pixel image resulted in 832 keypoints after searching for maxima and minima in the DoG scale, but the number of extrema found depends strongly on the input image.

After locating the true extrema and eliminating keypoints that are not robust, the same example image has 536 keypoints left, almost 300 keypoints less. Here, as the previous step, we see that we need to reduce keypoint data set. The same solutions also applies here as in the previous section, either do a serial step, looping over every keypoint or invalidate elements in a parallel execution, with the result of either creating invalid kernel instances in the next step or use a serial kernel in between to create a new dense field array.

5.2.3 Assigning Orientation

In the last two steps, we identified issues with reducing data sets used in data parallel exectuion. In this step, we might need to grow the data set. Because a keypoint can only have one dominant orientation, we need to add a new keypoint for every peaking orientation we find. As stated earlier, about 15% of the keypoints have multiple peaking orientations.

When shrinking a data set in parallel execution, we could work around the problem by invalidating elements. However, there is no workaround for increasing the data set in a data parallel execution, we need a serial step iterating over all keypoints in the C++ code block, and potentially growing a new field array dynamically with P2G's `put()`-function.

5.2.4 Creating SIFT Descriptor

Creating SIFT descriptors can benefit from data parallel execution as the creation of each descriptor is independent, and lets us create one kernel instances for evey keypoint indetified in the previous step. This step relies on reading orientations and gradient magnitudes for a region around the keypoint, but these values are already computed and available. When every descriptor are computed, a kernel could fetch every descriptor and write the output to a file or the screen.

5.3 Implementation in the P2G Framework

In the previous section, we discussed some issues regarding the design and tried to reason about how we could create a SIFT workload in P2G. In this section and the next subsections, we describe what we discovered when we implemented some of our ideas.

The *Init*-kernel is the only kernel in our SIFT workload with no data dependencies (fetch-statements) and is therefore the first kernel to execute. The kernel starts by computing the sigmas needed to create the Gaussian kernels for convolution. What is important to notice is that the number of scales created in this step implies how many intervals images is created in each octave later.

5.3.1 Building Scale-space and Detecting Extrema

Representing Gaussian and DoG pyramids in C++ could be done by either creating a 4D image data pointer, or creating a 3D pointer from an image wrapper-class, encapsulating image data. See examples of both solutions below in code listing 5.3. However, to represent an image pyramid in Kernel Language, we let an age define an octave.

```
char pixel;
char ****data;
Image ***images;

// Allocating memory
...

// Indexing first pixels (index = 0,0)
// in second (index = 1) octave and fourth
// (index = 3) interval
pixel = data[1][3][0][0];

// Alternatively, like this:
pixel = images[1][3]->getPixel(0, 0);
```

Listing 5.3: Example of image pyramids in C++

A sequential outline of this step is depicted in figure 5.4. For simplicity, some fields, store- and fetch-statements are left out of the figure, e.g., the field containing scales (σ).

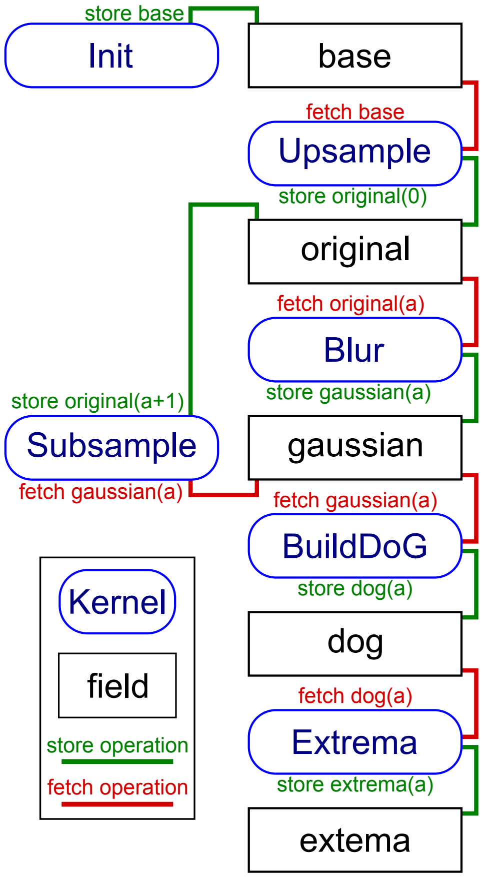

Figure 5.4: SIFT Kernel Language Outline

As you see, this step consists of five kernels, *(1)* `Upsample`, *(2)* `Subsample`, *(3)* `Blur`, *(4)* `BuildDoG` and *(5)* `Extrema`.

The `Upsample`-kernel fetches the input image and store to the initial age $a = 0$, which is the first level of the scale-space pyramid. `Upsample` stores as many images in the first octave as the number of scales (σ) implies. At this point it creates one kernel instance for every image, but we would rather see that it stores one result to multiple field positions in a field array. This is more efficient than creating multiple instances that produce the exact same result. When an image is stored to a position, the `Blur`-kernel fetches the image and convolves it with a Gaussian kernel. The number of instaces created from this kernel equals the number of interval images in one octave.

We stated in our design that we wanted to process every row in parallel before processing ever column in parallel, when convolving the image. However, we could not seem to find any good solutions to the issues with edge cases and fetching of data for the whole Guassian kernel.

The `Subsample`-kernel fetches the second to last image from the any octave (see figure 5.1) that is blurred and stores new subsampled images in the next octave. This kernel loops and initiate all of the octaves. From figure 5.4, we see that that *BuildDoG*-kernel has the same data dependencies as *Subsample*, which means that it starts building the DoG-pyramid at the same time as the subsequent octaves are being initiated. This is pipeline parallelism as discussed in section 2.2, which [56] does not recognize. We have outlined three stages in a modified version of [56]'s algorithm in listing 5.4, for simplicity some of the code is excluded. In this loop, *Stage 2* is dependent of *Stage 1*, but *Stage 3* is not dependent on *Stage 2*, thus these two stages can be executed in parallel, e.g., by spawning two independent threads, meaning that building the DoG and the rest of the algorithm can potentially be processed in parallel with building of the remaining building of the Gaussian pyramid.

```
for all octaves
{
  // Stage 1: Blur and Create Gauassian Pyramid
  for all scales
  {
    ConvolveImageGaussParallel();
  }

  // Stage 2: Create DoG Pyramid and Detect Extrema
  for all scales
  {
    BuildDoGParallel();
    // Detect Keypoint
    ...
  }
  ...

  // Stage 3: Downsample and Loop
  DownSampleImageParallel();
}
```

Listing 5.4: Pseudo-code for a pipeline parallel SIFT implementation

Build DoG

As stated in the design, when two neighboring images are blurred, their pixel differences can be calculated. The pixels in an image are independent of each other, which means we can instantiate one kernel per pixel. However, Kernel Language does not provide any way of expressing indices that depend on two and two images that overlap, i.e., when creating the first DoG image, one subtracts the pixel value from the two first images in an octave; when computing the next DoG you shift your data set by one image, and subtract pixel values in image two and three.

In this step, we found that we could use two fetch-statements. One fetch-statement that creates all the necessary instances and indices, and a second statement that fetches the whole data set. Every kernel instances can access the data set as it wants by using the indices, and by using the `index()`-function we have every index available in every instances, which can be used to model and fulfill more complex data dependencies. The drawback of using this method is the implicit data depedency barrier of waiting for the whole data set to be comlete.

Even though this solution would create a large number of kernel instances, we had to restrict this kernel to only one instance per image in each octave. This was due to undefined and undiserable behaviour in the P2G prototype when creating a substantial number of kernel instances.

Extrema Detection

As the computation of the DoG scale-space, the extrema detection can be executed in parallel by processing every octave independently. However, there are data dependencies within one octave, every image pixel tested needs to access its closest surrounding pixels at the same scale, but also the its nehighbors in the previous and next scale.

As with the previous step, we only created one instace per image in each octave, but here this decision was manily due to the lack of a set data structure as discussed earlier in the design. Given a 352×288 pixel image and $s = 3$ intervals in one octave, one could create $352 \times 288 \times 3 = 304,128$ instances. However, in a typical search one would expect to find about a $1 - 2,000$ extrema, depending on the image size and content. Thus, this creates more than $300,000$ elements marked as invalid.

