Aalto University
School of Science
Degree Programme of Computer Science and Engineering

Matias Piispanen

Simulating timing and energy consumption of accelerated processing

Master's Thesis
Espoo, May 11, 2014

Supervisor: Professor Heikki Saikkonen
Instructor: D.Sc. Vesa Hirvisalo

Aalto University
School of Science
Degree Programme of Computer Science and Engineering

ABSTRACT OF
MASTER'S THESIS

Author:	Matias Piispanen		
Title:	Simulating timing and energy consumption of accelerated processing		
Date:	May 11, 2014	**Pages:**	73
Professorship:	Software Technology	**Code:**	T-106
Supervisor:	Professor Heikki Saikkonen		
Instructor:	D.Sc. Vesa Hirvisalo		

As the increase in the sequential processing performance of general-purpose central processing units has slowed down dramatically, computer systems have been moving towards increasingly parallel and heterogeneous architectures. Modern graphics processing units have emerged as one of the first affordable platforms for data-parallel processing. Due to their closed nature, it has been difficult for software developers to observe the performance and energy efficiency characteristics of the execution of applications of graphics processing units.

In this thesis, we have explored different tools and methods for observing the execution of accelerated processing on graphics processing units. We have found that hardware vendors provide interfaces for observing the timing of events that occur on the host platform and aggregated performance metrics of execution on the graphics processing units to some extent. However, more fine-grained details of execution are currently available only by using graphics processing unit simulators.

As a proof-of-concept, we have studied a functional graphics processing unit simulator as a tool for understanding the energy efficiency of accelerated processing. The presented energy estimation model and simulation method has been validated against a face detection application. The difference between the estimated and measured dynamic energy consumption in this case was found to be 5.4%. Functional simulators appear to be accurate enough to be used for observing the energy efficiency of graphics processing unit accelerated processing in certain use-cases.

Keywords:	GPU, CUDA, OpenCL, parallel processing, energy efficient computing, high performance embedded computing, GPU compute, GPGPU
Language:	English

Aalto-yliopisto
Perustieteiden korkeakoulu
Tietotekniikan tutkinto-ohjelma

DIPLOMITYÖN
TIIVISTELMÄ

Tekijä: Matias Piispanen

Työn nimi:
Kiihdytetyn laskennan ajoituksen ja energiankulutuksen simulointi

Päiväys:	11. toukokuuta 2014	**Sivumäärä:**	73
Professuuri:	Ohjelmistotekniikka	**Koodi:**	T-106
Valvoja:	Professori Heikki Saikkonen		
Ohjaaja:	TkT Vesa Hirvisalo		

Suorittimien sarjallisen suorituskyvyn kasvun hidastuessa tietokonejärjestelmät ovat siirtymässä kohti rinnakkaislaskentaa ja heterogeenisia arkkitehtuureja. Modernit grafiikkasuorittimet ovat yleistyneet ensimmäisinä huokeina alustoina yleisluonteisen kiihdytetyn datarinnakkaisen laskennan suorittamiseen. Grafiikkasuorittimet ovat usein suljettuja alustoja, minkä takia ohjelmistokehittäjien on vaikea havainnoida tarkempia yksityiskohtia suorituksesta liittyen laskennan suorituskykyyn ja energian kulutukseen.

Tässä työssä on tutkittu erilaisia työkaluja ja tapoja tarkkailla ohjelmien kiihdytettyä suoritusta grafiikkasuorittimilla. Laitevalmistajat tarjoavat joitakin rajapintoja tapahtumien ajoituksen havainnointiin sekä isäntäalustalla että grafiikkasuorittimella. Laskennan tarkempaan havainnointiin on kuitenkin usein käytettävä grafiikkasuoritinsimulaattoreita.

Työn kokeellisessa osuudessa työssä on tutkittu funktionaalisten grafiikkasuoritinsimulaattoreiden käyttöä työkaluna grafiikkasuorittimella kiihdytetyn laskennan energiantehokkuuden arvioinnissa. Työssä on malli grafiikkasuorittimen energian kulutuksen arviontiin. Arvion validointiin on käytetty kasvontunnistussovellusta. Mittauksissa arvioidun ja mitatun energian kulutuksen eroksi mitattiin 5.4%. Funktionaaliset simulaattorit ovat mittaustemme perusteella tietyissä käyttötarkoituksissa tarpeeksi tarkkoja grafiikkasuorittimella kiihdytetyn laskennan energiatehokkuuden arviointiin.

Asiasanat: GPU, CUDA, OpenCL, rinnakkaislaskenta, energiatehokas suorittaminen, korkean suorituskyvyn sulautettu laskenta, grafiikkasuoritinlaskenta, GPGPU

Kieli: Englanti

Acknowledgements

I would like to thank my instructor, Vesa Hirvisalo, for all the support and guidance during this thesis and the later stages of my studies. I am especially grateful for getting the opportunity to learn about different aspects of high-performance embedded computing beyond the offerings of regular university teaching. Additionally, I would like to thank Professor Heikki Saikkonen for finding the time to supervise my thesis.

I would also like to extend my thank you to all my colleagues at the department of Computer Science and Engineering at Aalto, especially the members of the Embedded Software Group for all the help and support I received during the thesis, and for all the coffee room discussions that were often entertaining and sometimes informative.

Finally, I would like to thank my parents for their patience, and for the overwhelming support I have received throughout my life.

Trondheim, May 11, 2014

Matias Piispanen

Abbreviations and Acronyms

API	Application programming interface
CISC	Complex instruction set computing
CPU	Central processing unit
CTA	Cooperative thread array
CUDA	Compute unified device architecture
DMA	Direct memory access
DSP	Digital signal processor
DVFS	Dynamic voltage and frequency scaling
FIFO	First in, first out
FPGA	Field-programmable gate array
GPU	Graphics processing unit
GPGPU	General-purpose graphics processing unit computing
HPEC	High performance embedded computing
ioctl	intout/ouput control
ISA	Instruction set architecture
JIT	Just-in-time
MIMD	Multiple instruction, multiple data
MISD	Multiple instruction, single data
OpenCL	Open Computing Language
RISC	Reduced intruction set computing
SIMD	Single instruction, multiple data
SIMT	Singe instruction, multple thread
SISD	Single instruction, single data

Contents

Abbreviations and Acronyms **5**

1 Introduction **9**

2 Background **11**
2.1 Parallel computing 11
2.1.1 Trends in parallel computing 11
2.1.2 Granularity of parallelism 12
2.1.2.1 Instruction-level parallelism 12
2.1.2.2 Task parallelism 13
2.1.2.3 Data parallelism 14
2.1.3 Flynn's taxonomy 14
2.1.4 Amdahl's law 15
2.1.5 Multithreading on symmetric multiprocessors 15
2.1.6 SIMD computation 16
2.2 Efficient accelerated processing 17
2.2.1 Accelerators 17
2.2.2 Energy efficient computing 17
2.3 Execution timing 18
2.3.1 Dynamic program analysis 18
2.3.2 Simulators 19
2.4 Linux device driver 20
2.5 Energy consumption 21
2.5.1 Power and energy 21
2.5.2 Power consumption 22

3 GPU computation environment **23**
3.1 GPU accelerated processing 23
3.1.1 Graphics processing unit architectures 23
3.1.1.1 General graphics processing unit architecture 23
3.1.1.2 Mobile graphics processing units 25

3.1.2 Parallel compute model on graphics processing units . 26
3.1.3 Programming models for GPU accelerated processing . 28
3.1.4 GPU execution context and scheduling 29
3.2 Host platform for GPU accelerated processing 29
3.2.1 Program compilation workflow 29
3.2.2 Runtime library and GPU device driver 31
3.2.2.1 GPU device driver and CUDA runtime library 31
3.2.2.2 Task scheduling 32
3.3 Tracing of accelerated computing 32
3.3.1 Tracing of host operations 33
3.3.2 Tracing of operations on GPU 33
3.3.3 Tracing with TAU Parallel Performance System 34
3.3.4 Profiling of GPU accelerated processing 35
3.3.4.1 CUDA Profiling Tools Interface (CUPTI) . . 35

4 Measuring and simulating GPUs 36
4.1 Instrumentation 36
4.1.1 Timing and energy model 36
4.1.2 Instrumentation 37
4.2 GPU Simulators 38
4.2.1 GPU simulator structure 38
4.2.2 GPU simulator survey 39
4.2.2.1 Barra 39
4.2.2.2 GPGPU-sim 39
4.2.2.3 Multi2Sim 39
4.2.2.4 GPU Ocelot 40
4.3 Parallel GPU simulation framework 40
4.3.1 Parallel simulation 40
4.3.2 GPU accelerated many-core simulator 41
4.3.3 Extending QEMU with a parallel simulator 43

5 Benchmarks 46
5.1 Microbenchmarks 46
5.2 Application benchmark 47
5.2.1 Motivation 47
5.2.2 Computer vision 47
5.2.2.1 Software modifications 48

6 Evaluation 50
6.1 Energy estimation method 50
6.1.1 Power model 50

6.1.1.1 GPU computation power model 51
6.1.1.2 PCI Express memory transfer power model . 54
6.1.1.3 GPU operating state modeling 54
6.1.2 Measurement setup 55
6.1.3 Trace analysis 56
6.2 Results 57
6.2.1 Microbenchmarks 57
6.2.2 Face detection 60

7 Simulating GPU energy consumption 62
7.1 GPU energy consumption observations 62
7.2 Simulating GPU energy consumption 63
7.3 Simulating future GPU architectures 64

8 Summary 66

Chapter 1

Introduction

The time of continuous increase in the serial execution performance of processors has come to an end. It is no longer possibly to simply crank up the clock speeds of processors to gain more performance. We have reached the limits imposed by physical laws. It is increasingly more difficult to cool down modern high-end processors, which is why it is important to decrease their energy consumption. The problem chip manufacturers now face is not how to make faster processors but how to make more efficient ones.

The landscape of high-performance computing is inevitably changing. A clear paradigm shift is taking place from sequential processing towards parallelism. The first step was the introduction of multi-core processors in the form on symmetric multiprocessing, but this approach has proven not to be scalable. While there are fields where domain specific many-core accelerators have already been used, it was not until recently when many-core processing truly hit the mainstream. This was made possible by the realisation that there already is a many-core accelerator present in most modern computers, the graphics processing unit (GPU).

The programming models available for graphics processing unit programming were understandably meant for the rendering of graphics. The first attempt to bring more general-purpose computing to GPUs was to map computation to graphical primitives and performing computation by essentially drawing the primitives on the screen. This was a very tedious and ineffective way to perform computation, which led to more suitable programming models and languages, such as the CUDA and OpenCL standards.

The problem with developing GPU accelerated programs lies in the closed nature of GPUs. It is difficult to get meaningful feedback of the execution. Essentially GPUs are treated as a black box system that is given program code and data for execution and after a while you receive some sort of an output. Recently GPU vendors have improved the support for program pro-

filing, but these tools and performance counters are meant for debugging system performance only. Especially in the field of embedded computing, programmers are often interested in other matters as well, such as the energy efficiency of processing.

The focus of this thesis is exploring different tools and methods for observing both the host platform and GPU execution. These tools make it possible to profile and instrument GPU accelerated programs to learn more about the timing and energy consumption of program execution. In many cases the only solution is to rely on simulation tools, which may even be desirable for developers as they may not have access to the real hardware they are developing for. We also present a power model that allows the estimation of energy consumption of GPU accelerated programs. The model is validated by comparing it against the energy consumption of real GPU hardware running a real-world application benchmark from the field of computer vision.

We will start off by describing the background information related to this thesis in Chapter 2 that consists mostly of basic concepts of parallel processing, computer architecture and energy consumption of electronics. In Chapter 3 we explain the current computation and programming models of desktop GPUs, as well as explain the structure of software components related to GPU accelerated computing on the host platform, and present some program profiling tools and methods. Then, in Chapter 4 we explore the methods needed to instrument GPU execution, which is needed for the estimation of the energy efficiency of execution. The instrumentation must be generally done by using simulators. The chapter focuses on presenting a number of GPU simulators with a different focus, including a state-of-the-art parallel GPU simulator that is being developed. In Chapter 5 we present the benchmarks needed for configuring and validating the power model presented in Chapter 6. On top of the power model this chapter also presents the measurement setup and the obtained estimated and measured energy consumption results. Finally, in Chapter 7 we discuss the overall landscape of estimating energy consumption of GPU accelerated computing by means of simulation before wrapping up our finding in the summary in Chapter 8.

Chapter 2

Background

This section describes the general background of this thesis. The topics covered in this chapter include basic concepts of parallel processing, computer architecture, simulation, and power and energy consumption of electronics. A well informed reader may skip this section. As a reference to the general concepts and problems of parallel computing you can use the report by Asanovic et al. [2006] on the landscape of parallel computing.

2.1 Parallel computing

2.1.1 Trends in parallel computing

For many years the increase in performance followed the so called Moore's law [Moore, 1965] that states that the number of transistors on integrated circuits tends to increase by the factor of two approximately every two years. Many other characteristics of computers, such as capacities of hard drives, clock speeds of processors and even the overall computational performance, seemed to follow this trend for a long time. It is clear that this sort of exponential growth cannot continue forever due to the laws of physics. In his article "The Free Lunch Is Over" Sutter [2005] famously declared that the time of growth of the serial processing performance of microprocessors is now over, which will force manufacturers and developers to focus on more parallel computing solutions. In case of mobile devices, the available battery power is the limiting factor that will likely force them towards multi-core, and eventually many-core, architectures [van Berkel, 2009].

The trend in recent years has been to increase the number of processing cores in processors instead of raising the clock speeds of processors. It seems that the four gigahertz clock frequency is becoming a glass wall that main-

stream processor manufacturers are not able to exceed without unorthodox cooling methods. Modern computers also usually contain other domain specific processing elements such as graphics processing units and digital signal processors. As the amount of processing elements increases, so does the total potential computing power of computers.

The consensus seems to be that parallel processing will be come more important in the future. Research is already focusing on microprocessors that have hundreds, if not thousands, of low-power cores on the same chip. It is likely that as the price of manufacturing many-core chips decreases, the cores will become more and more specialized cores so that only parts of the chip are active at any given time. Current heterogeneity in systems is mostly limited to having different kinds of processing elements in the system.

2.1.2 Granularity of parallelism

There are many possible ways to achieve parallelism in computing. Parallelism can occur at different levels of granularity. Coarse-grained parallelism is usually more visible to the programmer than fine-grained parallelism. Different types of parallelism are typically divided into classes of instruction-level parallelism, task parallelism and data parallelism.

