UNIVERSITY OF OSLO
Department of Informatics

Efficient parallelisation techniques for applications running on GPUs using the CUDA framework

Master thesis

Alexander Ottesen -
alexao@ifi.uio.no

Contents

List of Figures

Abstract

Modern graphic processing units (GPU) are powerful parallel processing multi-core devices that are found in most computers today. The increase in processing resources of the GPU, coupled with improvements and flexibility of the programming frameworks, has increased the interest in general purpose programming on the GPU (GPGPU).

In this thesis, we investigate how the GPU architecture and its processing capabilities can be utilised in general purpose applications using the NVIDIA compute unified device architecture (CUDA) framework. With the large number of CUDA applications being developed, we investigate how CUDA applications can share the GPU resource and see what challenges are connected with concurrent applications executing on the GPU. As a basis for our investigation, we implement the advanced encryption standard (AES) to learn how to use the framework, and how to increase performance of a CUDA application.

Our results show that there is little support for concurrency in the CUDA framework at present time, as the GPU accesses are serialised, with no support for preemption or sharing of the resource. We have implemented a static scheduler to see if it could improve concurrency, but due to the hardware abstraction CUDA offers we could not control the scheduling as we wanted. When developing the AES application we saw the importance of memory optimisations in CUDA applications, and we present ways of optimising memory accesses together with optimisation concurrent execution between the CPU and GPU.

Acknowledgements

I would like to thank my advisors Håkon Kvale Stensland, Carsten Griwodz and Pål Halvorsen for their valuable feedback, discussion and help. Without them this thesis would not have been possible. In addition, thanks to Paul Beskow and Håvard Espeland for taking their time in reading this thesis and providing valuable comments.

Thanks to my fellow student Magne Eimot for the discussions and support when working with the CUDA framework. And also the guys and girls at the Simula lab for providing motivation, input and making the lab a nice workplace.

Finally, I would like to take the opportunity to thank my friends and family for always supporting me.

Oslo, May 4, 2009

Alexander Ottesen

Chapter 1

Introduction

1.1 Background and motivation

From the introduction of the integrated circuit in 1958, there has been a continuous need for increased processing power. According to Moore's law, the number of transistors on an integrated circuit has increased exponentially, almost doubling every two years. This trend has continued until this day, and is likely to continue for at least the next years. Physical properties in single core CPUs limit the number of transistors that can fit into a circuit. In addition, issues like heat design and power density limits the clock frequency of a CPU. This has introduced computers with multiple CPUs and CPUs with multiple cores, known as multi-core. The focus of the microprocessor industry has shifted from maximising single-core performance to using multiple cores on a die.

There are both advantages and disadvantages of using a multi-core architecture on a CPU. The main advantage is the improved response time of CPU intensive applications, and the possibility for concurrency. One a single-core CPU, it is more likely for processes to be a victim of starvation as CPU intensive applications will be assigned to the same core for computation. A disadvantage with the multi-core model is the need for adjustments in operating systems, and software to better utilise the computational resources available. The operating system can perform load balancing between different cores, but to fully utilise the multi-core CPU processors, applications have to be designed with multiple threads in mind.

The need for more processing power and the development of parallel architectures on CPUs, has increased the development of parallel applications and focus on parallel al-

gorithms. Developing applications that can run in parallel is more challenging than sequential code, as the concurrency introduces potential issues like race conditions and challenges with synchronisation and communication. There is also the challenge in finding parts of an algorithm that can be executed simultaneously. Usually when designing parallel algorithms, the algorithm is decomposed into smaller tasks that can be executed simultaneously, known as partitioning. The partitions are usually designed based on the data dependencies in the algorithm. A dependency occurs if a value cannot be computed until a previous element has been calculated. An application cannot be run more quickly than the longest chain of dependent calculations. Before applications can be offloaded to multi-core architectures, the developer needs to identify data dependencies and adjust the algorithm to contain partitions that can be computed in parallel.

With the increased focus on parallel computation, offloading computation to co-processors with parallel capabilities, have followed the same pattern. This is not a new idea as network processors have been used to handle parts of the network stack, and graphics cards have handled graphics processing for a long time. However, using the devices for general purpose applications have not always been applicable due to programming APIs and hardware being limited in functionality, and not designed for general purpose programing.

Modern day graphical processing units (GPUs) are powerful parallel processing units designed to transform input data in the form of geometric shapes into pixels on the screen. Rendering a typical High Definition image today that contains $1920x1080$ pixels at least 30 times per seconds, preferably 60 times per second, requires a huge amount of processing power. The GPU is specially designed for compute-intensive, highly parallel computations, which is what rendering graphics requires. Therefore, the hardware is designed to have a larger amount of transistors devoted to data processing, unlike the CPU that needs many transistors for flow control and data caching. Traditionally, the GPU has been programmed through graphic APIs like OpenGL [13] and DirectX [14]. These frameworks are suitable for graphics, but can be challenging when mapping general purpose applications to the APIs. Recently GPU vendors have tried to mitigate this challenges for GPGPU developers by creating frameworks that are aimed at general purpose applications. NVIDIA and Advanced Micro Devices (AMD) are vendors that have focused on this area, and have released frameworks based on the C programming language aimed at GPGPU developers for easier development. CUDA [1] and FireStream [15] from NVIDIA and AMD are the most common used frameworks today.

GPGPU has been an important topic in research for several years [2]. The release of high level programming APIs like CUDA have increased the number of applications and made it easier for developers to use the GPU for processing. Curry et al. [16] have used CUDA in Reed-Solomon coding in RAID 6 systems with GPUs and gained up to over 5 times the throughput than an Intel Core 2 Quad Q6600. Falcao et al. [17] have shown that CUDA can be used in developing a decoding algorithm and data structure suited for the GPU in Low Density Parity Check (LDPC) decoding. They received a performance increase up to three orders of the magnitude compared to a modern day 2,4 GHz CPU. LDPC codes are powerful for error correcting codes, used in satellite broadcasting systems and in WMAN standards like WiMax. Another research project using CUDA is the distributed computing folding@home project designed to perform computationally intensive protein folding [18]. There are also examples of commercial products using CUDA. Elemental Technologies have developed an application using CUDA to accelerate H.264 encoding in Adobe Premiere Pro [19]. The application is designed to free up the CPU for other tasks than encoding video, by offloading it to the GPU with CUDA.

The main focus of CUDA applications has been to increase the performance of a single application by offloading computation to the GPU. We want to investigate how the GPU can be used as a non-exclusive resource, by examining how multiple applications can share the many cores of the GPU. To achieve this we need to understand how concurrent applications in todays framework are executed.

1.2 Problem statement

Most of the CUDA applications today assume that the application will have exclusive access to the GPU, but with the increase in applications offloading processing to the GPU there might be a need for the applications to share the GPU resource. We want to investigate how multiple applications are executed with the framework today, what challenges are connected with concurrency on the GPU, and see if we can improve the performance of concurrent applications. To reach our goal of scheduling the resources among CUDA applications, we want to understand how to use the CUDA framework and how to gain optimal performance with CUDA applications.

1.3 Main Contributions

In this thesis, we have focused on how the CUDA framework can be used most efficiently. With the growing number of CUDA applications released, there is a higher probability of multiple applications requesting the GPU resource concurrently. We investigate how multiple applications are scheduled by the CPU and GPU when they offload processing concurrently, and see how this affects the performance. To improve efficiency, we create a prototype of a static scheduler to combine the GPU processing from different applications to run concurrently on the GPU. We discuss the alternative to this approach, and investigate the limitations.

To be able to schedule different applications on the GPU, we want to gain experience in using the framework. In this thesis, we examine the processing capabilities of the GPU and how they can be used with the CUDA framework. We achieve this by porting two different implementations of the advanced encryption standard (AES). The implementations have different characteristics, and therefore different requirements to achieve good throughput. By examining the requirements of the applications, we show how to increase the throughput of the applications by focusing on memory access patterns of the application and memory placement. We provide insight in how to experiment with applications to gain optimal occupancy of the GPU, and see what challenges the developer needs to be aware of in doing so.

As optimisations are important in CUDA applications, we examine how memory access patterns are efficiently executed on the GPU, and what properties in algorithms the different memory spaces are optimised for. We look at the hardware requirements for memory accesses on the different memory spaces, and how an algorithm can be mapped to fit the requirements.

Additionally, we investigate the asynchronous capabilities of the GPU, where the CPU and GPU can execute concurrently and see how this affects the performance of applications. We use our AES implementations to show how concurrent execution between the devices can increase the performance, and what properties affect the amount of concurrency achieved.

1.4 Outline

The rest of this thesis is organised as follows; Chapter 2 introduces the GPU architecture we use, and the different GPGPU frameworks available. In chapter 3 we describe

the CUDA framework and how to use it for developing applications. Chapter 4 is a study of two block cipher encryption implementations of AES for the GPU using CUDA. In chapter 5, we look at optimisations of memory accesses on the GPU, and the memory spaces CUDA offers. Chapter 6 focuses on CUDA applications used in a system where multiple applications use CUDA, and how this affects the performance. In chapter 7 we investigate how the CPU and GPU can execute concurrently to increase the performance of the AES implementations in chapter 4. Finally we discuss our results and what we have learned in chapter 8, before concluding in chapter 9.

Chapter 2

Graphic processing units

In this chapter, we look at the many-core architecture of the GPU. We focus on the latest generation of GPUs from NVIDIA, AMD and the future Intel Larrabee architecture. By investigating the different architectures, we can highlight the features and limitations of the GPUs. In addition, we look at the development of GPGPU frameworks, and what is available today to determine what framework we want to work with.

2.1 Introduction

The GPU is a dedicated hardware device for rendering graphics on computers like PCs, game consoles and mobile devices. It can be found as a part of a dedicated graphics card, or integrated directly into the motherboard of a computer.

Through time the GPU has evolved into a highly parallel architecture, with many cores and very high memory bandwidth. Computer graphics can easily benefit from a parallel architecture, as thousands of pixels in an image can be computed in parallel as they are independent of each other. This feature leads to a processing unit that can outperform a CPU in operations per second, as seen in figure 2.1. The latest chips from the big vendors in the GPU market contain up to 240 cores as of February 2009 [11].

Owens et al. [2] ask the question why graphics hardware performance is increasing more rapidly than that of CPUs, as semiconductors for both architectures have the same amount of transistors available. The disparity in the performance can be attributed to fundamental architectural differences between CPUs and GPUs. The CPU is optimised for executing high performance sequential code, so many of the transistors are dedicated for flow control, branch prediction and caching. As for the GPU, it

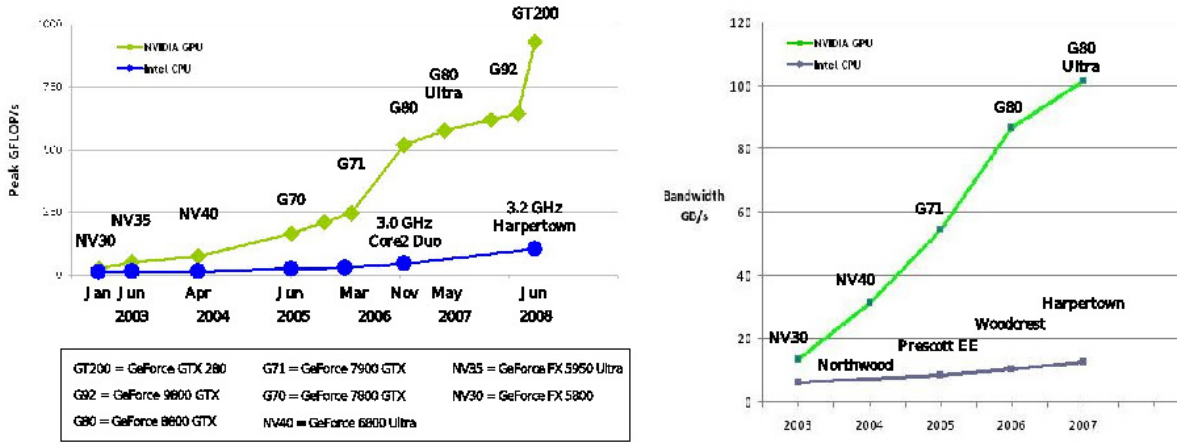


Figure 2.1: *Floating-point Operations per Second and Memory Bandwith for the CPU and GPU [1].*

has a more parallel nature for its applications, enabling the transistors to be used for computation as illustrated in figure 2.2.

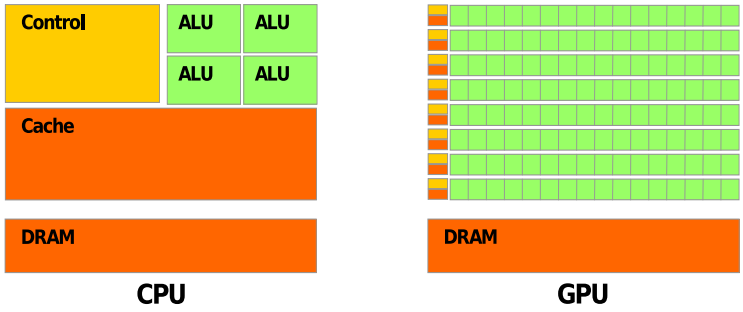


Figure 2.2: *The GPU devotes more transistors for data processing [1].*

2.2 Evolution of GPUs and their impact on GPGPU frameworks

Early GPUs had fixed-function graphic pipelines, which were limited to a predefined set of functions used for images and graphics. The pipeline design used was hardwired on the GPU to accomplish specific tasks. The goal was to maintain high computation rates through the use of parallel execution. Each stage in the pipeline is a designated piece of hardware on the GPU chip. The pipeline has evolved from being fixed to flexible. This was accomplished by making the vertex and fragment stages programmable as illustrated in figure 2.3.

Vertex and fragments are not terms often seen in general purpose applications, as most

programmers work with standard programming APIs that do not have specialised code distinctions like texture mapping and geometric objects. The vertex and fragment stages in the flexible pipeline can be programmed by running custom made shader programs, which were originally intended to alter vertex shapes and colours of fragments. One of the first GPUs that offered programmable shaders was the GeForce 3 chip from NVIDIA [20].

The graphics pipeline is designed to take a three-dimensional scene or image as input and output the image as a rasterized 2D image. Vertices are used as input, which are coordinates used as building blocks of primitives in 3D geometry, e.g., a three-dimensional scene or image. They are handed to the vertex processor to undergo transformation before they are processed by the rasterizer, which outputs fragments. A fragment is a piece of data that is used to update a pixel in the frame buffer at a specific location. The fragment processor determines the final pixel values before outputting to the frame buffer. The frame buffer is a video output device that contains a complete frame of data, typically consisting of colour values for every pixel in an image.

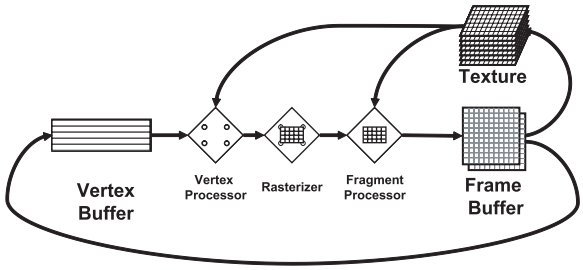


Figure 2.3: *The modern graphics hardware pipeline, vertex and fragment stages are programmable [2].*

Programming APIs

Most of the programming APIs that have been available until recently focused on graphics, and have been mapped to use the graphics pipeline pictured in figure 2.3. The intended use of the APIs were for designing graphics, but as developers started using the GPU for non-graphical applications the term general purpose computation on GPU (GPGPU) was founded in 2002. Developers successfully managed to map general purpose applications to the graphical API and gained performance in their applications by doing so due to the processing power of the GPU. Mapping general purpose applications to graphical APIs is challenging, as the API is not designed for general purpose applications.

OpenGL [13] is an example of a typical API that is designed for graphics. By combining it with the OpenGL Shading Language (GLSL) [21], one can fully program the graphics pipeline. GLSL is a high level shading language based on the C programming language and is meant to provide an easier approach of programming shaders than using low level assembly languages. However, GLSL does not remove the necessity of having detailed knowledge about the graphics pipeline and preferably experience writing 3D applications. Thus limiting, and creating added challenges for the developer when using the APIs for GPGPU development [2]. Microsoft have also developed a similar API, called DirectX [14]. The latter is a collection of APIs that Microsoft offer to handle multimedia applications on Microsoft Windows. Direct3D is the API that is used for graphics.

Graphical mapping, parallelisation of algorithms and adapting to a new hardware architecture imposes challenges for many developers when writing GPGPU code. To make the GPU an attractive development platform for other applications than graphics, there is a need for a programming model that adapts easily and scales to use the computation power the GPU offers. The learning curve for the programmer should be low so the developer can focus on adapting the tasks to the GPU, rather than details of the graphics APIs.

NVIDIA and AMD both offer ways of programming the device aimed at GPGPU developers. They focus on giving the developer a framework using industry standard programming languages to develop applications. NVIDIA has a development framework containing a compiler and development tools for writing applications to execute on the GPU. It is based on the C programming language, with extensions suited for the GPU architecture. CUDA was made public in a beta version in 2007 [22] for Windows and GNU/Linux. As of 2008, it has been released as a stable version and added support for Mac OS X.

CUDA is not the only API that has tried to make life easier for GPGPU developers. AMD is NVIDIA biggest rival in the GPU market. Together with NVIDIA, AMD have also focused on trying to get their technology used for general purpose programming. ATI technologies (ATI) started off by developing Close To Metal (CTM) [23], before they were acquired by AMD in 2006. CTM is no longer in use by ATI, but is available as an open source project. CTM lets the developers interface the GPGPU directly, through a low-level programming API. AMD used the CTM framework as their basis for their FireStream SDK [15]. FireStream was released in 2007, after rewriting the software stack to enable both a low level and high level programming API. FireStream uses Brook+, which is a hardware-optimised version of the Brook programming language

developed at Stanford University [24]. Brook is an open source language developed to offer GPGPU capabilities for the GPU independent of the manufacturer. Unlike CUDA, it is not an extension to the C language, but a variant. It is also released under a BSD license, so it is free to use for everyone.

Intel is developing a GPU based on the x86 architecture called Larrabee. For the programmer, it will appear as a set of x86 processors with SIMD units. The programming model for Larrabee will consist of an x86 compiler that will generate applications for the Larrabee x86 instruction set [25]. As Larrabee is based on the x86 architecture, many applications can be recompiled and work on Larrabee [25]. Such application portability can be of advantage when trying to parallelise existing code. Larrabee will have the same challenges that other GPGPU programming models have, as certain tasks are not always suited for the parallel architecture. For Larrabee to support existing GPU software, libraries to handle DirectX and OpenGL have to be developed, as the graphics pipeline will be different than on a traditional GPU. A software renderer will be used so applications designed for the graphics pipeline used today, will run on the Larrabee architecture. This shows that Larrabee is not a GPU in the traditional sense, but rather a set of x86 compatible cores that are connected. By having front-ends for DirectX and OpenGL, it will enable Larrabee to work like a GPU. The programming model proposed, is very flexible and lets the developer specify thread affinity with a particular core [25]. By using the well known POSIX threads API (P-threads) [26] and extending it to fit Larrabee's needs, Intel will offer a familiar programming API to developers. Intel anticipate that developers will use the Larrabee programming model to implement higher level programming models that may automate some aspects of parallel programming [25] in addition to offering low level thread and core control. More details on the programming API for Larrabee will most likely be released closer to launch of the architecture.

Open Computing Language (OpenCL) [12] is an open standard for parallel programming of heterogeneous systems, managed by the Khronos group. Khronos is an industry consortium that manages open standards like OpenGL. OpenCL provides a programming environment to write code that will run on a series of devices all from CPUs, GPUs, embedded devices and the cell broadband engine (CBE). It is the first step in providing a common standard to be able to develop platform-neutral applications for heterogeneous systems. OpenCL was created by a working group that recruited a number of participants from the likes of AMD, NVIDIA, Intel, Apple, Texas Instruments, Sony and ARM. They managed to create an API that was released at the end of 2008, however at present time there is no public runtime environment or compiler

available. Like CUDA, the OpenCL language is an extension of the C programming language.

2.3 General GPU architecture overview

To make it easier to understand the differences in the various GPU architectures, it is useful to understand a few concepts and definitions that are general to the architectures we investigate.

2.3.1 Abbreviations / definitions / concepts

Single instruction multiple data (SIMD) is a technique employed to achieve data level parallelism where one instruction is applied to many instances of data in parallel.

Kernel is a set of instructions run on the GPU in several threads. Whenever we refer to a kernel we mean code being run on the GPU unless explicitly stated.

Stream processing is a computer programming paradigm closely related to SIMD. It that allows applications to use multiple processing units like the GPU offers, without having to take into memory allocation, synchronisation and communication between the units. Usually a set of data to be computed is given as a stream, and a kernel function is applied to every element of the stream.

Stream processor (SP) is an individual stream processor designed to handle lightweight threads running a programmed kernel. A GPU normally contains multiple SPs.

Arithmetic logical unit (ALU) is a digital circuit that performs arithmetic and logical operations like addition, multiplication, subtraction and division. Together with bitwise operations, these are essential operations needed for computation.

The GPU is often considered a stream processor as it can only process independent vertices and fragments, but can process many of them in parallel. In the same sense, the GPU can run a single function in parallel on many data elements. This function is known as a kernel. The framework used determines how the kernel is programmed and executed, but in general the kernel is executed as many threads that run in parallel on the GPU. Detailed examples will be discussed in chapter 3.