Developing this kernel, we started by creating a kernel instance per octave, but had to restrict the parallelism due to the prototype suddenly and arbitrarly stopped processing (stalling and not using any CPU), and sometimes segfaulting. The prototype also used a lot of time to execute the kernels.

5.3.2 Interpolating Extrema and Eliminating Keypoints

The `Elimination`-kernel feches from the `extrema`-field populated in previous step and creates as many kernel instances as there are extrema and process every extremum independently. This kernel uses a sparse field solutions as discussed earlier in the previous section. Every extrema that is eliminated is replaced by invalid values in stored to the `keypoints`-field.

However, when developing this kernel, we started to experience a lot of trouble with the P2G prototype, and at the same point, the development of the prototype we were using was branched out an modified to better support scalability. However, this had implications for our SIFT workload as the new prototype do not yet support all features needed by our workload, meaning that the implementation of SIFT is delayed until it does.

5.3.3 Assigning Orientation

This step can segregated in two smaller tasks. First, we have the task of calculating orientations and magnitudes, then, we have the task of accumulating the orientations in a histogram and assigning the dominant orientations to keypoints. However, the task of calculating orientations is task parallel with calculating the magnitude, every pixel can be processed independently, and further these two tasks can be started whenever an image has been Gaussian smoothed. More precisely, these tasks can be modelled with two independent kernels, which can be started right after the *Blur*-kernel has stored an image. This will create a large number of independent kernel instances.

When the orientations and magnitudes, and the keypoints from the *Elimination*-kernel are stored, the *Orientation*-kernel can start its process of building the histogram and selecting the dominant orientation(s). As mentioned in the design, every keypoint can have a multiple of dominant orientation, which creates a new keypoint. We must create one instance that iterate over every keypoint and `put()` the existing keypoints, in addition to new keypoints in a new field array.

5.3.4 Creating SIFT Descriptor

With each keypoint having a dominant orientation, position and scale, the computation of the final image descriptor can start. As soon as the `Orientation`-kernel has finished storing the new keypoints with dominant orientations, the `Descriptor`-kernel can fetch this and start the creation of this keypoint's descriptor. Contrary to the previous kernel, this step can index the field array, creating as many kernel instances as there are keypoints.

5.4 Results

Due to the challenges with our version of the P2G prototype, we did not manage to implement our SIFT workload to a full extent. However, we did manage to *(1)* upsample the intial input image, *(2)* downsample the initial image and such create an image pyramid. We also did implement *(3)* Gaussian blur and *(4)* DoG computation, so that we were able to create a DoG scale-space. However, at the time we finished our work, our extrema detection only found about 60% of the same extrema points as a reference implementation[1], in addition to some false positives. This is likely due to our implementation of Gaussian smoothing, and has nothing to do with the evaluation the feasibility of the P2G framework and approach.

5.5 Lessons Learned

During this work, we wanted to validate P2G's ideas, the current implementation of it, and the expressive power of its Kernel Language, and even though we did not finish our SIFT implementation, we learned a lot about the current version of the P2G framework.

The lessons learned from this design and partial implementation are, *(1)* creating an image pyramid in Kernel Language was challenging, it could be useful to either have double aging, a jagged field array, or custom data structures. We also learned that, *(2)* fetching data ranges are necessary in certain scenarios, and *(3)* that it is hard to shrink and grow data sets when using field arrays and exploiting data parallelism at the same time. These lessons are discussed further in the next chapter.

5.6 Summary

In this chapter we presented the design of our desing and partial SIFT implementation. We analyzed the algorithm to indentify and expose parallelism, before we considered how this could be designed using P2G and Kernel Language. We managed to implement some core features of SIFT, but due to problems with our prototype implementation, we did not finisih. Yet, we learned a great deal through this work, which we discuss in the next chapter.

[1]http://www.vlfeat.org/ vedaldi/code/sift.html

Chapter 6

Discussion

In chapter 4 and 5, we explained our design and implementations of MJPEG and SIFT in the P2G framework and reviewed our results. In this chapter, we discuss our experiences while using P2G and Kernel Language, we look at how the source code for our workloads can be improved and propose some possible solutions for how to do it. The first section in this chapter discusses how it was to use Kernel Language on a general basis, then we take a look at the data types and means to model data structures. In the third and fourth section, we discuss how I/O is handled in P2G, before we look at other issues in the last section.

It is important to have in mind while reading this chapter, that the limitations discussed are derived from the current prototype implementation of P2G, which we have been working with. The P2G framework is ongoing work, its concepts and future implementation spans way beyond the prototype. Troughout the work with this thesis, the P2G development team has contributed to and influenced this discussion.

6.1 Expressing Parallelism

As we stated in chapter 2, it is difficult to develop large parallel and distributed workloads from low-level concurrency intrinsics. There are many considerations, bugs can be difficult debug, correct, and sometimes even harder to register. Explicitly setting up communication channels in message passing, mutual exclusion when sharing data, and thread synchronization is also time consuming.

In the next section, we evaluate how it was to use kernels and data set indexing to express parallelism. First, we look at kernels, fields, and the data communication and synchronization between them. Then, we discuss index- and age-variables, before we propose a solution on how to use index-expressions.

6.1.1 Kernels and Fields

Kernels are used to express task parallelism in P2G and Kernel Language. During our work, we became very comfortable with this model, it has proven to be effective. In our opinion, expressing task parallelism by creating indepedent code segments in terms of kernels, and let a runtime execute them, is simple and intuitive. It allowed us to exploit task and data parallelism in both workloads. Even though we believe kernels are a good abstraction when writing parallel and distributed code, there are some challenges when distributing to other networked execution nodes; this is discussed later in section 6.4.

Fields resembles traditional global variables, which are familiar constructs to most developers. Fetch and store statements used to express data dependencies are also fairly easy to understand. What can be more challenging is the write-once semantics and the use of aging. Expressing data parallelism by indexing field arrays is similar to how OpenCL and CUDA use indices to exploit data parallelism, but for a developer who is familiar with loops and not indexing data sets, this concept can be challenging at first. Having two kernel instances of one kernel is also more abstract than two distinct and different kernels.

P2G encourages the developer to express as fine parallel granularity as possible, which means, in terms of task parallelism, identifying as many independent tasks and creating as many kernels as possible. By studying an algorithm one can also find subtasks within a "naturally defined task" that can be executed independently of one another, e.g., like in SIFT's calculation of pixel orientation and gradient magnitude. In terms of data parallelism, fine granularity means working on the smallest data elements, i.e., using as fine index granularity as possible, like working with single pixels in an image.

By comparing the kernel model to traditional concurrent and distributed programming, P2G's Kernel Language model is simplistic, more effective and yet powerfull at exploiting parallelism. The model abstracts away concurrent programming and let developers focus on indentifying parallelism in algorithms.

6.1.2 Data Dependencies and Implicit Execution

As the authors of Dryad [2], we believe that forcing developers to explicitly and actively separate data sets and analyze data dependencies, make them succeed in data parallel programming. Thinking in terms of kernels which, as stated in section 3.2.1, is implicitly executed when data its dependencies are fulfilled introduce such force. In addition to this, we believe that separation of data into multiple fields in conjunction with the kernel model also aids identification of task parallelism. However, implicit execution of a kernel when its data dependencies are fulfilled, is perhaps unusual to developers who are used to explicitly calling functions or creating concurrent threads.

A P2G application's dependency graphs represent its data flow. The runtime derives this overview of every data depedencies only by evaluating fetch- and store-statements. We believe that using fetch- and store-statements to express data dependencies between kernels and fields is simple and easy to understand. We mentioned that the store- and fetch-statements are logical, and do not necessarily model data move or copy, it can be inferred as moving a pointer or involve message passing operations to a distributed node.

6.1.3 Index- and Age-Variables

As we noted earlier, index- and age-variables are most likely unfamiliar concepts to most developers, as it was to us. However, using index-variables to express slicing and data parallelism took some time to get used to, but exploring their usage in some small examples leveled the learning curve.

Index-variables used to fetch distinct, individual element from a field is not a problem, but if a developer needs to fetch a range of data, P2G has no way of expressing this in Kernel Language. For example, a computation of element x_i, where i denotes the current index, is depedent on neighboring data elements to index i, i.e., from element x_{i-1} and x_{i+1}. Developer might benefit from such *index-expressions* as it more precisely defines the data dependencies.

Also, as we noted in section 5.1.1, being able to have more than one age-dimension of a field could be a much needed feature. To explore this option, we propose a solution using *multiple aging* when building a image pyramid.