2.1.2.1 Instruction-level parallelism

Execution on general purpose central processing units (CPU) at a high level of abstraction can be said to consist of three separate phases: instruction fetch, instruction decode and instruction execute. In practice these phases can be further split into multiple stages, such as memory operations, mathematical or logical operations, or other intermediate stages inside a processor. Each of these stages can take one or more clock cycles to execute. The typical length of an execution pipeline in current CPUs can range from two to even thirty stages. [Patterson and Hennessy, 2007, p. 370–374]

The stages in the execution pipeline are ordered and typically different stages use different functional units on the CPU. This makes it possible to have multiple instructions in execution simultaneously on the same processor as long as they are all in a different stage of execution and there are no dependencies between instructions. A new instruction can be issued every cycle as long as the instruction does not depend on the results of any of the instructions currently being executed. The pipeline length also affects how long it takes to fill a pipeline to reach maximum utilization. [Patterson and Hennessy, 2007, p. 370–374, 412–415]

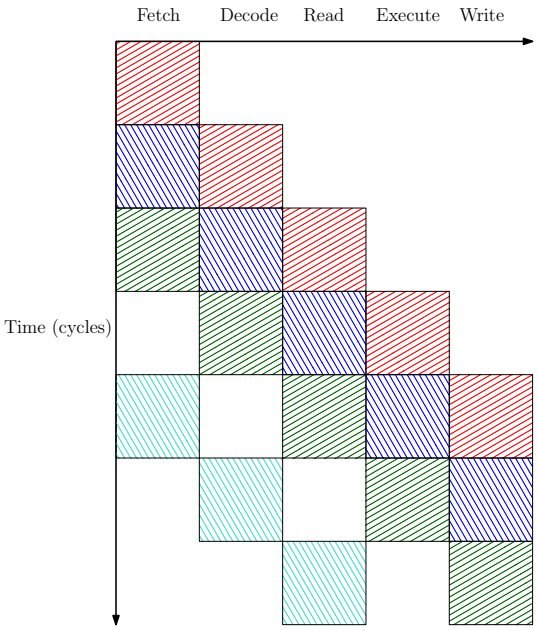


Figure 2.1: Example of pipelined execution. Squares of the same colour represent the execution of the same instruction.

Figure 2.1 shows an example of pipelined execution on a processor that has a five stage execution pipeline. The five stages are instruction fetch, instruction decode, read memory, instruction execute and write-back to memory. The x-axis represents time as clock cycles and y-axis represents the five pipeline stages. Boxes of the same colour represent the execution of a single instruction overtime. If a new instruction can be issued every clock cycle, a processor with five pipeline stages can be executing five different instructions at every given time. If new instructions cannot be issued due to for example data dependencies, it creates a so called bubble in the pipeline and maximum parallelism cannot be reached. In figure 2.1 the last instruction has to wait for one cycle before it can be issued for execution which creates a bubble in the pipeline.

2.1.2.2 Task parallelism

Splitting up a program so that different computational units execute different tasks is called task parallelism. Task parallelism is a more coarse-grained form of parallelism. It is the task of the programmer to write task parallel code by splitting the execution of the program into multiple processes or threads. An example of task parallelism would be dividing the work so that

one thread handles the rendering of graphics on the screen independently and in parallel to other threads in the process. Writing scalable task parallel code is very challenging, especially in presence of communication and synchronization between threads. [Chapman et al., 2007, p. 192]

2.1.2.3 Data parallelism

Often there is a need to perform an identical operation for all the data items in a dataset. The difference to task parallelism is that the work partitioning is such that the same *operation* is performed concurrently and in parallel, but on *different data*, instead of performing completely different tasks. [Chapman et al., 2007, p. 191–192] It is usually the responsibility of the programmer to partition the program execution for data parallelism, but under some circumstances a compiler can find opportunities for data parallelism and compile the program for parallel execution.

2.1.3 Flynn's taxonomy

Flynn's taxonomy [Flynn, 1972] is a traditional classification system for computer processor architectures. The taxonomy is used to classify processors into four categories on two axes, the amount of data being processed and the number of instruction streams being executed. The table of classifications is shown in figure 2.1.

	Single instruction	Multiple instruction
Single data	SISD	MISD
Multiple data	SIMD	MIMD

Table 2.1: Flynn's taxonomy.

Flynn's taxonomy classifies processors into following categories:

- **Single instruction, single data (SISD)** processors are typical sequential processors. A single instruction stream is being executed that operates on a single set of data. Currently high-performance processors are moving towards more parallel architectures.
- **Single instruction, multiple data (SIMD)** processors execute a single instruction stream on multiple processing elements in lockstep, but each processing element operates on its own data. An example of SIMD processing is vector operations that are executed on special vector processing elements on processors. The SIMT execution model presented in 3.1.2 has a close relation to the SIMD model.

- **Multiple instruction, single data (MISD)** processors execute multiple instruction streams on separate processing elements, but they all operate on the same data. In reality there are very few consumer processors of this category. MISD architecture processors are generally used in systems that require high fault tolerance.
- **Multiple instruction, multiple data (MIMD)** processors execute multiple instruction streams on separate processing element and they all operate on their own data. This sort of execution is typically task parallel as described in section 2.1.2.2. [Wolf, 2007, p. 68]

2.1.4 Amdahl's law

It is usually very difficult, if not impossible, to write completely parallel programs. In some cases parts of the program simply must be executed sequentially. For example, synchronization events may cause some threads to stall and wait for other threads to finish their task. The sequential parts of execution often dominates the overall execution time which results in often surprisingly bad overall performance. The guideline for finding the maximum expected improvement through parallelism is called Amdahl's law. [Amdahl, 1967]

Amdahl's law is presented in the formula

$$\text{Speed-up} = \frac{1}{(1-P) + \frac{P}{N}}, \tag{2.1}$$

where P is the proportion of the program that can be made parallel, $(1-P)$ is the proportion of the program that is sequential and N is the number of processing elements available.

As an example, let's observe the speed-up of a program where 75% of the program can be made parallel and it is processed on a four core processor. The expected maximum speed-up in such case given by Amdahl's law is as follows:

$$\frac{1}{(1-\frac{3}{4}) + \frac{\frac{3}{4}}{4}} = 2.286. \tag{2.2}$$

The performance of such program can only be expected to double on a four core processor through parallelization.

2.1.5 Multithreading on symmetric multiprocessors

One of the most common ways for a programmer to achieve parallelism in an application is to write a multithreaded application. Multithreading refers

to applications that consist of multiple instruction streams, called threads, that are executed concurrently and potentially in parallel. The creation and context switching of threads is much faster than those of processes, which is why they are preferred for concurrent computing. Parallelism can be achieved by executing threads on multiple processing elements. [Stallings, 2008, p. 161–165]

The most straightforward use for threads is to exploit *task parallelism* in applications [Wolf, 2007, p. 82]. In such programming model one task is usually assigned to one thread. Regular threads can also be used for *data parallel* computation. Data parallel computing models such as OpenMP follow a so called fork-join programming model, where the initial sequential thread creates a team of worker threads, or *forks*, and the parallel work is divided between the worker threads [Chapman et al., 2007]. When the parallel region ends, computation returns to the original sequential thread.

All threads belonging to the same process share the same resources. This introduces a need for synchronization between threads. Threads should not for example be allowed to write to the same location simultaneously. [Stallings, 2008, p. 166–167] The stalls in execution caused by synchronization essentially make processing sequential, which dramatically affects performance negatively, as stated by Amdahl's law in section 2.1.4

2.1.6 SIMD computation

Many modern processors contain special hardware that can perform SIMD style vector operations. In SIMD computation, a single instruction performs an operation on multiple data items simultaneously. Vector operations are useful because many real-world applications contain computation that are prime candidates for SIMD computation. For example, most operations than manipulate pixel colour values are a good fit for SIMD computation, because pixel colours are typically represented as four separate 8-bit values (red, green, blue and alpha) and the same operation is applied to all four values. Special vector instructions are used for vector operations and it is often the programmers duty to explicitly declare vector operations in the code. [Wolf, 2007, p. 80–81]

2.2 Efficient accelerated processing

2.2.1 Accelerators

Accelerators are computational units in computer systems that are designed to extend the functionality of the primary processor of the computer. Accelerators are designed to perform specific tasks, or at least specific types of tasks, more efficiently than the general-purpose host CPU. In the accelerated computing model, the host CPU offloads parts of the computation to accelerators. The motivation behind using accelerators is gaining either a considerable performance increase or reaching better energy efficiency.

Many different topologies are possible for accelerated systems. Accelerators may locate on the same chip as the primary processor or be on a separate chip attached to the CPU via a bus. Accelerators may either share memory with the primary processor or have their own memory.

Traditionally accelerators have been used to perform relatively simple fixed-function functionality. Examples of domains where accelerators have been widely used are accelerating complex mathematical operations, such as floating point arithmetic, signal processing and computer graphics. Recent trend has been to develop accelerators with more general-purpose computational capabilities. Examples of such processors are modern graphics processing units, the Cell microprocessor and most recently Intel's x86 many-core coprocessor Xeon Phi. The previously mentioned accelerators are mainly focused on exploiting data parallelism.

2.2.2 Energy efficient computing

For a long time, improving performance was the most important goal for commercial processor manufacturers. Since hitting the so called *power wall*, manufacturers have turned their focus on improving the efficiency of processors. Instead of plain performance, metrics of *performance per watt* and *performance per area* have become increasingly important.

There are two different viewpoints to improving the efficiency of processors. Power consumption directly affects the amount of heat processors generate [Wolf, 2007, p. 21]. Heat generation is one of the biggest reasons for the rise of CPU clock speeds declining [Sutter, 2005]. Sequential performance can only be increased by improving the performance per watt ratio. On the other hand we can talk about energy efficiency when we want to stress the fact that embedded systems are often powered with batteries. Efficient computation increases the battery life of the whole system. [Wolf, 2007, p. 21]

Dynamic voltage and frequency scaling (DVFS) is a technique for lowering the power consumption of processors by scaling the clock speed and voltage of processors dynamically. There is a quadratic relationship between the energy consumption and operating voltage of a processor. In an ideal setting this would mean that by halving the clock speed and the operating voltage of a processor, you would get to one fourth of the original energy consumption. In practice there is a significant static component in the energy consumption of a processor that does not depend on the operating speed and voltage, which is often called leakage power. [Wolf, 2007, p. 86–87]

Simply adding more cores to CPUs feels intuitively like a good solution, but in practice it has been shown that this method does not scale well. Large portions of computer programs consist of sequential execution and writing completely parallel programs is difficult, if not impossible. Growing the number of cores also makes hardware design and memory management increasingly more difficult. A more realistic approach is to add more specialized processing units on computers, such as the accelerators discussed in section 2.2.1. Specialized hardware is usually simpler in design, which lowers the energy consumption and often leads to better performance. Heterogeneous computing, where parts of computation are offloaded to specialized processing units, tries to address the need for high performance and low energy computing. [Wolf, 2007, p. 267–275]

2.3 Execution timing

2.3.1 Dynamic program analysis

Dynamic program analysis means the analysis of programs by observing their execution either on a real processor or in a simulated environment. Program execution can be observed by either *instrumenting* the program by injecting code into the program that records desired information, or by explicitly monitoring the state of the real or virtual processor or memory during execution. There is usually a considerable amount of overhead associated with dynamic program analysis. The more often the state of execution is observed and logged, the bigger the overhead is.

Software profiling is a form of dynamic program analysis that aims to count the occurrences of different types of events, such as the number of times a function, basic block or instruction has been executed. Software profiling is typically used for optimizing program performance and identifying bottlenecks. [Ball and Larus, 1994]

Software tracing is a more detailed form of dynamic program analysis that

does not only count the number of different events, but records the sequence of events that have occurred during execution. Like with software profiling, execution tracing can be performed at different levels of granularity. A full execution timing model can be constructed by adding a timestamp to each event in the trace. [Ball and Larus, 1994]

2.3.2 Simulators

GPUs are much more closed systems compared to general-purpose CPUs. It is not possible to generate as thorough traces of programs that are being executed on the GPU. Vendors provide some tools that generate traces that allow developers to evaluate the performance of their applications. We are interested in building an estimated power model based on the execution traces, which requires more detailed tracing. Currently the only way to inspect the execution of GPU accelerated programs at the required level of detail is to use a GPU simulator.

The terms *emulator* and *simulator* are often used interchangeably. Both can be used to execute programs in place of an abstract or real machine. The difference between the terms lies in the model they are mimicking. The term *emulator* is used when the aim is to reproduce the behaviour of a real system. The term *simulator* is used when the system executes an abstract model. [Hirvisalo and Knuuttila, 2010, p. 2]

Simulators have numerous different use cases. Most commonly they are used to analyze the performance or the energy consumption of programs. Simulators can also be used to analyze the behaviour of processors before there is an actual physical processor available. Simulators can also be used for software debugging purposes. Simulation can be done by directly executing on the host processor and keeping track of the state of the guest processor, or by implementing an explicit simulator program that performs the execution and tracks the state of the guest processor. [Wolf, 2007, p. 126–130]

There is a clear trade-off between the speed of simulation and the accuracy with respect to the the characteristic of execution that is under analysis. *Functional simulators* simulate the instruction set of a processor, but they do not model the microarchitecture of processors. A more accurate, but also much slower, simulator type is a so called *cycle-accurate simulator*. Cycle-accurate simulators model the state of the processor at microarchitecture level at the temporal granularity of a clock cycle. They also often keep track of processors resources, such as registers and memory, more accurately. The cost of cycle accuracy is a significant loss in performance. For accurate power modeling with simulation it is often necessary to also simulate other components of the computer, such as memory buses and memory. [Wolf,

2007, p. 131–132]

There are two basic methods for executing code in a simulator. *Binary interpretation* and *binary translation*. Interpreting code in a simulator works much like a fetch-decode-execute loop in a real processor. This results in quite poor performance. A faster method is to translate the binary into the instruction set architecture (ISA) of the host processor either statically ahead of time or dynamically in blocks just-in-time (JIT) before the code is executed. While binary translation has a significant overhead, it is usually a much faster method compared to binary interpretation. [Hirvisalo and Knuuttila, 2010]

2.4 Linux device driver

Device drivers are computer programs that work as an interface between user programs and hardware. They provide a set of operations related to the hardware device they govern to user programs and manage the hardware resource so that it cannot be accessed in an illegal way. In Linux device drivers are typically implemented as a part of the kernel. User programs can invoke device services through a set of system calls. It is also the duty of the device driver to handle interrupts raised by the device.

Graphics processing units are typically character devices, meaning they are accessed as a stream of bytes like files. The typical system calls provided by a character device driver are *open*, *close*, *read* and *write*. Non-standard operations and hardware control operations are usually handled using the *ioctl* (input/output control) system call. Devices can have a large set of operations that are accessed through the ioctl system call, and they are typically device specific operations. [Corbet et al., 2005, p. 1–7, 135–140] Typically user applications do not access device services directly through the system call interface, but they call functions of user-mode libraries that provide a higher level interface for applications.