2.3.2 NVIDIA GPU architecture

In figure 2.4, we see the GPU core of the NVIDIA architecture. To understand the figure and the rest of this thesis, the following abbreviations, definitions and concepts are essential to have in mind:

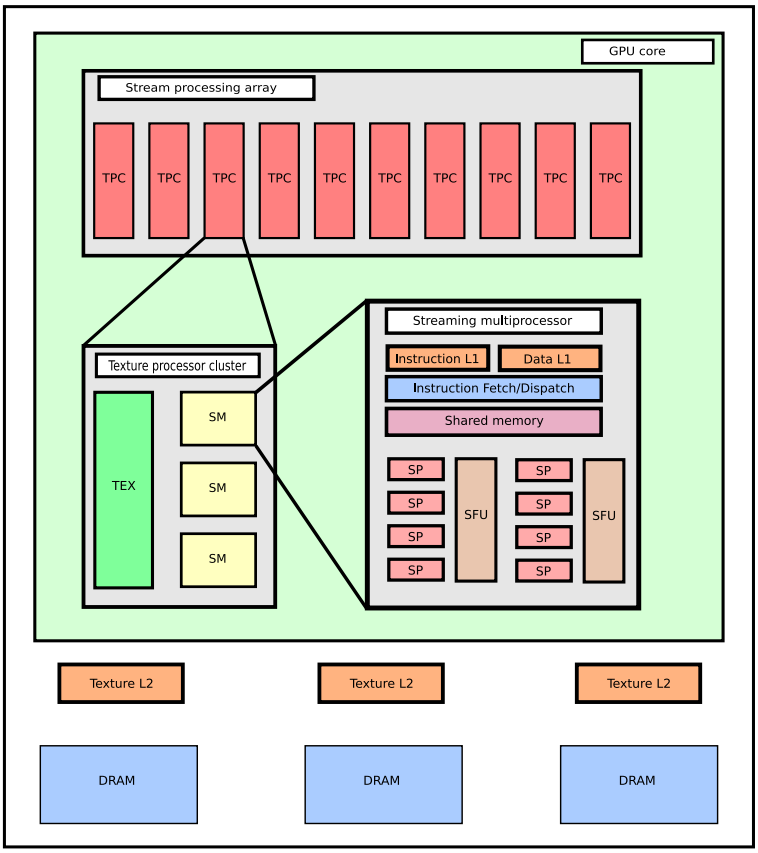


Figure 2.4: *Overview of the NVIDIA GT200 GPU architecture. Figure based on [3]*

Stream multiprocessor (SM) is a multiprocessor that contains 8 streaming processors, on-chip shared memory and instruction and data/constant cache.

Texture unit (TEX) is used for fetching instructions and data to the streaming multiprocessors.

Texture processing cluster (TPC): is a cluster containing multiple streaming multiprocessors and a texture unit.

Stream processing array (SPA): is the GPU core, also known as the stream processing array, and is a set of texture processing units.

Super function unit (SFU) is responsible for executing transcendental functions, sinus, cosines and other mathematical functions.

The GPU core of the NVIDIA GT200 also known as the SPA, can be seen in figure 2.4. It consist of a number of TPCs, each containing a group of SMs. An SM is made up of 8 individual SPs. These SPs are the cores that make up the processing power of an NVIDIA GPU. An SM has 16kB shared memory available for the SPs to share data across the cores in the SM, without having to read or write to or from external DRAM memory subsystems. The DRAM is located off-chip, and therefor has a higher latency than the on-chip shared memory. Every SM includes a texture unit used for graphics processing, but can be useful for general purpose applications. It assists in speeding up memory reads when the data is mapped as textures by using a level 2 texture cache to combine memory access achieving a higher bandwidth. A picture of the die can be seen in figure 2.5, where the core is illustrated together with the texture units, frame buffer and raster operation units (ROP). The ROP is a processor responsible for compression and decompression of textures.

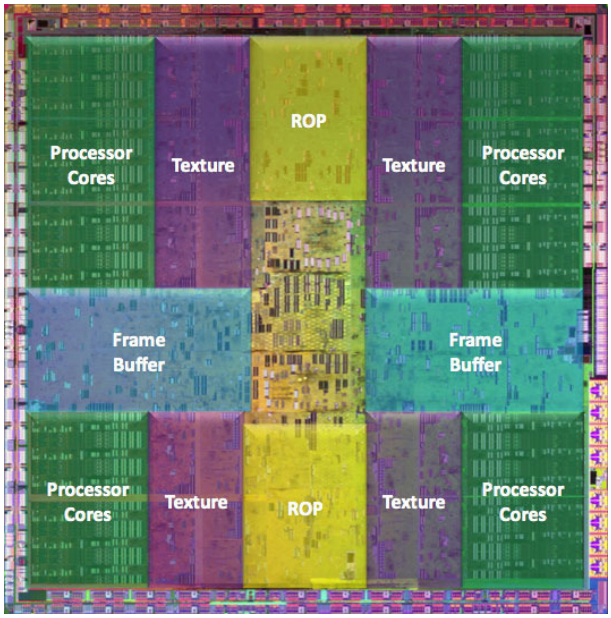


Figure 2.5: *Illustration of the GT200 die*

NVIDIA G92 and GT200 architecture

NVIDIA released the GeForce 8800 GT [27], in October 2007. It was the first GPU to support PCI-express 2.0 and to use a 65nm process. The GPU core used in the 8800GT is the same as in the G92 core. The GT200 was released in 2008 together with GeForce GTX 280, built with a 65 nm process, but uses a 512-bit Graphics Double Data Rate 3 (GDDR3) interface. The G92 has in total 112 SPs, divided on 8 TPCs with 2 SMs on each TPC. This gives a total of 128 cores, but on each SM one SP is disabled as the G92

is basically a die shrinkage of the G80. The GT200 has 10 TPCs, each with 3 SMs and all 8 SPs enabled giving a total of 240 SPs available.

2.3.3 AMD GPUs

AMD uses the same terms for their architecture, but call the SM a SIMD Core. The SFU is referred to as a special functions unit, but performs the same operations. AMD develop GPUs, but sell them under the ATI brand. The first AMD FireStream processor was announced on the 15th of November 2006. It was first sold as an add-on unit to existing ATI chips, and designed to solve scientific and general purpose tasks. The second generation FireStream code named 9170 [28], is based on the RV670 core. The GPU core of the RV670 contains 64 SPs. The SP on an AMD core differs from a SP on an NVIDIA chip, as the AMD SP contains 5 ALUs, giving a total of 5 * number of SPs ALUs on the GPU. The SP designed by AMD is pictured together with other GPU processing cores in figure 2.8. The 5 processing units, enables the AMD SP to handle five instructions in parallel, while executing the same thread. The units that are similar to ALUs, are pictured in figure 2.8. The x, y, z and w are similar to ALUs, but the t unit is a special functions unit. The SPs are grouped into a SIMD core, as depicted in figure 2.7(a).

The third generation FireStream, code named RV770 has a total of 10 SIMD cores, giving a total of 160 SPs. ATI Radeon HD4800 series uses the RV770 core, and can in theory produce a total of 1 TFLOPS on single precision floating point numbers with its 160 SPs [29].

2.3.4 Intel Larrabee

Intel are keeping details to a minimum on their future x86 many core GPU, as it will not be released until early 2010. However, a few details are known. As seen in figure 2.6, it will differ from Intel's regular line of GPUs, as it is not going to be integrated into the motherboard chipset, but will be released as a stand alone graphics card. Larrabee will first and foremost be a many-core x86 architecture designed for parallel processing and visual computing. It will use an extended version of the x86 instruction set. Even though the Larrabee core supports the x86 instruction set, it will not support branch prediction and out-of-order execution. However, this is functionality that is not found on regular GPUs [25].

According to Anand Shimbi et al. [30] at Anandtech, there will be room for between 16 and 32 cores on the die. The architecture is built on the original Pentium, but uses a 45 nm process instead of the original 800 nm. Unlike the original Pentium, Larrabee will have an L2 cache on the die, while the latter relied on external memory, which was placed on the motherboard. Modifications were made to the Pentium core to support 4-way simultaneous multi-threading (SMT), which is a mechanism to improve the efficiency of modern processors by hiding memory latency and letting instructions from more than one thread execute at any given time in the pipeline.

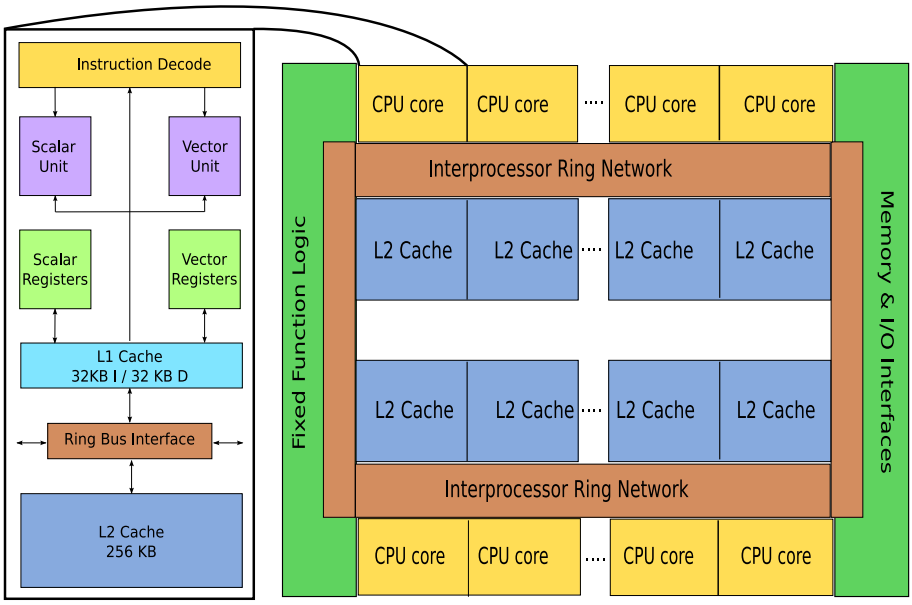


Figure 2.6: *A schematic of the Larrabee many-core architecture, with the CPU core on the left.*

Each Larrabee core will contain a vector ALU that can process 16 floating point operations in parallel, known as the vector processing unit illustrated in figure 2.8. The vector processing unit gives high parallel throughput of data due to the wide ALU. The design of the Larrabee core is essential for Intel to be able to achieve the processing power a GPU provides, as 16 – 32 cores is not enough to compete with the amount of instructions that can be executed on an AMD or NVIDIA GPU.

The Larrabee architecture contains a ring bus that connects the cores. The ring is a high-bandwidth interconnect network between the fixed function logic, memory I/O interfaces and an unique 256 KB coherent L2 cache per core. The ring bus is used to handle cache coherency between the cores, and for communication. Each core will have access to its own subset of the L2 cache, and can facilitate communication between the cores through the ring bus. If there is a cache miss in the local L2 cache, the request will be put out onto the bus, and checked to see if the data is in the L2 cache of the other cores. If found, it is transferred back to the correct cache. The L1 cache is built up

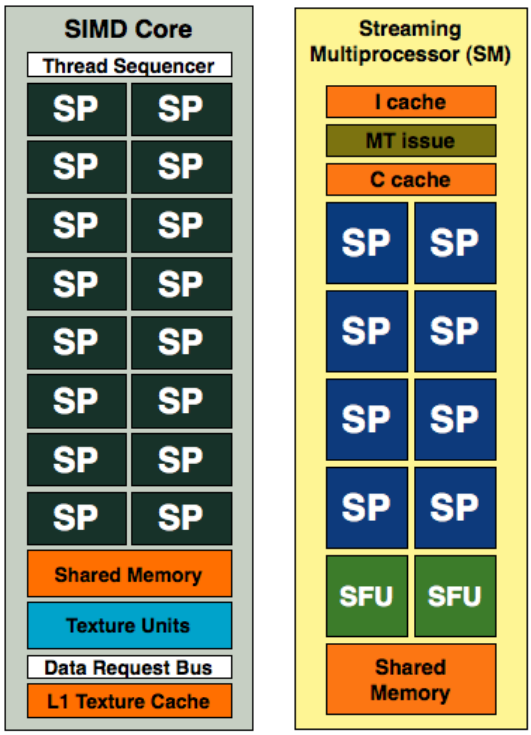
by an I-cache (instruction cache), and D-cache (data cache), giving a total size of 64KB. This enables support for execution of four threads in parallel per CPU core.

2.4 Capacities and processing

The GPUs all offer very high computation power, but differ in architectural design. NVIDIA have more cores in their GPUs compared to AMD due to the different design of the SP. As seen in figure 2.8, the ALU of the SP on an AMD GPU is wider than on the NVIDIA SP. AMD offers a higher theoretical processing power with their RV770 chip compared to the NVIDIA GT200, due to the ALU of the AMD SP that can in theory issue five instructions in parallel. The GT200 contains 240 SPs, but the ALU of the NVIDIA SP is limited to one instruction.

The design of each SP is important when considering what kind of workload is assigned to each core. The simpler SP of NVIDIA will work best if it can execute thousands of simple instructions in parallel, while the AMD architecture prefers workloads that are instruction heavy as it can process five instructions in parallel. This enables AMD to use instruction level parallelism (ILP) instead of using thread level parallelism as NVIDIA. ILP is parallelism that can be extracted from a single instruction stream. If there is a sequence of instructions that are independent, they can be executed in parallel. However, the use of ILP is challenging both for the hardware and the compiler, and may not always be beneficial. Another reason for why NVIDIA have opted for a simple core design is due to the clock frequency of each core. A simpler core design can run a higher clock frequency.

The Intel Larrabee architecture will be a very different GPU architecture. It uses x86 cores, with a 16-wide vector ALU that can process 16 instructions in parallel. The number of ALUs in each SP determines the theoretical computation power of the GPU, it also creates a challenge when there is a wider pipeline to fill. Since both the Intel Larrabee and AMD architecture will have a wider processing unit than the NVIDIA architecture, it may be challenging when it comes to pipelining data so that one instruction can be run on each of the ALUs in parallel. In the case of Larrabee, it will be challenging for the compiler to find instructions that can be executed in parallel. As the Larrabee GPU is yet to be released, it is impossible to determine if this will be case when released.



(a) (b)

Figure 2.7: *The AMD SIMD core in figure a), and the NVIDIA SM core in figure b)*

2.5 Summary

GPUs have evolved into a highly parallel architecture, with many cores and high memory bandwidth to fill the demand for real-time and high-definition 3D graphics. There has also been a growing interest in using the processing power of the GPU for general purpose applications. In this chapter, we have taken a closer look at the architecture of the GPU offered by NVIDIA and AMD see how they differ and how they can achieve high processing ability. In addition, we look at the future Larrabee GPU from Intel. We have investigated the development of the programming APIs, and seen what is available today and in the future.

After investigating the different frameworks and GPUs, we have decided to work with the NVIDIA CUDA framework to investigate the challenges connected to GPGPU development. NVIDIAs CUDA framework is the framework with the largest market share at present time. The number of applications using CUDA, the advancement of the framework, the quality of the documentation and the experienced development community are important reasons for choosing CUDA. We believe that if there is a market leader at present time, it is CUDA.

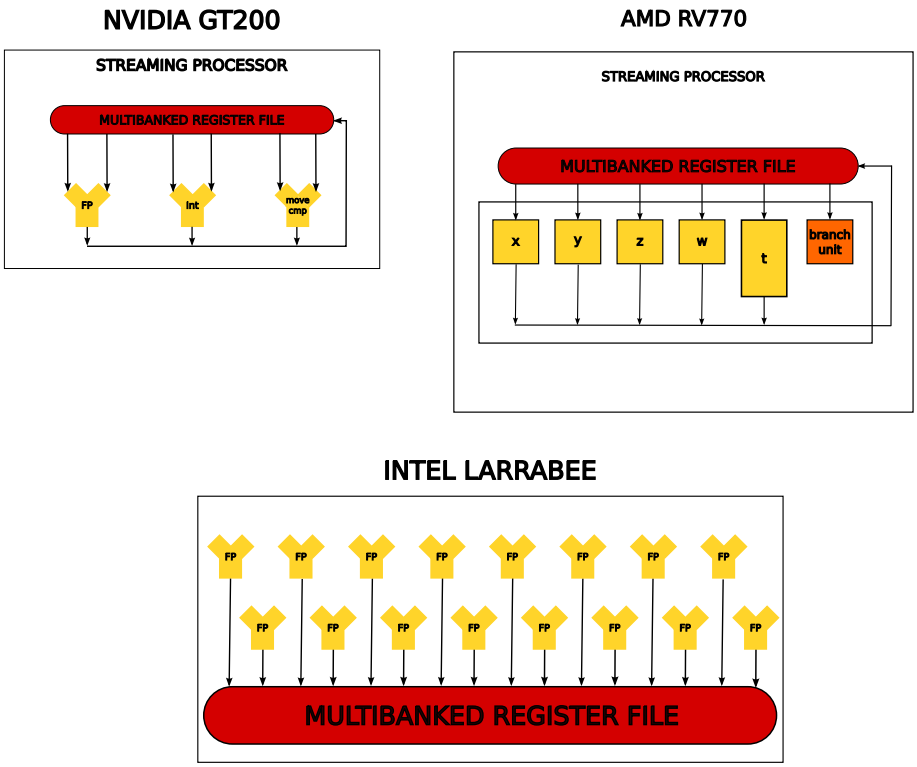


Figure 2.8: *The cores of NVIDIA, AMD and Intel Larrabee GPUs*

The best choice would have been to use OpenCL, as it will be a platform-neutral framework with support for other hardware devices. We believe OpenCL will bring more attention to the research area of GPGPU computing. However, at the time of writing there is no public working runtime environment or compiler.

Chapter 3

NVIDIA Compute unified device architecture

In this chapter, we investigate the NVIDIA Compute Unified device architecture (CUDA) framework and software environment. CUDA is aimed to assist developers who want to use the processing power of the GPU for general purpose applications. We examine what the framework offers in terms of tools, and how the developer should map algorithms to run on the GPU.

3.1 Introduction

CUDA was released by NVIDIA in November 2006. It is a general purpose parallel architecture aimed at overcoming the challenges of creating parallel applications that scale their parallelism to multi-core GPUs. The architecture consists of a software environment that lets developers use C as a high-level programming language, but will support C++, OpenCL and Fortran in the future [1]. By using the C programming language, CUDA provides a familiar programming model for developers unaccustomed to graphics APIs like OpenGL. CUDA extends C by using qualifiers to tell the compiler what code is assigned to the GPU.

To offload processing to the GPU using CUDA, developers define normal C functions called kernels. The kernels should be data-parallel compute intensive functions to get the benefit of the parallel hardware used, and are run as lightweight threads on the GPU. Data parallelism is a form of parallelisation where the same function is executed on different sets of data. Typical candidates for a kernel are functions that are executed

many times, but on independent data. This is also known as SIMD, where the same code is run in every thread, but the threads compute on different data. The CUDA programming guide uses matrix multiplication as a typical application that can fit this pattern. The kernels are executed `n` times on the GPU depending on how many times the programmer wants the kernel to run. In normal applications a function call usually means the function is executed once, unless it is a recursive function. In CUDA an execution configuration is used to specify the number of times the function should be executed as threads on the GPU. The execution configuration also specifies the how the threads are organised, known as the thread hierarchy. Kernels are run as threads on the GPU, while memory copying and the execution configuration is done by the CPU.

This separates the code into code executed on the CPU and code executed on the GPU. CUDA refers to the CPU and GPU as host and device. Both the host and the device manage their own memory space, but data can be copied between them. In a typical CUDA application the host allocates memory on both the host and device. This is done due to the fact that the GPU has its own DRAM on the video card. Data therefore has to be copied from the host to the allocated device memory. Similarly, data has to be transferred back to the host when computation is terminated. So in simple steps a typical CUDA applications starts off with the host allocating memory and doing the necessary steps, before executing the kernels that are run as a large number of threads on the GPU. When the kernel is terminated, the host can continue execution and transfer data back to the host from the device.

3.2 Thread organisation

To manage the large number of threads executed on the GPU, CUDA uses a thread hierarchy to identify and organise the threads. The execution of the threads is also referred to as a kernel launch. As all the threads launched execute the same function, they rely on unique coordinates to distinguish themselves from each other, and to assist the programmer in addressing memory. For every launch, the threads are lined up in a grid. The grid is divided into a two level hierarchy of thread blocks and threads, identified by coordinates called `blockIdx` and `threadIdx` given to them by the CUDA runtime system. These coordinates are accessible in the kernel to identify the different threads.

To give a brief overview on how the the hierarchy looks like. Two example grids are illustrated in figure 3.1, where the grids are divided into thread blocks of threads. The

```
1 kernel_function<<< gridDim, blockDim, sharedMemory >>>(kernel_param);
```

Listing 3.1: *Example of a execution configuration*

thread blocks use the `blockIdx` as coordinates, while threads within a thread block use the `threadIdx` for indexing. The combination gives the opportunity to index a thread across the grid.

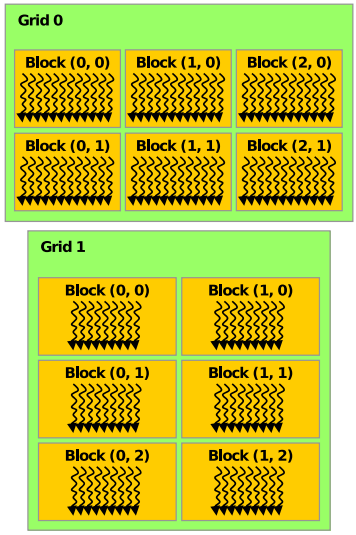


Figure 3.1: *Example of CUDA thread organisation [1]*

In figure 3.1, the thread blocks are lined up in a 3x2 and 2x3 array of thread blocks on the grid, where each thread block has coordinates consisting of `blockIdx.x` and `blockIdx.y`. The threads in a thread block use `threadIdx.x` and `threadIdx.y` as coordinates. The grid layout is determined by the parameters of the execution configuration, which can be one or two-dimensional. An example of a execution configuration launching a kernel can be seen in listing 3.1. The parameters `gridDim` and `blockDim` determine the number of thread blocks and threads in each thread block. The `sharedMemory` parameter is optional if the programmer wants to dynamically allocate shared memory for each thread block.

We have seen how the thread blocks are organised in a grid, but threads within a thread block may also use multiple dimensions to organise the threads. The threads can be organised in a one, two or three-dimensional array. It should be noted that each thread block is of the same dimension. To specify the dimensions in the execution configuration, the `gridDim` and `blockDim` variables must use the built-in variable `dim3`,

```
1 dim3 gridDim(64, 64);
2 dim3 blockDim(8,8,2);
3 ...
4 ...
5 int gridDim = 4096;
6 int blockDim = 256;
```

Listing 3.2: *Example of a grid setup*

which is an integer vector type. This is to provide a natural way to invoke computation across elements in a domain such as vector, matrix or field [1]. In listing 3.2 we see two different examples of setting up a hierarchy within a grid of $64x64 = 4096$ threads blocks with $8x8 = 256$ threads in each. The first approach uses a two-dimensional grid of thread blocks, with a three-dimensional array of threads within each thread block, while the second approach uses both a one-dimensional grid and thread block setup.

The developer should note that there are limitations in the dimensions of the execution configuration. In one kernel launch a dimension in the grid cannot exceed 65535 and the size of a thread block is limited to 512.

One of the reasons for the grouping of threads in blocks is hardware related, but it is also to give the developer easier ways to synchronise computation. CUDA allows threads in the same thread block to coordinate their processing steps through using a barrier to synchronise computation, by using a built-in function called `__syncthreads()`. In many applications there are data dependencies, meaning data may depend on other data elements to be computed before it can proceed, in these cases synchronisation is essential to make sure data has been computed. If not, a race condition might occur. The barrier is implemented with a single instruction to make sure it is not a bottleneck for performance. This ensures all the threads in a thread block wait until all the threads have reached the same part of the code before continuing. An advantage with the the thread block abstraction, is that threads are processing in close proximity limiting the cost of synchronisation.