Index-expressions

Using arithmetic expressions together with index-variables in fetch-statements could fetch elements that is in relation to the index. We mentioned fetching the elements before and after current index, another example is fetching an element that is five positions ahead of the current index (x_{i+5}).

More complex index-expressions would be required when image data is concatenated into a one-dimensional field array and each kernel instance were to work with a single image row, i.e., the first instance would want fetch the range from x_0 to $x_{\texttt{width}-1}$ and the next instance would fetch from $x_{\texttt{width}}$ to $x_{(2\times\texttt{width})-1}$. There are also scenarios where one would like to fetch overlapping data elements, e.g., one kernel instance would depend on x_0, x_1 and x_2 while the next kernel instance depended on x_1, x_2 and x_3. This example might seem very similar to the example of fetching x_{i-1}, x_i and x_{i+1} as they, in some cases would provide the same data. However, their edge cases would differ. Egde cases and how to handle them are discussed later in this section.

The ability to fetch such ranges, or multiple neighboring elements in a field, can be necessary in some workloads. This idea of *array slicing* is implemented several programming languages, e.g., Fortran, Ada, Perl and Python. In Python, given a one-dimensional array with five elements, $[1,2,3,4,5]$ array slicing can return subsets of this array, see example of different slices below in listing 6.1.

```
array = [1,2,3,4,5]

slice = array[:2]   # returns [1,2] range 0-1
slice = array[1:4]  # returns [2,3,4] range 1-3
slice = array[-2:]  # returns [4,5] range 3-4

slice = array[::2]  # returns [1,3,5] every other element
slice = array[2::3] # returns [3] evey third element from 2
```

Listing 6.1: Examples of array slicing in Python

We believe index-expressions can be used to, provide additional data for the kernel instance to work with, but we also believe that in some cases, index-expressions should also create less indices and kernel instances. This makes good sense in scenarios where the developer wants coarser data parallelism. For example, in the the case of fetching image rows, creating an instance for every pixel when only one instance per row is needed would most likely waste computational resources. To be able to do so, one must have language support for increasing index stepping. In listing 6.1, we see that Python has array slicing for returning every other element. A similar feature in Kernel Language could also be used to decrease the number of indices and kernel instances. We propose how to write fetch-statements with index-expressions in code listing 6.2, heavily influenced by Python's array slicing.

```
// Fetching current index and
// two neighboring elements
fetch data_ = data[x-1:x+1] wrap;

// Fetching range of three
// adjacent data elements
fetch data_ = data[x-1:2] pad;

// Fetching 352 elements with
// a stepping (stride) of 352
// (e.g, image width = 352)
fetch data_ = data[x:352:352];
```

Listing 6.2: Examples of fetch-statements with index-expressions

Index-expressions can some times try to fetch beyond the field array's boundaries, e.g., in the first fetch statement, when creating a kernel instance of the first data element $(index = 0)$, it would want to also fetch from the previous $(index = -1)$. This problem with edge cases can be solved by, *(1)* do not create kernel instances that fetch across field boundaries, *(2)* create kernel instances fetching across data boundaries, but zero pad the fetched data, make it point to NULL or similar, or *(3)* create instances, but wrap around the edges, e.g., fetching $(index = -1)$ would rather fetch the last element in the field, i.e., $index = size - 1$.

Not creating the kernel instances is aligned with how data dependencies work in P2G at the moment, the data dependecy for the exceeding element is not fulfilled. However, depending on the application, a developer could want to create the instance even though there are no data present. To solve this, we propose adding keywords to the fetch-statement, examples of which is in listing 6.2. The `wrap` keyword would wrap around the data set, `pad` would zero pad the data set, while no keyword would yield the expected result of not creating instances where the expression overstepped field array boundaries.

In the current prototype of P2G, Kernel Language has no way of expressing this using index-variables, but would be very useful, e.g., in SIFT's Gaussian smoothing and MJPEG's DCT (Discrete Cosine Transform) computation of a macro block.

When building the Difference of Gaussian (DoG) scale-space in our SIFT workload, we used two fetch-statements to be able to access adjacent scales. One fetch-statement fetched all the data, while another fetch-statement created as many kernel instances as pixels. Using index-expression instead would probably result in a more efficient execution as the kernel instances could start executing whenever the data covering their individual data dependencies are fulfilled, i.e., in our example, the kernel instances does not have to wait for the implicit barrier created by fetching all of the scales. In a distributed scenario, using index-expressions instead of our solution would proably also reduce the amount of data sent across the network.

Uninitialized Ages

To create age-loops with the current prototype of P2G, a developer must initiate the loop with an intialization-kernel. Initialization in this case means storing to the inital age $(age = 0)$ as noted in section 3.2.5. To illustrate this, we have included code example 6.3 where the developer wants to read a sequence of images and use age to model the sequence, every new age represents one new image.

```
uint8 sequence[200][200] age;

Init:
  local uint8 image_[200][200];

  %{
    // Some code that reads
    // the first image ...
  %}

  store sequence(0) = image_;

ReadLoop:
  age a;

  local uint8 image_[200][200];
  fetch image_ = image(a);

  %{
    // Some code that reads
    // the subsequent images ...
  %}

  store sequence(a+1) = image_;
```

Listing 6.3: Example of initializing an age

This code first executes `Init` as it has no data dependencies, which initializes `sequence` by storing to $age = 0$. The `ReadLoop`-kernel then forms a loop and reads the subsequent images, adding them to `sequence`; similar to our MJPEG workload. We believe that having one general kernel handling the age-loop makes the source code less complex (and more elegant). This discussion is related to the evaluation of fetch- and store-statements in section 6.3.3.

Multiple Aging

As we briefly mentioned earlier in this section, we could be able to better solve the problem of creating an image pyramid with the concept of multiple aging, in the case two levels of aging was needed. In code example 6.4, we propose a simple outline of how double aging could be used to create an image pyramid. The kernel initializing the age is deliberately left out to keep the example brief.

```
uint8 pyramid[][] age;

OuterLoop:
  age a;

  local uint8 image_[][];
  fetch image_ = pyramid(a)(0);

  %{
    // Some code that reads
    // subsequent images ...
  %}

  store pyramid(a+1)(0) = image_;

InnerLoop:
  age a, b;

  local uint8 image_[][];
  fetch image_ = pyramid(a)(b);

  local uint8 resized_[][];

  %{
    // Some code that uses 'put()'
    // dynamically grow 'resized_'
  %}

  store pyramid(a)(b+1) = resized_;
```

Listing 6.4: An example of multiple aging

We can see from the code that this forms a nested loop, but what is unique about these loops is that the outer loop does not have to wait for the inner loop to finish before it can start reading a new image into the first age-level.

6.1.4 Pipeline Parallelism

In chapter 2, we explored different types of parallelism, among them pipeline parallelism as described by Michael I. Gordon et al. [11] and William Thies et al. [13]. During our work, we discovered that P2G and Kernel Language is great at not only expressing task and data parallelism, but also pipeline parallelism. With a number of tasks in a pipeline, we could create an age loop "supplying" tasks with data, analogous to a factory pipeline.

To illustrate this, we have included an example in listing 6.5. Here we see three kernels that are task independent, but not data independent; every kernel needs input from the previous kernel's output to do its computation. In the first iteration ($age = 0$), there are no parallelism. However, in the second iteration ($age = 1$), `Stage1` reads the image while `Stage2` finishes up its processing on $age = 0$. Finally, in the third iteration, the data has propegated to `Stage3`, which then processes $age = 0$ while `Stage2` is processing $age = 1$, and `Stage1` reads a new image and stores to $age = 2$. The observant reader will also notice that `Stage2` is data parallel.

```
uint8 stage1[][] age;
uint8 stage2[][] age;
uint8 stage3[][] age;

Stage1:
  age a;

  local uint8 stage1_[][];

  %{
    // First stage in pipeline,
    // reads image ...
  %}

  store stage1(a) = stage1_;

Stage2:
  age a;

  index x, y;
  local uint8 stage2_;
  fetch stage2_ = stage1(a)[x][y];

  %{
    // Second stage in pipeline
  %}

  store stage2(a)[x][y] = stage2_;

Stage3:
  age a;

  local uint8 stage3_[][];
  fetch stage3_ = stage2(a);

  %{
    // Third stage in pipeline
  %}

  store stage3(a) = stage3_;
```

Listing 6.5: An example of pipeline parallelism with P2G

6.2 Data Structures and Types

In the previous section, we discussed how it was to use Kernel Language on a general basis. In this section, we look more specifically on how to use other data structures and data types in P2G than the one discussed in section 3.2.2. First, we see how the image pyramid can be created with a jagged array field and compare it with the version created with two-level aging. Then we take a look at how custom data structures could enrich Kernel Language. At the end, we discuss if the sparse array fields we created in SIFT workload could be replaced by a set data structure.