Most hardware devices need to notify the kernel about certain events that have occurred on the device. Devices can interact with the kernel using *interrupts*, that signal that there is an event that may require immediate attention. It is the duty of the device driver to implement functionality to handle interrupts raised by the hardware. There *interrupt handlers* are often split into kernel and user mode parts, often called the top and bottom halves. The kernel mode interrupt handler routine is run when an interrupt has been caught. The kernel mode interrupt handler routines should execute in a minimum possible amount of time because the execution blocks all other processes. What is actually required to be done in an interrupt handler

depends on the nature of the interrupt. The handler should at least make sure the device can continue normal operation and that the interrupt will be handled appropriately. If the interrupt handling requires long computations, it can be split into one or more tasklets, that are user mode processes that will handle the computation. Tasklets are schedules like regular processes, so they do not interfere with regular execution of processes. [Corbet et al., 2005, p. 258–278]

2.5 Energy consumption

2.5.1 Power and energy

Energy and power are terms that in everyday speech are often used interchangeably. *Energy* means the amount of work that a physical system is able to do on another system. The SI unit of energy is joule (J). In this thesis we are especially interested in the energy efficiency of computation, because energy consumption has a direct effect of battery lifetime in embedded systems. *Power* on the other hand is used to describe the rate of energy dissipation. The SI unit of Power is watt (W), which is joules per second. [Young et al., 2006]

We can describe the relationship between power and energy by defining *instantaneous power* as follows:

$$P = \lim_{t \to 0} \frac{\Delta E}{\Delta t} = \frac{\delta E}{\delta t}. \tag{2.3}$$

In this thesis we are interested in the *electromagnetic energy* dissipation of the computation of many-core processors such as GPUs. However, we cannot measure energy or power consumption directly. We can, however, measure the current (I) and voltage (V) of electronic circuits. Instantaneous power can be described as a function of voltage and current by using *Ohm's law* and *Joule's first law*, as defined in for example physics handbooks [Young et al., 2006]. Ohm's law is defined as:

$$I = \frac{V}{R}. \tag{2.4}$$

Joule's first law is:

$$P = I^2 \cdot R. \tag{2.5}$$

By substitution, instantaneous power can be defined as a function of voltage and current at a given point in time t as follows, assuming a constant supply voltage:

$$P(t) = I(t) \cdot V = \frac{V^2}{R}. \tag{2.6}$$

Energy dissipation over a time interval T can be then defined through integration:

$$E = \int_0^T I(t) \cdot V dt. \tag{2.7}$$

2.5.2 Power consumption

The overall power consumption of a processor can be described with the following equation:

$$Power = Dynamic\_power + Static\_power \tag{2.8}$$

Dynamic power is caused by the switching of transistors during execution of programs. As the name suggests, dynamic power consumption varies based on the nature of execution and the workload of the processor. Static power is the result of hardware architecture design and operating temperature, and it can be considered to be more or less constant, unless execution affects the operating temperature. [Hong and Kim, 2010]

Dynamic power in the above equation can be described to be proportional to the following formula:

$$P = ACV^2 f, \tag{2.9}$$

where A is the activity factor, C is total capacitance, V is operating voltage and f is the frequency of the clock. The above formula explains why power-saving techniques such as DVFS, described in Section 2.2.2, can yield significant energy savings. It if more efficient to "race to idle" by completing a task at a high peak frequency and then running at a reduced clock frequency and operating voltage during idle states.

Chapter 3

GPU computation environment

In this chapter we present both the hardware and software platforms related to GPU accelerated processing. The GPU compute model differs quite dramatically from typical program execution on CPUs. Section 3.1 presents the abstract hardware architecture of graphics processing units and the programming models for GPU accelerated processing. After that we take a look at the software stack on the host platform. Finally, we look at some methods on how it is currently possible to instrument and profile the execution of GPU accelerated programs.

3.1 GPU accelerated processing

3.1.1 Graphics processing unit architectures

3.1.1.1 General graphics processing unit architecture

Graphics processing units are high throughput processors. Their processing power is achieved by having a large number of simple processing elements that perform computations in parallel. Originally graphics processing units were designed specifically for rendering of graphics, but as the graphics rendering pipeline became more programmable, the structure of GPUs became more suitable for general-purpose computing.

Figure 3.1 shows the high-level architecture of current desktop graphics processing units. A GPU consists of multiple processing cores. Typically the number of cores can vary from just a few up to a few dozen cores per GPU. The processing cores each execute their own stream of instructions. The actual computation is performed by the processing elements (PE) on the processing cores. Recent consumer GPUs have had up to 48 processing elements on each processing core. The trend on both mobile and desktop

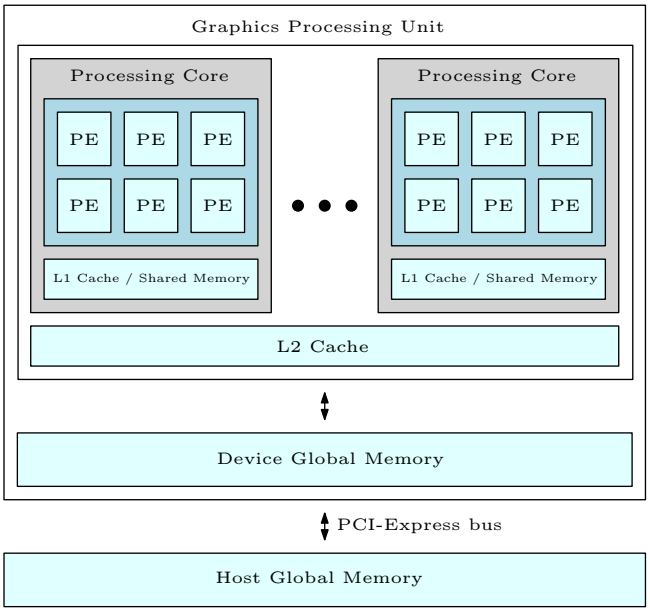


Figure 3.1: High-level GPU architecture

GPUs has been to increase the total number of processing elements on the GPU to achieve higher throughput.

Current desktop GPUs contain a small cache, that can also be used as a programmable shared memory, on each processing core [NVIDIA, e, p. 10–11]. In the first generation GPUs that were used for general-purpose computation, only a smaller shared memory was available on the processing cores. The main method for hiding memory access latency is having a large number of threads ready to run and having a fast thread switching mechanism implemented in hardware. [Wang, 2010, p. VI-620] The current generation GPUs also have a larger L2 cache shared by all the processing cores for hiding global memory access latency.

Graphics processing units have a much larger number of registers compared to general-purpose processors. Each processing core typically has thousands of registers. The registers allocations are divided between a large number of threads to avoid the need of context switches in the sense they are performed on CPUs. Having a fixed allocation of registers for each thread that is ready to be executed on a processing core simplifies the hardware needed for scheduling. [Kanter, 2009, p. 5–6]

Scheduling logic is located on the processing cores. Instructions are dispatched to the core's processing elements that execute the same instruction in parallel operating on their own data. This makes the execution funda-

mentally SIMD-style computation as described in 2.1.6. [Kanter, 2009, p. 5–7]

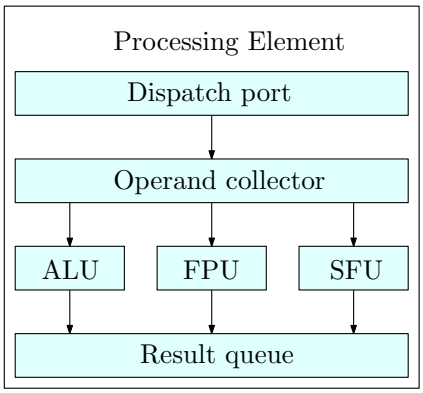


Figure 3.2: Abstract processing element architecture

Processing elements on the processing cores consist mostly of functional units. The number of different functional units may differ. GPUs that are focused on graphics rendering performance typically have high single-precision floating point performance, while GPUs aimed for general-purpose computation and scientific computing may have more double-precision floating point units. [Kanter, 2009, p. 5–7] GPUs capable of general-purpose computation also have integer arithmetic logic units and Special Function Units (SFU) that are responsible for performing special math operations such as square root, sine and cosine [Wong et al., 2010]. There may be other units on the processing element that aim to reduce memory operation latencies, such as the operand collector and result queue units in the example processing element structure presented in figure 3.2 [Kanter, 2009, p. 5–7].

3.1.1.2 Mobile graphics processing units

Graphics processing units on mobile devices such as mobile phones are very similar to desktop GPUs. They are expected to perform mostly similar tasks, although their performance is not on par with modern desktop GPUs. Mobile GPUs need to be extremely energy efficient, because they are powered by batteries and there is typically no active cooling elements such as fans cooling them down. [Akenine-Moller and Strom, 2008]

The high-level memory architecture of a typical mobile system on a chip (SoC) can be seen in figure 3.3. The biggest difference between mobile and desktop GPUs is that mobile GPUs are typically located on the same chip as the host CPU processor. The CPU and GPU also often share the system main

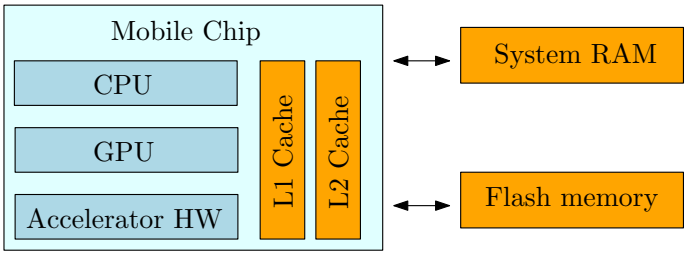


Figure 3.3: Memory architecture of a typical mobile chip. [Akenine-Moller and Strom, 2008]

memory which is located off the chip. Sharing memory eliminates the need for explicit memory transfers between CPU and CPU memories. Mobile devices usually have domain specific accelerators for multimedia operations, such as video encoding and decoding. The accelerators aim to improve performance and energy efficiency. [Akenine-Moller and Strom, 2008]

3.1.2 Parallel compute model on graphics processing units

The parallel compute model on GPUs, originally called general-purpose graphics processing unit computing (GPGPU), is a data parallel computing model. A commonly used term for the execution model is single instruction, multiple thread (SIMT), which is a marketing term coined by NVIDIA. The slight difference compared to SIMD vector computing is that the threads in the SIMT model perform the computations on the data items of the given vector lane sequentially one by one as scalar operations. While the threads execute the same instructions simultaneously, each thread can still have their own logical control flow by masking out computations in divergent branches. The GPU compute model is close to the single program, multiple data (SPMD) model where data is partitioned for parallel execution on different processors [Algorithms and of Computation Handbook, 1999].

In the current compute model it is the responsibility of the programmer to manage memory on the GPU and explicitly handle memory transfers between host and device memories. The input data needs to be transferred to the GPU before computation can be done, and the results usually must be transferred back to the host memory. A typical application that uses the GPU as an accelerator takes the following steps:

- Allocate buffers in host and device memories.

- Copy data from host to device buffers.
- Launch compute kernel for computation on the GPU device.
- Copy results back from device to host buffers.
- Deallocate buffers. [Gaster et al., 2011, p. 16–26]

The parallel regions of code that can be efficiently executed on a GPU are typically loop nests that iterate over a large amount of data. These parallel regions are implemented as so called *kernel functions*. The kernels are offloaded to the GPU for execution using a large number of threads which are all executing the same kernel function. The only difference between the threads is that they all have a unique index number which is used to determine their control flow and the data the thread will be working on. [Gaster et al., 2011, p. 16–19]

In addition to specifying the number of threads that perform the computation, the programmer needs to explicitly divide the threads into blocks called cooperative thread arrays (CTA) or workgroups [Gaster et al., 2011, p. 17–19]. GPUs schedule threads of the same CTA for computation on the same processing core. The motivation behind dividing threads into blocks is to avoid context switches in the traditional sense during computation. As presented in section 3.1.1.1, GPUs hide memory access latency by having a large number of threads ready to run. When a CTA is scheduled for execution on a processing core, registers and shared memory are allocated for all the threads in the CTA so that they are all potentially ready to be executed. On the other hand, the amount of threads in a CTA limits the amount of registers and memory that can be allocated to a single thread. Finding the optimal CTA size is an application specific problem.

Another reason for dividing threads into blocks explicitly is to achieve scalable computation. CTAs can be scheduled to any processing core for computation, because by definition the computation of one CTA must be able to be done independently of others. The block structure makes it possible to accelerate the execution efficiently on GPUs with a different number of processing cores. [Gaster et al., 2011, p. 17–19]

As the threads of the same CTA are executing on the same processing core, they also share the same memory local to the core. This also allows the threads belonging to the same CTA to synchronize. However, synchronization between threads in different blocks is possible only by exiting the kernel execution and returning to sequential host code for synchronization. [NVIDIA, e, p. 6]

A more fine-grained scheduling unit in the execution model is what NVIDIA calls a *warp* and AMD calls *wavefront*. A warp is a group of threads of the same CTA, in current compute model 32 threads, that are scheduled for execution on a processing core in parallel. The threads in the same warp always execute the same instruction with their own data in lockstep. Branching is handled by masking out the computation of threads that are not taking the branch currently in execution. [Wong et al., 2010]

3.1.3 Programming models for GPU accelerated processing

General-purpose computing on graphics processing units (GPGPU) emerged as an exploitation of vertex and pixel shader programs used for computer graphics. In the shader based GPGPU programming model, input data was represented as a texture and it was drawn into a scene on a plane facing the camera so that each pixel colour value on the view plane represented a data item in the input data array. Pixel shaders were written to sample the input data texture, perform computation and write the output to another texture that could be read back to the host. While it was possible to get noticeable speedups, the programmability of shader based GPGPU computation was really low.

The first programming model actually designed for general-purpose GPU accelerated processing to emerge was NVIDIAs CUDA (Compute Unified Device Architecture) [NVIDIA, a]. In CUDA, the kernel functions, described in section 3.1.2, are implemented using CUDA C or CUDA C++. CUDA C/C++ are subsets of C and C++ languages. Compared to shader program based programming, the programmability of CUDA programs is dramatically better. However, the kernel functions are still written using a relatively low level language. CUDA programs can only be executed on NVIDIA's graphics processing units.

As a response to CUDA, an open framework called OpenCL [Khronos Group] was developed. OpenCL is governed by Khronos Group. OpenCL resembles the functionality of CUDA very closely. Kernel functions are written using OpenCL C, which is a subset of C99 language. The host side library also has a very similar interface. The biggest difference to CUDA is that OpenCL programs can be executed on GPUs of multiple vendors, CPUs, field-programmable gate arrays (FPGA) and other accelerator hardware.

Recently, other parallel processing programming models have also emerged. There is a growing need for a high-level programming model that would allow widespread adoption of accelerated parallel processing. OpenACC [Ope-

nACC] is one standard that aims to provide an easy and familiar programming model that would still reach performance that is comparable to lower level computation models. OpenACC uses a similar compiler directive based approach as OpenMP, an existing parallel processing framework aimed for shared-memory parallelism on CPUs.

On mobile devices there are currently at least two programming models that have expressed the intent of supporting parallel computation on mobile GPUs, OpenCL and Renderscript [Google] on the Android platform. There are already multiple OpenCL comformant mobile GPUs on the market. GPU compute support was included to Renderscript in Android version 4.2.

3.1.4 GPU execution context and scheduling

Computation is scheduled on GPUs hieracrhically with different levels of granularity. On the coarsest level the global scheduler schedules blocks of kernels for execution on the processing cores. Processing cores may be scheduled to execute blocks belonging to different kernels simultaneously on modern GPUs. [Kanter, 2009, p. 4] However, it is not possible to have kernels from different streams executing at the same time [Kato et al., 2012]. Executing multiple kernels simultaneously can improve utilization.

Finer grained scheduling is performed by the processing cores. The cores issue instructions from warps that are ready to run. GPUs do not perform context switching like traditional CPUs. They rely on having enough warps ready to run so that there are always instructions ready to be issued for execution.

Another big difference compared to traditional CPUs is that GPU computation in current generation GPUs is non-preemptive. Once a compute kernel has been launched for computation, the GPU will keep processing the kernel until all threads belonging to the kernel launch have finished computation. [Kato et al., 2012]

3.2 Host platform for GPU accelerated processing

3.2.1 Program compilation workflow

This section presents the compilation flow of NVIDIA's LLVM based CUDA and OpenCL compiler called *nvcc*. Other vendors, such as AMD, may have different terminology and they may use different intermediate languages and compiler frameworks for compilation, and they ultimately target a different

ISA, but the overall structure of the workflow still mostly applies to them. This section considers NVIDIA's implementation because most of the GPU simulators are based on the PTX intermediate language used by them. The compiler is open source as a part of the LLVM compiler project up to the PTX code generation phase. The final ISA code generation is performed by a closed source library.