Even though the grid and thread blocks are hardware abstractions for the developer, they are closely connected to the hardware architecture of the GPU. When the grid of thread blocks are launched, each thread block is assigned to an arbitrary SM. Every thread within the thread block is computed on the same SM. This means the thread blocks can be computed in any order relative to each other even if there are synchroni-

sation points in the code, as the synchronisation is done within an SM. Each thread in the thread block is assigned an SP within the SM the thread block is assigned to.

Scalability

Another advantage by assigning the thread blocks to an arbitrary SM, is the scalability of the application. As GPUs differ in the number of SMs they contain, the thread blocks will still be computed in the same way, but with another amount of thread blocks per SM as seen in figure 3.2. In this example there are two different GPUs computing the same kernel, but with a different allocation of thread blocks to different SMs. This is a useful feature if CUDA is to scale to mobile devices, and future devices with an increased number of SMs. The developer just has to make sure enough thread blocks are generated to keep the cores busy. The parallelisation is scalable due to the way thread blocks are assigned, with more SMs available the faster the code will run.

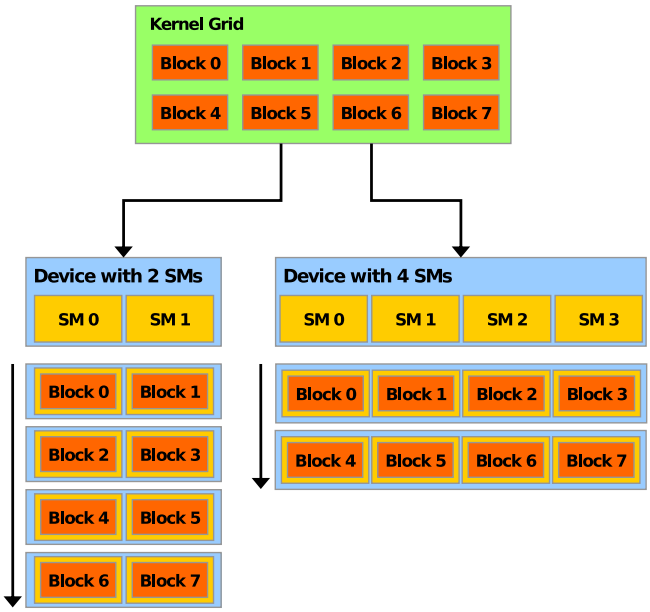


Figure 3.2: *The scalability of CUDA* [1]

3.3 Single-instruction multiple-thread

We have mentioned how the thread blocks are assigned to the SMs for execution, but not how the threads assigned to each SM are scheduled by the SPs available on the SM. To schedule the threads the SMs use a technique NVIDIA call single-instruction

multiple-thread (SIMT). It is similar to SIMD, but SIMT specifies the execution and branching behaviour of each thread. This enables the programmer to write thread-level parallel code, and to coordinate operations between threads unlike vector machines [1]. All the threads issue the same instructions, but the programmer can specify what data to compute and may also control the execution of each thread.

The SMs are allocated thread blocks, and the SIMT unit splits the threads into groups of 32 threads called *warps*. Each warp is created in the same way, and consists of threads with consecutive increasing thread IDs with the first warp of a thread block containing thread ID 0-31. For every clock cycle, the SM chooses a warp that is ready to execute for scheduling, and hands each SP a thread. Across the warp the same instruction is performed, meaning best performance is achieved if all the threads within the warp follow the same execution path and avoid branching in the code.

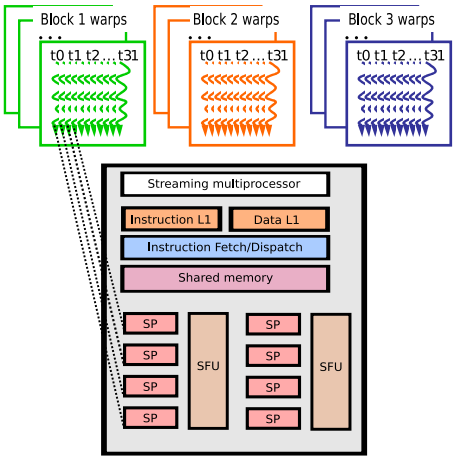


Figure 3.3: *An illustration on warps of threads assigned to an SM* [3]

To make sure all the SMs have threads to compute the number of thread blocks should be large. Figure 3.3 illustrates how different thread blocks are assigned to the SM, as there are usually a larger number of thread blocks than SMs. The SIMT unit creates warps from different thread blocks, and can schedule warps from different thread blocks interleaved. This is done for efficient scheduling if one of the warps is waiting on a memory fetch or a barrier.

It should be noted that there is a limit to how many thread blocks an SM can handle concurrently. The run-time system in CUDA maintains a queue of thread blocks to be processed, and assign new thread blocks to the SMs when a thread block has finished computation.

Memory space	Access time	On chip	On card
Register - dedicated HW	single cycle	yes	no
Shared memory - dedicated HW	single cycle	yes	no
Local memory - DRAM, no cache	slow	no	yes
Global memory - DRAM, no cache	slow	no	yes
Constant memory -DRAM, cached	1-100 of cycles depending on location	no	yes
Texture memory - DRAM, cached	1-100 of cycles depending on location	no	yes

Table 3.1: *Access times for the different memory spaces on the GPU [3]*

3.4 Memory model

The memory model in CUDA differs from earlier legacy GPGPU programming models. In OpenGL memory accesses are performed as pixels through the API, and it did not support scatter/gather operations. CUDA offers scatter/gather read and write operations, enhancing programming flexibility and resembles memory accesses seen in standard programming languages. Like the organisation of threads, the memory model in CUDA is also a hardware abstraction. As seen in figure 3.4, there are different memory spaces, but the CUDA memory model differs from physical memory seen in figure 3.3.

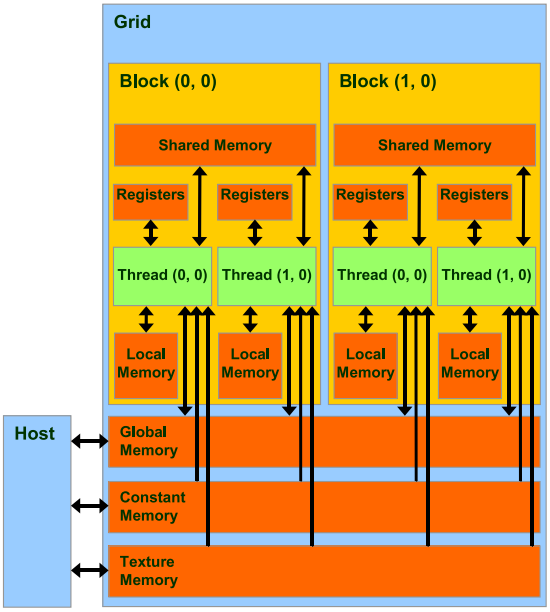


Figure 3.4: *The CUDA Memory Model [1]*

There are three main types of memory in CUDA. Global memory, shared memory that is on-chip on each SM and the cached memory spaces constant and texture memory. The accesses time for each memory space can be seen in table 3.1.

The memory spaces at the bottom of figure 3.4 are the ones the host has read and

write access to. Global memory is the most frequently used memory space, as it is the only memory space that is both read and writable for both host and device. Memory can be allocated in global memory from the host using CUDA API functions like `cudaMalloc` and freed with `cudaFree`.

Constant and texture memory is read-only on the device, and is setup from the host for read-only data. The constant and texture memory is located in the same physical memory (DRAM) as global memory, but uses the texture unit in combination with a cache available to each SM to increase speed up reads from the memory spaces. Each SM has an 8kB working set of constant and texture memory that is cached, making it useful for algorithms using data patterns that are read-only. Global memory, texture and constant memory can all be accessed from any thread on the grid.

Shared memory is the only memory space that is available on the actual GPU core as it resides on each SM. As it is on-chip, shared memory is local to each SM, meaning it is limited to be accessed within a thread block. The shared memory is very quick, but limited in size with 16 kB on each SM. The shared memory is optimal for sharing data across a thread block, or for performing computation before writing back to global memory. Due to the latency of global memory, a good way to compute data is to load data from global memory into shared memory, perform the computations in shared memory, synchronise each thread block and then write the result to global memory for the host to read when done. The developer can explicitly use the fast on-chip memory in CUDA, which is something that OpenGL does not offer, increasing programming flexibility.

The local memory pictured in figure 3.4 is not an actual hardware component, but local as in the scope of variables available for each thread. The actual variables may be allocated in global memory if there is not enough room in registers or shared memory. Local memory is not a memory space the developer can explicitly use, but the runtime places the data in the memory space if there are limited resources available.

The memory model and the thread organisation are closely linked together. To hide memory latency from global memory, other warps are scheduled while waiting for memory fetches. The thread block distinction is also designed for easier use of the shared memory space. For some algorithms this may enforce a redesign to fit the memory spaces to gain optimal performance. In a scenario where an algorithm may need synchronisation across thread blocks, synchronisation will need to be performed through global memory, as it is not possible to synchronise across thread blocks. A global memory mutex is not something very efficient, and it would be wise to launch more than one kernel as an alternative. It is therefore important to make sure the algo-

rithm fits the pattern of threads blocks to be able to benefit from shared memory.

3.5 GPU occupancy and capabilities in CUDA

NVIDIA use the term occupancy to describe how much parallelisation is achievable on the GPU in an application. It is the number of warps running concurrently on an SM divided by the maximum number of warps that can run concurrently. This means that a higher occupancy increases the probability of hiding memory latency in an efficient manner. There are many factors that determine the occupancy of the GPU, but the most important are:

- Number of thread blocks on the grid
- Size of the thread block
- Register usage per thread
- How much shared memory is used per thread block

As multiple thread blocks are assigned to an SM, each SM will have multiple thread blocks to compute. As the devices increase the number of SMs, it is recommended to have a large number of threads blocks assigned to each SM to scale to future devices. With multiple blocks running concurrently, it is important that each thread block does occupy all of the resources available on the SM. If they do, it limits the parallel capability. To assist in finding how well a kernel occupies the GPU, NVIDIA have released a occupancy calculator [31] to assist in finding the right balance of thread blocks and threads based on how much resources a kernel uses.

Compute capability

Compute capability is determined by how much of the CUDA API is supported by the GPU. At present time there are four different versions of compute capability ranging from 1.0 to 1.3. The main differences are how memory transactions are executed by the memory controller, the support for atomic instructions and better precision in floating-point numbers. There is no documentation on what will be supported in future compute capabilities, but since CUDA's release there has been an continuous development in what functionality the GPUs support. Thus creating new compute capabilities that are backwards compatible, and making it easier for GPGPU developers to increase per-

formance of their applications. We show examples of how the compute capabilities can affect the performance in chapter 4 and 5.

3.6 Software stack

The CUDA framework is composed of several layers as illustrated in figure 3.5. The developer can use the runtime API or driver API. Using the latter is harder to program as it provides more control in how the framework is used, while the runtime API is aimed at delivering a higher-level API that is run on the top of the CUDA driver. The framework is used as a stack, as the runtime API functions are automatically mapped to the driver API, easing the development process for developers who are not interested in low level instructions.

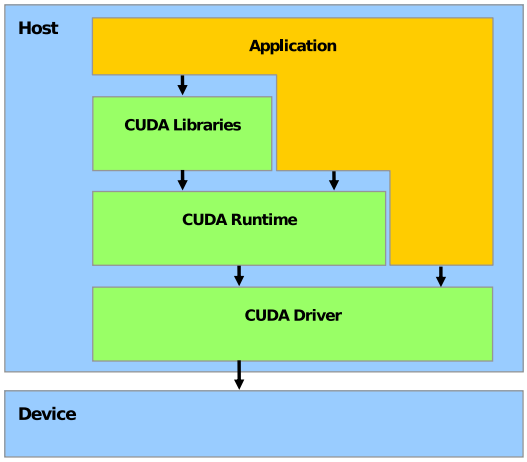


Figure 3.5: *The CUDA software stack [1]*

3.7 Language extensions

The CUDA programming framework provides simple extensions for the developer to be able to write code to run on the GPU. This is accomplished by a small number of extensions to the C language. The extensions provide qualifiers that specify what functions are run on the GPU, either through the use of the `__device__` or `__global__` qualifiers. Global is the qualifier that determines a kernel, but a kernel can use functions or variables declared with `__device__` qualifiers. The kernel is callable from the host only, and is executed on the device. If the developer wants to specify parts of the

code that is only available for the host, the `__host__` qualifier is available. Variables and functions with these qualifiers are assumed to be part of the host code.

To use a specific memory spaces, the developer must specify the memory space using a qualifier. The `__constant__` or `__shared__` qualifier will allocate variables into the given memory space. The global memory space does not use a qualifier, as it is allocated through API calls. Texture memory is setup by the host through specific API calls and declarations that are CUDA specific.

3.8 The CUDA toolkit and compiler

CUDA comes with a toolkit containing a compiler, the mentioned occupancy calculator and a visual profiler that analyses a kernel in runtime. The CUDA visual profiler gives results from hardware profile counters on CUDA-enabled devices. The profiler reads counters from one SM on the GPU during execution of the kernel. Each counter contains the number of occurrences of a transaction executed on the given SM. Since only one SM is used, it is necessary to launch many thread blocks, so the counters can work on a good percentage of the total work. We use the visual profiler in chapter 5.

The CUDA framework uses the NVIDIA-written compiler, `nvcc` [32]. CUDA code contains a combination of host dedicated for the GPU and the CPU. The nvcc compiler therefore needs to delegate the right parts of the source file to the right compiler. A view of the nvcc tool chain can be seen in figure 3.6

A CUDA source file has a .cu extension, the nvcc compiler supports several extensions, but triggers the CUDA compilation trajectory when working on a .cu extension. The CUDA compilation trajectory starts with invoking the CUDA front-end (*cudafe*), which is a preprocessor that splits the application into a CPU and a GPU part. Cudafe supports both C++ and C in the .cu file, but translates the code into C. Only C is supported for the code running on the GPU. The host code is compiled by the native C host compiler, and the GPU code is processed by NVIDIAs Open64 [33] based compiler *nvopencc*. Open64 is an open source optimising compiler, and is often used by compiler and computer architecture researchers.

The nvopencc compiler takes the GPU part from cudafe, as .gpu files. Depending on the parameters given to nvcc, these can be written as output files, or just be used as intermediate steps. The .gpu files are processed by nvopencc, which emits an assembly language called Parallel thread execution (PTX) [34]. Since CUDA can be run

on several GPU architectures, PTX has an advantage of generating a virtual machine model that can be used independent of NVIDIA GPU architecture. Once generated, the PTX output it is passed to a PTX assembler (ptxas), which outputs to the optimised code generator (OCG). It handles register allocation, scheduling and peephole optimisations that are tasks specific for a particular chip. Peephole optimisation is a compiler optimisation that is performed over a very small set of instructions that can be replaced to generate a leaner instruction set.

The PTX code is useful, as the programmer cannot always predict the NVIDIA architecture of the GPU that will run the application at compile time. There are several ways of making sure you are in control of the architecture. One way is to make sure to generate PTX files as intermediate code, and let the CUDA runtime system compile it for the current GPU. Alternatively, it is an option to compile executables for different architectures and let the runtime system decide what executable to use. In nvcc terms, this is is referred to as a device code repository [32].

The code generated from the OCG is GPU object code, also known as a CUDA binary (*cubin*). The cubin file is linked with the output from the host compiler into an executable. A cubin file contains not only GPU object code, but details about the resource usage for each thread. This is helpful when using the CUDA occupancy calculator.

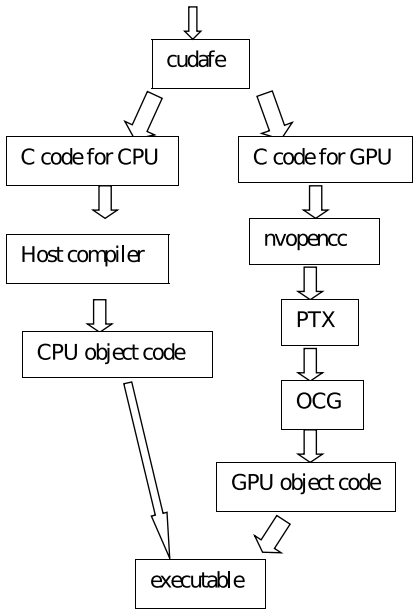


Figure 3.6: *The nvcc toolchain [4].*

The host code is compiled by the nvcc supported native host compiler for compilation and linking. A list of supported host compilers is shown in table 3.2.

Operating system	Compiler
`GNU/Linux and OS X`	The GNU compiler, gcc [35]
`Microsoft Windows`	The Microsoft Visual Studio compiler, cl

Table 3.2: *List of supported host compilers [32]*

Emulation mode

A problem with the execution model of CUDA, is that there is no stack for the code to use on the GPU [4]. This prevents recursive functions and frequently used functions that use a stack based calling convention like `printf()` from working. The latter creates a challenge when it comes to debugging CUDA code. NVIDIA has tried to solve this problem by supporting a *device emulation mode*. The mode runs only on the CPU, and tries to emulate the execution pattern of the GPU. It removes the need for an actual NVIDIA and CUDA supported GPU, as each thread is emulated using a host thread.

When running in device emulation mode, the developer has access to all the native debugging tools that host applications offer, e.g., GDB [36]. As code compiled for the host has a stack available, host functions like `printf()` work. The emulation mode runs all the threads in sequence, meaning it cannot emulate the parallel execution of the GPU. This causes problems when trying to debug race conditions. Additionally, the emulation mode does not offer proper support for a memory space like texture memory.

To generate an executable that uses the device emulation mode, there is a flag `nvcc -deviceemu` that must be set during compilation. Since there is no code generated for the GPU, there will also be a slight difference in code generated by the compiler since there are different instruction sets for the different architectures.

NVIDIA have also released their own GDB version for CUDA called CUDA-GDB [37]. It is currently in beta, but lets the developer use a familiar GNU GDB interface to debug threads running on the GPU in runtime. However, not all memory spaces are available through the debugger, and 64-bit applications are currently not supported.

3.9 Summary

In this chapter, we have investigated the CUDA framework and how it offers developers a scalable programming API for the multi-core architecture of the GPU. By ex-

amining how CUDA organises threads, memory and code structure, we can develop applications that benefit from the parallel capabilities of the GPU architecture. In chapter 4, we investigate the challenges a developer faces when using CUDA, and how much performance increase can be achieved by implementing two different versions of the advanced encryption standard (AES). Two different implementations are ported to see how different characteristics and requirements fit the memory spaces and thread organisation. Additionally, We focus on how the applications map to the GPU architecture explained in chapter 2.

CUDA is also the focus in chapter 6, where we examine how CUDA applications are executed from a system perspective to see how multiple applications use the framework concurrently.

Chapter 4

Advanced encryption standard

In this chapter, we look at the advanced encryption standard (AES) and show how the algorithm maps to the parallel architecture of the GPU. We explain the algorithm, and look at the modes of operation that can be used with AES for parallelisation. As the AES algorithm itself is not our focus, we base our work on existing single threaded implementations run on the x86 architecture, and port the code to the GPU using CUDA. We implement two versions of the algorithm to test different characteristics of the GPU, and assess our results. Our goal is to implement AES to learn how to use the CUDA framework, and discuss how to obtain good performance.

4.1 Background information

4.1.1 Rijndael

AES is a symmetric block cipher encryption algorithm developed by two Belgian cryptographers, Joan Demen and Vincent Rijmen. The standard is also known as Rijndael, which was its original title when submitted for the AES selection process. It is called a block cipher as the data being encrypted is divided into blocks that are encrypted individually. The standardisation process started on January 2nd, 1997, when the national institute of standards and technology (NIST) announced that they wanted a successor to the data encryption standard (DES). DES was becoming vulnerable to brute force attacks, since it only uses a short 56-bit key, which limits the possible combinations, thus easier to break. NIST chose a successor after holding a competition where fifteen different designs were submitted from several countries and research communities. Each design was investigated by cryptographers, for security issues and on performance in

various scenarios. They were looking for a design that would be able to run on devices with different capabilities, ranging from normal computers to devices with limited, and slow resources like smart cards.

The fifteen designs were narrowed down to five finalists after NIST held two conferences to solve the matter. The finalists were then again subjected to a new round of reviews, which resulted in Rijndael being announced as the winner on October 2nd, 2000. By the end of 2001, Rijndael was standardised and became known as the AES standard.

4.1.2 AES Cipher overview

In cryptography, an algorithm for performing encryption or decryption is referred to as a cipher. The algorithm in the cipher usually consists of a series of steps that are implemented as one or more functions. In encryption, plain text is converted to a cipher text by passing the plain text together with a key to the cipher, and getting a cipher text in return. Different keys used with the same plain text give different cipher text, making the original message unreadable for anyone who does not have the key to decrypt the message. In AES, depending on the mode of operation (explained in section 4.1.3), the same function can be used as a cipher for encryption and decryption. When using a symmetric-key algorithm, the same key is used for both encryption and decryption, while asymmetric algorithms use different keys for encryption and decryption.

The AES standard [38] specifies the Rijndael algorithm, a symmetric block cipher that processes data blocks of 128 bits, using a cipher key of length 128, 192 or 256 bits. Data is encrypted in blocks. In AES the block size is fixed at 128 bits, which are organised in a 4x4 byte array referred to as the *state*.

The cipher in AES is a repetition of processing steps, called rounds. The number of rounds is determined by the cipher key length (128, 192, 256), respectively 10, 12 or 14 rounds. Each round consists of the following processing steps:

1. Substitute Bytes (`SubBytes`)
2. Shift Rows (`ShiftRows`)
3. Mix Columns (`MixColumns`)
4. Add Round Key (`AddRoundKey`)

Before any of these rounds are executed, there is an initial phase called the Key Ex-

pansion or Rijndael key schedule. Here the cipher key is expanded into an unique key for each round, hence it is referred to as the round key. All the steps are performed in every round, apart from the first and in the last round. In the first round only the `AddRoundKey` step is performed, while the `mixColumns` is omitted in the last. The steps are explained in detail below.

Key Expansion

The AES algorithm takes the cipher key and performs a key expansion routine to generate unique keys for each round. The number of keys depends on the size of the cipher key. This step is referred to as the Rijndael key schedule, described in detail in the AES standard [38].

Substitute Bytes

In the `SubBytes` transformation, each byte in the state is updated using an 8-bit substitution box, also called the Rijndael S-Box. Each byte is looked up in the table by using the same index in the state array (Illustrated in figure 4.1).