6.2.1 Jagged Fields

In the previous section, we outlined a scenario where we used multiple aging, because the current impementation of P2G does not support having different 1D array sizes within a multi-dimensional field array, e.g., in a 2D field array, if the first position in the first index points to a 1D field of size 5, the next position cannot point to a new 1D of size 10. We also described this problem when creating image pyramids in section 5.2.1. However, it would be interesting to know how our code would look like if such jagged fields were supported at this point.

```
uint8[][] image age;
uint8[5][][] pyramid age;

OuterLoop:
  age a;

  local uint8[][] image_;
  fetch image_ = image(a);

  %{
    // Some code that reads
    // image sequence ...
  %}

  store image(a+1) = image_;

InnerLoop:
  age a;
  index x;

  local uint8[][] resized_;

  local uint8[][] image_;
  fetch image_ = image(a);

  local uint8[][][] octave_;
  fetch octave_ = pyramid(0)[x];

  %{
    // Based on value of index,
    // resize images using
    // 'put()' to grow 'resized_'
  %}

  store pyramid(a)[x] = resized_;
```

Listing 6.6: An example using jagged arrays

When comparing this jagged array code to the version with two level aging (listing 6.4), we see that the "inner loop" in this version is not a loop. Resizing of independent images benefits from data parallelism, which makes the jagged array solution better then the additional age-loop. In a loop scenario where an iteration depends on input from the previous iteration, we see that multiple aging can be used to write such loop, but it can also be written in the kernel code block using a C++ for loop. With our experience, we believe that a developer would benefit more from jagged field arrays than from multiple aging.

6.2.2 Custom Data Structures

In C and C++, we are allowed to create custom data stuctures, which helps us encapsulate data. See a simple example of a RGB data structure in code example 6.7.

```
struct rgb_data
{
  uint8_t red, blue, green;
};
```

Listing 6.7: Example of a regular C-struct

In listing 6.8, we propose how one could use custom data structures together with field arrays, if P2G allowed using structs-like data structures in Kernel Language as an alternative to its predefined data types. In listing 6.7, we propose how to create and use a RGB image by using the struct.

```
rgb_data[200][200] image;

RgbImage:
  index x, y;

  local rgb_data pixel_;
  fetch pixel_ = image[x][y];

  %{
    // Accessing struct members
    pixel_->red = 0;
    ...
  %}
```

Listing 6.8: Example of usage of custom data structures

One must have in mind that encapsulating data like the RGB struct can make it difficult to exploit parallelism, e.g., if the data transformation was independent of the color channel, one would benefit from having three data fields (one for each channel) as we did in our MJPEG workload. However, if this is not the case, using custom data structures could reduce source code size and make the code easier to read and understand.

Implementing the use of custom data types can be problematic in a distributed environment with heterogeneous arhcitectures where endianess and byte alignment are subject to change, this problem out of the scope of this thesis and is left for further work.

6.2.3 Set Structures

A problem arises in workloads that creates multiple kernel instances and where some of the instances do not need to store the output, e.g., some of the input values are invalid cannot be used in further processing. In our SIFT design, we discarded extrema (discussed in section **??**) and created a sparse field array by flagging invalid data with a predefined value. Here we used a field array as a low-level list where the index position of an element is negligible (see section 3.2.2).

There are two problems with sparse field arrays, *(1)* they might need a serial step to create a new dense field array, or *(2)*, the subsequent data parallel kernels depending on this field would create unnecessary kernel instances and would have to filter out instances with data is flagged as invalid. There is a tradeoff between these two solutions, the serial step forces synchronization of threads and a loop, while instances processing invalid elements occupy worker threads.

There is no way of utilizing field arrays, in any reasonable way, in the opposite scenario, where a kernel instance need to store more than one data element, i.e., to grow the data set by adding more elements than it originally had. To achive this, some new set structure is needed.

An example of such scenario can be found in our SIFT design. When computing orientations, we create as many kernel instances as there are current keypoints. However, because we potentially need to add extra keypoints by identifying more than one dominant orientation, we end up with more keypoints than kernel instances and thus need to grow the data set.

We have included simplistic versions of our two problems in listing 6.9. Remember that we are using the field array as a list, their position and associated index has no importance when creating kernel instances.

```
int32[5] input age;
int32[] output age;

DecreaseDataset:
  age a;
  index x;

  local uint32 element_;
  fetch element_ = input(a)[x];

  %{
    // Less than zero is invalid,
    // which is flagged ...
    if (element_ < 0)
      element_ = INVALID;
  %}

  store output(a)[x] = element_;

IncreaseDataset:
  age a;
  index x;

  local uint32 element_;
  fetch element_ = input(a)[x];

  %{
    // On condition a new element
    // needs to be added ...
    if (condition)
      // There is no way of adding
      // an extra element ...
  %}

  store output(a)[x] = element_;
```

Listing 6.9: Example of sparse array

Utilizing field array as a list-structure is not a good solutions as it introduces new problems and there is no way of growing a field array dynamically. We propose a new data structure allowing increasing and decreasing elements dynamically, which, like fields, could be used in data parallel computations is needed. The discussion on wether to create a regular set structure or a multiset (also known as bag) structure is a discussion on allowing equal elements. A multiset is a set structure that allow "identical elements to be repeated a finite number of times" [59]. The different structures have different applications and could be implemented side by side if believed to be necessary.

Implementing a set could be done with a list-structure. Donald E. Knuth has defined operations one might want to do on a list [60]; a subset of these can be found below.

1. Access a specific element
2. Insert a new element
3. Delete a specific element
4. Combine two lists into one list
5. Split one list into two lists
6. Determine the number of elements

In a set, elements does not have an order, but the set could be implemented with an iterator, allowing elements to be distinguished from each other and to be used for achieving data parallelism. This set structure could inherit the fetch-statement used with field arrays and such allow creation of kernel instances in the same manner.

From set-theory, we recognize the *union* operation ($\cup$) for representing the collection of elements in multiple sets, which is the same as the fourth list-operation we listed. Instead of reusing field's store-statement, we propose using a *union-statement* to join several smaller *local set fields* together into one large *global set field*. The local and global set fields are analogous to local and global fields. See an example of how this could be used in listing 6.10, where we also propose tentative language syntax. The example would create $3 \times 5 = 15$ data elements in the output set.

```
int32 set input;
int32 set output;

MultipleElements:
  local int32 set elements_;

  %{
    // Add some values to the set
    for (int i = 0; i < 5; i++)
      insert(elements_, value);
  %}

  union input = elements_;

DataParallel:
  iterator i;

  local uint32 set input_;
  fetch input_ = input[i];

  local uint32 set output_;

  %{
    // Add processed input value
    insert(input_, value);

    // Need to add even more values
    for (int i = 0; i < 2; i++)
      insert(output_, value);
  %}

  union output = input_;
```

Listing 6.10: Example of how to use an union statement

Code example 6.10 only expands data sets. We stated earlier that workloads that creates multiple kernel instances, and where some of the instances find the input values invalid and do not store them are problematic when using fields. Using sets and the union-statement, this is handled elegantly as the union of empty sets is a valid operation, thus set structure can be used in these scenarios as well. There are a lot that needs to be investigated with respect to set-structures, which are subject for further work, among them are the possible use of intersection- ($\cap$) and complement-statements ($\setminus$).

6.3 Source- and Sink-kernels

Source- and Sink-kernels are special case kernels that produce and start a data stream (data source) or receive at the end of data stream (data sink). As stated in section 3.2.6, I/O handling is done within a kernel code block, which means that, depending on wether it is a source or sink kernel, fetch- and store-statements are not needed.