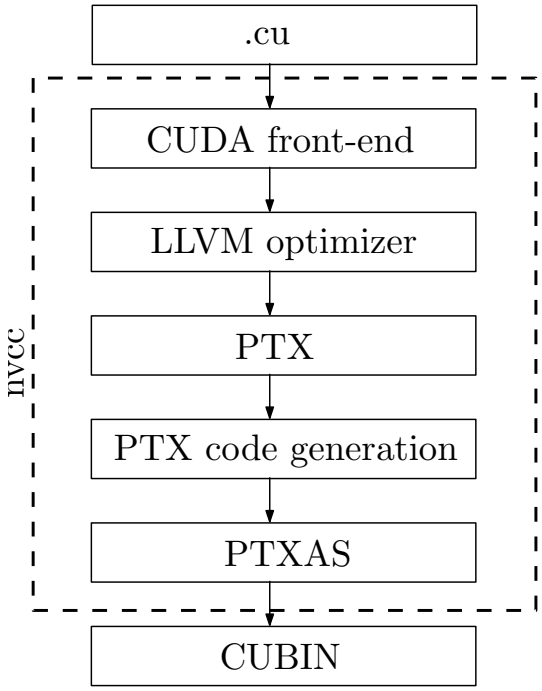


Figure 3.4: Workflow of the nvcc compiler. [NVIDIA, f]

The nvcc compiler workflow for a CUDA application has multiple steps as shown in figure. Kernels written with CUDA C or OpenCL C are first separated from the host code by the language specific front end. The kernels are then translated to the NVVM intermediate representation. NVVM IR is based on the LLVM intermediate language. NVVM extends the LLVM IR with a set of rules and intrinsics related to the parallel compute model, but it is fully compatible with existing LLVM IR tools. Standard LLVM optimizer passes also work on the NNVM IR. [NVIDIA, f]

The last phase of the open source part of the nvcc compiler is the PTX intermediate language code generation phase. The PTX code is already generated to match the targeted GPU architecture, which makes the generation of native ISA binaries easier. Final ISA code generation is performed by a separate proprietary host assembler PTXAS which can either compile the

binary offline or just-in-time as a part of the CUDA runtime. PTX representation is embedded with the binaries enabling JIT compilation. [NVIDIA, f]

Using a virtual ISA that is fairly close to the final ISA has some benefits. While the generated PTX code is optimized for the targeted hardware architecture, the PTX IR itself is machine independent. This makes the PTX representations backwards compatible and it can be used to generate binaries for different generations of hardware and even for hardware of different manufacturers. From the vendor's point of view it gives other parties a virtual ISA to target without releasing the specifications of their hardware's ISA.

3.2.2 Runtime library and GPU device driver

3.2.2.1 GPU device driver and CUDA runtime library

Currently the device drivers for the graphics processing units of major GPU manufacturers are closed source and little documentation related to their functionality and interfaces are provided. There has been some efforts to reverse engineer the NVIDIA device driver and to implement open source alternatives. The Nouveau [Nouveau] open source drivers are already widely used as alternative graphics drivers for NVIDIA's proprietary binary drivers. pscnv [PathScale] is a project forked from the Nouveau project, developed by PathScale, that aims to provide an open source driver for graphics and GPGPU computing.

The NVIDIA GPU driver is split into user-mode and kernel parts [NVIDIA, b]. Communication between the user-mode module and the kernel module occurs using system calls. GPU drivers generally use a large number of different *ioctl* system calls to invoke different operations. The ioctl calls used by GPU device drivers are not documented, but the previously mentioned open source projects have backwards engineered some of them. In facts, there is nothing that stops user applications from sending ioctl calls directly to the kernel driver to invoke GPU operations [Kato et al., 2012].

NVIDIA's GPU driver offers a low level interface called CUDA driver API for directly configuring the GPU and launching compute kernels. CUDA driver API is an alternative to the higher level API provided by the runtime library giving slightly finer grained control over the GPU. The runtime library is built on top of the driver API. [NVIDIA, b]

A higher level interface compared to the Driver API is provided by the CUDA runtime library. The CUDA runtime library allows programmers to perform the basic GPU operations described in 3.1.2 without having to initialize the GPU explicitly. The runtime also allows native bindings for

other programming languages. [NVIDIA, b]

3.2.2.2 Task scheduling

GPU operations, such as memory copy operations and compute kernel launching, are issued to channels that are called command queues or streams. Current generation GPUs do not allow multiple channels to access the GPU simultaneously, but channels can coexist and the GPU can switch from one channel to another. Streams are essentially First In, First Out (FIFO) queues of operations. [Kato et al., 2012]

Memory transfer operations are performed as either synchronous or asynchronous DMA transfers. Typically data transfers are performed synchronously, because the results of the data transfer are often needed either on the GPU or CPU before further computation can be performed. Asynchronous memory transfers can overlap compute operations on the GPU. [Kato et al., 2012] Compute kernels are usually launched to the GPU asynchronously.

Proper synchronization in current GPU accelerated computing can be performed only on the host CPU. An explicit *synchronize* call waits for all commands that have been sent to the GPU to finish. Alternatively synchronization can be performed by using events that are defined in both CUDA and OpenCL programming models. Compute kernels for example can be queued so that their computation will not start before a set of specified events have been received. Events can be dispatched at the start and end of operations. [Kato et al., 2012]

3.3 Tracing of accelerated computing

The host and the accelerator have a different view of the processing of an application. The host sees accelerated blocks of computation and memory transfers as a set of operations. The accelerator performing the computation has a more detailed view of the execution. If we want to follow the overall execution of an application, we need to be able to combine the host and device views.

For following the execution from the host's point of view, we need to track the start and end times of operations. The timing of operations can be done using different methods. Operations can either be measured indirectly from the host's point of view, or directly on the accelerator if hardware support for such measurements is implemented. Measurements made on different devices using different clocks must be synchronized.

3.3.1 Tracing of host operations

Programs initiate accelerated compute operations by calling functions in the runtime libraries. Some of these operations, such as allocating buffers in host memory, take place on the host platform. The parameters of the function calls may also reveal details of operations that may not be observable on the accelerator device. Host events can be observed by library wrapping of the runtime API [Malony et al., 2011]. The library wrapper can then intercept function calls before calling the actual runtime library.

3.3.2 Tracing of operations on GPU

The operations we wish to observe on the accelerator are the start and end times of the computation of compute kernels and memory transfers between host and accelerator memories. It would be desirable to also track host CPU events related to the accelerator operations. For example, launching a compute kernel on the accelerator actually consists of multiple different function calls to the driver API [Malony et al., 2011].

There are different methods for observing the start and end times of CUDA and OpenCL compute kernels. Execution can be observed using the *synchronous*, *event* or *callback* methods. Support in both hardware and software may limit which instrumentation methods can be used.

Synchronous method is the simplest way to instrument compute kernel execution. By launching computer kernels for execution in a blocking manner, the host can measure the execution time using its own clock. The synchronized method is inaccurate because there is a delay between launching a kernel and when the execution actually starts, and also between the end of execution and when the host's synchronization point. [Malony et al., 2011]

Event method relies on the event feature that is specified in both the CUDA and OpenCL language specifications. In the event method special *event kernels* are queued to execute immediately before and after the measured compute kernel. The event kernels record the state of the accelerator, meaning the event method uses the device clock for measurement. The event method at least in theory gives a more accurate measurement of the compute kernel execution time. The measured timestamps need to be synchronized with the CPU clock to obtain full system timing information. The event method needs support from hardware, which means that especially older hardware cannot use events for instrumentation. [Malony et al., 2011]

Callback method is the most accurate method for instrumenting accelerator operations. Callbacks are defined in the recent versions of both the CUDA and OpenCL specifications. Like with the event method, the callback

method requires support from hardware and the device driver. The accelerator triggers callback functions on the host when certain types of events occur. For timing, the events are the start and end events of compute kernels, but callback functions can be used to gather and handle performance measurements of other types too. [Malony et al., 2011]

On NVIDIA's GPU hardware, CUDA Performance Tool Interface (CUPTI) provides an API for event and callback based instrumentation. CUPTI can be used to also read GPU device counters containing different performance metrics. [Malony et al., 2011] In CUDA toolkit versions prior to version five, there was a limitation that compute kernels could not be executed concurrently when instrumenting execution using CUPTI.

3.3.3 Tracing with TAU Parallel Performance System

TAU Parallel Performance System [Shende and Malony, 2006] is a profiling and tracing tool for parallel programs. It is capable of instrumenting programs ranging from regular multi-threaded applications to distributed computing at different levels of granularity. The recent versions of TAU have also included the capability to instrument GPU accelerated CUDA and OpenCL applications.

TAU implements heterogeneous computing instrumentation by wrapping CUDA and OpenCL libraries and dynamically preloading the wrapped libraries [Malony et al., 2011]. TAU was recently updated to use CUPTI fo instrumentation, which makes it possible to instrument kernel execution using any of the three instrumentation methods described in section 3.3.2.

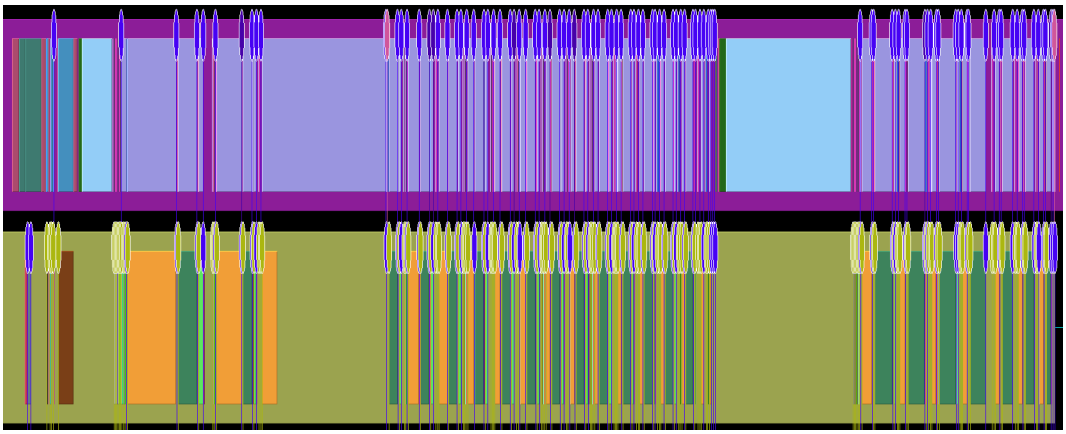


Figure 3.5: A visualisation of a TAU trace

Figure 3.5 shows an example of a trace output from the TAU tool visualised with the Jumpshot visualisation tool that is distributed with TAU. The application that was being traced was a modified face detection application using the OpenCV computer vision library. The application is presented in detail in Section 5.2.2. The coloured rectangles represent the execution of different functions over time. The upper row represents execution on the CPU and the lower row execution on the GPU. Keep in mind that execution of a function on the GPU in this context means a kernel launch of hundreds of threads executing the same function.

3.3.4 Profiling of GPU accelerated processing

3.3.4.1 CUDA Profiling Tools Interface (CUPTI)

CUDA Profiling Tools Interface (CUPTI) is a set of APIs that enable the creation of profiling and tracing tools, such as the TAU tool presented in the previous section. The four APIs provided by CUPTI are Activity API, Callback API, Event API and Metric API. Together they allow the tracing of calls to the CUDA runtime libraries, timing of the execution of CUDA kernels and collecting performance counter metrics of kernel execution.

The Metric API provides access to a set of metrics collected from actual execution of applications on a GPU. The set of reported metrics include metrics about number of instructions issued and executed, cache behaviour metrics and memory throughput metrics. The metrics are mostly suitable for evaluating the utilisation of the GPU. [NVIDIA, c]

Chapter 4

Measuring and simulating GPUs

This section covers some methods that are currently available for observing the execution of GPU accelerated programs. We are especially interested in observing the timing and energy consumption of program execution. As GPUs are relatively closed systems, we have no choice but to use simulators to unveil details of the execution. At the end of this chapter we present a number of different parallel processing and GPU simulators that can be used to instrument parallel processing.

4.1 Instrumentation

4.1.1 Timing and energy model

It is often not convenient, or even possible, to measure the energy consumption of a processor directly during execution. To overcome this observability issue, simulators are often used to trace the execution of a program. Execution traces can be used for building an abstract energy model for the system.

Power consumption consists of two parts: dynamic power and static power. Static power does not depend on the execution. It is a direct consequence of the chip design and operating temperature of the processor. Dynamic power is the part that changes with runtime events of execution. Basically each component that is activated on the processor consumes power. Static power, including power needed for the GPU memory, typically dominate the overall power consumption of a GPU. [Hong and Kim, 2010]

Accurate energy models are usually constructed using a cycle-accurate simulator that also simulates the memory and cache hierarchy. Such simulators are currently too slow to be used for software development purposes. Ac-

cording to Miettinen and Hirvisalo [2009] the power consumption is roughly proportional to the cycle count of the processor. This makes it viable to use fast functional simulators for constructing approximate energy models if we can verify that the loss in accuracy is within acceptable bounds.

Different instructions take a varying time to execute and they consume different amount of energy. In practice we can group instructions into classes such as integer operations, floating point operations or memory load and store operations [Hirvisalo and Knuuttila, 2010]. For the energy model, we should count the amount of instructions belonging to each instruction class during execution. This can be done quite trivially during the execution loop in simulators. The power consumption of instruction classes can be calibrated through microbenchmarking which we will present in section 5.1.

Memory accesses have a significant impact on the overall power consumption. Simulating the full memory and cache hierarchies is fairly difficult and slow. Functional simulators often use abstractions of the memory hierarchy and simulate only a part of the cache hierarchy, if at all. For our purposes, it would be desirable to have at least the L1 cache simulated as there is a considerable difference between accessing L1 cache compared to other levels of the hierarchy.

Instruction counting is only applicable for estimating the power consumption of a single processor. For a many-core processor like a GPU, we must also keep track of *where* the computation takes place. Collange et al. [2009] have shown that the power consumption of a GPU rises piecewise linearly as the amount of active processing cores is increased.

4.1.2 Instrumentation

We wish to generate traces of the execution of kernels on GPUs. A trace is a record of a sequence of events that occurred during the execution of a program. We need to be able to count the number of executed instructions of different types and the number of different types of memory accesses.

The most common techniques for generating execution traces are instrumentation and using simulators. Instrumentation generally means injecting code that emits events related to the execution. Simulators can generate executiong traces as they perform the execution of the program. For GPU accelerated programs, it is more convenient to generate traces of executed instructions by using a simulator.

It should be noted, that generating traces typically always slow down the execution of programs. Since the speed of execution even in a simulated environment is important to us, we should make sure that the overhead caused by tracing is as low as possible.

4.2 GPU Simulators

The motivation for using GPU simulators is that it is otherwise not possible to trace the execution of GPU accelerated programs at a fine-grained level of detail. Profiling tools released by GPU vendors are only suitable for application performance analysis and tuning. In this section we cover the current state of state of the art GPU simulators. In section 4.2.1 we present the typical structure of popular GPU simulators and reasoning behind it. In section 4.2.2 we present some state of the art GPU simulators.

4.2.1 GPU simulator structure

There are two major hurdles GPU simulators need to overcome. The first follows from the sheer number of processing units on graphics processing units. Simulating a many-core processor that has hundreds, if not thousands, of processing units on a multi-core processor with up to 6 physical cores with sufficient performance is intuitively a difficult problem to overcome. In reality simulators often work as sequential applications, which makes things even worse. The second problem is that the instruction set architecture of GPUs is not publicly available knowledge. While the ISAs have been partially backwards engineered, in practice simulators often work on the intermediate language representations of programs instead of binaries. Emulating an intermediate language results in a loss in accuracy.