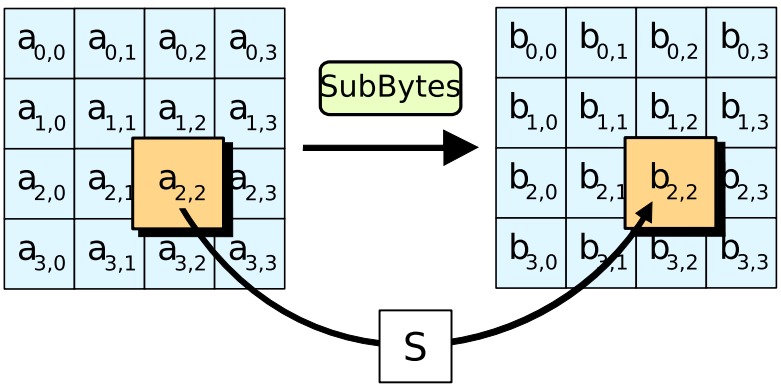


Figure 4.1: *The `SubBytes` step. The function S is a lookup operation in the S-box table. Illustration from Wikipedia [5].*

The `SubBytes` transformation provides non-linearity in the cipher, and is important to prevent cryptographic attacks based on algebraic properties. The S-Box is created as described in the AES standard [38].

Shift Rows

The `ShiftRows` step (shown in figure 4.2) operates on the rows of the state, it shifts each byte on the row by a certain offset. The first row is left unchanged, the second row is shifted one space to the left, the third row is shifted two spaces and the fourth row is shifted three spaces. This effect moves the highest byte in the row to the lowest position, and the other way around.

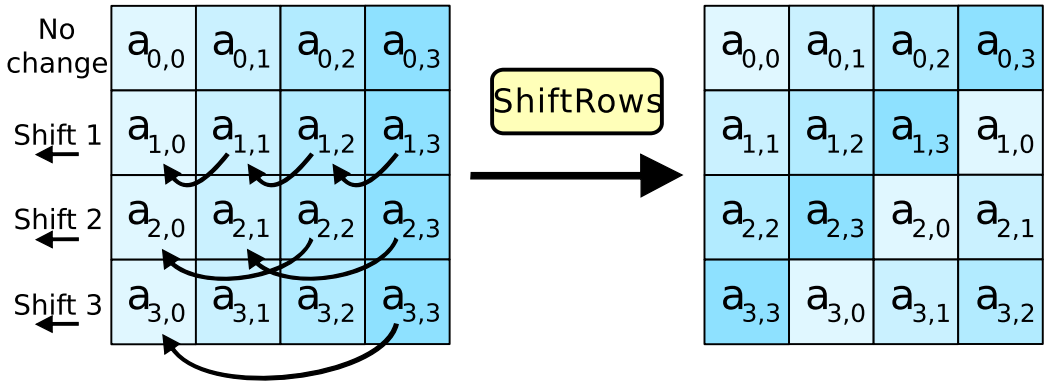


Figure 4.2: *The `ShiftRows` step. Illustration from Wikipedia [6].*

Mix Columns

`MixColumns` is a transformation that operates on the state on column-by-column basis, where the four bytes of a column are combined using a polynomial function described in section 4.3 in the AES standard. The `MixColumns` function takes four bytes as input and output four bytes, where each byte affects all the others (illustrated in figure 4.3).

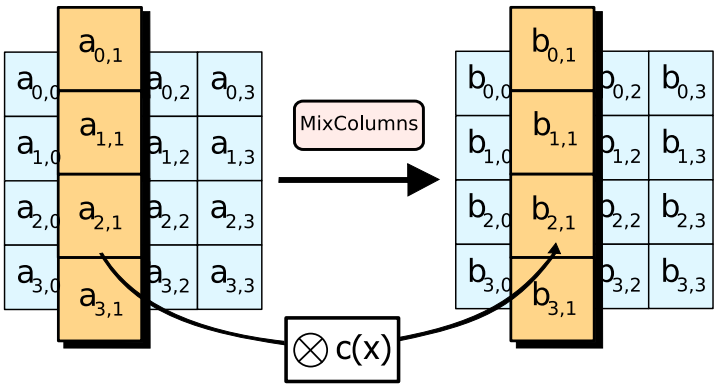


Figure 4.3: *The `MixColumns` step. Illustration from Wikipedia [7].*

The combination of the three mentioned functions creates an output that has a cryptographic diffusion property. According to Shannon, diffusion is associated with dependency of bits of the output on bits of the input [39]. In a cipher with good diffusion, flipping an input bit should change the cipher text significantly. This is referred to as the Strict Avalanche Criterion (SAC). If a block cipher does not exhibit the criterion to a certain degree, it suffers from poor randomisation and can be subject for an attack by a cryptoanalyst.

Add Round Key

The sub keys that are generated from the key expansion step are used in the `AddRoundKey` function. The sub keys are referred to as the round key in every round. The round key is added to the state by a simple bitwise XOR operation on every byte of the state to a corresponding byte of the round key. An illustration is shown in figure 4.4.

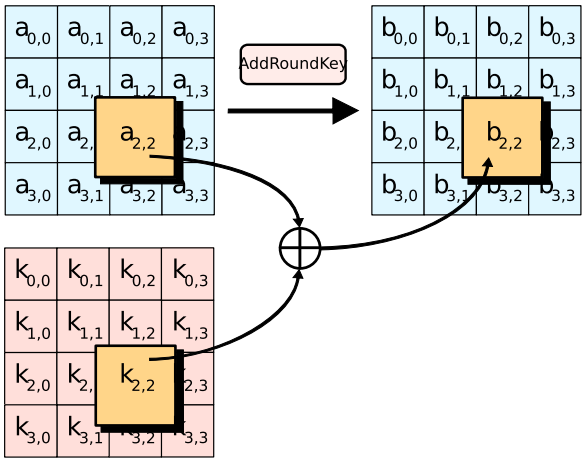


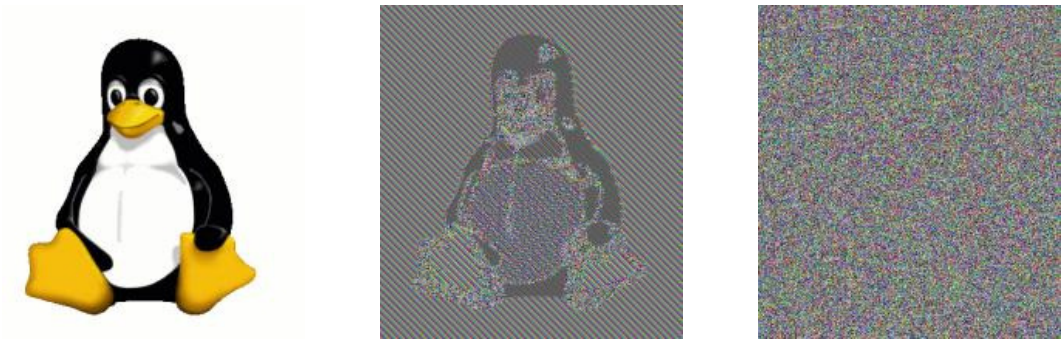
Figure 4.4: *The `AddRoundKey` step. Illustration from Wikipedia [8].*

4.1.3 Block cipher modes of operation

Different modes of operation provide confidentiality for messages of arbitrary length. When encrypting different cipher blocks that contain the same byte pattern and key, the output will be the same. The modes of operation are designed for different purposes, and therefore vary in how well they are suited for parallelisation. Important properties for parallelisation is to try and avoid data dependencies between the blocks.

In the simplest mode of operation, electronic codebook (ECB), the plain text is divided into blocks, and each block is encrypted individually. There are no chaining or depen-

dencies between each block, so if a similar data pattern exists, then plain text will generate the same cipher text. An example of ECB is shown in the encryption performed from the original figure 4.5(a) to figure 4.5(b).



(a) Original

(b) Encrypted using ECB mode

(c) Encrypted using CTR

Figure 4.5: *Comparison on encrypting using two different modes of operation*

One way to hide data patterns is to provide some randomisation for each block. Figure 4.5(c) shows how randomisation can give a better encryption result All the modes of operation apart from ECB require that an initialisation vector (IV) is used to provide an unique cipher text if the same key is re-used. The Cipher-block chaining (CBC) mode of operation is an example of how an IV can be used. However, the mode of operation uses dependencies between the blocks to assure randomisation. Figure 4.6 illustrates the dependencies between the blocks. Each cipher block is XORed with the previous cipher text block before being encrypted, giving a dependency between each block, which will force the algorithm to be run sequentially. The IV is used on the first block to make each block unique. One of the issues with CBC is that a single bit error can corrupt subsequent blocks, as the bits are used in the next sequence.

The ECB mode of operation does not hide data patterns in the encrypted message, and since CBC has sequential dependencies it is not optimal to parallelise due to dependencies between blocks.

One mode of operation that can be easily parallelised is the counter (CTR) mode of operation. It avoids the problems from both ECB and CBC, and benefits from the independent blocks as seen in ECB and the randomisation of CBC. An example is picture in figure 4.7.

Normally the plain text is encrypted in a cipher, but in the counter mode a counter value is used. The counter is initialised with a *nonce* (a random value used with each IV), and is incremented for every block. The combination of the counter and the nonce

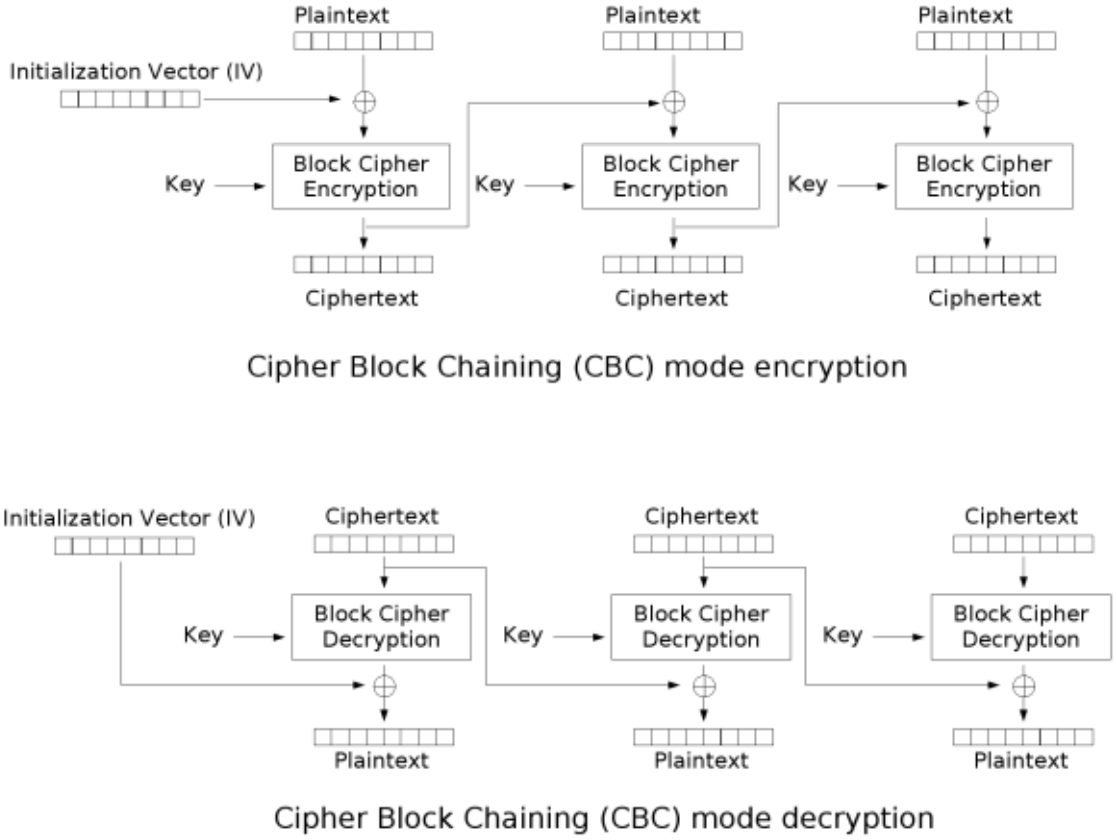


Figure 4.6: *Cipher Block Chaining (CBC) encryption and decryption. Illustration from Wikipedia [9].*

is the data that is encrypted, and not the plain text. The encrypted counter is then XORed with the plain text that results in a cipher text. By encrypting a counter that is separate for each cipher block, with no dependencies between the blocks, the algorithm can run in parallel and produce a cipher text that hides the original message. An advantage with the counter mode, is that the same algorithm can be used for both encryption and decryption, because of the properties of the XOR operation. We will therefore refer to both encryption and decryption as encryption from now on.

4.2 Implementations - software basis

There exist implementations of AES for the GPU implemented both using OpenGL [40] and CUDA [41]. The OpenGL version is an example on how OpenGL can used as a GPGPU framework using newly added support for integer operations and bitwise logical operations. Yamanouchi [40] shows a significant performance gain over CPU im-

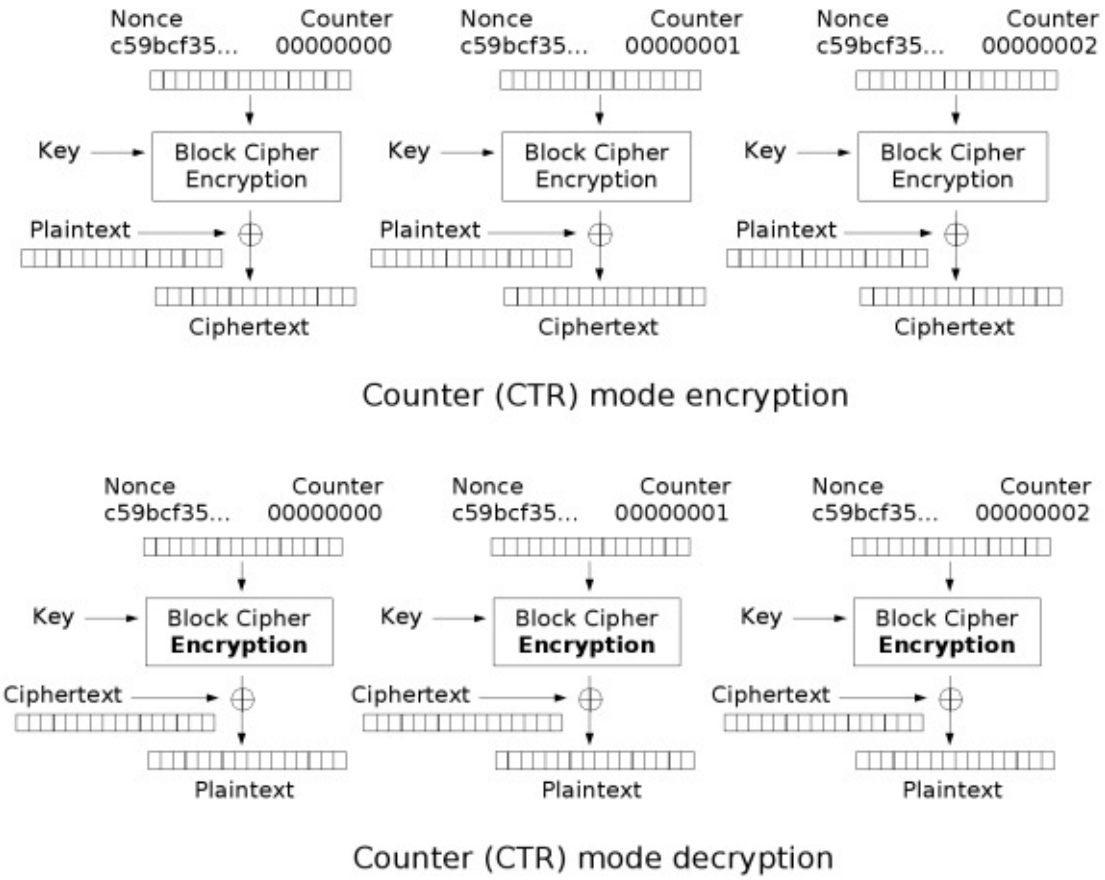


Figure 4.7: *Counter (CTR) mode encryption and decryption. Illustration from Wikipedia [10].*

plementations in his implementation executing on the GPU. The CUDA implementation by Manavski [41] uses an optimised implementation of the algorithm and reaches a throughput of 8.28 Gbit/s on a NVIDIA 8800 GTX.

As our goal was to learn the challenges connected to developing with CUDA, we wanted to write our own application. But as our focus is not on the AES algorithm itself, we wanted to base our work on code that was easy to understand and not already ported to CUDA.

To increase our understanding of the algorithm, we first ported an implementation that followed the AES standard to point with regard to names on functions and definitions. Then, to further benefit from the processing power of the GPU, we ported an implementation that focused on efficiency. This left us with two AES implementations, one written for efficiency, using pre-defined lookup tables,and one implementation focusing on simplicity and following the AES standard [38]. The lookup tables are used to combine the `SubBytes`, `ShiftRows`, and `MixColumns` steps into pre-calculated values that are suitable to fit in a header file. Each round in the cipher requires 16 table

lookups that are XORed 12 times, followed by an XOR operation with the round key.

With two implementations, taking such different approaches, we test both the processing power of the GPU, and how it deals with tasks that are more memory bound. We will refer to the lookup table-based implementation as *AES-lookup* and the standard implementation as *standard-AES*. The AES-lookup implementation has a higher requirement to efficient memory accesses, as it access the lookup tables in every round, while the standard-aes implementation has a high processing requirement as it does most of the calculation in the cipher. This means we have a memory access bound algorithm in the AES-lookup implementation, and a computational bound algorithm in the standard-AES implementation.

The choice for the standard-AES implementation fell on an open source project written by Niyaz PK [42], which is an implementation that follows the AES standard implementation manual. It was easy to understand and modify to the counter mode of operation.

The AES-lookup implementation is based on a public domain implementation written by Philip J. Erdelsky [43]. Unlike the standard implementation, it focuses on efficiency and not simplicity. We used a modified version of this code [44] written by Håvard Espeland at Simula Research Laboratory. He used this version to run AES on the Cell Broadband Engine Architecture.

The AES-lookup and standard AES implementations were ported to run on the GPU using CUDA. We also modified the x86 implementations to use the counter mode of operation and use P-threads to compare the parallel environments the architectures offer. Giving us four different implementations. We refer to standard implementations as GPU-S and CPU-S, while the lookup table implementation is referred to as GPU-L and CPU-L.

4.3 GPU implementation using CUDA

As we see in section 4.1.3, the AES algorithm using the counter mode of operation is suitable for parallelisation. Each cipher block can be computed in parallel, making the cipher a clear candidate for a kernel in our GPU implementations. When designing CUDA code, the developer needs to be careful to specify suitable code to be run on the GPU (kernel), and what part of the code should be executed on the CPU. As we based our work on existing code, we did not have to modify the algorithm, but we

had to map the right parts of the algorithm to the GPU and CPU. We used the given cipher as a kernel, modifying it to be executed in parallel, meaning each thread computes one cipher block. An alternative would be to follow the AES implementation in Manavski [41]. It uses four threads per cipher block. This was not applicable without modifying the lookup tables used in the code. Both implementations use lookup tables, as the standard implementation uses lookup tables in the `SubBytes` step.

The modifications done in the code is to map memory accesses to the GPU in the kernel, and setup from the host to make sure memory is allocated and the kernels are launched correctly. For the memory accesses, it is important to consider what variables are used frequently in the algorithm, and how they may affect the access patterns. In both implementations there are four variables that are used in every round. They are the round key, the counter, the nonce and the state. As they are accessed in every round, it is important that they do not affect the performance by using a memory space with latency. The nonce and round key are calculated once by the CPU, and are not altered during execution. This makes them suitable to be placed in a read-only memory space like texture memory, as they are calculated on the CPU, limited in size and can be copied to the memory space before the kernels are launched.

The combination of the counter and nonce is the data that is encrypted. In each thread, the nonce is read from texture memory, and the counter is calculated by the index values available from the CUDA runtime. The computation is done on the state, allocated for each thread, in the remaining steps of the cipher. The state is also independent for each cipher block, meaning there is no need for synchronisation between thread blocks. This makes the state a good candidate to be placed in the on-ship shared memory space. Each thread allocates 16 bytes of shared memory, which fits into the shared memory space even if each thread block consists of the maximum number of threads CUDA permits.

The amount of global memory available on the device restricts how much data can be processed in one kernel launch. To be able to encrypt an arbitrary large file, we split the encryption into a number of launches, where each launch fits on the GPU. The launches always use the same number of threads in a thread block, but the number of thread blocks may differ. To implement the counter in the counter mode of operation, we use a combination of the block and thread ID that are built in the CUDA runtime system. As these identifiers are reset for each launch, we add an offset for each launch to get the right counter.

In our implementations the host reads data from a file, allocates memory on the GPU before copying the data over. Each thread then calculates a cipher block, reading data

GPU	8800 GTX	8600GT	8800GT	8800GT-OC	GTX 280
Chip	G80	G84	G92	G92	GT200
Stream processors	128	32	112	112	240
Core Clock (MHz)	575	540	600	660	602
Shader Clock (MHz)	1350	1180	1500	1674	1296
Memory Clock (MHz)	900	700	900	950	1107
Memory Amount (GDDR3)	768MB	256MB	256MB	512MB	1024MB
Memory Interface	384-bit	128-bit	256-bit	256-bit	512-bit
Memory Bandwidth (GB/sec)	86.4	22.4	57.6	60.8	141.7
Compute capability	1.0	1.1	1.1	1.1	1.3
PCI express bus used	1.0	1.1	1.1	2.0	2.0

Table 4.1: *Hardware specifications for the GPUs we have run tests on [45] [11]. The 8800GT-OC is over-clocked by the manufacturer.*

from an unique place in memory based on a combination of the thread block and thread index. The cipher is then executed on the GPU, and the result written to global memory for the CPU to read.

Both implementations follow the mentioned pattern, but they differ in how they use the cached memory space for lookup tables.

4.3.1 Standard AES

The standard implementation uses a substitution box (`sbox`) in the `SubBytes` function, which is used for byte substitution operation explained in the AES standard [38]. The `sbox` is placed in the cached constant memory space, as it is read-only, small in size and used in every round.

4.3.2 Lookup table-based AES

The lookup table implementation uses lookup tables to efficient calculate the steps of the cipher. A total of five lookup tables are used, each 1kB in size making it a good candidate for the constant memory space.

4.4 Testing

We test the GPU implementations on the GPUs listed in table 4.1, to see how the number of cores and compute capability affects the performance. To see how well the algorithm scales to the many cores of the GPU, we compared the results with our CPU

implementations. We are also interested in seeing how the different characteristics of the algorithms affect the results. The standard implementation focuses on computation, while the lookup table is more memory access bound to the number of lookups.