6.3.1 Source-kernels

Resources that source-kernels typically reads from can be a input from a keyboard, files, or network connections. One can differentiate between three types of data sources, *(1)* complete data sets (e.g., files), *(2)* incomplete or partial data sets (e.g., streams), and *(3)* requested subsets of data (e.g., database). Depending on what kind of data source a kernel is reading from, there might be arbitrary delays, but what these data source have in common is that the data flow goes via the OS. A developer must call OS systems functions to get read access to them network connections or files, and when a developer tries to read from a resource via a system call, the OS might decide to *block* the the process or thread. Note that source kernels do not have to rely I/O resources to produce data, other data streams could be created, e.g., from a random number generator.

Blocking I/O

When a function blocks, the thread or process that calls the blocking function is put on a waiting queue by the OS, but this occupies one of the available worker threads, making it unavailable for any of the other kernel instances. Looping over the I/O resource querying it multiple times can be even worse, as this consumes a lot of CPU time in addition to stealing a worker thread. The developer can sometimes specify that the function call is not allowed to block, but this is neither desirable as the function call would not actually provide any data.

One solution to this problem could be to wrap the standardized I/O system calls in P2G I/O library. This library could include custom P2G I/O-calls executed by the runtime. When the rutime receives an I/O-call, the Low Level Scheduler (LLS) can put the kernel instance on a waiting queue, analogous to what the OS does, and then it can use the worker thread for another kernel instance. When the blocking function returns, the LLS take the waiting kernel of the queue and start executing again. Depending on the read-pattern of the kernel, the kernel instances might be put on and taken of the runtime waiting queue several times.

Using non-standard I/O calls is more challenging, e.g., I/O calls that are a part of a hardware driver or any other third party library. Such library call can either wrap standard I/O calls or block waiting for something else. Catching such calls, either their are I/O or not, is difficult and out of the scope for this thesis.

6.3.2 Sink-kernels

Resources that a sink-kernel use can be standard output to a terminal, writig to files or network connections or similar. These resources can also block and occupy one worker thread. The same solutions of wrapping I/O functionality, as we proposed in the section sources kernels, can be applied here as well. However, when writing from a kernel, the data is available, meaning that the I/O library write functions can *buffer* this data.

Buffered I/O

It is hard to predict how an implementation of this might look like, but in the case of writing to output, one might be able to tweak the wrapping write-functions by letting the runtime buffer output data. By having all of the data buffered it can execute a new kernel instances in the worker thread, without have to reschedule the writing instance later. In case of such solution, we propose that the monitoring manager keeps track of buffer queue size and report on discarded I/O.

6.3.3 Evaluation of Fetch- and Store-statements

In a source-kernel, you would not need, and therefore do not have any fetch-statements, as in a sink-kernel, you would not have any store-statements. In the current prototype of the P2G framework, it is not possible to create multiple instances of a kernel if the fetch-statement does not include a index. This is because store-statement is not evaluated when creating kernel instances. A developer might want multiple kernel instances reading from the same I/O resource and use an offset to operate indepedently on different data segments. One could also read from multiple I/O resources, e.g., reading from multiple cameras. Reading from a shared resource does not create a lot of problems, but writing to a shared resource needs to be synchronized and can possibly introduce resource contention. Because of the possibility kernels without fetch-statements and store-statements, we believe that both these statements should be evaluated when deciding on how many kernel instances to create.

6.4 Distributed Kernels and I/O

In the previous section, we discussed source- and sink-kernels and how they are the source and sink of a data stream, and we know that I/O handling is done within a kernel's code block. We stated in section 3.2.6 that at this point in development, kernels that access I/O resources are always on a local machine. However, problems arises when kernels are to handle I/O resources when executing in a distributed environment, this is because the I/O resource might not be connected to the execution node processing the kernel code. I/O resources a source-kernel would read from are most likely keyboards, hard drives, network interfaces or cameras, all of which is typically bound to a certain computer, which implies that a kernel reading from such device with standard I/O libraries also become machine bound. We believe it is fair to say that network interfaces can be exempt from this discussion, as this is I/O devices which every node needs to be able to communicate and operate in a distributed fashion. This is reinforced by the fact that every node has its own network manager.

In its current version, P2G and Kernel Language do not give satisfactory support for I/O bound source- and sink-kernels. Even though workloads are not distributed at this point, it is important to have a strategy on how to solve problems that follows enabling of distribution.

There are three different solutions to problems that occur in such scenarios, *(1)* P2G can restric access to I/O on other machines than the master, *(2)* kernels accessing I/O must specify on which machine they wish to do so, or *(3)* P2G can distribute I/O resources so that they are accessible from every execution node. We discuss these possible solutions in later sections, but first we the problem of keeping state from iterative I/O operations.

6.4.1 I/O State in Iterative Kernels

We looked into I/O state in section 3.2.6 and stated there is a problem with keeping this state in looping kernels, created by using an age-loop as explained in section 3.2.5. The problem with using OS support for I/O operations is that OSes are not designed to handle kernels, but processes and threads, and because P2G is supposed to work on continous streams of data and not just batch process files, keeping I/O state is important.

We wanted to simulate a data stream in our MJPEG encoder workload, where the `Read`-kernel creates an age-loop, reading one frame at a time. The frame being read is stored to a global field, which the kernels doing JPEG transformation depend on and fetch from. Independent of this, the `Read`-kernel starts reading the next frame and initiates the second JPEG transformation. For the second execution to be able to read from the correct file position, the file's byte position or the active file descriptor from the first execution must be kept. In an OS, file descriptors are kernel object handles used by processes or threads to access I/O resources. These handles are specific to one process and is not shared. Since P2G developer has no knowledge of how or even on which machine a kernel is executed, the file descriptor received when opening a file cannot be kept and accessed in the next age-iteration, it must be discarded (closed) in the same kernel code that created (openend) it. There are of course many other implications of distributed kernels, but this is discussed later in this chapter.

Similar examples are found when creating TCP sockets. Given that a kernel creates an incoming TCP socket and another application connects to it, when the kernel is done executing, it has no way of keeping the connection open and must close the socket. This could be a undesirable feature in some applications as the connected application would have to reconnect. The opposite scenario, where a kernel creates a connecting socket, is also problematic as creating, connecting and then destroying TCP sockets might create extra overhead and have negative effect on network performance.

P2G is targeting streaming applications, which often make use of network sockets and therefore it must be able to handle them in a graceful manner. In section 6.3 we discussed wrapping I/O functionality in a special P2G library executed by the runtime, and this solution could also help with keeping I/O state from looping kernel executions. In the case of source- and sink-kernels, the library functions would be used to prevent a kernel instance for occupying a P2G worker thread by handling the real I/O operations on behalf of the kernel code. It is obvious that since these function already handle OS resources, they can also be utilized for keeping state associated with the same OS system calls. With respect to keeping information about file positions, appending data to a file when writing is a special case as opening files in append mode solves this. However, opening and closing of the file descriptor between every iteration is still necessary, as discussed in section 4.5.

This section discussed the implication of handling I/O when accessing a data source in an iterative fashion. The next section deals with which machines the I/O resource are connected to in a distributed environment.

6.4.2 Master Only I/O

The current implementation of P2G lets developers write kernels in a general way, there is no language constructs specifying where a kernel must be executed, it can be executed on any execution node. We believe this gives the developer freedom to focus on modelling and exploiting parallelism in his workload's algorithm. Concerns about how and where a kernel is executed complicates writing of distributed workloads and can distract the developer.

Perhaps the simplest way of keeping the simplistic model in a distributed environment is by only allowing I/O operations to be done on the master node. This would put a restriction on the P2G framework, but is it an unreasonable restriction, which excludes certain workloads. Perhaps of interest, is also how a developer can work around such limitation.

As far as we could think of, there are two factors which point towards using multiple computers to handle I/O, *(1)* by using many I/O devices there might be limitations on a single computer, and *(2)* the devices producing I/O streams are physically so far apart that they need to be connected to different computers, only reachable by network access.

The number of I/O devices used and their spatial placement is dependent on what type of application P2G is used for, and not easy to predict; with a low number of and close I/O resources we do not see any reason for not restricting I/O to the master node, especially because implementing distributed I/O most likely is much more complex that restricting I/O to the master node. To work around the restriction, one could create workloads which only purpose are to provide I/O for the main workload. This approach could use the I/O resource every distributed node definitly have in common, network interfaces.

To allow interactive P2G application and feedback from the execution, keyboard input (*stdin*) and terminal output (*stdout*) needs to be redirected to one computer. These two I/O resources is perhaps the most natural resources to restrict to the master node (initiating the execution).