The simulators can be split into two different categories. Some simulators aim for purely functional simulation. Functional simulators model the execution at instruction level. They do not keep track of the internal state of the simulated GPU. Functional simulators also usually do not simulate memory or cache behaviour. The other category is architectural simulators that try to model the internal state of GPUs at the granularity of a clock cycle. Architectural simulators also usually have at least L1 level cache simulation and some sort of a device global memory simulation. Architectural simulators are usually aimed for timing and energy analysis and they can be an order of magnitude or more slower than functional simulators.

Most of the found simulators are aimed for analysing programs written with NVIDIA's CUDA framework. Some of them also had support for the OpenCL standard. All the simulators included a software component that basically replaces the CUDA or OpenCL runtime library that intercepts the IO requests made by the application running on the host processor.

4.2.2 GPU simulator survey

4.2.2.1 Barra

Barra is a functional simulator built on top of the UNISIM simulator framework [August et al., 2007]. The execution model of the simulator is based on NVIDIA's older G80-based GPUs. The simulator can run unmodified CUDA binaries. [Collange et al., 2010] Barra does not appear to be under active development.

Barra differs from other GPU simulators by simulating programs at assembly language level. The simulator works with the Tesla ISA, which is an older version of the ISA used on NVIDIA's GPUs. The Tesla ISA specification is based on the work done by the decuda project [van der Laan] and extended by the authors of the simulator. Collange et al. [2010] claim that by using the native assembly language instead of the PTX intermediate language, Barra can simulate real hardware more accurately.

4.2.2.2 GPGPU-sim

GPGPU-sim is a GPU simulator that can perform both functional and cycle-accurate simulation. Its focus is on cycle-accurate microarchitecture simulation. The functional simulation is mostly supported for software debugging purposes. The simulators execution model tries to follow NVIDIA's GPUs execution model. It is also possible to modify the topology and other characteristics of the simulated GPU architecture. [Bakhoda et al., 2009]

GPGPU-sim takes NVIDIA's PTX intermediate language as its input. Unlike other GPU simulators, GPGPU-sim can execute programs written with both CUDA and OpenCL. [Bakhoda et al., 2009]

4.2.2.3 Multi2Sim

Multi2Sim is a simulation framework that has a couple of notable differences compared to other simulators presented in this section. First of all it is the only simulator targeting AMDs GPUs as its simulation model. Multi2Sim also works as a simulator for multi-core x86 CPUs. By combining the simulated execution of CPU and GPU code Multi2Sim can potentially reach better accuracy in estimating the performance and energy efficiency of executed software. [Ubal et al., 2012]

Similar to Barra, Multi2Sim can run unmodified binaries and perform simulation at assembly language level, which is claimed to improve simulation accuracy. The simulator can execute programs compiled to the AMD Evergreen ISA. The simulator can perform both functional and architectural

simulation. The simulator currently supports execution of programs written with OpenCL. [Ubal et al., 2012]

4.2.2.4 GPU Ocelot

GPU Ocelot is a dynamic compilation framework for heterogeneous systems. The framework can take programs in PTX intermediate representation as input and compile them just-in-time for execution on multiple platforms. Currently GPU Ocelot supports execution on AMDs and NVIDIA's GPUs, general purpose x86 processors through translation to LLVM's intermediate language and execution on Ocelot's own PTX emulator. The emulator can be used to analyze the fine grained details of execution on GPUs.

Compared to other GPU simulators, Ocelot is the least concerned with simulating the fine-grained microarchitecture of real life GPUs. Ocelot's PTX emulator can be used to retrieve detailed instruction traces of execution and exploring the state of the simulated abstract GPU model at different stages of execution. The program can also be augmented with instrumentation code before dynamic translation and execution on real hardware.

GPU Ocelot's strength is that it has a modular structure which makes it easy to write custom trace generators for it. Trace generators for the emulator can be written as event-handlers that receive an event after the execution of every instruction that allows the observation of the state of the emulated GPU.

4.3 Parallel GPU simulation framework

4.3.1 Parallel simulation

There are several existing sequential emulators that have reached a good maturity level. There has been some attempts to extend their functionality to make them suitable for the emulation of multi-core and many-core processors. Efficient emulation of many-core guest processors on a host that has a smaller number of cores is difficult. Ensuring the correctness of parallel execution in such parallel simulator is even more difficult and easily affects the emulation performance negatively.

QEMU [Bellard, 2005] is a popular and fairly mature open source sequential emulator. QEMU's strengths are its relatively good emulation performance through binary translation. It can function as a full system simulator or as a process emulator. The difference is that full system simulators simulate also the hardware of the system. To run user-mode processes, the

simulator must also execute an operating system on top of the simulated hardware. Process simulators also simulate the operating system by forwarding the system calls of the processes to the host operating system.

Sequential emulators typically emulate multi-core guest processors by time-slicing the execution of different cores in round-robin fashion. In QEMU this time-slicing occurs at the granularity of basic blocks which makes synchronization and atomic operations are easy to handle. Parallelizing existing sequential emulators requires the implementation of proper synchronization mechanisms to enable parallel programs to function correctly. [Wang et al., 2011]

Projects such as COREMU [Wang et al., 2011] and PQEMU [Ding et al., 2011] have attempted to extend QEMU for multithreaded emulation on multi-core processors. They both map the emulation of virtual processors to different cores on the host processor. The actual emulation is still handled using QEMU for sequential emulation. Thread safety has been accomplished by implementing lightweight mechanisms for atomic operations and serializing signal handling. Parallel execution also complicates the scheduler required for parallel emulation. Virtual cores executing a thread that is currently holding a spin-lock should not be preempted, because that could potentially interfere with the execution of other threads and have a significant impact on performance. COREMU has solved this by modifying the scheduler to detect when guest processes are holding a lock and not allowing their preemption until they have released the lock. [Wang et al., 2011]

Both of the parallel emulators based QEMU were able to scale efficiently to emulate a few dozen of virtual cores [Wang et al., 2011; Ding et al., 2011]. However, they do not scale efficiently beyond that so that the emulation of many-core processors with hundreds or thousands of cores would be viable. Even if they did scale, emulating hundreds of cores of a guest processor on a small number of of host processor cores would be really slow.

4.3.2 GPU accelerated many-core simulator

Existing CPU simulators are not a good fit for the simulation of many-core processors. Intuitively the only way to reach parallel simulation of many-core processors is to run the simulation on a comparable number of processing units. There have been attempts to parallelize simulation by using distributed computing. Such solutions do not scale well due to synchronization overhead. A logical approach would be to use a many-core processor for the simulation of guest many-core processors. GPGPU computing is currently the easiest way to take advantage of many-core processors.

The state of the art of using GPUs to accelerate simulation of many-core

processors is the simulator infrastructure developed by Raghav et al. [2012b] called *SIMinG-1k*. Their simulator is implemented as a CUDA kernel running on a GPU where every virtual core is mapped to an individual thread. The simulator aims to provide fast and scalable functional simulation. The trade-off is that the simulator does not track the state of the processor at microarchitecture level. SIMinG-1k currently supports ARM and x86 instruction sets, but it is designed to be able to model a variety of different architectures. This is achieved by using an offline compiler to translate the instructions in the original ISA to a set of simple micro-operations that are executed in the parallel simulator.

The simulation loop of each thread follows the steps taken by a classical processor. The loop fetches the bytecode of the next micro-operation instruction, decodes the instruction and executes its semantics. The usage of the micro-operation instruction set makes it possible to perform the decode phase as a simple look-up in one step. The execution phase is implemented as a *switch-case* construct, where the instruction that is being executed decides which branch is taken. [Raghav et al., 2012b]

There is a number of challenges that must be taken into account when implementing GPGPU software. Raghav et al. [2012b] identified some challenges for their simulator implementation and presented possible solutions for them. The challenges are a direct consequence of the CUDA programming model and the structure of GPU hardware that all GPGPU programs need to take into account. The first issue follows from the so called SIMT execution model of GPUs. Diverging control flow within a warp can have a dramatic effect on performance. The worst case where all threads take a different branch leads to completely serial execution. The second issue that can have a huge impact on performance is inefficient use of memory. Threads should exploit data locality as much as possible by using the shared programmable cache on each processing core. Programs should also aim to minimize memory accesses and memory bank conflicts. The last challenge is to minimize interaction between CPU and GPU as it is a very costly process.

The previous version of SIMinG-1k's implementation suffered from control flow divergence during the instruction decode phase of complex instruction set computing (CICS) ISA instructions, such as x86 ISA [Raghav et al., 2010]. This follows from the need to decode CISC instructions in multiple conditional stages. The complex decode structure has been avoided by translating the instructions ahead of execution to micro-operations that can be decoded in single step with a simple table look-up. Control flow divergence cannot be avoided when executing task parallel programs where each thread has a separate instruction stream. Executing data parallel programs, such as programs meant to be executed on a GPU in the first place, can be ex-

ecuted efficiently on the simulator because efficient GPU programs should have mostly coherent control flow. [Raghav et al., 2012b]

Global memory operations are very costly. Performance can be increased dramatically by exploiting data locality by moving often accessed data to the on-chip shared memory. SIMinG-1k utilises shared memory by copying the virtual processor context structures to shared memory. Copying data to shared memory should be optimized by coalescing memory operations so that the threads in the same warp are copying a contiguous area in memory. This way the copy operation can be performed using a small number of actual memory operations as threads are accessing data on the same memory line. [Raghav et al., 2012b]

Synchronization across threads is an important feature needed for shared memory parallel programming. The GPU execution model only offers synchronizations for blocks belonging to the same cooperative thread array. Hardware synchronization primitives such as *test-and-set*, *spin locks*, *wait-for-event* and *signal-event* can be implemented using CUDA atomic operations and using global memory for communication and messaging. Naive busy waiting implementations can lead to deadlocks if there are less processing elements available on the host GPU than there are simulated virtual processors. This occurs because the busy waiting threads do not allow blocked threads to begin execution. Proper inter-thread synchronization is needed for yielding the simulation for other threads, which is possible only by halting the execution of the GPU simulator and returning to host code for synchronization. As this is a very costly operation in terms of performance, synchronization should be avoided if possible. Raghav et al. [2012b] have proposed to solve this issue by preempting the execution of a virtual processor through synchronization only when a synchronization instruction has been detected during execution or alternatively after some thread has been busy waiting for an event for a certain time period.

In their benchmark tests Raghav et al. [2012b] have found that it is possible to reach good scalability by simulating many-core processors on GPUs. SIMinG-1k simulation scales well up to 2048 simulated cores and can match, or even outperform, state-of-the-art simulators such as OVPSim [OVP] that use binary translation as their simulation method when simulating processors with over one thousand cores.

4.3.3 Extending QEMU with a parallel simulator

The SIMinG-1k parallel many-core simulator described in section 4.3.2 describes the implementation methods of a many-core processor simulator that is executed on a GPU. Raghav et al. [2012a] describe how this parallel simu-

lator can be paired together with QEMU, an existing sequential full system simulator. The goal is to allow the sequential simulator to offload the simulation of the parallel regions in program code for simulation on the parallel simulation running on the GPU, just as in native execution the parallel region would be offloaded for execution on the GPU. In should be noted that this method differs fundamentally from the parallelization efforts of QEMU described in section 4.3.1 as the simulation in QEMU itself remains sequential.

The implementation relies on a feature called semihosting [ARM] for extending QEMU with the parallel simulator. Semihosting is a feature originally developed for ARM guest processors for redirecting IO system calls to host's debugger interface. The semihosting feature is used to intercept IO calls meant for the simulated accelerator and triggering the execution of the external GPU simulator process. The software component that handles the forwarding of semihosting calls to the host processor is a device driver for linux that is called everytime an application wishes to use the simulated GPU device. The driver calls a special semihostring interrupt and sets the associated parameters which the QEMU process running on the host will intercept. [Raghav et al., 2012a]

Another software component that is needed in order to write programs that can offload computation to an accelerator is a parallel programming API library. Raghav et al. [2012a] have implemented a custom API library called libGPUSim that roughly resembles the functionality of OpenCL, but a direct replacement for CUDA or OpenCL runtime library would be possible too. The library needs to perform all the necessary calls to the device driver. The library also needs to construct the required parameter structures for the IO calls, such as providing pointers to the parallel function entry or to a data region. The different operations that a parallel API library should be able to perform in current GPU computing model is summarized in the following list:

- Allocate and deallocate device buffers.
- Transfer data between host and device memories.
- Offload kernel execution to the device.
- Wait for execution to finish.
- Release the accelerator when computation is finished.

The QEMU process needs to be modified with capability to intercept and handle semihosting calls directed for the external GPU simulator. It is

also the responsibility of QEMU to transfer required parameters and data between the guest and host environments. Upon the first request directed at the parallel simulator QEMU forks a daemon process that launches the GPU simulator kernel and starts waiting for incoming commands. [Raghav et al., 2012a]

Chapter 5

Benchmarks

A set of different benchmarks were used to calibrate and evaluate our measurement setup. In this chapter we present three different types of benchmarks and explain the motivation behind the selections. First we present microbenchmarks, which are used to analyse the microarchitecture of the hardware. After that we present a real world application benchmark to demonstrate how our system can be used to evaluate real world software.

5.1 Microbenchmarks

Microbenchmarks are used to stress different parts of architectural components on hardware. Microbenchmarks can be used to assess different characteristics of hardware, such as the performance or energy consumption of individual hardware components. A common use for microbenchmarks is for calibrating power and energy models by implementing a programs that continuously stress a single architectural component of a device and measuring the power consumption during execution.

Another use-case for microbenchmarks is to verify the information given by hardware vendors and explore the functionality of hardware in more detail. Since GPUs are typically very closed devices, using microbenchmarks is a good way to gain information about the hardware. As an example, Wong et al. [2010] used microbenchmarking to gain detailed information about the hardware architecture of NVIDIAs GT200 GPU. They managed to extract information about the hardware that was not documented by the vendor.

The microbenchmarks implemented for finding out per instruction energy consumption execute the same instruction in a loop continuously, counting the number of loop iterations. Each loop contains 500 statements of the same type. The benchmarks were implemented for the integer type and both

32- and 64-bit floating point types. Benchmarks were implemented for the following instruction types:

- Arithmetic instructions (Multiply and addition)
- Division
- Special instructions (Square root, sine, cosine)
- Control operations (Comparison, branch)

Memory operations were tested with similar benchmarks that copied memory around between variables either in global or shared memory.

5.2 Application benchmark

5.2.1 Motivation

As a proof of concept, we want to test and demonstrate our setup with an application benchmark that behaves similarly to a real world application. Many of the benchmarks included in benchmark suites are algorithms that are typically used in applications running on desktop computers, or even super computers and computing clusters. We are more interested in examining algorithms that are more likely to be used in mobile applications and devices.

5.2.2 Computer vision

We have chosen to evaluate our setup using an algorithm from the field of computer vision. Face and object recognition is being used increasingly much on mobile devices as the processing power rises. Currently most computer vision algorithms are implemented as sequential applications. This limits the complexity and accuracy of the algorithms that can be used on devices with limited processing power, such as mobile phones. However, computer vision and image processing algorithms are a prime candidate for accelerated data parallel computation as they are by definition so called embarrassingly parallel algorithms.

There are many use cases for object recognition algorithms. Face detection and recognition is already used in many camera applications on mobile phones and stand-alone cameras. Augmented reality applications also rely on object recognition and tracking. Face recognition is also used as an alternative phone unlocking mechanism on some platforms. All these algorithms are currently implemented using fairly simple and inaccurate algorithms.