To avoid an unfair comparison of the implementations, we only measure the runtime of the cipher. This is due to the fact that our CPU implementations use threads on the CPU, while the CUDA implementations are single threaded. Another difference is that the implementations differ in how data is transferred to the cipher. An option would be to avoid using multiple threads in the CPU implementations, but as CPUs have an increasing number of cores available we believe it is interesting to test the parallel capabilities of ordinary multi-core CPUs. To test we encrypted a 512MB file and measure the throughput of the encryption, we run the test ten times and plot the average values.

As mentioned previously, the file is divided into a number of launches due to memory constraints. The launches are similar in how many threads are used in each thread block, but the number of thread blocks in each launch vary.

The CPU implementations were tested on a $4x$ 2,6 GHz AMD Opteron 8218, with the number of threads equal to the number of CPU cores.

4.4.1 GPU Lookup table based AES (GPU-L)

Figure 4.8 shows the throughput of our GPU-L and CPU-L implementations. Note that the CPU implementations are plotted on the y-axis due to the limited number of threads. The throughput in mbit/s is shown on y-axis, and the number of threads in each thread block is shown on the x-axis. As seen in the figure, only one GPU manages to execute the kernels when using more than 384 threads in each thread block. This is due to resource constraints on the GPUs. The resource needs are dependent on each threads register usage, the amount of shared memory used by the thread block and how many threads are executed on the device. Usually, when there is not enough registers available, data will be placed in local memory that is located in the same physical location as global memory. This increases memory latency, and the performance might be affected. In the case of not enough shared memory, the kernel will fail to launch. Shared memory usage is not only determined by how much is used per thread block, but the parameters of the kernel, the indexes used in the grid all allocate shared memory.

The graph indicates that the GPUs with compute capability 1.1 or lower (all except

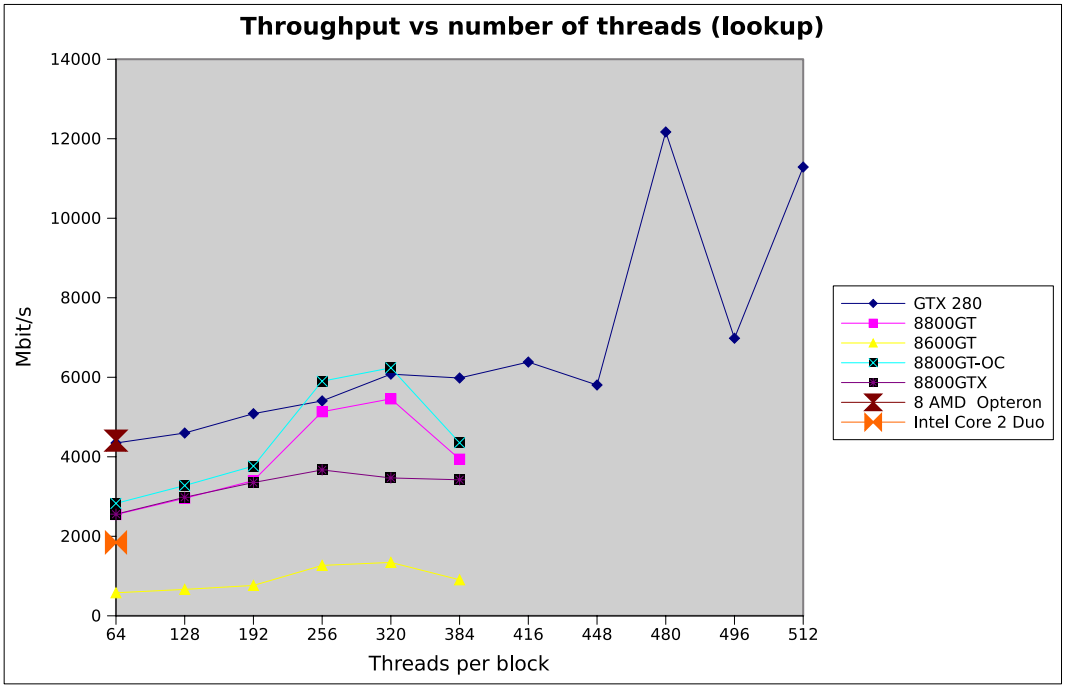


Figure 4.8: *Throughput of the GPU-L and CPU-L implementations. Note the non-linear x-axis*

GTX 280), scale the same way when comparing the number of cores on the GPUs. An example can be seen when comparing the 8600GT and the 8800GT. They follow the same pattern using different thread block sizes, but the added number of cores and higher clock frequency assists in achieving better performance for the 8800GT. We see that the 8800GTX has a higher amount of cores than the 8800GT(-OC), but yet it does not reach the same throughput. This is because the cores on the 8800GTX run at a lower core clock speed, which decreases the performance on an algorithm that is more memory access bound than computation-bound. A lower clock frequency on the core will also slow memory reads. This shows that in some algorithms the number of cores is not always the most important, but the clock frequency can play an important role.

To achieve good performance on the GPU, the processors need threads to process when waiting for memory latency. This means having a good occupancy of threads. As the lookup table algorithm is memory access bound, it is important to try and hide the memory latency by having a high occupancy of the GPU. A low occupancy makes it difficult to hide the latency, as computation will wait for memory fetches. By studying the performance of the GTX 280. It is by far the most powerful GPU, yet it achieves a lower throughput than for instance the 8800GT. This indicates that with the resource usage of our implementation, the memory access bound algorithm does not perform very well with a low occupancy.

However, with an increased thread block size, the GTX 280 performs excellent. It achieves a large speedup compared to the CPU implementation. From the GTX 280 performance we can see how small changes in the execution configuration can affect the throughput. The performance increase using a thread block size of 480 and 512 is most likely due to a good occupancy of the GPU. The dip when using 496 threads is not easy to explain, but we believe a plausible reason is because warps are scheduled in groups of 32 threads. This means that the last warp of the 496 threads on each SM will only run with 16 threads as $496\%32 = 16$, causing a divergent branch in the warp. Another explanation for the surges could be that the data placed in the cached memory spaces are not actually cached when using the specific amount of threads, and that every read suffers from the global memory latency.

As for the 8600GT, it generally performs bad in all scenarios. The GPU is cheap, and not aimed at high-end users. This is confirmed by looking at the hardware specifications. It does not have a large number of cores, runs at a slow clock speed and has low memory bandwidth.

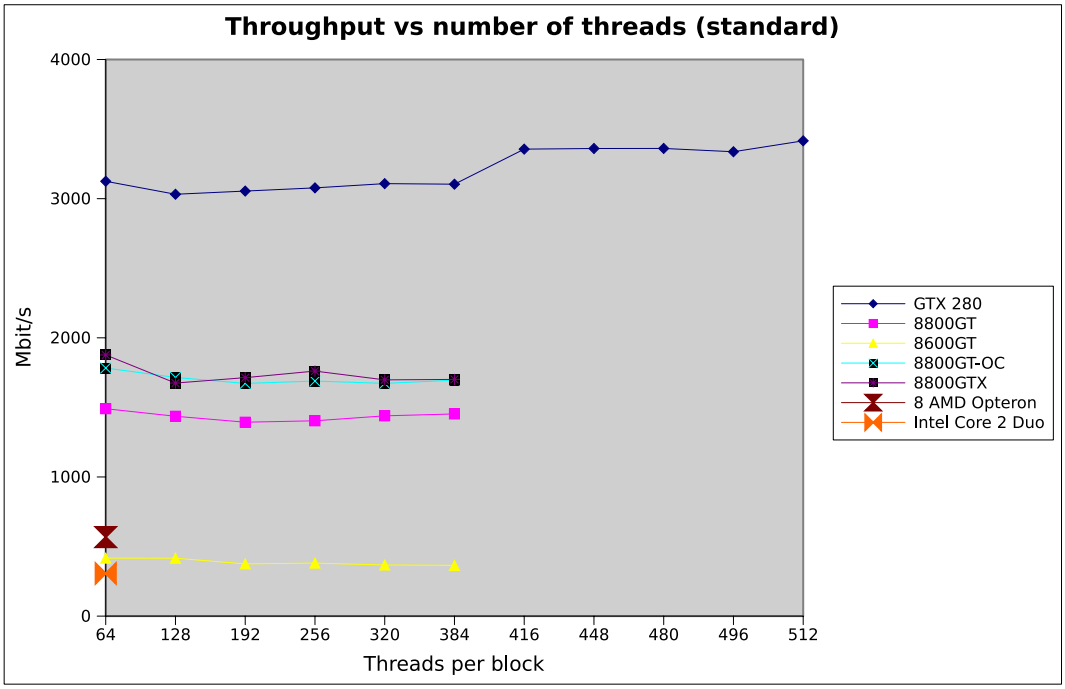


Figure 4.9: *Throughput of the GPU-S and CPU-S implementations. Note the non-linear x-axis*

4.4.2 GPU Standard AES (GPU-S)

The GPU-S and CPU-S implementation results are illustrated in figure 4.9. As with the lookup table implementation, it is only the GTX 280 that manages to run with a thread block size higher than 384. The graph shows that the implementation is computation-bound, as the GPUs with the highest number of cores produces the highest throughput. The performance of the implementations are similar when using a different size of threads per block, indicating that memory latency is not an large issue as all the threads use shared memory on computation and limit the use of the cached memory spaces. As the implementation does not fetch a lot of data from global memory or the cached memory spaces, there is a higher probability that data can be loaded directly into registers and reside in shared memory.

The difference between each GPU is about the same as in the lookup table implementation, the 8600GT suffers because of the lack of processors and slow clock frequency. The 8800GTX benefits from the extra 16 processors compared to the 8800GT. And as expected, the GTX 280 gives the best throughput due to the many cores.

4.4.3 Memory spaces

During development of the two implementations, we experimented with the different memory spaces CUDA offers, by changing the placement of data. The constant memory space was used for lookup tables, while the texture memory space was used for the nonce and round key. Additionally, we tested to see how the performance was affected when the state was placed in global memory. The difference in throughput can be seen in figure 4.10 for the GPU-L implementation. As expected, a memory access bound algorithm achieves an increase in the performance of caching mechanisms.

One optimisation that did not give any improvement on the GPU-L implementation, was using shared memory. We believe this due to the memory latency of the other memory spaces used in the implementation, meaning the benefit of shared memory is negligible due to the latency from the other memory spaces. The GPU-S implementation though benefits from using shared memory, as seen in figure 4.11. We can see from the output of the cubin file, that the GPU-S implementation uses a larger amount of registers than the GPU-L implementation. When not using shared memory, there is not enough registers available, so data is placed in global memory. This is most likely due to the fact that the GPU-S uses (`unsigned char`) as data type, whereas the GPU-L uses 32-bit words (`uint32`). A register on the device is 32 bits large, while the word

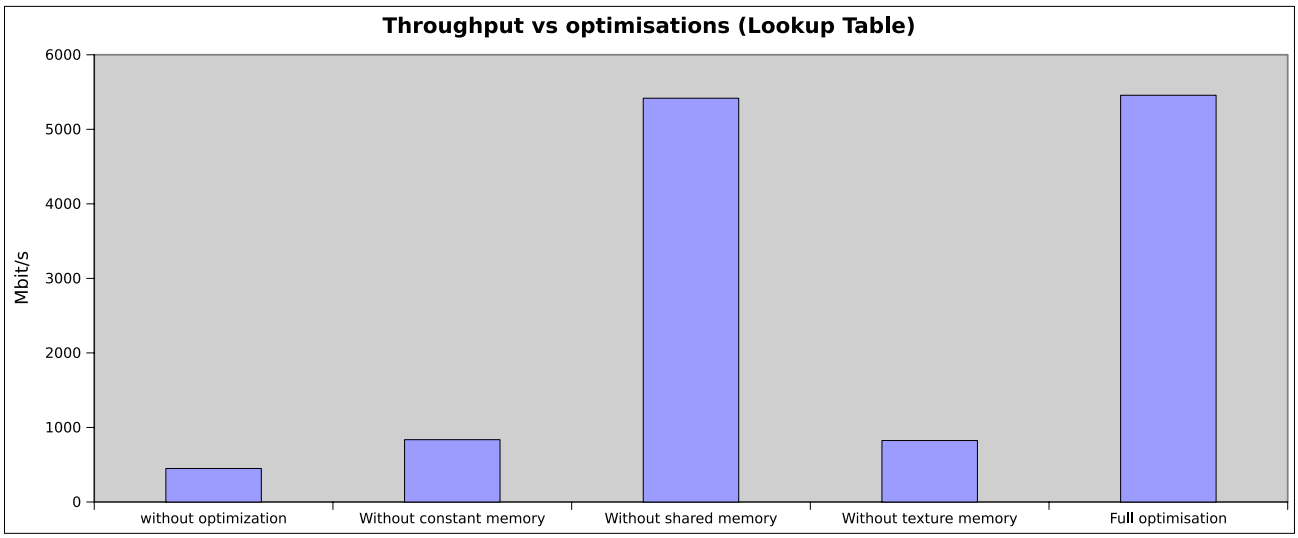


Figure 4.10: *Using different memory spaces affect the performance on the lookup table implementation*

size in the standard AES implementation is 8 bits large. Depending on the compilers' optimisations, this could mean that the compiler needs to use four registers for one operation, rather than one 32-bit register in one operation. A higher number of registers in use, increases the probability of data being placed in global memory.

By not using shared memory in both implementation, the number of registers used drops from 15 to 10 while in the GPU-S implementation, while the GPU-L implementation decreases from 18 to 15. We believe that both implementations try to place data in registers as often as possible, when not using shared memory. Since the GPU-S implementation has a higher amount of computations, and uses a shorter length on each word, it needs a larger amount of registers and therefore place more of the data in global memory. We will discuss the affects of using an 8-bit data type in chapter 5.

In conclusion, we believe that using shared memory is more beneficial for the GPU-S implementation because the state is accessed more often in this implementation, and using variables with a word length that does not fit the size of registers on the device. The GPU-L implementation has fewer computations and has a longer word length, which gives a more optimal usage of the registers, and thus limiting the need for shared memory.

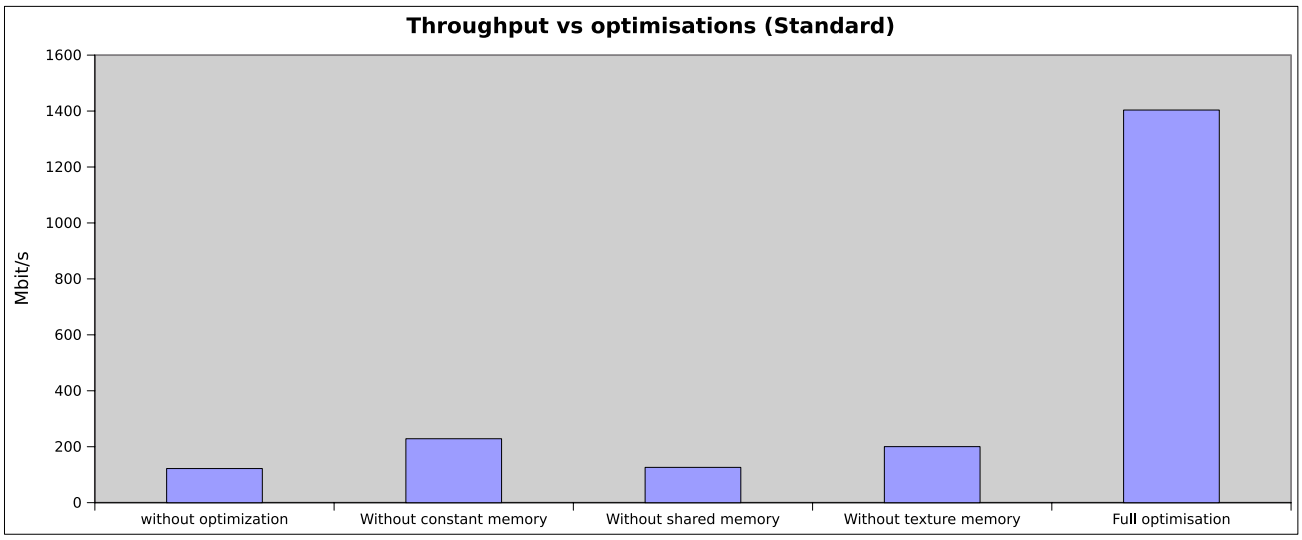


Figure 4.11: *Using different memory spaces affect the performance on the standard implementation*

4.5 Lessons learned

There are many ways to offload an algorithm to the GPU. As seen in our approaches, both implementations experience an increase in the performance by offloading the computation to the GPU. However, the optimised lookup table implementation does not experience the same increase in the performance compared to the standard implementation, like the implementation did on the CPU. The lookup table-based implementation is memory access bound, which is a challenging area for the GPU when it comes to performance. A GPU is designed to perform well on computation, which is shown in the performance increase of the standard AES implementation on the GPU compared to the CPU implementation. It shows that to achieve good performance using CUDA it is essential to have a highly parallel algorithm that utilises the large number of cores the GPU offers. This is achieved by hiding global memory latency by using a large number of threads, and making sure the right memory space is used for the data being computed. Hiding the latency is done by using the right balance between the number of thread blocks, the size of the thread block and trying to limit each thread's resource usage. The more resources a thread uses, a lower number of threads can be run in parallel.

As seen in figures 4.8 and 4.9, the performance differs with the same grid setup when run on different GPUs. Therefore, it is difficult to give a general indication for what is a good grid setup and resource usage for each thread. The occupancy calculator lets the developer experiment with various grid setup and to see how much of the resources

are used in each permutation.

We used the different memory spaces for optimisations, but we did not think about how to optimise the access pattern within each memory space. The compute capability of the GPU also affects how the access pattern is transferred intro transactions by the memory controller.

The NVIDIA GTX 280 compute capability 1.3 GPU, gave us such an increase in speed, that we investigated if the cause was only down to the increased number of cores. To do this we used the Visual Profiler [46]. When running our AES implementations in the profiler, it reported that our access patterns in the kernel were not optimal. Therefore, we will investigate access patterns and the use of the different memory spaces in chapter 5.

Another optimisation we did not make was on how to transfer data to and from the GPU. We could have implemented a *double buffering scheme* to limit the delay of input and output operations to the GPU. A simple optimisation would be to transfer data to the GPU while reading new data from a file, and to output data to file while waiting for data to be copied from the GPU.

The GPU also offers asynchronous transfers to the GPU through the CUDA stream API. This is done by using asynchronous memory copy operations and kernel launches. Due to time constraints, we have not had time to test a double buffering scheme, but we investigate the use of streams further in chapter 7.

Offloading this algorithm showed us that the easiest way to work with CUDA is to start off by implementing a simple version of the algorithm, and look at optimisations in steps after the code is working. This is done to achieve good performance, and is an easier way for the developer to debug CUDA code. A typical CUDA development process involves a lot of experimenting with the number of threads, fitting the algorithm to the memory model and using the right access patterns in the different memory spaces.

4.6 Summary

In this chapter, we have investigated how to use the CUDA framework to offload the AES algorithm to the GPU. We have gained knowledge on using the framework, and seen ways to best utilise the resources on the GPU. The AES algorithm is explained, and shown how the different modes of operation used in AES are suitable for parallelisation. By implementing two different versions of AES, we see how different char-

acteristics in the algorithms affect the performance gain on the GPU, and what the developer needs to consider.

Results show an increase in the performance for both implementations, but also pinpoints the importance of optimising code when it comes to memory placement. We have evaluated the resource usage of each implementation, and seen how it affects the occupancy of the GPU and the performance. The results show us that we need to further investigate optimisation of memory access patterns, and see what the differences are in the various compute capabilities. In the next chapter we focus on memory optimisations, and further investigate the properties of the memory spaces.

Chapter 5

Optimisation of memory accesses in CUDA

In this chapter, we investigate how to efficiently use the memory spaces CUDA offers. We look at the properties of each memory space, and how they are optimised for various access patterns. From the results of our AES implementation in chapter 4, we see that the access patterns used in our kernels are treated differently on GPUs with different compute capabilities. Therefore, we run tests on the different memory spaces, with specific access patterns to investigate how performance is affected on GPUs with different compute capability, and see what patterns are optimal.

5.1 Introduction

To get the best possible performance from an application ported to the GPU, developers need to be careful when it comes to resource usage. Registers per thread, occupancy of the GPU, memory placement and access patterns are all properties of a kernel that are crucial for achieving optimal performance. The results from our different AES implementations in chapter 4, indicate a large difference in how the memory accesses are handled by GPUs with different compute capabilities. We will therefore look at the requirements for optimising memory access patterns in the various memory spaces CUDA offers.

5.1.1 Half-warps and coalesced accesses

To have an understanding of optimising memory accesses, the developer needs to be aware of how memory instructions are executed by the memory controller. This is especially the case for global memory, as it is used by every thread, and is the memory space with the highest latency. In chapter 3, we mentioned how threads are scheduled in groups of 32 threads called warps. To make the scheduling more flexible, the memory transactions from a warp are executed on a half-warp basis. This is due to the design of shared memory, and to ease the handling of memory transactions from threads in a divergent warp. Divergence within a warp means threads execute different instructions that can be caused by branching in the code, or idling of threads.

The half-warps are most efficient when memory accesses from simultaneous running threads can be combined into a single memory transaction on global memory. This is known as an coalesced memory transaction. There are certain requirements the half-warp must oblige to be able to coalesce the memory transaction. These requirements are determined by the compute capability of the GPU. The compute capability also affects how the transactions are issued if the requirements are not met. This is referred to as an uncoalesced memory transaction. An example of a coalesced and uncoalesced access pattern is illustrated in figure 5.1(a) and 5.1(b). In this example, the coalesced access is achieved by having each thread access a 32 bit word in sequence within a 64 byte segment. The uncoalesced access reads values from different segments, which is not possible for the memory controller to coalesce.

In the next section we explain the different requirement for coalesced transactions on global memory and examine the different memory spaces.

5.2 Memory spaces

The memory spaces available in CUDA are global, shared, constant and texture memory. In this section we will take a closer look at what sort of data they are designed for, and how the memory spaces should be used in the different compute capabilities. We mentioned the latency of the memory spaces in chapter 3, where we showed the latency in clock cycles in table 3.1.