6.4.3 Machine Bound I/O

We noted that kernels accessing I/O can be distributed and end up executing on computers where the I/O resources they depend on are not present, if no measures to prevent this are taken. The previous section discussed restricting I/O to the master node, another solution is to bind kernels that access I/O to the computer where the I/O resources are present and avoid the problem.

The solution sounds simple, but it requires that the developer has great knowledge about the execution environment and that there are ways to uniquely identify an execution node. A developer can use the IP address of a machine for identification, but this makes the workload rigid and also breaks the generality of how a workload is written. For example, a workload written for one setup can perhaps not be used in another similar setup if it is not identical, and if the I/O machine changes its IP address the workload needs to be recompiled. That said, configuration files can be used to create a mapping between machine identification and where to pin it.

What is also interesting to discuss is how using I/O on multiple computer affects the elasticity and fault tolerance of P2G. Elasticity, or P2G ability to issue more worker threads on distributed execution nodes in case of poor performance, can be hurt as these worker threads needs to be pinned to specific machines and cannot be distributed and executed anywhere else.

Comparing this solution to the solution where I/O was restricted only to the master node, a recommendable feature of P2G, fault tolerance, might also be harder to achieve. The more computers a distributed execution system is dependent on, the more difficult it can be make the system fault tolerant.

6.4.4 Distributed I/O

We have identified two problems with I/O resources in P2G, *(1)* an age-loop cannot keep state, which prohibits reuse of file descriptors used in file reading or writing, network communication, and *(2)* in distributed scenarios there is a problem accessing I/O resources that are bound to a specific computer.

The general way of writing kernels make the kernels easy to distribute, they are not specific to one computer and can be executed on any. We build on this when we in this section explore distributed I/O. As we see it, there are two prominent solutions of solving distributed I/O in P2G, either by introducing a distibuted file system or by creating special I/O-fields.

Distributed Filesystem

One could adapt the solution of a distributed file system used in both Dryad and MapReduce, and then map other I/O resources, i.e., create virtual files of keyboard, terminal, or video input on the file system, making it available for every kernel. The virtual files would then be accessed by calling P2G library functions in the kernel code block.

See examples of how a developer could use these function calls in listing 6.11.

```
Sink:
  local int8[128] data_;
  fetch data_ = data;

  %{
    // Write 128 bytes to a file
    p2g_write("/path/to/file", data_, 128);
  %}

Interactive:
  local int8 char_;

  %{
    // Read one character from keyboard
    p2g_read("/path/to/stdin", char_, 1);

    // Write line to terminal
    p2g_write("/path/to/stdout", "Hello, World!\n");
  %}
```

Listing 6.11: Examples of P2G libary calls

This solution addresses the problem with file descriptors and state as that information would be kept in the virtual file system and abstracted away from the kernels. The virtual file system would need to handle potential synchronization between kernels reading from and writing to the same shared file, this is similar to the concurrency discussion in section 2.3.

Special IO Fields

A second solution, could be to give the developer special global I/O fields, and use fetch- and store-operations for reading from them and writing to them in a kernel, see below for code examples.

```
// Create a file field called "output" with
// write access.

file "/path/to/file" output;

Sink:
  local int8[128] data_;
  fetch data_ = data;

  %{
    //  Use "data_" as any normal field array...
  %}

  store output = data_;

// Create both standard input and output
// fields, called "keyboard" and "terminal".

stdin keyboard;
stdout terminal;

Interactive:
  local int8 char_;
  fetch char_ = keyboard;

  local int8[] hello_;

  %{
    // Use "hello_" as any normal field array
    // and insert "Hello, World!\n" ...
  %}

  store terminal = hello_;
```

Listing 6.12: Special IO Fields

With this solution, one can utilize the field architecture already created for dissemnination of data through the *storage manager*.

Comparing special I/O fields to a distributed file system, a distributed file system is easier to grasp, for most developers, mainly because a field is a new P2G programming construct. However, as we stated earlier in this chapter, using global fields is not much different from using a global variable in other languages.

Both global I/O fields and a distributed file system needs configuration for mapping

I/O resources that are not standard OS resources, e.g., when using driver libraries for special hardware such as cameras or other kinds of sensors. Either way, such configuration could be an abstraction level between kernels and distributed I/O, meaning that once the I/O resource is configured, the developer does not need to take into consideration which node the kernel is executed on.

Earlier, we discussed wrapping I/O calls in P2G functions, both to prevent blocking I/O from occupying execution node's worker threads and for keeping I/O in between iterating kernel executions. The usage of a distributed file system outlined in the previous section (listing 6.11) maps well with these thought. However, we do believe that global I/O can handle the same challenges, in either way, it is the runtime that handles the I/O flow.

There are also other challenges with distributed I/O, one of them is network throughput. Large data set of raw unprepared information sent over the network can create bottlenecks or flood links. Choosing one distributed I/O solution over another, needs a thorough analysis and is subject for further work, but what we believe tip the scales in favour of global I/O fields are two factors. First, we believe using global I/O fields are more aligned with the general P2G way of thinking, kernels using I/O should be scheduled when its I/O data dependencies are fulfilled and not "try and potentially block". Secondly, we believe that using global I/O fields makes it easier to do I/O depedency analysis. Such analysis give the runtime better overview of which kernels are using I/O, which again can be of interest to the High-Level Sheduler (HLS) when it is making decisions, especially since it also has information about each execution node's network throughput (section 3.1).

6.5 Miscellaneous

6.5.1 Kernel Namespaces

In C++, Java and C#, some form of namespace is used to separate code and data structures in different contexts, and can be a way of organizing source code. In Kernel Language there are no contexts, every field and kernel is in the same namespace. We believe that introducing some namespaces or contexts in Kernel Language could be beneficial. First, it would allow the developer to organize code, and secondly it could give the runtime and schedulers hints about which kernels and fields that are tightly connected and should be bundled together, e.g., it could prevent a graph partitioning algorithm making bad decisions.

6.5.2 Setting Field Sizes

When developing in Kernel Languge, we did not always know the how large the field arrays would be up-front, which meant that we had to leave them blank and use `put()`

to dynamically grow them. However, often we could find their exact sizes early in the kernel code block. This observation made us want a P2G function call that could set the correct size of the field so that we could use the faster `puts()`-call instead of `put()`.

6.5.3 Other

Fault Tolerance: To make sure that execution of workloads finish, even though changes in topology and failures occur, fault tolerance is needed. As we noted earlier in section 2.7, MapReduce and Dryad both have fault tolerance built-in. We believe it is an imporant feature which would be good to see in P2G as well.

Fault Tolerance: A nice-to-have feature in Kernel Language would be enumeration, similar to the enumerated type `enum` in C++ or macros in C.

P2G Functions: The implementation of the P2G `get()` and `set()` function calls is not optimal. These functions should be in a namespace or prefixed in some way as they interfere with other functions with the same name, e.g., when calling function from a library or legacy code.

6.6 Summary

In this chapter, we discussed and touched upon some potential areas of improvement in the current P2G prototype, and also proposed some new solutions, e.g., in data structures, and I/O handlind and distribution. The next chapter, we summarize our work and conclude this thesis.

Chapter 7

Conclusion

7.1 Summary

During this work, we have looked into principles of parallel and distributed execution, and reviewed some of the options a developer has when writing such software. Frameworks relieving the burden of low-level concurrent programming increase the efficiency of the developer, but from a multimedia point of view, current frameworks are missing some features, such as loops, branching and soft real-time support, or do not have openly available implementations.

Being able to express workloads with loops, inherent multimedia support, and the ability to do both task parallelism and data parallelism in an easy manner separates the P2G framework from other distributed execution systems like MapReduce and Dryad. The development of P2G is ongoing work and it has a constantly moving target as new requirements are added continuously. Verification of its concepts and programming model is therefore important in this process.

Our task have been to implement multimedia workloads and evaluate the P2G concept and prototype in its current form compared to more low-level approaches to concurrency. This work includes a MJPEG encoder adapted to P2G, a SIFT pipeline and partial implementation written from scratch.

7.2 Contributions and Conclusion

During our work designing and implementing workloads in P2G, we gained a lot of experience. Developing concurrent workloads in P2G is effective compared to low-level concurrent intrinsics, like threads, locks and message passing. The Kernel Language is powerful, and it provides good abstractions and support to developers that needs to express both task parallelism and data parallelism.