We have chosen a simple face detection algorithm as our application benchmark. Face detection is an algorithm that detects human faces in an image, but it cannot recognise who that person is. A face detection algorithm tries to find characteristics and shapes that are typical for human faces, such as eye, nose and mouth shapes, that are arranged in a certain way related to each other. Our implementation is based on the face detection sample application included in the OpenCV computer vision library. The application has been modified to use the CUDA implementations of the image processing and pattern recognition functions included in the GPU module of OpenCV. As an input the application takes an image or a video stream. The application detects one or many faces in the input image and tries to also detect eye positions within every subimage containing a detected face.

Computer vision algorithms perform rather simple calculations on the pixel colour of nearby algorithms. As such, they can be implemented to use caches and memories close to processors efficiently. As pixel colours are described using integer values, the benchmark is expected to stress the integer arithmetic and comparison units more than their floating point counterparts.

5.2.2.1 Software modifications

Some modifications were required for accelerating the face detection algorithm on a GPU. The *Mat* data structure used for representing matrices of data in OpenCV was changed for the *GpuMat* data structure included in the GPU module of OpenCV. The GPU module also contained GPU accelerated versions of all the OpenCV functions used in the face detection application implemented with CUDA.

GPU Ocelot, the simulation tool we chose for tracing the execution of GPU accelerated applications, was initially not able to execute the face detection application. Some changes were required in the simulator framework and the OpenCV library to work around bugs and unimplemented features.

NVIDIA's CUDA compiler *nvcc* version 4.2 generated PTX code that was not fully compliant with the PTX language specification. The rounding modifiers for the floating-point division, square root and reciprocal operations were in some cases ordered incorrectly in the statements. We overcame this problem by modifying the PTX language grammar definition of the parser of GPU Ocelot.

As GPU Ocelot currently has not implemented asynchronous memory transfer operations, its source code was modified so that those operations were forwarded to the synchronous memory transfer functions. Performing the memory transfers synchronously affects the performance of the application negatively.

The execution of the application also crashed when being executed by GPU Ocelot's emulator when a memory transfer operation to constant device memory was issued. Constant memory was used in one kernel function in OpenCV. The constant memory array was changed into a shared memory array, which allowed the application to work correctly. As shared memory is slower than constant memory, this affects the performance of the application negatively.

Chapter 6

Evaluation

In this chapter we present the mathematical model that we will use to estimate the energy consumption of GPU accelerated program execution. The model assumes that different architectural components are either on or off, and thus have discrete levels of power consumption over time. After presenting the model, we describe the measurement setup and trace analysis methods used for estimating the energy consumption. Finally we present the results from the microbenchmarks and validate the results and the power model against the face detection benchmark.

6.1 Energy estimation method

6.1.1 Power model

In this thesis, we aim to model the energy consumption of a computing platform offloading data parallel computation to a many-core accelerator, which in our case is a GPU. The energy consumption of the accelerated computation can be split into two parts, execution of compute kernels on the GPU and data transfers over a PCI Express bus. We can consider the sequential execution of the program to form the third component of the total energy consumption E_{total}. We can thus describe the total energy consumption with the following formula:

$$E_{total} = E_{CPU} + E_{GPU} + E_{data\_transfer} \tag{6.1}$$

In the following sections, we describe a power model used for estimating the energy consumptions of the compute kernel execution, E_{GPU}, and PCI Express data transfers, $E_{data\_transfer}$. Energy estimation of sequential execution on CPUs using similar methods, namely using fast instruction-accurate

simulators, has already been shown to be feasible [Miettinen and Hirvisalo, 2009].

6.1.1.1 GPU computation power model

As described in Section 2.5.2, the total power consumption is split into static and dynamic parts. Isci and Martonosi [2003] demonstrated a way to estimate the overall power consumption of a processor based on performance-counter metrics. Their model describes power consumption as a sum of the power consumption of architectural components of the processor. The power model is expressed as:

$$\begin{aligned} Power = \sum_{i=0}^{n} (&AccessRate(C_i) \times ArchitecturalScaling(C_i) \times MaxPower(C_i) \\ &\times NonGatedClockPower(C_i)) + IdlePower. \end{aligned} \tag{6.2}$$

AccessRate describes how often an architectural unit is accessed over time. The other components in the equation, such as $MaxPower$ and $ArchitecturalScaling$ are determined heuristically.

Equation 6.2 was used to express the power model of a traditional Pentium 4 CPU. Hong and Kim [2010] extended this model for expressing the power model of a GPU. Their GPU power model can be described as:

$$GPU\_power = RuntimePower + IdlePower \tag{6.3}$$

$$\begin{aligned} RuntimePower &= \sum_{i=0}^{n} RP\_Component_i \\ &= RP\_SMs + RP\_Memory \end{aligned} \tag{6.4}$$

The above power model expresses power as the sum of the power consumptions of all the processing cores (RP_SMs), or streaming multiprocessors (SM) and the power consumed by the memory (RP_Memory).

The power consumption caused by operations by processing cores, RP_SMs, can be expressed with the following equation:

$$RP\_SMs = Num\_SMs \times \sum_{i=0}^{n} SM\_Component_i, \tag{6.5}$$

where the $SM\_Component$ term describes the power consumption of individual architectural components, such as integer or floating-point units, on a processing core and Num_SMs is the total number of processing cores.

The dynamic power consumption RP_Memory can be described with the following equation:

$$\begin{aligned} RP\_Memory &= \sum_{i=0}^{n} Memory\_component_i \\ &= RP\_GlobalMemory + RP\_LocalMemory. \end{aligned} \tag{6.6}$$

Modeling the dynamic power consumption of memory operations is done identically to other types of operations.

Since current generation GPUs do not employ clock gating, Hong and Kim [2010] propose that the architectural component power can be described with the following equation:

$$RP_{comp} = MaxPower_{comp} \times AccessRate_{comp}. \tag{6.7}$$

The above equation can be interpreted so that an architectural component consumes a certain amount of dynamic power when active, defined by $MaxPower$, and it consumes only idle power when it is inactive.

The $MaxPower$ component in equation 6.7 can be determined by constructing microbenchmarks that stress a certain architectural unit and measuring the power consumption during execution. For example, floating-point unit power consumption can be observed with a benchmark that performs floating-point operations repeatedly in a loop. Similar microbenchmarks can be constructed for other architectural units.

The access rates are computed by using the emulator in the GPU Ocelot framework described in Section 4.2.2.4 to count instructions that are executed by different architectural components. The different instruction classes categorized by the implemented kernel profiler are:

- Integer arithmetic
- Integer division
- Integer logical
- Integer comparison
- Float arithmetic
- Float division

- Combined float and double comparison
- Double arithmetic
- Double division
- Transcendental and other special operations
- On-chip memory accesses
- Off-chip memory accesses
- Control flow operations
- Parallelism operations
- Other operations.

The contents of most of the instruction classes in the above list should be intuitively apparent. The *parallelism operations* class contains reduction, vote and barrier instructions. The *other operations* class consists currently of a single *cvta* instruction, which is used to convert addresses between different address spaces. Division operations have been separated from other arithmetic operations as it is a much more expensive operation. In the analysis phase the *control flow*, *parallelism* and *other* operations are grouped together with *logical operations* as they are difficult to microbenchmark separately and they are assumed to have similar energy consumption characteristics. This may cause some inaccuracy to the model.

The access rate of architectural component i can be described using the following equations:

$$TotalCycles = \sum_{i=0}^{n} InstructionCount_i \times CyclesPerInstruction_i \quad (6.8)$$

$$AccessRate_i = \frac{InstructionCount_i \times CyclesPerInstruction_i}{TotalCycles}, \quad (6.9)$$

where $InstructionCount_i$ is the number of instructions executed on the architectural unit i and $CyclesPerInstruction_i$ is the duration it takes to execute those instructions in clock cycles. $CyclesPerInstruction_i$ depends on the modeled hardware.

6.1.1.2 PCI Express memory transfer power model

Data transfers over PCI Express consume a significant ratio of the total energy consumption of GPU accelerated computing. The data transfers activate many different components of the system, including GPU and CPU memories, DMA controller and the PCI express bus components. It if not possible to directly measure the effects of data transfers on the power consumption of these components. However, we can measure and model the effect of the data transfers on the total power consumption of the system.

The time of PCI Express data transfers depends on the amount of data to be transferred. There is also a fixed latency associated with every data transfer operation. The bandwidth may also depend on the size of data. [Hovland, 2008] The time of PCI Express data transfers can be estimated with the Hockney model [Hockney, 1994], which can be expressed as:

$$Time = Latency + Bandwidth \times DataSize. \tag{6.10}$$

The increase in power consumption during PCI Express data transfers is assumed to be more or less constant. The power consumption is measured by implementing a simple program that transfers memory between device and host memories and comparing that to the idle power consumption.

6.1.1.3 GPU operating state modeling

Current generation GPUs do not employ clock gating, but they do employ a coarse grained DVFS functionality. GPUs can operate at a small number of discrete performance levels. NVIDIAs Fermi cards for example can operate at three different performance levels with differing clock frequencies and operating voltages. The lowest operating level can perform typical operating system user interface tasks, but when more computing performance is needed, the GPU driver immediately raises the operating level to the maximum performance operating level. There is additionally a third operating level that sets the clock frequency to such level that the GPU is capable of encoding and displaying high-definition video.

The GPU remains in the current operating state for some time even after GPU load has dropped to such level that moving to a more energy saving operating state would be possible. This is done for hysteresis reasons, because switching between operating states is a costly operation and especially operations that draw graphics to the screen tend to occur cyclically. These tails states can be quite long in desktop GPUs. For example, NVIDIAs Fermi generation cards with current Linux drivers (version 304.54) take 10 seconds to drop from the highest operating level to the middle one and further 5

seconds to switch to the lowest operating level. Similar behaviour has been observed in mobile GPUs

Because static power consumes a significant portion of the total power consumption, power consumption at these operating levels must be measured and modeled. Measuring the power consumption is very straightforward. The GPU simply must be stressed with a short compute load and then measure the power consumption as the GPU switches down to the lowest operating level.

6.1.2 Measurement setup

The measurements are done using a Corsair AX760i power supply that is capable of measuring and logging instantaneous power consumption. The power supply is connected to a separate measuring computer with a USB cable that handles logging of the results. The measurement setup is pictured in Figure 6.1.

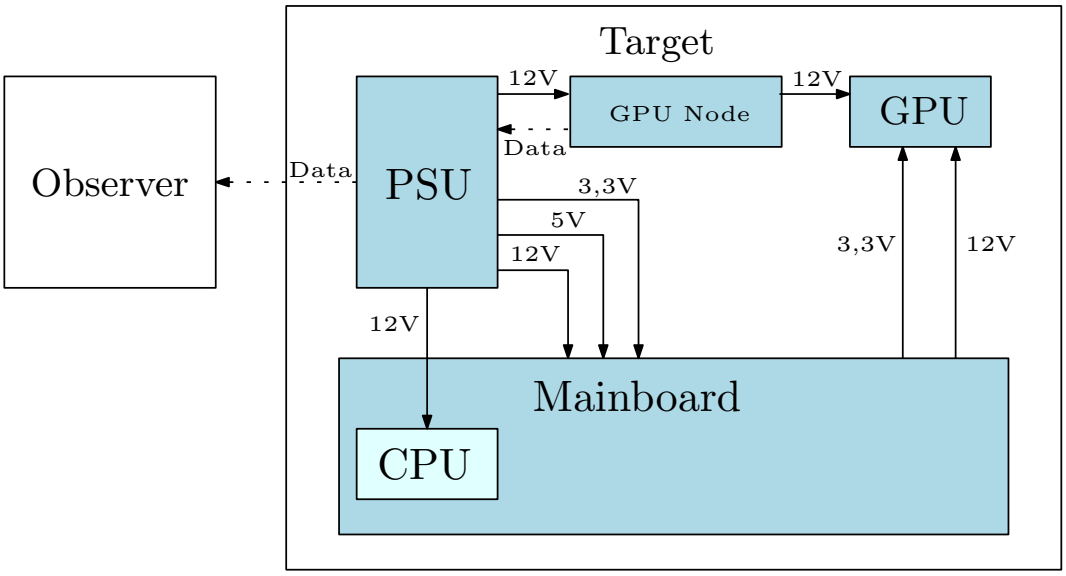


Figure 6.1: Measurement setup.

The power supply is capable of measuring the voltages and currents of power rails separately. However, the current values are only reported if the current exceeds 3A current. Instantaneous power consumptions are reported at all time for the PSU input and output power. By using a separate GPU Node the PSU also reports the instantaneous power consumption for the

auxiliary PCI-Express power inputs of the graphics processing unit. The power consumption can be logged at the frequency of one second.

The energy consumption of the implemented microbenchmarks described in Section 5.1 is measured over a period of 10 minutes. A warm-up period of one minute is used to make sure the system reaches a steady state before the measurement begins. Additionally the idle energy consumption of the system is measured similarly with the GPU set to run at the different operating states that were presented in Section 6.1.1.3.

6.1.3 Trace analysis

The instrumented traces from all instrumentation tools output a plaintext trace that were parsed using a custom-made Python script. The implementations consisted of rather basic CSV, or similar format, parsing. The figures and charts were visualised using Matplotlib plotting library.