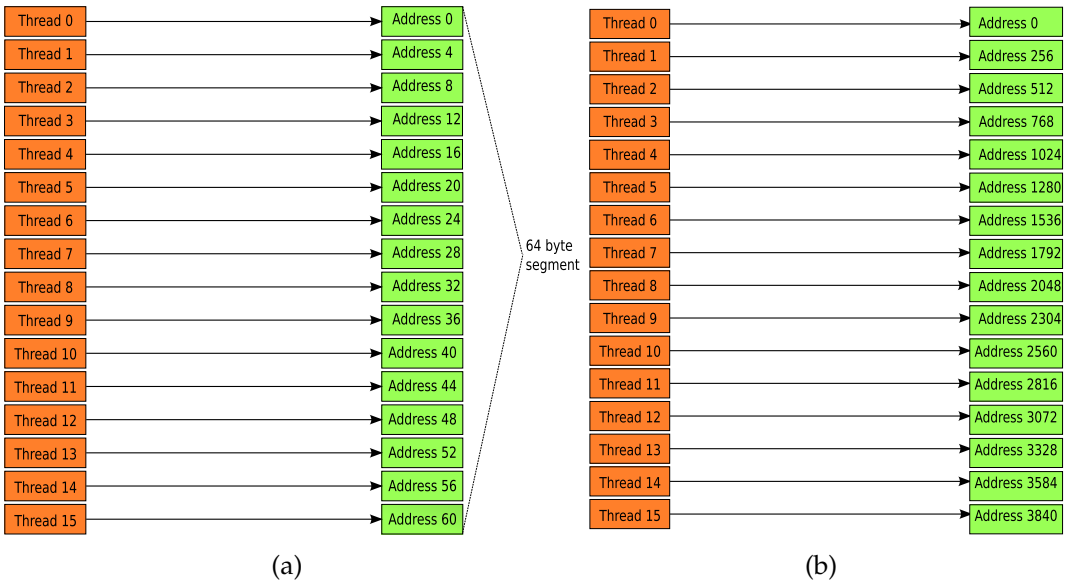


Figure 5.1: *A coalesced access pattern in figure a) and an uncoalesced access pattern in figure b).*

5.2.1 Global memory

Global memory is commonly used by every thread executed on the GPU as it is the only writable memory space that is shared across thread blocks, and the only memory space accessible for the CPU. The memory space is not cached and has high latency, so it is important to use the right access pattern to limit latency.

The GPU is capable of reading 32, 64 and 128-bit words from global memory into registers in a single instruction. To have the compiler generate assignments into a single instruction the data type must be such that the `sizeof(type)` is equal to 4, 8 or 16 bytes. If using an array structure, it is important to have the variables aligned to the same size, meaning the address for each index is a multiple of `sizeof(type)`. The alignment requirements can be automatically fulfilled for built-in types supported by the CUDA API, or by the use of `__align__` quantifier. This can be useful if a kernel needs to collect data in structures, where size and alignment can be enforced by the compiler using the align quantifier.

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

Compute capability 1.0 and 1.1

When accessing global memory, the data type used for the variables being accessed is important. This is referred to as the word size as it is a fixed size group of bits that are treated as one word. To achieve an efficient coalesced access pattern there are some requirements that need to be fulfilled for kernels running on a compute capability 1.0 or 1.1 GPU:

- Threads must access 32, 64 or 128-bit words resulting in one 64, 128 or two 128-byte memory transactions.
- All the 16 words being accessed by the half-warp must reside in the same memory segment of size equal to the memory transaction size, or twice the size if accessing 128-bit words.
- Threads must access the words in sequence within the segment, meaning the k^{th} thread of a half-warp must access the k^{th} work.

If the requirements are not fulfilled, a separate memory transaction will be issued for each thread, and throughput will suffer. According to the CUDA programming guide [1], a coalesced 64-bit access delivers slightly lower bandwidth than a coalesced 32-bit access, and a 128-bit coalesced access delivers a noticeably lower bandwidth.

Compute capability 1.2 and higher

There are similar requirements for higher compute capabilities, but they are more flexible when it comes to access patterns and the amount of transactions needed if requirements are not met. Threads must access 8, 16, 32 or 64-bit words that reside in the same memory segment of size 32, 64 or 128 bytes. Unlike lower compute capabilities, the access pattern within the segment does not have to be sequential. If the access pattern accesses `n` different segments, it will lead to `n` transactions as a transaction for each segment is needed. In lower compute capabilities 16 different transactions are executed as soon as `n` is greater than one. Additionally, compute capability 1.2 and higher can execute a coalesced access if multiple threads access the same address, unlike lower capabilities where accesses need to be in sequence within the memory segment. An example of a scattered pattern within a segment can be seen in figure 5.2(a), and access patterns across segments can be seen in figure 5.2(b)

Another improvement in compute capability 1.2 is the reduced waste of bandwidth if there are idle threads in a half-warp. To reduce waste of bandwidth, the memory

controller will automatically issue the smallest memory transaction possible for the requested words. Waste is reduced by issuing the smallest memory transaction possible in a half-warp. There is no need for a 128-byte memory instruction if a half-warp only issues requests for words that reside in the upper side of the memory segment. In this case it would be more efficient if the hardware issued a 64-byte instruction. Figure 5.3 shows an example of an access pattern with idle threads.

The protocol for memory transactions is as follows:

- Find the memory segment that contains the address requested by the lowest numbered active thread.
- Find all other active threads whose requested address lies in the same segment
- If possible, reduce the transaction size
- Carry out the transaction and mark the serviced thread as inactive
- Repeat until all threads in the half-warp are serviced

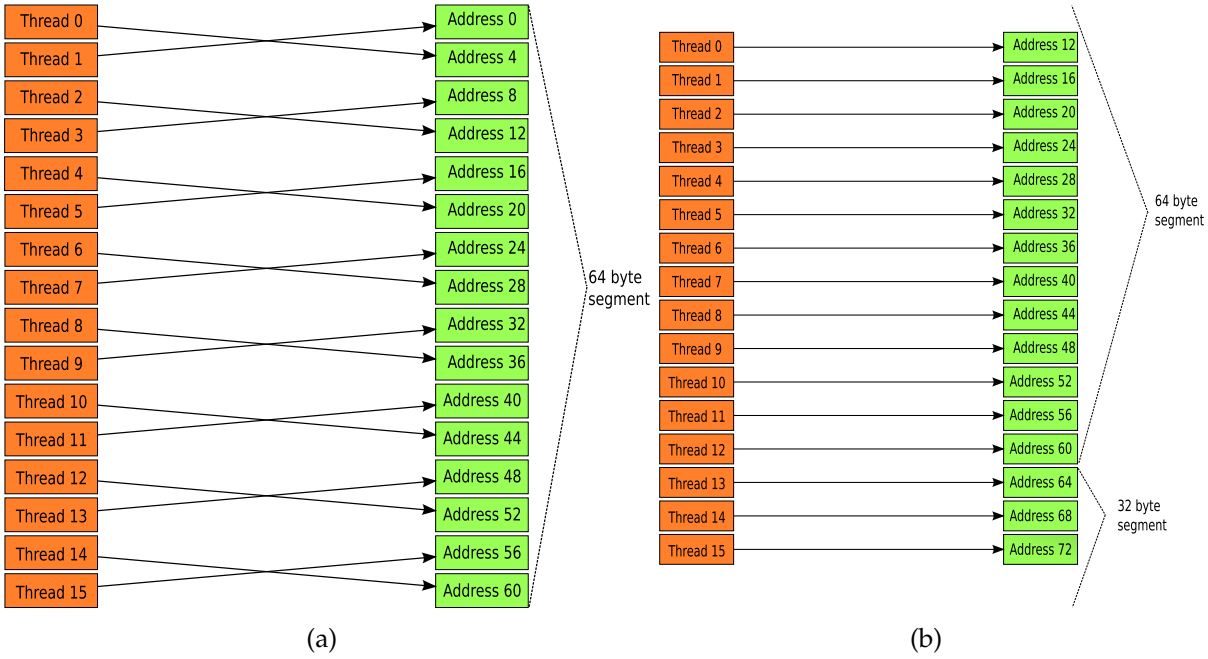


Figure 5.2: *Figure a) showing a scattered pattern within a memory segment. Figure b) shows how the memory transaction protocol will issue memory transactions across segments.*

A common access pattern for CUDA applications, which also ensures coalesced access, is when each thread (identified by the `tid = threadIdx.x`) accesses a value from global memory located at a `base_address`. The value should be read into shared memory, by fetch data from the following address: `base_address + tid`. The `base_address` is common for all the thread blocks as it is normally given as

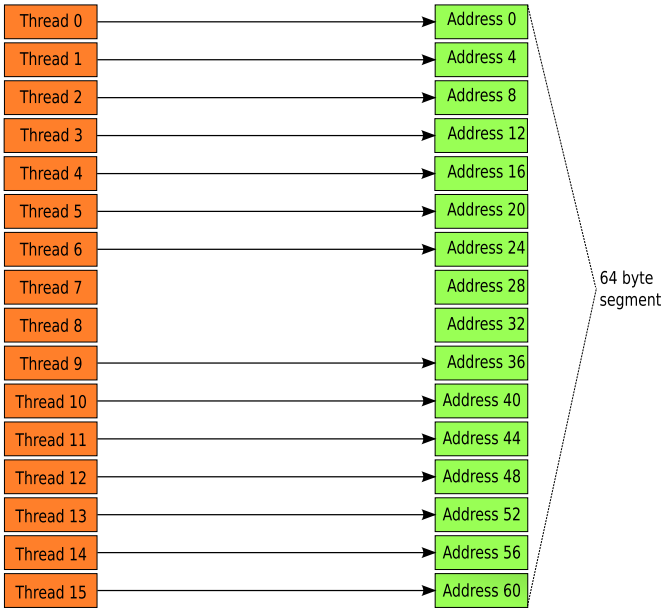


Figure 5.3: *A coalesced access pattern with idle threads.*

a parameter to the kernel, or can be computed in each thread. Each thread block will do the same operations, but on different values, so the `base_address` needs an offset for each thread block giving the following declaration: `base_address += blockIdx.x * num_threads`.

5.2.2 Constant memory and texture memory

The constant and texture memory space are designed for read-only data structures that have elements that reside close together in memory. The memory spaces are limited in size, are read-only and therefore not always applicable for certain applications. Both memory spaces use a caching mechanism, where a up to 8kB cache is available for both texture and constant memory on each SM. If there is a cache miss, a read costs the same as a fetch from global memory, as both memory spaces are subsets of global memory. An advantage of using these read-only memory spaces is that the requirements to get optimal performance are not as strict as in global memory, but there are recommendations on how to access the memory spaces to avoid unnecessary latency. Threads of a warp that read texture addresses that are close together will achieve the best performance, so mapping the read-only data to fit this alignment is a good optimisation.

The texture and constant cache differ in the kind of locality that they are optimised for. According to the programming guide [1], the constant cache is as fast as reading a register, as long as all the threads in a half-warp read the same address, and the cost

scales linearly with the number of different addresses that are read. Additionally, it is recommended that a whole warp reads the same address, and not just one half warp as future devices will require this access pattern for full speed read.

Texture cache is a more flexible cache, as it does not require each thread to read the same address to achieve full speed. However, it is recommended to have threads read addresses that are close to each other as the cache is optimised for 2D spatial locality used in imaging. Texture memory is normally used for storage of texture data used for rendering of images. The memory space is designed for mapping 2D images onto a 3D model and work together with the texture units on the GPU. CUDA offers mechanisms so GPGPU developers can benefit from this feature for read-only data used in their applications, and also using 1D data like a linear array. Additionally, the programmer has an advantage that texture memory can be set up very dynamically by the host, and has several different options for storing data in different patterns.

5.2.3 Shared memory

One way to limit the latency from global memory is to stage computation by loading data from global memory to shared memory for processing. Alternatively, the source data can be read from one of the cached memory spaces depending on the application. Shared memory is not accessible across thread blocks, as it is per SM only. This leads to the need of a synchronisation point after loading the data. The synchronisation point is implemented as a barrier, meaning threads of a thread block must have reached the same point in the code, before executing further. This ensures that data is loaded correctly.

The on-chip memory space is almost as quick as accessing a register depending on what the data is used for and how it is accessed. To achieve high memory bandwidth, the shared memory space is divided into equally-sized memory modules called banks, which are designed to be accessed simultaneously. If `n` shared memory requests fall into `n` different banks, they can all be served simultaneously. However, if two or more accesses from the same half-warp fall into the same bank (same address or multiple of the address that maps to the same bank), we have a bank conflict and the accesses have to be serialised. Examples of a warp with and without bank conflicts can be seen in figure 5.4(a) and 5.4(b).

Successive 32-bit words in shared memory are mapped to successive banks. In the current compute capabilities 1.x, the warp size is 32 and the number of banks is 16, but

this might be subject to change in future revisions. The number of banks is a reason why the warps are divided into half-warps, to make it is easier to avoid bank conflicts as requests are split into one request for the first half and one for the second. An advantage of this model is that it is not possible to have bank conflicts between different half-warps, as the requests will not be served simultaneously.

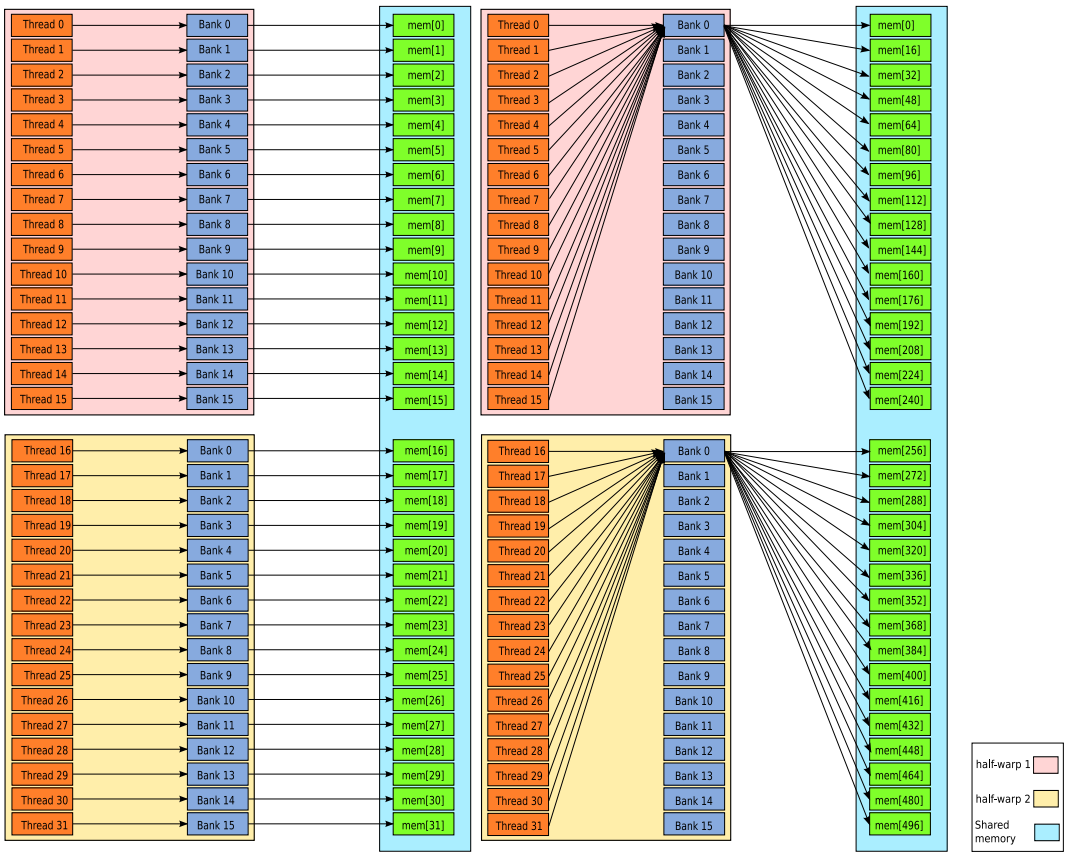


Figure 5.4: *Figure a) shows an access pattern from a warp of threads with no bank conflicts. Figure b) shows a warp of threads with maximum possible bank conflicts*

It should be noted that shared memory is optimised to use 32-bit words, as a `__shared__ char data[32]` declaration will have a bank conflict if different threads try to access `data[0], data[1], data[2]` ... `data[15]` in a half-warp. This pattern will cause a 4-way bank conflict as `data[0], data[1], data[2]` and `data[3]` all belong to the same bank. One solution is to access the data with a certain stride that avoids accesses falling into the same 32-bit bank. Our GPU-S implementation in chapter **??** uses an 8-bit data type, which affects the performance due to the number of bank conflicts.

Shared memory also features a broadcast mechanism, which reduces bank conflicts by letting several threads read the same shared memory address by broadcasting the data to all the threads after one read. The hardware automatically detects a broadcast word,

and it is automatically passed to other threads reading from that word.

5.3 Tests

To test various access patterns on the different memory spaces mentioned in the previous section, we developed a simple CUDA application that is based on code used in a paper by Boyer et al. [47]. The authors developed a tool to detect bank conflicts in CUDA kernels. They showed how bank conflicts can decrease performance in a kernel. We wanted to extend their work to show similar trends on other memory spaces, and to test access patterns we used in our AES implementation, and to see how kernels react when adapting to the requirements set by the compute capabilities mentioned. Listing 5.1 shows pseudo-code of how the kernels test the memory spaces.

We used the kernel in listing 5.1 as a basis, and they do essentially the same, the difference being the memory space they use as source, and where they perform computation before writing the result back to global memory. It should be noted that the `tmp` variable is unnecessary, but is added to spend some time on additional computation. The initialisation phase is used to add to the number of accesses performed. Each thread should read a value in a loop, for `number_reads` times. The value will be read from a source, incremented and then stored. In all the kernels apart from ones using shared memory, the computation is performed in global memory, but data is not necessarily read from global memory. Every value is stored in different addresses in global memory, giving a total of `threads * blocks * num_reads` write and read operations. Global memory has more space available than the cached memory spaces, and the latter are designed to have threads read the same values or values close together in a smaller memory space. So constant and texture memory allocates a smaller amount of memory, and works on a smaller set of values.

We developed the following kernels with different benchmarking targets:

- **global coalesced** is a kernel designed to show how fast a coalesced read and write pattern on global memory can perform. The kernel follows the pattern of figure 5.1(a).
- **global idle coalesced** is similar to **global coalesced**, but with 2 idle threads per half-warp. Tested to see how different compute capabilities handle warps with idle threads. The kernel follows the pattern of figure 5.3.
- **global uncoalesced** is similar to **global coalesced**, but with an uncoalesced read

```
#include "mem.h"

__global__ void memtest_kernel(int *data, int iterations, int threads
    ,
        int blocks, int number_reads)
{
    int i,j;
    int index = blockIdx.x * threads + threadIdx.x;
    int tmp = 0;

    /* All active threads initalize result data to 0.
     * If using shared memory, data is a shared memory location
    */
    for (i=0; i < number_reads) {
        if (coalesced)
            data[index + blocks * threads * i] = 0;
        else
            data[index * number_reads + i] = 0;
    }

    /* All active threads read and write data. Memory_src can be
     * global memory, texture memory, constant memory or shared
     * memory. If we are using shared memory, data is a shared
     * memory location
    */
    for (i=0; i < iterations; i++) {
        for (j=0; j < number_reads; j++) {
            if (coalesced) {
                tmp = memory_src[index + blocks * threads * j] + 1;
                data[index + blocks * threads * j] = tmp;
            }
            else {
                tmp = memory_src[index * number_reads + j] + 1;
                data[index * number_reads + j] = tmp;
            }
        }
    }
    /* If using shared memory, write result from global to
     * shared memory
    */
    for (i=0 i < number_reads; i++) {
        ....
    }
}
```

Listing 5.1: *A pseudo code of the test kernels*

and write pattern. The kernel follows the pattern of figure 5.1(b).

- **global shared** is a kernel that performs computation in shared memory and writes the results back to global memory in a coalesced manner. Benchmarked to see how fast doing computation in shared memory can be performed. The kernel follows the pattern of figure 5.4(a).
- **global shared bank** is similar to **global shared**, but with maximum possible bank conflicts on shared memory. The kernel follows the pattern of figure 5.4(b).
- **constant caching** is a kernel where each thread block reads from the same address in constant memory to try and see the benefits from the constant cache. The result is written back to global memory in a coalesced manner, showing the benefits of reading data from a cached memory space. The kernel is very similar to **global coalesced** apart from the fact that the block index is used as index in the constant memory space to limit the number of different accesses and fit a pattern that is suitable for the memory space.
- **constant various** is a kernel where each thread reads the address equivalent to its thread index from constant memory, and writes the result to global memory in a coalesced manner. This shows the same as **constant caching**, but with a spread read pattern on the cache. The kernel is similar to **constant caching** but every thread will use the thread id as index in the constant memory space.
- **constant uncoalesced** is similar to **constant various**, but writes the result back to global memory in an uncoalesced manner.
- **texture coalesced** is similar to **constant caching**, but each thread reads different values from texture memory.
- **texture uncoalesced** is similar to **texture coalesced**, but reads from the texture in an uncoalesced manner, and writes the result to global memory in an uncoalesced pattern. The kernel is designed show how an uncoalesced pattern can kill performance from an efficient source.
- **scatter coalesced** is similar to **global coalesced**, but in each half-warp the access are crossed with each other to see how the different compute capabilities optimise the pattern. The kernel follows the pattern of figure 5.2(a).
- **segment coalesced** is a kernel where we test how accesses across different segments in a half-warp is executed using the memory transaction protocol in compute capability >= 1.2. The kernel follows the pattern of figure 5.2(b).

- **char coalesced** is similar to **global coalesced**, but using an 8-bit data type to show the latest compute capability supports coalesced 32-byte transactions.

The kernels are designed to benchmark how different access patterns perform on the different memory spaces, so we want to ignore any operations that are connected to the host. To achieve this we used the NVIDIA visual profiler [46] and monitored the following hardware counters from the counters it offers:

- **gld_incoherent** is the number of uncoalesced loads on global memory.
- **gld_coherent** is the number of coalesced loads on global memory.
- **gst_incoherent** is the number of uncoalesced stores on global memory.
- **gst_coherent** is the number of coalesced stores on global memory.
- **local_load** is the number of local loads from local memory.
- **local_store** is the number of local stores from local memory.
- **branch** is the number of branched events taken by threads.
- **divergent_branch** is the number of divergent branches within a warp.
- **instructions** is the number of instructions.
- **warp_serialize** is the number of threads that are serialised because of address conflicts on constant memory or shared memory.
- **cta_launched** is the number of threads blocks that have been profiled.

In addition, the profiler outputs the total time the profiled kernel spends on the GPU, CPU and the occupancy of the GPU. We are looking for the number of coalesced and uncoalesced instructions. We are also interested in investigating the number of bank conflicts. So the counters we are interested in are the *gld_incoherent*, *gld_coherent*, *gst_incoherent*, *gst_coherent*, *warp_serialize* and *cta_launched*. Not only do we want to see how the different values affect the total gputime, but also see how the different compute capability properties can improve the number of loads and stores needed. The loads and stores indicate if instructions have been combined into a coalesced access. The lower number of instructions means that the GPU has managed to coalesce the instructions.