The MJPEG workload's parallel design and P2G implementation have been tested and benchmarked, and have also contributed to a paper [5] pending review. Additionally, these ideas and results have been presented and discussed in a poster-session at Eurosys. The results show that inherent parallelism in MJPEG is possible to exploit using P2G, and that it scales this workload and achieves an speed-up when executing in parallel. Furthermore, we have provided a design and partial implementation of a parallel execution of SIFT feature extraction. Through our work, we also indentified potential limitations and possible improvements in the current prototype of P2G and proposed some new solutions in the discussion.

7.3 Future Work

For future work, it would be interesting to investigate a distributed multiset data structure as discussed in section 6.2.3. How can distributed multiset elements affect multiset operations, e.g., insert or delete a element. Future work also includes supporting more customized, user specified data structures, like C structs and perhaps also C++ objects, and investigate how this would affect distribution and slicing, with respect to kernel instances and data parallelism, but also distribution and heterogeneous architectures.

Using index-expressions to slice data is another concept that needs further work. A clear definition of how index-expressions should help the developer more precisely express the true data dependencies and relationship of data sets, perhaps exploring index-expressions in multi-dimensional fields. There is also further work on I/O handling in P2G. The implications of distribution are prominent when it comes to reading and writing I/O, but preventing blocking I/O from using available computing resources inefficiently is also important.

Furter, the P2G implementation of SIFT should be finished, tested and evaluated against other implementations, both in terms of keypoint accuracy and execution time performance. In the MJPEG workload, we could expose other parallel parts, e.g., entropy coding. At last, while developing P2G and Kernel Language, it is important to keep implementing workloads in P2G to verify and validate its concepts as it can reveal other operations and functionality that is difficult to predict.

Chapter 8

Appendix

8.1 MJPEG: Motion JPEG

```
uvDct:
  age a;

  index x, y;

  local int8[] uvInput_;
  local int16[64] uvResult_;

  fetch uvInput_ = uvInput(a)[x][y];

  %{
    uint32_t blockSize = extent(uvInput_, 0);
    uint8_t *UV = new uint8_t[blockSize];
    int16_t *uvDct = new int16_t[blockSize];

    for (uint32_t i = 0; i < blockSize; i++)
      UV[i] = get(uvInput_, i);

    // dct and quantization (uQantTbl == vQuantTbl)
    Workloads::dctQuantize2d(UV, 8, 8, uvDct, uQuantTbl);

    for (uint32_t i = 0; i < blockSize; i++)
      puts(uvResult_, uvDct[i], i);

    delete UV;
    delete uvDct;
  %}

  store uvResult(a)[x][y] = uvResult_;
```

Listing 8.1: How the `uvDvt`-kernel slices a 2D CbCr (uv) field array

Bibliography

[1] Jeffrey Dean and Sanjay Ghemawat. Mapreduce: simplified data processing on large clusters. *Commun. ACM*, 51:107–113, January 2008.

[2] Michael Isard, Mihai Budiu, Yuan Yu, Andrew Birrell, and Dennis Fetterly. Dryad: distributed data-parallel programs from sequential building blocks. *SIGOPS Oper. Syst. Rev.*, 41:59–72, March 2007.

[3] John E. Stone, David Gohara, and Guochun Shi. Opencl: A parallel programming standard for heterogeneous computing systems. *IEEE Des. Test*, 12:66–73, May 2010.

[4] John Nickolls, Ian Buck, Michael Garland, and Kevin Skadron. Scalable parallel programming with cuda. *Queue*, 6:40–53, March 2008.

[5] Håvard Espeland, Paul B. Beskow, Håkon K. Stensland, Preben N. Olsen, Ståle Kristoffersen, Carsten Griwodz, and Pål Halvorsen. P2g: A framework for distributed real-time processing of multimedia data.

[6] David G. Lowe. Distinctive image features from scale-invariant keypoints. *International Journal of Computer Vision*, 60:91–110, 2004. 10.1023/B:VISI.0000029664.99615.94.

[7] P.B. Beskow, H. Espeland, H.K. Stensland, P.N. Olsen, S. Kristoffersen, E.A. Kristiansen, C. Griwodz, and Halvorsen. Distributed real-time processing of multimedia data with the p2g framework.

[8] G. K. Manacher. Production and stabilization of real-time task schedules. *J. ACM*, 14:439–465, July 1967.

[9] William Thies, Michal Karczmarek, and Saman P. Amarasinghe. Streamit: A language for streaming applications. In *Proceedings of the 11th International Conference on Compiler Construction*, CC ’02, pages 179–196, London, UK, 2002. Springer-Verlag.

[10] Gene M. Amdahl. Validity of the single processor approach to achieving large scale computing capabilities. In *Proceedings of the April 18-20, 1967, spring joint computer conference*, AFIPS ’67 (Spring), pages 483–485, New York, NY, USA, 1967. ACM.

[11] Michael I. Gordon, William Thies, and Saman Amarasinghe. Exploiting coarse-grained task, data, and pipeline parallelism in stream programs. *SIGOPS Oper. Syst. Rev.*, 40:151–162, October 2006.

[12] Guilherme Ottoni, Ram Rangan, Adam Stoler, and David I. August. Automatic thread extraction with decoupled software pipelining. In *Proceedings of the 38th annual IEEE/ACM International Symposium on Microarchitecture*, MICRO 38, pages 105–118, Washington, DC, USA, 2005. IEEE Computer Society.

[13] William Thies, Vikram Chandrasekhar, and Saman 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, Washington, DC, USA, 2007. IEEE Computer Society.

[14] A. S. Tanenbaum. *Modern Operating systems: Second Edition*. Prentice-Hall, New Jersey, 2001.

[15] Paul E. Mckenney et al. Read-copy update. In *In Ottawa Linux Symposium*, pages 338–367, 2001.

[16] Maurice Herlihy and J. Eliot B. Moss. Transactional memory: architectural support for lock-free data structures. *SIGARCH Comput. Archit. News*, 21:289–300, May 1993.

[17] Nir Shavit and Dan Touitou. Software transactional memory. In *Proceedings of the fourteenth annual ACM symposium on Principles of distributed computing*, PODC '95, pages 204–213, New York, NY, USA, 1995. ACM.

[18] Manuel Fähndrich, Mark Aiken, Chris Hawblitzel, Orion Hodson, Galen Hunt, James R. Larus, and Steven Levi. Language support for fast and reliable message-based communication in singularity os. *SIGOPS Oper. Syst. Rev.*, 40:177–190, April 2006.

[19] David Wentzlaff and Anant Agarwal. Factored operating systems (fos): the case for a scalable operating system for multicores. *SIGOPS Oper. Syst. Rev.*, 43:76–85, April 2009.

[20] Andrew Baumann, Paul Barham, Pierre-Evariste Dagand, Tim Harris, Rebecca Isaacs, Simon Peter, Timothy Roscoe, Adrian Schüpbach, and Akhilesh Singhania. The multikernel: a new os architecture for scalable multicore systems. In *Proceedings of the ACM SIGOPS 22nd symposium on Operating systems principles*, SOSP '09, pages 29–44, New York, NY, USA, 2009. ACM.

[21] M.J. Flynn. Very high-speed computing systems. *Proceedings of the IEEE*, 54(12):1901–1909, dec. 1966.

[22] J. M. Tendler, J. S. Dodson, J. S. Fields, H. Le, and B. Sinharoy. Power4 system microarchitecture. *IBM J. Res. Dev.*, 46:5–25, January 2002.

[23] D. Molka, D. Hackenberg, R. Schone, and M.S. Muller. Memory performance and cache coherency effects on an intel nehalem multiprocessor system. In *Parallel Architectures and Compilation Techniques, 2009. PACT '09. 18th International Conference on*, pages 261–270, sept. 2009.

[24] Shekhar Borkar. Thousand core chips: a technology perspective. In *Proceedings of the 44th annual Design Automation Conference*, DAC '07, pages 746–749, New York, NY, USA, 2007. ACM.

[25] Rob F. van der Wijngaart, Timothy G. Mattson, and Werner Haas. Light-weight communications on intel's single-chip cloud computer processor. *SIGOPS Oper. Syst. Rev.*, 45:73–83, February 2011.

[26] Håkon Kvale Stensland, Håvard Espeland, Carsten Griwodz, and Pål 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, New York, NY, USA, 2010. ACM.