GPU Ocelot was used to simulate and trace the GPU program execution. The implemented trace generator counts instructions of certain types per kernel and at the end of kernel executions it prints out the counters. The following is an example of the GPU Ocelot trace format:

```
Kernel name: _Z9transposeIjEvPT_jS1_j10NcvSize32u
Integer_arithmetic: 9041472
Integer_logical: 828432
Integer_comparison: 1378080
Memory_offchip: 526338
Memory_onchip: 2724930
Control: 1943793
Parallelism: 278784
```

The instantaneous power consumption traces generated by the Corsair Link software are in simple comma-separated list format. Note that the decimal point in the following format is also comma. The data on each line was selected to be date and time, PSU efficiency, power in, power out, and GPU supplementary PCI-E power. The following line is an example of the output format of the trace:

```
15.7.2013 11:36:16,20,7W,89,8766315921014,151,9921875,136,605458408151
```

Instruction	Total (nJ)	Main (nJ)	Supplementary (nJ)	Main (%)
Integer arithmetic	0.11	0.03	0.08	27 %
Integer division	2.08	0.30	1.78	15 %
Integer SFU	1.99	0.54	1.44	27 %
Integer other	0.62	0.17	0.45	27 %
Float arithmetic	0.08	0.01	0.07	19 %
Float division	4.67	0.77	3.90	17 %
Float SFU	1.72	0.52	1.20	30 %
Float other	0.83	0.24	0.59	29 %
Double arithmetic	0.79	0.25	0.54	31 %
Double division	12.94	3.59	9.35	28 %
Double SFU	1.75	0.55	1.20	31 %
Double other	0.97	0.26	0.71	27 %
Global memory	17.95	4.55	13.40	25 %
Local memory	0.20	0.05	0.15	25 %

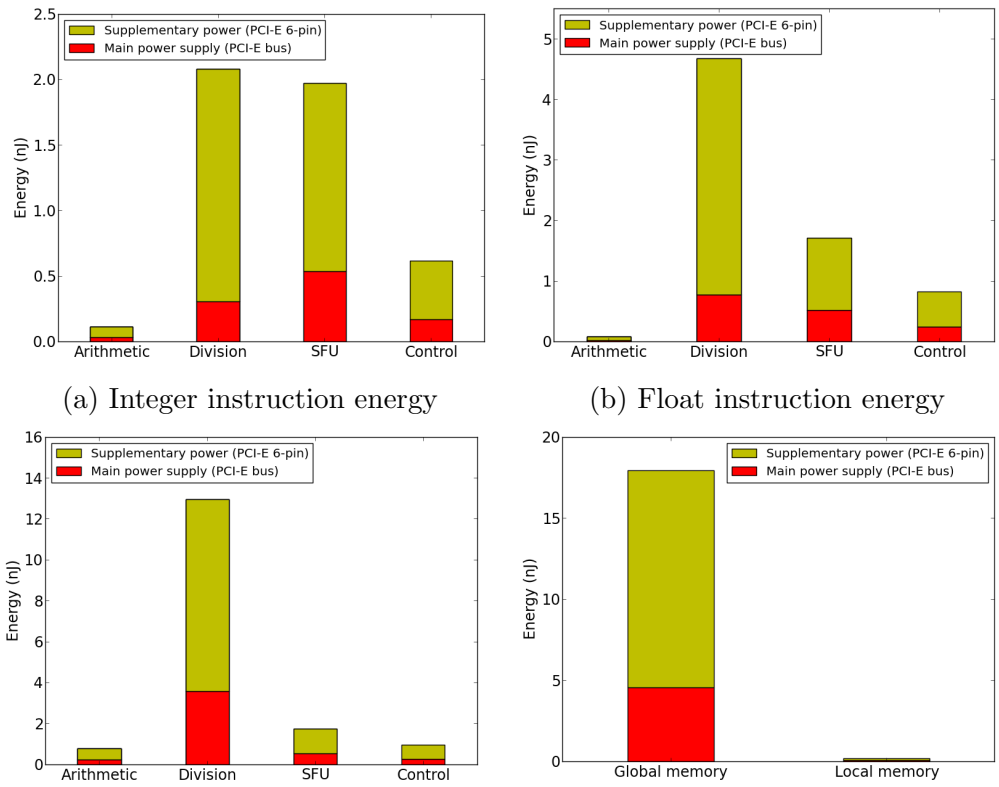
Table 6.1: Per instruction dynamic energy consumptions

6.2 Results

6.2.1 Microbenchmarks

The per instruction dynamic energy consumption of different instruction classes are listed in Table 6.1 and visualised in the following figures. The table shows the total energy consumption per instruction as well as energy consumptions through the main PCI-E bus and the supplementary power input to the GPU. It also shows the ratio of energy drawn through the main power input. Figure 6.2a shows the per instruction energy consumption for integer operations. It was discovered empirically that the division operation is a much more expensive operation than other arithmetic operations for all data types, which is the reason for measuring it separately. The division operation turns out to consume an order of magnitude more energy than other arithmetic operations.

The energy consumption of 32-bit integer and floating point operations appear to be rather close to each other except for the division operation which was measured to consume about twice the energy of integer division. Special functions and the *other* instruction class energy consumptions were observed to be similar for all data types. This makes sense as the special functions are typically implemented using look-up tables which makes the execution similar for different data types.



(a) Integer instruction energy

(b) Float instruction energy

(c) Double instruction energy

(d) Memory instruction energy

Figure 6.2: Energy consumption per instruction

Arithmetic operations for double-precision floating point data type were measured to be an order of magnitude more expensive than single-precision operations. The magnitude of this difference is rather notable. The cost of double precision operations may be the reason why the double precision performance of NVIDIA's GPUs have been lagging behind up until Kepler GPUs. The energy consumption of control instructions is also similar for all data types, which is to be expected. The similarity of energy consumptions for these instruction types add confidence for the sanity of the measurements.

Overall, these energy consumption figures seem believable and logical. NVIDIA [d] has said that the Fermi GPUs consume about 200 picojoules of energy per instruction, including static power, which is more or less in agreement with our results. Lucas et al. [2013] measured 40 pJ for integer instructions and 75 pJ for floating point operations for an older Tesla series GPU. Our measured results are in the same vicinity.

As expected, accessing the main memory of the GPU is the most energy consuming operation. As a rule of thumb, the farther away memory is in

the memory hierarchy, the more energy is consumed when accessing it. The actual energy consumption of accessing the global memory is probably higher than the measured value due to cache hits in the benchmark. Shared memory, which resides on the chip, is much cheaper to access. The difference of energy consumption per memory access is visualised in Figure 6.2d.

The power consumption of continuous host-device memory transfer over the PCI-Express bus can be seen in Figure 6.3. The instantaneous power consumption is about 40 watts, which is a significant addition to the total power consumption of the system. It should be noted that memory transfers can occur concurrently with computation on the GPU. *Supplementary* power in the figure refers to the power draw through the supplementary 6-pin PCI-Express power cables and *main* power refers to power consumption increase in the rest of the measured system that includes power drawn through the PCI-Express bus, host memory activation and the DMA controller power consumption.

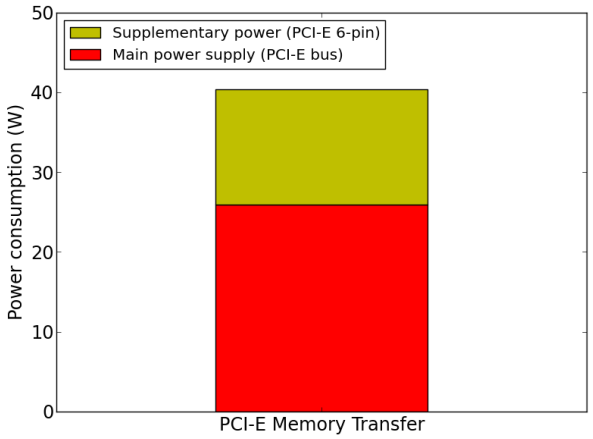
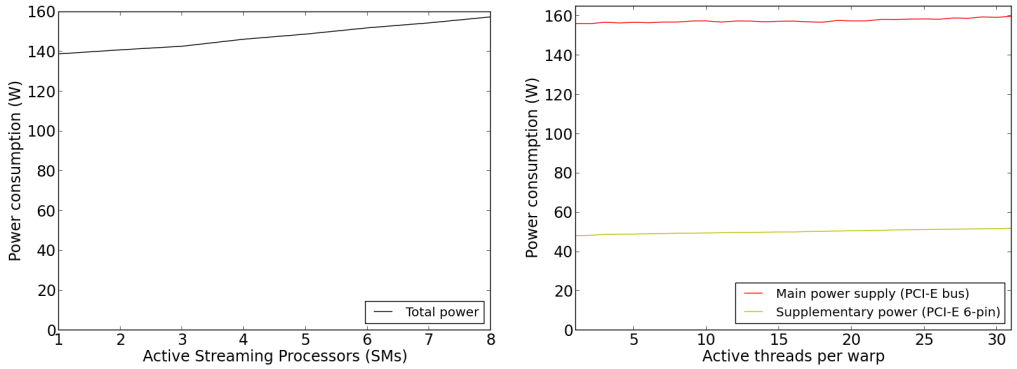


Figure 6.3: Power consumption of continuous host-device memory transfer.

Figures 6.4a and 6.4b show how the number of active processing cores or processing elements affect the overall power consumption of the GPU. In both cases it seems that when the number of active elements is increased, the power consumption rises linearly. The difference between one active processing core versus all eight cores actively processing data in this case is about 20 watts. The used benchmark performs a mixture of different arithmetic operations on data. The number of active processing elements per processing core due to thread branching has a much smaller effect on the power consumption. The difference between one thread in a warp actively executing versus all 32 threads in a warp actively executing is only about 3



(a) Power consumption of SMs (b) Power consumption of PEs

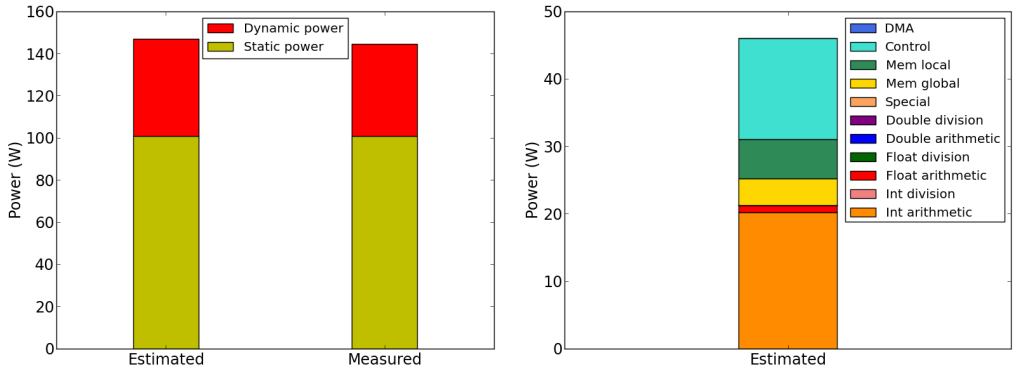
Figure 6.4: Power consumption with respect to SM and processing element activations

watts with this benchmark. The small difference can be explained by the fact that even the threads that have not taken the branch execute the instructions inside the branch, but the results of the execution are not written to memory. The results were still a bit surprising. It is quite easy to see that the static power dominates the energy consumption even under moderately heavy load. The linear growth of energy consumption has also been observed by others [Wang and Ranganathan, 2011; Hong and Kim, 2010].

6.2.2 Face detection

The face detection application presented in Section 5.2.2 was used to evaluate our power model and the measured architectural component base power consumptions. The base powers and the access rates obtained by simulating the execution of the application with GPU Ocelot were fed into the equations we presented in Section 6.1.1. The power model predicts an increase of 46.1 watts over the idle power consumption with the GPU in the highest power state. The overall predicted power consumption of the whole system is 147.0 watts. The measured power consumption for the whole system was 144.5 watts. The difference between the predicted and measured power consumption was 2.5 watts, which is 5.4% of the predicted dynamic power increase. The predicted and measured power consumptions are also visualised in Figure 6.5a.

Figure 6.5b shows the breakdown of the contributions of different types of instructions to the predicted dynamic power consumption. *Arithmetic integer instructions* and the *other instructions* class contribute most to the power



(a) Estimated and measured power consumption

(b) Dynamic power consumtion per instruction type

Figure 6.5: Power consumption of the face detection application

consumption. Global and local memory operations also have a significant impact on the predicted power consumption. DMA memory transfers do not seem to affect the power consumption a lot in this case, which is explained by the small size of the used input image and the relatively low frequency of DMA memory transfers.

Chapter 7

Simulating GPU energy consumption

The focus of this thesis so far has been in measuring and estimating the energy consumption of a real and existing graphics processing unit. The preferred targets for energy consumption estimation are non-existing or unavailable GPUs. In this chapter we will try to discuss the suitability of the presented model and explored simulation techniques in the energy estimation of different GPU architectures.

7.1 GPU energy consumption observations

The measurements in Section 6.2 confirmed the rather obvious assumption that the activations of different hardware components correlate with the overall energy consumption of GPUs. What was somewhat surprising was that the effect of the activations of individual processing elements was relatively small. The amount of computation performed by a processor mattered far less than the time the processor was computing anything at all. Current desktop GPUs do not employ very sophisticated energy saving mechanism. The GPUs are typically running at the highest power state even when there are pauses between computation. The static energy consumption of the highest power level is by far the biggest factor to the overall energy consumption of GPU accelerated processing.

Since the activations of the whole GPU and its individual processors correlate with the energy consumption, there is also a very direct relationship between the time a program is executing on the GPU and the energy. On existing systems finding out the timing can be done trivially by profiling the program execution. If real hardware is not available, a simulator must be

used.

The second most important factor to both performance and energy efficiency is the efficient use of caches and shared memory and thus minimizing accesses to global memory. The further away data must be fetched, the more energy is spend during the operation. Memory accesses are also slow, which may stall the execution of the program, which causes further energy losses. The memory usage can be quite architecture dependent even for the execution of the same program. Different GPUs have different memory configurations that affect the energy consumption and the timing of the program execution.

There are plenty of differences between GPUs as well that affect timing and energy efficiency. For example, NVIDIA's latest desktop GPU architecture, Kepler, improved computational capabilities of each core significantly, especially double floating point performance, but since the memory hierarchy did not get a similar boost the memory bandwidth has become more of a bottleneck for general purpose computation loads [Kanter, 2012]. There can also be more subtle architectural differences, for example differing scheduling or power management practices, but such details are usually not published by vendors and must be observed through reverse engineering means.

Mobile GPUs are fundamentally different due to much more severe energy consumption restrictions. Traditionally the mobile GPUs have been focused exclusively on graphics rendering, but recently the trend has been to include general purpose computation capabilities as well. The biggest difference between current mobile and desktop GPUs is that mobile GPUs typically share memory with the CPU, and that memory is usually slower than traditional desktop GPU memory. Mobile GPUs are much more reliant on using local shared memories and minimizing global memory accesses.

The trend seems to be that mobile and desktop GPU architectures are becoming more and more similar with each other. The Heterogeneous System Architecture (HSA) [AMD] pushed by AMD and some other hardware manufacturers promotes and architecture with stronger coupling of the CPU, GPU and other accelerators. On the otherhand, NVIDIA has been demonstrating a version of their Kepler GPU architecture that is also suitable for mobile computing [Alben, 2013].

7.2 Simulating GPU energy consumption

One of the biggest issues with using GPU simulators to predict the energy consumption of program execution is the performance of simulation. The sheer amount of threads and processing elements on a GPU make the simu-

lation very slow. Existing energy predicting simulators have been based on cycle-accurate simulation, which is not even nearly fast enough for the use of most software developers. We have found instruction-accurate simulation to be accurate enough to give a good idea of the energy consumption of a GPU. However, the approach we used is not without flaws.

Our energy estimation model relies solely on executed instructions. The mostly unmodified version of the GPU Ocelot emulator we used uses a simplistic execution model. Especially the estimation of the memory access energy consumption could be improved greatly by simulating the cache behaviour as well. GPU Ocelot includes a simple L2 cache simulator, but it was not used as it is based on an old GPU architecture and cannot be configured to match more recent architectures. However, implementing a cache simulator for the full cache hierarchy of GPUs would make the overall simulation even slower.

However, since simple instruction counting is already sufficient for estimating energy consumption at an acceptable accuracy, it may be possible to simulate the program execution once and output energy consumption estimates for multiple target platforms simultaneously. What is needed is information about the execution times and architectural component energy consumptions relative to a known and existing GPU. That way we would be able to project the energy consumption data of a known platform to non-existing GPU architectures.

Considering the real-time nature of a large portion of common GPU accelerated algorithms, increasing the GPU simulator performance is an essential step for making the simulation techniques practically usable. Previous parallelization efforts have proven the feasibility of parallel GPU simulation. Such approaches should be applied to the more mature GPU simulators.

The performance and accuracy trade-off is always present when we talk about simulation. However, as the performance counters of current GPU architectures do not expose enough detail about the execution, we have no choice but to rely on simulation to obtain more detailed information about program execution on GPUs.

7.3 Simulating future GPU architectures

The flexibility of GPU simulators makes it possible to adapt them for future GPU architectures. While current desktop GPUs have been following the CUDA compute model quite religiously in their hardware design, this may not be the case for mobile GPUs and future GPU generations. The modular structure of GPU Ocelot is a good example of a framework that is a good fit

for architectural exploration and can be easily modified to match the changes in GPU architectures. The way GPUs are divided into processing cores and processing elements varies a lot between different manufacturers and GPU types. GPUs are also equipped with different memory architectures, which affects the overall performance and timing of events. These are exactly the sorts of effects that could be examined more easily with a configurable simulator.