We profiled the kernels on a compute capability 1.1 GPU (NVIDIA 8800GT-OC) and a compute capability 1.3 GPU (NVIDIA GTX 280), with the parameters `number_reads` and `iterations` set to 4 and 10000 – 40000 respectively. The reason we only choose 4 as the `number_reads` was because the combination of register usage per thread and

the block size limited the amount of shared memory available. The results plotted are the average values of ten runs of each kernel, with a grid size of 60 thread blocks containing 256 threads.

5.4 Results

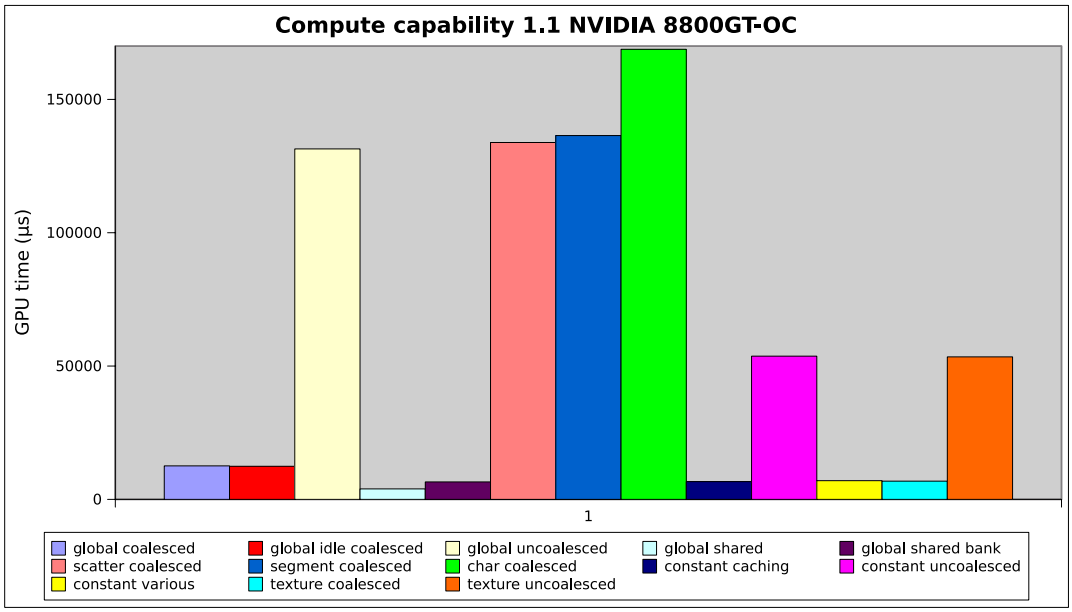


Figure 5.5: *The total gputime for the kernels running with* 10000 *iterations on a compute capability 1.1 GPU*

The total GPU time from the kernels running on a compute capability 1.1 and a 1.3 GPU can be seen in figure 5.5 and 5.6. The figures are plotted separately for readability. There is a significant difference in the GPUs when it comes to processing ability as it contains more processors, so the actual GPU time will always be lower on the GTX 280 card. The speed difference becomes very clear by looking at the scale on the y-axis for both figures. We will focus on how the access patterns are treated differently on the GPUs, and how the compute capability affects performance rather than the number of cores.

To help explain the GPU time results, we have also plotted the number of load and store transactions that are issued on an SM for each of the kernels in figure 5.7 and 5.8. As a coalesced load or store will issue considerably less transactions.

Both figures show that a coalesced access will improve performance due to the reduced number of transactions. In the **global coalesced** and **global uncoalesced** kernels there

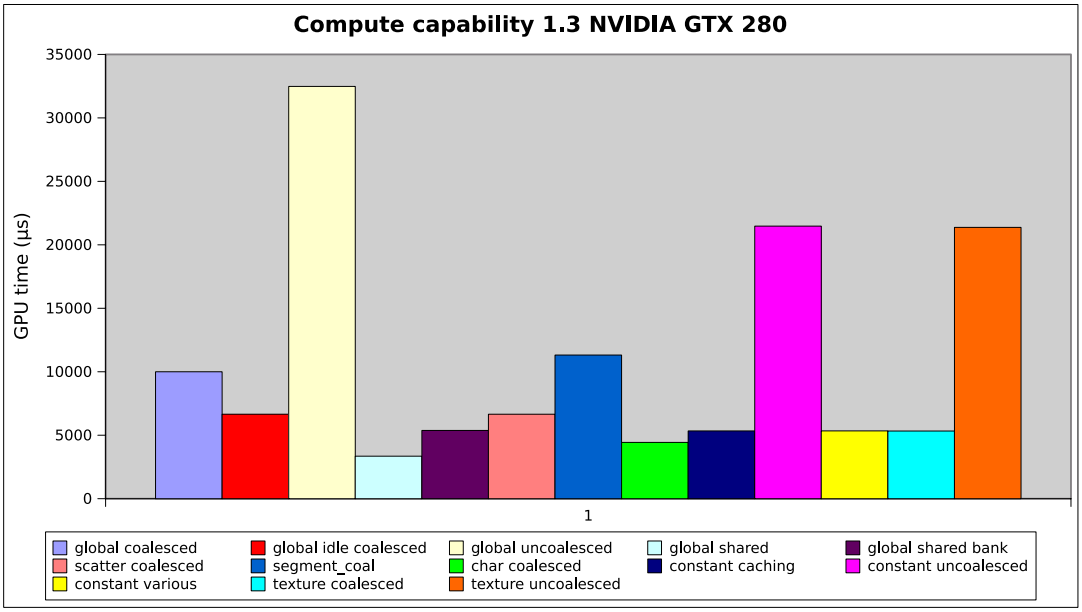


Figure 5.6: *The total gputime for the kernels running with* 10000 *iterations on a compute capability 1.3 GPU*

is no difference in the compute capabilities, but the runtime of the kernels are affected by the number of transactions issued.

In the **global coalesced idle** kernel, bandwidth is wasted in compute capability 1.1, as each thread spends time processing data even if it does not need it. The improvement in how to split memory transactions is seen in the GPU time of the same kernel in compute capability 1.3. The number of loads are equal, but due to a smaller transaction size the speed improves.

The kernels **char coalesced**, **scatter coalesced** and **segment coalesced** show how performance is affected by not following the requirements for a coalesced access in compute capability 1.1. Breaking the requirement of word size, sequential accessing across threads and reading words within a certain segment size gives the same kind of performance as an uncoalesced kernel. The requirements are not as strict in compute capability 1.3, which is shown by the GPU time of the different kernels in figure 5.6. The performance increase can be explained by the added support of 8-bit words, support for scattered transactions within a segment and having the memory transfer protocol issuing a smaller number of requests.

Reading from a cached memory space is quicker than reading from global memory. From our results, we do not see a difference between the access pattern in the constant or texture memory. However, we see that writing back to global memory in an uncoalesced fashion is expensive as to be expected. We were surprised to see that the stride

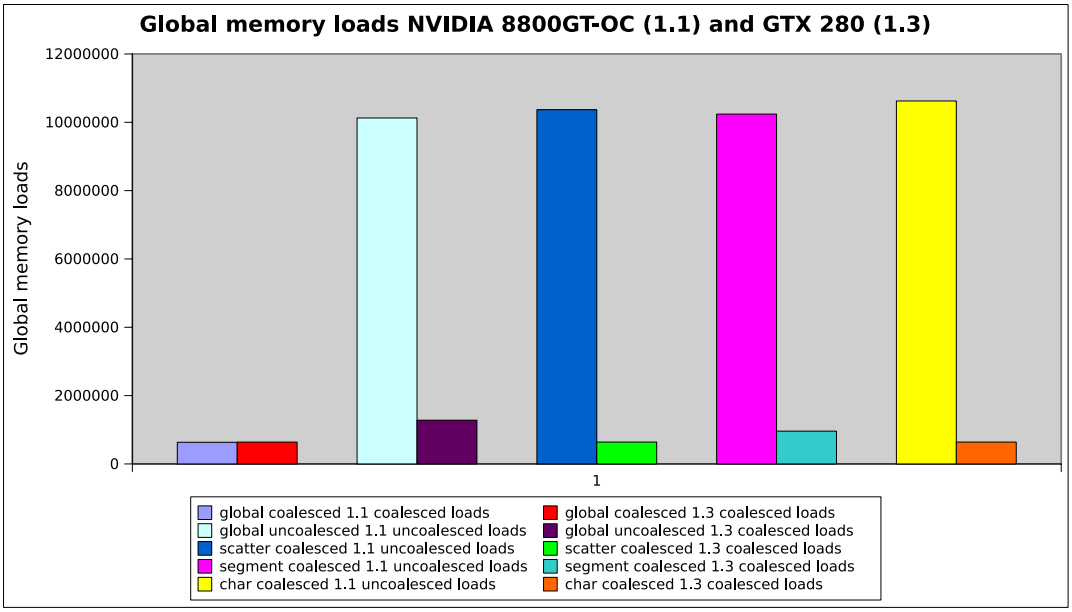


Figure 5.7: *The number of global memory loads in each kernel. Loads from constant and texture memory are not listed in the profiler, and therefore not graphed.*

used in **constant various** did not reduce performance compared to **constant caching**. These results show that the use of cached memory spaces also allow a more flexible access pattern, as the different patterns do not show any different results. We believe this may be due to the size of the memory allocated, and that all the data elements were placed in the cache after the first cache miss. Due to time constraints we have not been able to test this at present time.

As illustrated in figures 5.5 and 5.6, doing computation in shared memory is very efficient. Even with a high amount of bank conflicts the kernels perform very well. Normally there would be the added transactions of moving data from global memory to shared memory, which would increase the total GPU time. But as we wanted to show how quick the shared memory space is for doing computation, we did not add this step. We compared how the shared memory space performs isolated, with and without bank conflicts. Our results did not show that bank conflicts have as a severe affect on performance as Boyer et al. [47] illustrated. They claim a kernel with the maximum possible bank conflicts can perform as bad as the same kernel only using global memory. However, they have not explained how they tested their code to see the affect of the bank conflicts.

From our results we can conclude that a bank conflict pattern is almost as twice as inefficient, and with a higher number of computations the performance would decrease noticeably and we might have seen the same pattern as Boyer et al. [47]

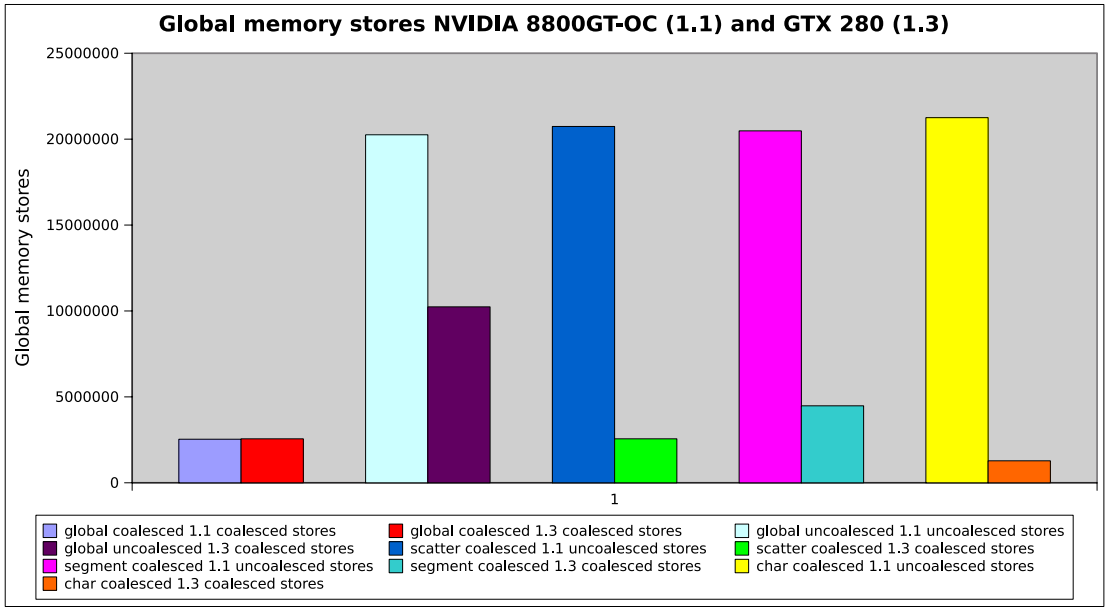


Figure 5.8: *The number of global memory stores in each kernel. The kernels that are not graphed follow the same pattern as similar coalesced or uncoalesced kernels*

Figure 5.9 shows how the different kernels experience bank conflicts. It should be noted that when profiling the compute capability 1.3 shared memory kernels, we only managed to get a GPU occupancy of 0.75, compared to all the other kernels where we got an occupancy of 1. Not reaching full occupancy means that the SMs might spend time idling, because resources are not available, and the CUDA runtime cannot find warps to schedule. This means a thread will in average have to wait for a longer time to get processed, but as a side effect will avoiding as many bank conflicts as there are not as many threads trying to access the same bank as with a full occupancy of the GPU.

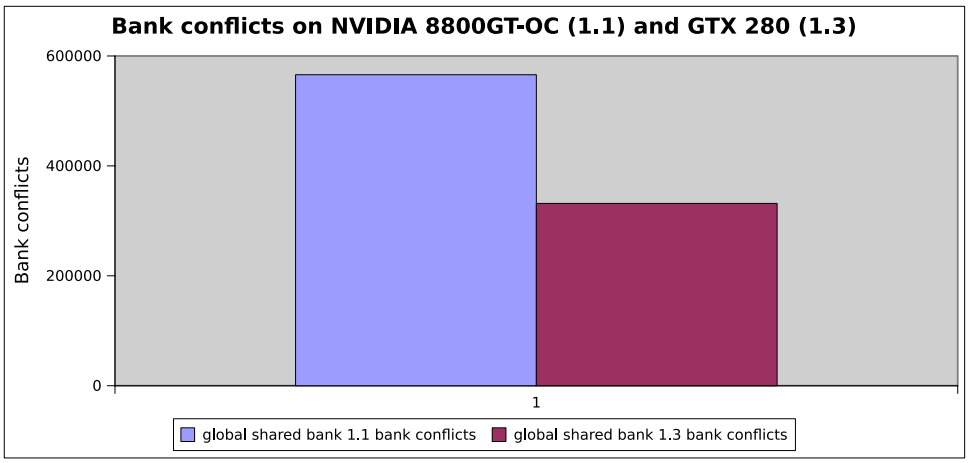


Figure 5.9: *The number of bank conflicts in the kernels running on different GPUs*

5.5 Lessons learned

We have seen what access patterns perform well in the different memory spaces, and what usage the memory spaces are designed for. Our results indicate that following requirements for coalesced memory accesses on global memory are crucial for the performance of an algorithm. However, the added flexibility in support of different access patterns in compute capability 1.3 helps the developer limit memory latency. This clarifies some of the performance issues regarding our AES implementations in chapter 4, and gives us indications on how to map our algorithms.

It should be noted that it is not always possible to adapt the algorithm to fit the requirements for to gain the optimal accesses. In some cases the algorithm itself is not suited for GPU offloading, but we have seen that one way to limit memory latency is to use the cached constant or texture memory space. They are restricted in size and are read-only, but offer good performance even when requirements are not followed 100%. Unlike global memory, which suffers in performance when the required pattern is not followed. Another solution for accesses that do not fit with the requirements, is to pad the allocated memory so alignment is assured, and to make the access patterns easier to program. This will use more global memory, but could be worth the tradeoff in performance.

Overall the memory spaces have their qualities, and different usage areas. In combination, they can give great performance, but also added complexity as the algorithm needs to be adapted to the access patterns and properties of each memory space used. From our results we can see an optimal pattern is to load data in a coalesced fashion from global memory into shared memory, or if applicable use a cached memory space as source. The use of the texture memory for reading can help the developer if the access patterns are difficult to adapt to global memory. The computation should be done in shared memory, without bank conflicts, which according to Boyer et al. [47] can give as high latency as using global memory.

To summarise, the developer needs to make sure every memory access is correct according to the specifications, and use the tools available like the CUDA profiler to see how the kernels performs. Boyer et al. [47] have proposed a tool that detects bank conflicts in CUDA code. Tools like these is something that would be of great benefit for CUDA developers. We also considered extending their work to detect coalesced and uncoalesced accesses, but we leave this for future work.

5.6 Summary

In this chapter, we have clarified how to gain optimal memory access in a kernel. We have run isolated tests to show how various access patterns in memory affect the performance on GPUs with different compute capabilities. We have explained how to obtain efficient and coalesced memory transactions, by following the requirements set by the compute capabilities. The results show that it is very important to follow the requirements, and that every memory transaction should be evaluated to see if it follows the an efficient pattern. If applicable, a cached memory space should be used for reading data into shared memory, while avoiding bank conflicts, for computation. The results should then be written back to global memory in a coalesced manner.

Additionally, we have seen that the latest hardware from NVIDIA eases the restrictions on the algorithms for easier performance gains making it easier to get good performance from the GPU.

Memory optimisations is important for efficient CUDA applications, but there is also optimisations that can be done when it comes to transferring data to the GPU for computation. In chapter 7 we investigate another possible optimisation we mentioned in chapter 4, which is to have memory transfers overlap with kernel execution through the use of the asynchronous API called streams.

Chapter 6

Concurrency with CUDA applications

In this chapter, we investigate how the CUDA framework executes applications that try to access the GPU concurrently. If CUDA is to become popular for offloading computations, it is important that applications can run concurrently without negating the performance the offloading was originally set to enhance. We look at scenarios similar to a situation where a CUDA application like our AES encryption is run concurrently with another CUDA application like the badaboom h.264 encoder [48]. We investigate alternatives for running concurrent processes, and look at how the applications use the resources in todays framework.

6.1 Introduction

As CUDA applications increase in popularity, there is a higher possibility that there will be a need for support of concurrent execution of kernels on the GPU. GPGPU applications use the GPU to increase performance in their applications, so it is important that concurrency does not diminish the advantage GPU processing offers. In CUDA at present time, CUDA applications are context switched like other applications on the CPU, but on the GPU the accesses are executed in first come first serve basis, without preemption or context switching between applications executing on the GPU. This leads to serialising of accesses by the CUDA driver as access to the GPU is exclusive and the applications executes until it is finished.

When executing CUDA applications, the CUDA memory copy operations block the CPU thread, and the kernel launches are asynchronous. Due to the asynchronous capabilities, we believe that it is possible to have CUDA memory copy operations and

kernel launches overlap between different CUDA processes. In addition to the fact that they are executed independent processes in the operating system. To see how the CUDA processes are run when executing concurrently, we ran a script that executes four identical CUDA processes simultaneously. The process used for testing is a generic workload that calculates a value based on the thread index and thread block index before writing the result to global memory. Our focus is on how the CPU schedules the applications when the GPU is busy, and not how the GPU executes the workload. We want to monitor how each process executes on the CPU when they compete for the GPU resource, to investigate the scheduling pattern of each application on the CPU. The monitoring was performed by outputting when a functionality like a kernel launch or memory copy was performed. We refer to the outputs as steps in the execution. The steps we monitor in each application are the following:

- `cudaMemcpyHostToDevice` operations, where data is transferred to the GPU for computation
- The asynchronous kernel launch
- `cudaMemcpyDeviceToHost` operations, where data is transferred from the GPU after computation
- Output to mark the end of an iteration

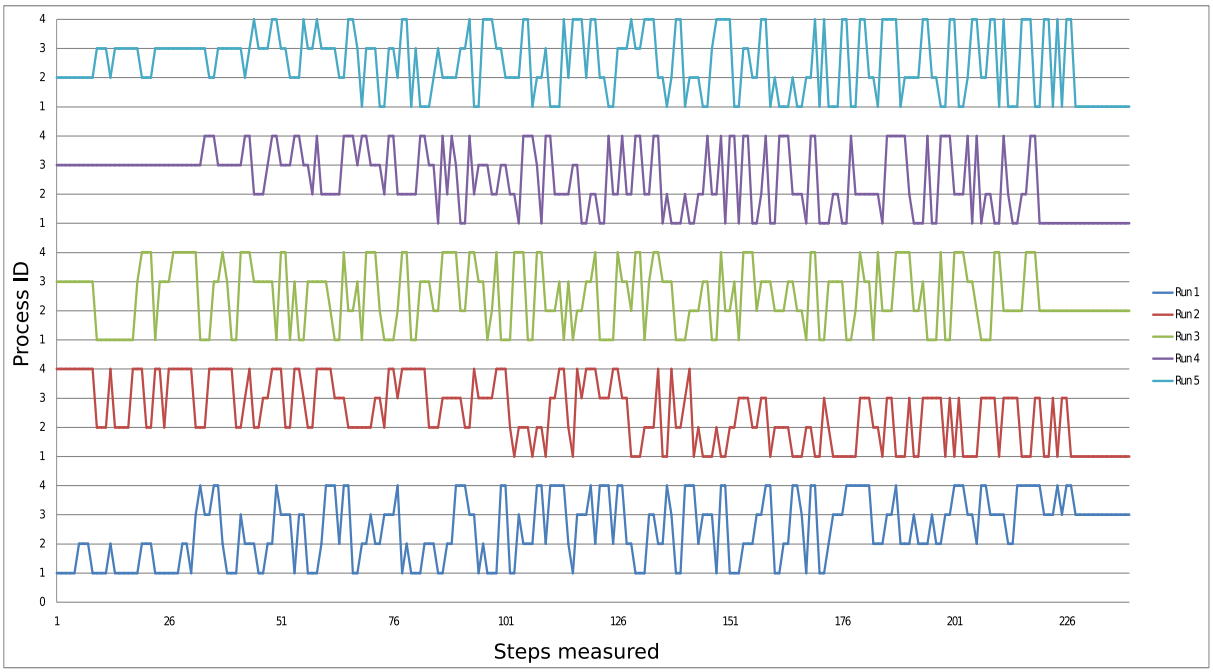


Figure 6.1: *An illustration of how concurrent CUDA processes are executed on the CPU.*

The steps were chosen to see a combination of synchronous and asynchronous calls,

and because they are functions that are used in most CUDA applications. We are not interested in monitoring all the context switches on the CPU, but want to see an outline on how the processes are scheduled on the CPU when multiple applications try to access the GPU. In each process we iterate the same code ten times giving a total of 240 steps. We ran the script five times to see if there was any variation of scheduling patterns.

The graph in figure 6.1 indicates an outline of how different CUDA processes may be scheduled on the CPU. The y-axis contains the process ID, and the x-axis shows the steps in each process to give a time line of the execution. Note that different processes may be context switched on the CPU between the steps measured, and that we only monitor the mentioned CUDA specific steps to see how they are scheduled compared to each other.