[27] T. Chen, R. Raghavan, J. N. Dale, and E. Iwata. Cell broadband engine architecture and its first implementation – a performance view. *IBM Journal of Research and Development*, 51(5):559–572, sept. 2007.

[28] M.D. Hill and M.R. Marty. Amdahl's law in the multicore era. *Computer*, 41(7):33 –38, july 2008.

[29] J. Dongarra, S. Huss-Lederman, S. Otto, M. Snir, and D. Walker. *MPI: The complete reference*. The MIT Press, 1996.

[30] William Gropp, Ewing Lusk, Nathan Doss, and Anthony Skjellum. A high-performance, portable implementation of the mpi message passing interface standard. *Parallel Computing*, 22(6):789–828, 1996.

[31] Jim Larson. Erlang for concurrent programming. *Commun. ACM*, 52:48–56, March 2009.

[32] Simon Peyton Jones, Andrew Gordon, and Sigbjorn Finne. Concurrent haskell. In *Proceedings of the 23rd ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, POPL '96, pages 295–308, New York, NY, USA, 1996. ACM.

[33] Anthony Discolo, Tim Harris, Simon Marlow, Simon Jones, and Satnam Singh. Lock free data structures using stm in haskell. In Masami Hagiya and Philip Wadler, editors, *Functional and Logic Programming*, volume 3945 of *Lecture Notes in Computer Science*, pages 65–80. Springer Berlin / Heidelberg, 2006. 10.1007/117374146.

[34] Cristian Perfumo, Nehir Sönmez, Srdjan Stipic, Osman Unsal, Adrián Cristal, Tim Harris, and Mateo Valero. The limits of software transactional memory (stm): dissecting haskell stm applications on a many-core environment. In *Proceedings of the 5th conference on Computing frontiers*, CF '08, pages 67–78, New York, NY, USA, 2008. ACM.

[35] L. Dagum and R. Menon. Openmp: an industry standard api for shared-memory programming. *Computational Science Engineering, IEEE*, 5(1):46–55, jan-mar 1998.

[36] Kevin O'Brien, Kathryn O'Brien, Zehra Sura, Tong Chen, and Tao Zhang. Supporting openmp on cell. *International Journal of Parallel Programming*, 36:289–311, 2008. 10.1007/s10766-008-0072-7.

[37] Karl FÃ¼rlinger and Michael Gerndt. Analyzing overheads and scalability characteristics of openmp applications. In Michel DaydÃ©, JosÃ© Palma, Alvaro Coutinho, Esther Pacitti, and JoÃ£o Lopes, editors, *High Performance Computing for Computational Science - VECPAR 2006*, volume 4395 of *Lecture Notes in Computer Science*, pages 39–51. Springer Berlin / Heidelberg, 2007.

[38] Michael Armbrust, Armando Fox, Rean Griffith, Anthony D. Joseph, Randy Katz, Andy Konwinski, Gunho Lee, David Patterson, Ariel Rabkin, Ion Stoica, and Matei Zaharia. A view of cloud computing. *Communications of the ACM*, 53:50–58, April 2010.

[39] Felix C. Gärtner. Fundamentals of fault-tolerant distributed computing in asynchronous environments. *ACM Computing Surveys (CSUR)*, 31:1–26, March 1999.

[40] Hung-chih Yang, Ali Dasdan, Ruey-Lung Hsiao, and D. Stott Parker. Map-reduce-merge: simplified relational data processing on large clusters. In *Proceedings of the 2007 ACM SIGMOD international conference on Management of data*, SIGMOD '07, pages 1029–1040, New York, NY, USA, 2007. ACM.

[41] M. de Kruijf and K. Sankaralingam. Mapreduce for the cell broadband engine architecture. *IBM Journal of Research and Development*, 53(5):10:1–10:12, sept. 2009.

[42] Sanjay Ghemawat, Howard Gobioff, and Shun-Tak Leung. The google file system. *SIGOPS Oper. Syst. Rev.*, 37:29–43, October 2003.

[43] David Tarditi, Sidd Puri, and Jose Oglesby. Accelerator: using data parallelism to program gpus for general-purpose uses. *SIGOPS Oper. Syst. Rev.*, 40:325–335, October 2006.

[44] William R. Mark, R. Steven Glanville, Kurt Akeley, and Mark J. Kilgard. Cg: a system for programming graphics hardware in a c-like language. *ACM Trans. Graph.*, 22:896–907, July 2003.

[45] G. Kahn. The semantics of a simple language for parallel programming. In *In Information Processing '74: Proceedings of the IFIP Congress*, pages 471–475. North-Holland, 1974.

[46] Vrba Ž., Pal Halvorsen, Carsten Griwodz, Paul Beskow, and Dag Johansen. The nornir run-time system for parallel programs using kahn process networks. *Network and Parallel Computing Workshops, IFIP International Conference on*, 0:1–8, 2009.

[47] Bugra Gedik, Henrique Andrade, Kun-Lung Wu, Philip S. Yu, and Myungcheol Doo. Spade: the system s declarative stream processing engine. In *Proceedings of the 2008 ACM SIGMOD international conference on Management of data*, SIGMOD '08, pages 1123–1134, New York, NY, USA, 2008. ACM.

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

[49] Ronnie Chaiken, Bob Jenkins, Per-Åke Larson, Bill Ramsey, Darren Shakib, Simon Weaver, and Jingren Zhou. Scope: easy and efficient parallel processing of massive data sets. *Proc. VLDB Endow.*, 1:1265–1276, August 2008.

[50] Tinne Tuytelaars and Krystian Mikolajczyk. Local invariant feature detectors: a survey. *Found. Trends. Comput. Graph. Vis.*, 3:177–280, July 2008.

[51] K. Mikolajczyk and C. Schmid. A performance evaluation of local descriptors. *Pattern Analysis and Machine Intelligence, IEEE Transactions on*, 27(10):1615–1630, 2005.

[52] Herbert Bay, Tinne Tuytelaars, and Luc Van Gool. Surf: Speeded up robust features. In Ales Leonardis, Horst Bischof, and Axel Pinz, editors, *Computer Vision - ECCV 2006*, volume 3951 of *Lecture Notes in Computer Science*, pages 404–417. Springer Berlin / Heidelberg, 2006. 10.1007/1174402332.

[53] Johannes Bauer, Niko Sunderhauf, and Peter Protzel. Comparing several implementations of two recently published feature detectors. *Proc. Int. Conf. Intelligent and Autonomous Systems IAV*, 2007.

[54] Sudipta Sinha, Jan-Michael Frahm, Marc Pollefeys, and Yakup Genc. Feature tracking and matching in video using programmable graphics hardware. *Machine Vision and Applications*, 22:207–217, 2011. 10.1007/s00138-007-0105-z.

[55] Leonardo Chang and JosÃ© HernÃ¡ndez-Palancar. A hardware architecture for sift candidate keypoints detection. In Eduardo Bayro-Corrochano and Jan-Olof Eklundh, editors, *Progress in Pattern Recognition, Image Analysis, Computer Vision, and Applications*, volume 5856 of *Lecture Notes in Computer Science*, pages 95–102. Springer Berlin / Heidelberg, 2009. 10.1007/978-3-642-10268-411.

[56] Qi Zhang, Yurong Chen, Yimin Zhang, and Yinlong Xu. Sift implementation and optimization for multi-core systems. In *Parallel and Distributed Processing, 2008. IPDPS 2008. IEEE International Symposium on*, pages 1 –8, april 2008.

[57] Jean-Michel Morel and Guoshen Yu. Asift: A new framework for fully affine invariant image comparison. *SIAM J. Img. Sci.*, 2:438–469, April 2009.

[58] Yan Ke and R. Sukthankar. Pca-sift: a more distinctive representation for local image descriptors. In *Computer Vision and Pattern Recognition, 2004. CVPR 2004. Proceedings of the 2004 IEEE Computer Society Conference on*, volume 2, pages II–506 – II–513 Vol.2, june-2 july 2004.

[59] Donald E. Knuth. *The Art of Computer Programming Volume 2: Seminumerical Algorithms, Third Edition*. Bonn: Addison-Wesley, 1998.

[60] Donald E. Knuth. *The Art of Computer Programming Volume 1: Fundamental Algorithms, Third Edition*. Bonn: Addison-Wesley, 1997.