The problem with most current GPU simulators is that they simulate an abstract GPU model. Simulators cannot be made fully accurate as long as details about GPU execution are not known. It seems like manufacturers are slowly reacting to this, and simulators such as Multi2Sim are starting to support the simulation of actual hardware instead of just abstract computation models.

There is a clear trend that CPUs and GPUs are being brought both physically and logically closer together. AMD has already introduced an architecture for desktop computers where a high-performance GPU is integrated to the same chip as the CPU. There is also plans to have a coherent shared memory model for different computing elements on the same chip. In such an architecture computing devices do not only shared the same physical memory, but also the same virtual address space. In the long run it is not enough to observe the GPU characteristics in isolation of the rest of the system. GPU simulation should be integrated into full system simulators.

General-purpose GPU computation on mobile platforms is an emerging field and there is not a lot of information available about either the performance or energy efficiency of executing programs on mobile GPUs. It would be essential to implement mobile GPU models in existing simulators and verifying the simulation results with real hardware. The first possible hardware targets are the Mali-T604 and Adreno 320 mobile GPUs that have had OpenCL drivers and libraries released on some devices.

Chapter 8

Summary

In this thesis we explored the platform operation, both on the host platform as well as on the GPU device, of GPU accelerated processing. As the native execution on the device is difficult to observe, we focused on presenting methods that were largely based on using GPU simulators for unveiling the more fine-grained details of execution. We also found that the more coarse-grained information can be obtained by observing the processing from the point of view of the host platform by reading the performance counters reported by the drivers.

As we saw in Chapter 4, there is a smorgasbord of different GPU simulators available and they all approach the problem from a slightly different angle. They are also generally speaking in the relatively early stages of development with plenty of desired features missing. Basically the simulators can be categorised either as detailed cycle-accurate simulators or faster instruction-accurate ones. This thesis concentrated more on fast, but relatively inaccurate, simulation which is more suitable for software developers' needs.

The results we obtained by estimating energy consumption using the power model presented in Chapter 6 were promising. We could show that even when the simulator ignores some of the details of execution on the GPU, we can still end up with a fairly accurate estimate. In the future, it would be desirable to speed up the simulation speed by means of parallel simulation, and improving the simulated GPU model to include functionality such as cache hierarchy simulation to improve simulation accuracy.

The next step from the results of this thesis would be to try to utilise the explored methods for other platforms, such as mobile GPUs. Estimating desktop GPU energy consumption is not very interesting, but using the more powerful GPUs as a basis for estimating energy consumption of embedded GPUs would be desirable. The recent emergence of mobile GPUs with

general-purpose processing support would make such research possible.

Bibliography

OVPSim — Open Virtual Platforms. Online. URL `http://www.ovpworld.org/`.

T. Akenine-Moller and J. Strom. Graphics Processing Units for Handhelds. *Proceedings of the IEEE*, 96(5):779–789, May 2008. ISSN 0018-9219. doi: 10.1109/JPROC.2008.917719.

Jonah Alben. Nvidia brings kepler world's most advanced graphics architecture to mobile devices. *NVIDIA Blog*, July 2013.

Algorithms and Theory of Computation Handbook. single program multiple data. In *Dictionary of Algorithms and Data Structures*. CRC Press LLC, 1999. URL `http://www.nist.gov/dads/HTML/singleprogrm.html`. Online.

AMD. What is Heterogeneous System Architecture (HSA)? Online. URL `http://developer.amd.com/resources/heterogeneous-computing/what-is-heterogeneous-system-architecture-hsa/`.

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. doi: 10.1145/1465482.1465560.

ARM. What is semihosting? Online. URL `http://infocenter.arm.com/help/index.jsp?topic=/com.arm.doc.dui0471c/Bgbjjgij.html`.

Krste Asanovic, Ras Bodik, Bryan Christopher Catanzaro, Joseph James Gebis, Parry Husbands, Kurt Keutzer, David A. Patterson, William Lester Plishker, John Shalf, Samuel Webb Williams, and Katherine A. Yelick. The landscape of parallel computing research: A view from berkeley. Technical Report UCB/EECS-2006-183, EECS Department, University of California, Berkeley, Dec 2006. URL `http://www.eecs.berkeley.edu/Pubs/TechRpts/2006/EECS-2006-183.html`.

D. August, J. Chang, S. Girbal, D. Gracia-Perez, G. Mouchard, D. Penry, O. Temam, and N. Vachharajani. UNISIM: An Open Simulation Environment and Library for Complex Architecture Design and Collaborative Development. *Computer Architecture Letters*, 6(2):45–48, February 2007. ISSN 1556-6056. doi: 10.1109/L-CA.2007.12.

A. Bakhoda, G.L. Yuan, W.W.L. Fung, H. Wong, and T.M. Aamodt. Analyzing CUDA workloads using a detailed GPU simulator. In *Performance Analysis of Systems and Software, 2009. ISPASS 2009. IEEE International Symposium on*, pages 163–174, April 2009. doi: 10.1109/ISPASS.2009.4919648.

Thomas Ball and James R. Larus. Optimally profiling and tracing programs. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 16(4):1319–1360, July 1994. ISSN 0164-0925. doi: 10.1145/183432.183527.

Fabrice Bellard. QEMU, a fast and portable dynamic translator. In *Proceedings of the annual conference on USENIX Annual Technical Conference*, ATEC '05, pages 41–41, Berkeley, CA, USA, 2005. USENIX Association.

Barbara Chapman, Gabriele Jost, and Ruud van der Pas. *Using OpenMP: Portable Shared Memory Parallel Programming (Scientific and Engineering Computation)*. The MIT Press, 2007. ISBN 0262533022, 9780262533027.

S. Collange, M. Daumas, D. Defour, and D. Parello. Barra: A Parallel Functional Simulator for GPGPU. In *Modeling, Analysis Simulation of Computer and Telecommunication Systems (MASCOTS), 2010 IEEE International Symposium on*, pages 351–360, August 2010. doi: 10.1109/MASCOTS.2010.43.

Sylvain Collange, David Defour, and Arnaud Tisserand. Power Consumption of GPUs from a Software Perspective. In *Proceedings of the 9th International Conference on Computational Science: Part I*, ICCS '09, pages 914–923, Berlin, Heidelberg, 2009. Springer-Verlag. ISBN 978-3-642-01969-2. doi: 10.1007/978-3-642-01970-8_92.

Jonathan Corbet, Alessandro Rubini, and Greg Kroah-Hartman. *Linux device drivers*. O'Reilly Media, Inc., 3 edition, 2005. ISBN 978-0-596-00590-0.

Jiun-Hung Ding, Po-Chun Chang, Wei-Chung Hsu, and Yeh-Ching Chung. PQEMU: A Parallel System Emulator Based on QEMU. In *Parallel and Distributed Systems (ICPADS), 2011 IEEE 17th International Conference on*, pages 276–283, December 2011. doi: 10.1109/ICPADS.2011.102.

Michael J. Flynn. Some Computer Organizations and Their Effectiveness. *Computers, IEEE Transactions on*, C-21(9):948–960, September 1972. ISSN 0018-9340. doi: 10.1109/TC.1972.5009071.

B. Gaster, L. Howes, D.R. Kaeli, P. Mistry, and D. Schaa. *Heterogeneous Computing with OpenCL*. Elsevier Science, 2011. ISBN 9780123877673.

Google. Renderscript. Online. URL `http://developer.android.com/guide/topics/renderscript/index.html`.

Vesa Hirvisalo and Jussi Knuuttila. pQEMU – Profiling with and ISA emulator. Technical report, Aalto University, 2010.

R.W. Hockney. The communication challenge for MPP: Intel Paragon and Meiko CS-2. *Parallel Computing*, 20(3):389–398, 1994.

Sunpyo Hong and Hyesoon Kim. An integrated GPU power and performance model. *SIGARCH Comput. Archit. News*, 38(3):280–289, June 2010. ISSN 0163-5964. doi: 10.1145/1816038.1815998.

Rune J. Hovland. Latency and bandwidth impact on gpu-systems. Technical report, Norwegian University of Science and Technology, 2008.

Canturk Isci and Margaret Martonosi. Runtime Power Monitoring in High-End Processors: Methodology and Empirical Data. In *Proceedings of the 36th annual IEEE/ACM International Symposium on Microarchitecture*, MICRO 36, pages 93–105, Washington, DC, USA, 2003. IEEE Computer Society. ISBN 0-7695-2043-X.

David Kanter. Inside Fermi: Nvidia's HPC Push. Online, September 2009. URL `http://www.realworldtech.com/fermi`.

David Kanter. Impressions of Kepler. Online, March 2012. URL `http://www.realworldtech.com/kepler-brief/`.

S. Kato, M. McThrow, C. Maltzahn, and S. Brandt. Gdev: First-class GPU resource management in the operating system. In *USENIX ATC*, volume 12, 2012.

Khronos Group. Opencl. Online. URL `http://www.khronos.org/opencl/`.

Jan Lucas, Sohan Lal, Michael Andersch, Mauricio Alvarez-Mesa, and Ben Juurlink. How a single chip causes massive power bills GPUSimPow: A GPGPU power simulator. In *Performance Analysis of Systems and Software (ISPASS), 2013 IEEE International Symposium on*, pages 97–106, 2013. doi: 10.1109/ISPASS.2013.6557150.

Allen D. Malony, Scott Biersdorff, Sameer Shende, Heike Jagode, Stanimire Tomov, Guido Juckeland, Robert Dietrich, Duncan Poole, and Christopher Lamb. Parallel Performance Measurement of Heterogeneous Parallel Systems with GPUs. In *Proceedings of the 2011 International Conference on Parallel Processing*, ICPP '11, pages 176–185, Washington, DC, USA, 2011. IEEE Computer Society. ISBN 978-0-7695-4510-3. doi: 10.1109/ICPP.2011.71.

Antti P. Miettinen and Vesa Hirvisalo. Energy-efficient parallel software for mobile hand-held devices. In *Proceedings of the First USENIX conference on Hot topics in parallelism*, HotPar'09, pages 12–12, Berkeley, CA, USA, 2009. USENIX Association.

Gordon E. Moore. Cramming more components onto integrated circuits. *Electronics*, 38(8), April 1965.

Nouveau. nouveau wiki. Online. URL `http://nouveau.freedesktop.org/wiki/`.

NVIDIA. Cuda compute unified device architecture. Online, a. URL `http://www.nvidia.com/object/cuda_home_new.html`.

NVIDIA. *NVIDIA CUDA Architecture*, b. URL `http://developer.download.nvidia.com/compute/cuda/docs/CUDA_Architecture_Overview.pdf`.

NVIDIA. CUPTI Metrics API. Online, c. URL `http://docs.nvidia.com/cuda/cupti/index.html#r_metric_api`.

NVIDIA. Doing More with Less of a Scarce Resource. Online, d. URL `http://www.nvidia.com/object/gcr-energy-efficiency.html`.

NVIDIA. *Fermi Compute Architecture Whitepaper*, e.

NVIDIA. *CUDA Compiler Driver NVCC*, f.

OpenACC. Openacc. Online. URL `http://www.openacc-standard.org/`.

PathScale. pathscale/pscnv. Online. URL `https://github.com/pathscale/pscnv`.

David A. Patterson and John L. Hennessy. *Computer Organization and Design: The Hardware/Software Interface*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 3rd edition, 2007. ISBN 0123706068, 9780123706065.

S. Raghav, M. Ruggiero, D. Atienza, C. Pinto, A. Marongiu, and L. Benini. Scalable instruction set simulator for thousand-core architectures running on GPGPUs. In *High Performance Computing and Simulation (HPCS), 2010 International Conference on*, pages 459–466, July 2010. doi: 10.1109/HPCS.2010.5547092.

Shivani Raghav, Andrea Marongiu, Christian Pinto, David Atienza, Martino Ruggiero, and Luca Benini. Full system simulation of many-core heterogeneous SoCs using GPU and QEMU semihosting. In *Proceedings of the 5th Annual Workshop on General Purpose Processing with Graphics Processing Units*, GPGPU-5, pages 101–109, New York, NY, USA, 2012a. ACM. ISBN 978-1-4503-1233-2. doi: 10.1145/2159430.2159442.

Shivani Raghav, Andrea Marongiu, Christian Pinto, Martino Ruggiero, David Atienza, and Luca Benini. SIMinG-1k: A thousand-core simulator running on general-purpose graphical processing units. *Concurrency and Computation: Practice and Experience*, 2012b. ISSN 1532-0634. doi: 10.1002/cpe.2940.

Sameer S. Shende and Allen D. Malony. The tau parallel performance system. *International Journal of High Performance Computing Applications*, 20(2): 287–311, May 2006. ISSN 1094-3420. doi: 10.1177/1094342006064482.

William Stallings. *Operating Systems: Internals and Design Principles*. Prentice Hall Press, Upper Saddle River, NJ, USA, 6th edition, 2008. ISBN 0136006329, 9780136006329.

Herb Sutter. The Free Lunch Is Over: A Fundamental Turn Toward Concurrency in Software. *Dr. Dobb's Journal*, 30(3):202–210, 2005. URL `http://www.gotw.ca/publications/concurrency-ddj.htm`.

Rafael Ubal, Byunghyun Jang, Perhaad Mistry, Dana Schaa, and David Kaeli. Multi2Sim: a simulation framework for CPU-GPU computing. In *Proceedings of the 21st international conference on Parallel architectures and compilation techniques*, PACT '12, pages 335–344, New York, NY, USA, 2012. ACM. ISBN 978-1-4503-1182-3. doi: 10.1145/2370816.2370865.

C. H. (Kees) van Berkel. Multi-core for mobile phones. In *Proceedings of the Conference on Design, Automation and Test in Europe*, DATE '09, pages 1260–1265. European Design and Automation Association, 2009. ISBN 978-3-9810801-5-5. URL `http://dl.acm.org/citation.cfm?id=1874620.1874924`.

Wladimir J. van der Laan. decuda. Online. URL `https://github.com/laanwj/decuda/wiki`.

Guibin Wang. Power analysis and optimizations for GPU architecture using a power simulator. In *Advanced Computer Theory and Engineering (ICACTE), 2010 3rd International Conference on*, volume 1, pages V1–619–V1–623, August 2010. doi: 10.1109/ICACTE.2010.5578942.

Yue Wang and N. Ranganathan. An Instruction-Level Energy Estimation and Optimization Methodology for GPU. In *Computer and Information Technology (CIT), 2011 IEEE 11th International Conference on*, pages 621–628, 2011. doi: 10.1109/CIT.2011.69.

Zhaoguo Wang, Ran Liu, Yufei Chen, Xi Wu, Haibo Chen, Weihua Zhang, and Binyu Zang. COREMU: a scalable and portable parallel full-system emulator. In *Proceedings of the 16th ACM symposium on Principles and practice of parallel programming*, PPoPP '11, pages 213–222, New York, NY, USA, 2011. ACM. ISBN 978-1-4503-0119-0. doi: 10.1145/1941553.1941583.

Wayne Wolf. *High-Performance Embedded Computing: Architectures, Applications, and Methodologies*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 2007. ISBN 012369485X, 9780080475004.

H. Wong, M.-M. Papadopoulou, M. Sadooghi-Alvandi, and A. Moshovos. Demystifying GPU microarchitecture through microbenchmarking. In *Performance Analysis of Systems Software (ISPASS), 2010 IEEE International Symposium on*, pages 235–246, March 2010. doi: 10.1109/ISPASS.2010.5452013.

H.D. Young, R.A. Freedman, and L. Ford. *University physics*, volume 1. Addison-Wesley Longman, 2006.