The figure 6.1 shows that CUDA applications are scheduled on the CPU in a random order, where some applications may suffer starvation before getting scheduled. What is important to note from the figure, is that even though GPU access are serialised, the CPU still schedules the other CUDA applications to limit the affect of serialisation of GPU accesses. In certain cases an application is not scheduled by the CPU before almost one quarter of the execution steps have passed. An example of this can be seen in run 4, where the monitored steps of process 1 is not scheduled before around step 80. In the same run process 3 is scheduled for a long time in the start of execution, which is not very fair towards the other processes. However, we can see from the various patterns that even though the GPU accesses are serialised, the CPU still schedules the other applications on the CPU. It should be noted that because of the random order of the CPU scheduling, in combination with the serialising of the GPU accesses, it is difficult to predict performance limitations.

6.2 Performance of concurrent CUDA applications

Accesses to the GPU are serialised, but as seen in the previous section, CUDA operations performed on the CPU can be overlapped between different applications. We want to investigate what affect context switching, like in figure 6.1, has on the performance of concurrent CUDA applications.

There are different ways to achieve concurrency for the applications. If an application needs to offload different functions or sets of data, the accesses can be offloaded in sequence by the algorithm. The difference in a concurrent launch compared to a

sequential launch is how many CUDA processes are run. In a concurrent approach, different processes try to offload computation to the GPU in the same fashion CPU processes try to access the CPU. In this case, the requests are serialised by the CUDA driver. In the sequential approach, the same computation is offloaded, but we execute the GPU requests in one process and the serialisation of the GPU accesses is controlled by the application. An example of an sequential kernel launch in an application can be

To see how the different approaches affect the performance of the applications, we have tested different combinations of CUDA processes using the mentioned approaches. The processes are referred to as workloads to give generic examples on applications that are offloaded to the GPU for computation. We choose the following workloads as they have different characteristics in how they use the GPU, to emulate a real world scenario where different applications might offload concurrently.. Our workloads explaining the characteristics are the following:

Workload1 is our GPU-L implementation from chapter 4. A memory bound application that uses lookup tables in the cached memory spaces and performs computation in shared memory.

Workload2 is our GPU-S implementation from chapter 4. A computation bound application that focuses on computation in shared memory.

Workload3 is a sequential execution of workload1 and 2. To illustrate an application using the GPU twice during execution.

Workload4 is a matrix multiplication, a typical application that the GPU can perform well. Rahmani [49] shows in his proposal that a CUDA based matrix multiplication implementation outperforms a CPU implementation with the same matrix dimension. Using a matrix dimension of 3072, the CPU has a runtime of 98.5 seconds while the GPU uses 3.67 seconds. Unlike the other applications, it uses a 2D grid of thread blocks, where each thread reads from global memory, computes in shared memory before writing the result back to global memory.

To test the affect of serialised accesses we run the four different workloads in various combinations. We compare the runtime of the concurrent execution with how the application runs when it accesses the GPU exclusively. To see if the scheduling of the different workloads on the CPU affects the runtime, we also run the workload combinations in one process offloading the computation of the workloads sequentially.

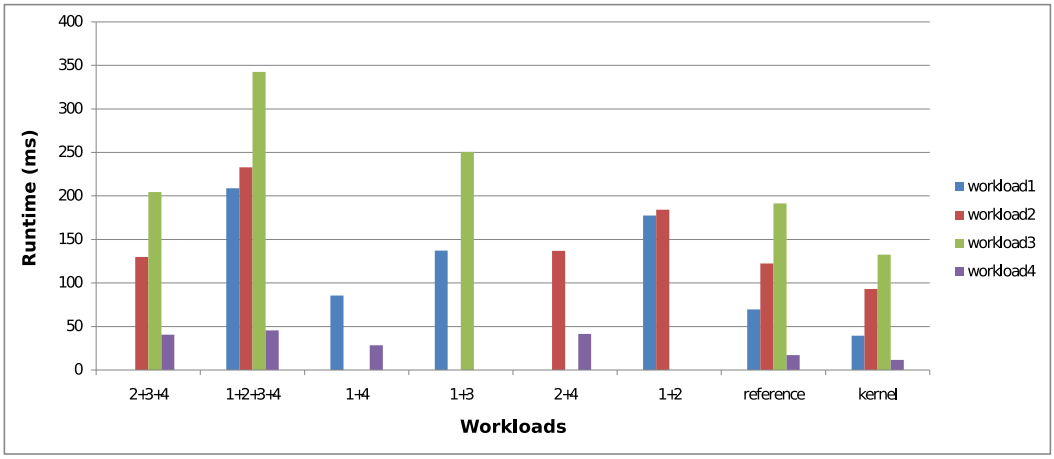


Figure 6.2: *Concurrent execution of different workloads in different processes. The runtime consists of GPU execution and memory transfers*

workload combination	runtime concurrent	runtime sequential	increase
$1+2+3+4$	8.0178s	8.5040s	6.0640 %
$2+3+4$	6.1412s	6.2164s	2.8865 %
$1+3$	5.8036s	5.9874s	3.1660 %
$1+2$	4.1088s	4.2274s	1.2245 %
$1+4$	2.3588s	2.3754s	9.1305 %
$2+4$	2.3416s	2.5554s	0.7037 %

Table 6.1: *The total runtime for the applications with different approaches*

Results

The results of concurrent execution can be seen in figure 6.2, where the *reference* column is the runtime of the workload when the workload has exclusive access to the GPU. The *kernel* column is the same as reference, but only showing the time spent executing on the GPU. The figure illustrates an increased runtime for all the workloads when GPU accesses are serialised. There are variations in how much increase each workload experiences, and what workload is affected the most. An example can be seen in the concurrent execution of *workload1 + workload3 (1+3)* compared to *workload2 + workload3 + workload4 (2+3+4)*, where the concurrent execution of the two processes 1+3 suffer more than 2+3+4. As we saw of the CPU patterns earlier in figure 6.1, we think it is plausible to believe that the difference in result can be caused by how the workloads are scheduled by the CPU. Table 6.1 shows the total execution time of processes launched concurrently compared to sequentially launching of kernels. There is not a large difference in the runtime. The concurrent approach benefits from having execution overlapped by the CPU and GPU, but pays a penalty in serialised GPU access and unpredictable scheduling from the CPU. The sequential approach does not overlap execution on the CPU, but experiences instant access to the GPU due to de-

sign of the algorithm and because there is only one process running. Even though the results are affected by the scheduling of the CPU, we can conclude that if a kernel is serialised, the increased runtime each workload experiences is equal to the runtime(s) of the kernel(s) that are queued ahead of the workload.

6.3 Static scheduling of concurrent applications on the GPU

In scenarios like real-time computing, reaching deadlines is critical. The overhead caused by the CUDA driver serialising kernel executions might not be acceptable in such systems. Therefore, we have looked at the possibility to run kernels from different applications in one kernel launch. This will ensure that the applications will gain access to the GPU concurrently instead of having the CUDA driver serialize the access requests for GPU processing. The concurrency is provided by having a generic kernel that combines two kernels into one, launches the generic kernel and statically schedules processes to different SMs based on the index of the thread block. An example can be seen in figure 6.3 where we have grouped 4 thread blocks to the same application. The GPU still only runs one kernel, but the generic kernel makes sure different applications can be processed concurrently by dividing thread blocks to different functions that are declared with `__device__` qualifiers and contain the original kernel code of the application. As there is one launch instead of multiple, the number of threads in each thread block will be the same for each process. This means the original kernels must support the same size of thread block, and the grid setup will be the same for both processes. This might require adaptation in a kernel so it can fit the grid setup used by the generic kernel. An example can be a 2D grid used in a matrix multiplication that needs to be combined with 1D grid setup used in AES encryption. Each kernel is mapped towards a specific grid setup, and will not work when the grid is setup differently, due to the different indexing of threads and thread blocks.

To avoid unnecessary idling of the GPU, each application should also use the same number of thread blocks. For instance, if an application needs 100 thread blocks, we will need to launch 200 threads for the indexing in each kernel to be correct. If the application is executed together with an application that only uses 50 thread blocks, there will be 50 thread blocks launched that would idle.

There are different alternatives in scheduling the thread blocks to different SMs. Since the information on how thread blocks are mapped to different SMs is disclosed by

NVIDIA, we have opted to try and see if we can execute different workloads by assigning a number of sequential thread blocks to the same application. In our approach, this number is variable because thread blocks assigned to SMs divide the threads into warps that can be scheduled at any time. We believe that if the thread blocks assigned to an SM belong to the same workload, there is a better chance of hiding memory latency. The processing steps in the warps are similar, and there is a better chance of achieving more efficient scheduling as there is less context switching on the SMs between applications. Context switching in this context, means that thread blocks from different applications are handed to the SM, and then scheduled in warps. Not context switching like when an application is preempted on the CPU. By grouping the assignment of thread blocks to applications, there is a better chance of having an SM containing a majority of warps assigned to a specific application.

Each process allocates its own memory on the GPU, and passes its own pointers to the generic kernel. The different processes are handed their parameters through calls to `__device__` functions that are in-lined during compilation. These functions contain the same code as the original kernels, including shared memory allocation, but have been adjusted to give the right thread block index. The latter is done due to the added number of thread blocks launched, and because the thread blocks are handed to different applications.

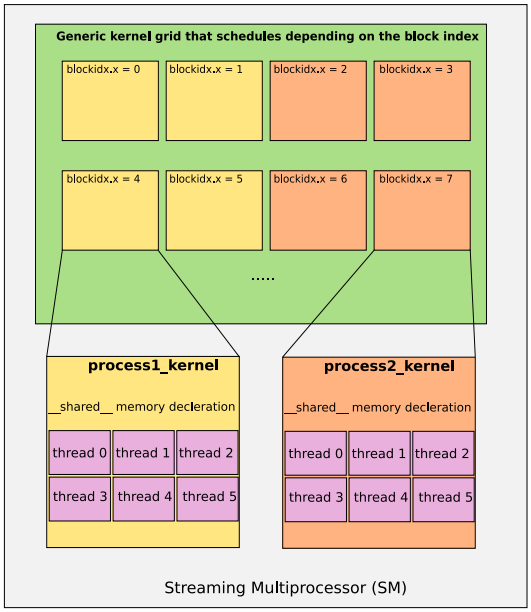


Figure 6.3: *A view of the grid of thread blocks when executing the generic-kernel.*

Results

Figure 6.4 shows how the combined kernel performs compared to a sequential launching approach, which is an alternative way of mapping the applications. Job split refers to the number of sequential thread blocks that are grouped together and assigned the same workload. Overall the sequential kernels perform better than the combined generic kernels. It is difficult to pinpoint the exact cause of this, but we believe a few factors may affect the performance. The first issue is that we do not know how the thread blocks are assigned to the SMs. We see a pattern when we use a larger group size of thread blocks with the same workload, as we get better performance compared to assigning every second thread block to a different application. This shows that it is easier to hide memory latency if warps from the same workload can be scheduled, rather than context switching between different applications on the GPU. However, there are also occurrences that indicate that a larger group may not be beneficial in the case of using 7000 thread blocks. Since the scheduling is like a black box performed by the driver and hardware units, we do not know enough of low level scheduling to give a clear answer to what is the cause.

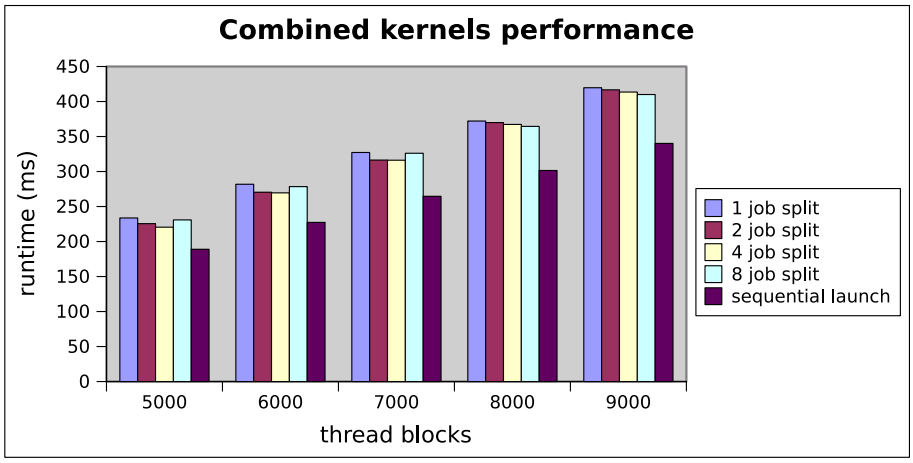


Figure 6.4: *Total runtime when using different approaches to combine workload1 and workload2 in one kernel launch.*

The scheduling of thread blocks will also affect caching. As both workloads use constant and texture memory, the applications will most likely suffer from cache misses if different workloads are using the caches concurrently. In other words, the cached memory spaces also suffer when there is frequent change in applications running on the SM.

Another reason is that the combined kernel has a larger resource demand than the sequential kernels, as it needs to allocate shared memory for both applications and

also has an increase in register usage because of the added logic needed in the static scheduler. This is not the case for the sequential offloading, as the kernels allocates GPU resources when they are launched.

6.4 Lessons learned

The CUDA framework today has a few limitations when it comes to concurrent execution of applications. We believe that the GPU should support concurrent execution of applications without killing the performance that was the motivation for offloading the algorithm in the first place. As accesses to the GPU are serialised if the GPU is executing another process. This affects the performance depending on the number of applications trying to use the GPU concurrently. Our attempt to limit the performance of serialisation with our static scheduler was not very successful. However, it should be noted that the generic kernel is not a typical approach for writing CUDA applications. As we have no control on how the thread blocks from the different applications are assigned to the SMs, it is plausible to believe that thread blocks from different applications are assigned to the same SM. This leads to context switching of applications when warps from different thread blocks are executed on the SM. As both of the applications use the cached memory spaces on the SM, and warps of threads from different applications are scheduled between each other, there is a chance that the data being requested is not in the cached memory space due to context switching of applications on the SM. So we believe that we need to be able to assign thread blocks from the same application on the SM to limit the performance penalty as it would mean all the warps are from the same application. Memory latency is also hidden in a better way if warps from the same application can combine execution. This is the case when the same application is run across all the SMs on the GPU, as different warps are scheduled when another warp is awaiting a memory fetch.

The structure of the code in our implementation is also very static, meaning it is not dynamically adjustable for other applications without rewriting and recompiling code. However, it is a way of showing that combining workloads is possible, even though it has to be done by hand as it is basically a combination of two applications. With better support in the API, or through another way of handing the GPU workloads, it is possible to run concurrent applications.

This leads us to believe that in systems where multiple applications will use CUDA concurrently, it is recommended to have multiple GPUs available. The framework has

good support for multiple GPUs by using an abstraction called *context*. A context can be attached to a certain GPU through the API making it easier allocate one GPU per thread if the application is threaded.

The multiple GPU approach might be feasible for supercomputers, but personal computers are still mostly installed with a single GPU unit. If CUDA increases in popularity, the serialising of the computation will affect the benefit of using CUDA. The same problem occurs today when CUDA applications are launched together with graphics applications like computer games. Both the performance of the CUDA application and computer games suffer in performance, but they are interleaved in a better fashion than concurrent execution of CUDA applications. According to NVIDIA in their CUDA 2.1 FAQ, CUDA is client of the GPU just like OpenGL and Direct3D drivers and share the GPU through time slicing. However, for the time slicing mechanism to be efficient for concurrent use, there is a need for interleaving of computation within CUDA applications. We discuss this matter further in chapter 8.

The support for time slicing on the GPU is interesting, and has been mentioned as an alternative for CUDA applications. There have also been rumours from NVIDIA 2008, NVIDIAs event to promote visual computing, that they have planned support for preemption of CUDA applications and a virtual pipeline. This might assist in achieving better performance for concurrent execution of CUDA applications in the future. However, it is not known how the preemption will be implemented, or what the virtual pipeline will offer.

In our view, context switching with preemption like on a CPU is too expensive to perform due to the nature of the architecture as the amount of state needed to be saved is too large. This could be solved by context switching between the execution of warps, but as shown in our results in section 6.3 we see that this affects the performance in a negative manner. Another issue is the limited runtime environment on the GPU. It would be better to preempt between contexts within an application, like between thread blocks. We discuss further alternatives to scheduling in chapter 8.

6.5 Summary

In this chapter, we have investigated how concurrent CUDA applications are executed both on the CPU and GPU. As CUDA applications increase in popularity there is a higher probability that applications will issue processing requests to the GPU concurrently. We have investigated how multiple requests for GPU are handled by the de-

vice driver in todays framework, and what impact this has on the performance. Our findings show that in concurrent executing CUDA processes on the CPU, performing computation on the GPU is exclusive for one application, meaning the accesses are serialised on a first come first serve basis without preemption or context switching between applications. While the GPU accesses are serialised, the CPU can execute other CUDA applications. We see a penalty in the performance depending on how many CUDA applications are competing for the resources, and how the CPU parts of the CUDA applications are scheduled by the CPU. In most cases we see a added runtime of the applications equal to the number of kernels that are executed while the application awaits GPU time. To see if we could increase concurrency of the applications, we created a static scheduler that computes two different applications in one kernel by trying to allocate a set of streaming multiprocessors on the GPU to each application. Our attempt was unsuccessful due to the limited control the developer has scheduling thread blocks on the GPU.

Chapter 7

Optimising applications with CUDA streams

In this chapter, we look at the asynchronous API in CUDA. It offers functionality for overlapping execution of memory transfers and processing on the GPU. We use the AES implementations from chapter 4 to investigate if streams can improve the runtime of the implementations. Additionally, as the asynchronous API requires the use of pinned system memory, we test the throughput of both pinned and pageable memory allocated through the CUDA API.

7.1 Introduction to CUDA streams

To improve the performance of applications using CUDA, we want to investigate how the CPU and GPU can concurrently execute operations. CUDA offers concurrent execution between the CPU and GPU through their asynchronous API called streams. CUDA streams should not be confused with the term stream processing, where a stream is a data-set to be computed by a kernel. A stream in this context is a sequence of operations that execute in the sequence they are added to the stream.

In order to facilitate concurrent execution between the CPU and GPU, some of the runtime functions in CUDA are asynchronous. We have mentioned previously that kernel launches are asynchronous, but the streams API offers asynchronous memory copy operations. These work the same way as normal memory operations, but the functions are suffixed with `Async`. So the asynchronous operations are; kernel launches, memory operations that are suffixed with `Async`, memory set operations and GPU to

GPU memory copy operations executed from the CPU.

Different streams in an application may overlap in execution, letting memory transactions overlap with a kernel launch of another stream. To enable concurrent execution, it is necessary to have multiple memory copy operations and kernel launches within an application. Listing 7.1 shows a simple code example on how streams can be setup. While, figure 7.1 illustrates how the overlapping of execution is performed compared to an application with one or multiple synchronous launches.

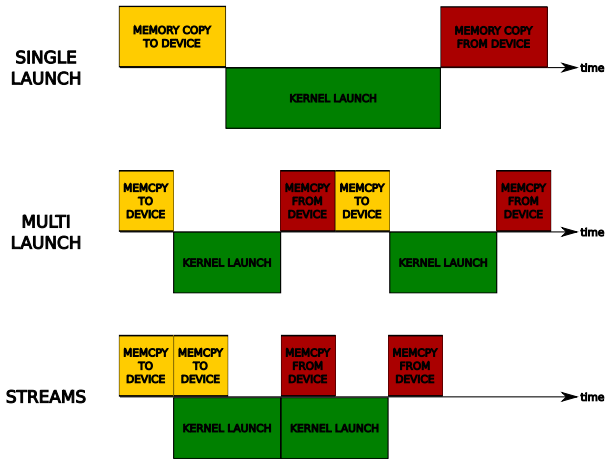


Figure 7.1: *How the CPU and GPU overlap in execution with the use of streams.*

A stream is created by using `cudaStreamCreate`, which creates a stream object that the runtime environment can attach operations. The execution within a stream is sequential, meaning each operation is not performed until all previous operations is completed. However, different streams may be overlapped in execution, meaning a memory copy can be performed in one stream, while the other stream performs computation on the GPU.

To ensure a stream has completed, the runtime can force the application to wait until all the tasks in the stream have completed by using `cudaStreamSynchronize`. It is a similar to `cudaThreadSynchronize`, which is a blocking function to make up for the asynchronous kernel launches. Otherwise it would not be possible to time the kernel execution.

The runtime also provides ways to monitor the progress of a stream, as well as record accurate timing by using *events*. They can be recorded at any given time in the application, and may use the built-in function `cudaEventElapsedTime` to calculate the elapsed time between two events.

Streams require that memory used on the CPU is allocated with `cudaMallocHost` as

```
__global__ void stream_kernel(int *in, int *out) {
    /* Performs computation on the GPU in parallel */
}

int main(int argc, char **argv) {
    /* Initialize event and stream objects */
    cudaEventCreate(&start);
    cudaEventCreate(&stop);
    for(i= 0; i < STREAMS; i++)
        cudaStreamCreate(&stream[i]);

    for(i=0; i < STREAMS; i++)
        cudaMemcpyAsync(gpu_input + offset * i, cpu_input + offset *
            i, mem_size, cudaMemcpyHostToDevice, stream[i]);

    /* Setup each stream with the same operations */
    for(i=0; i < STREAMS; i++)
        stream_kernel<<<100, 512, 0, stream[i]>>>(gpu_input + offset
            * i, gpu_output + offset * i);

    for(i=0; i < STREAMS; i++)
        cudaMemcpyAsync(cpu_output + offset * i, gpu_output + offset
            * i, mem_size, cudaMemcpyDeviceToHost, stream[i]);

    for(i=0; i < STREAMS; i++) {
        cudaStreamSynchronize(stream[i]);
    }
    /* Write result to file if needed */
}
```

Listing 7.1: *An example on how an application can use streams to try and increase the performance by having the GPU and CPU overlap execution*

concurrent execution cannot be performed without using page-locked memory. This means the memory is pinned in virtual memory, and cannot be swapped out like pageable memory allocated with `cudaMalloc`. The memory is also accessible from the GPU as the driver records the virtual memory addresses allocated with this function and automatically accelerates calls to functions such as `cudaMemcpy`. As the allocated memory can be accessed directly by the GPU, it can be accessed with a much higher bandwidth than pageable memory. However, allocating memory using `cudaMallocHost` is costly for system performance, as it reduces the amount of memory available to the system as the pages cannot be swapped out if other applications request memory.

7.2 Tests

To see how the use of pinned memory in applications affects the performance, we have benchmarked memory transfers between the CPU and GPU using both pinned and pageable memory. The test is based on a bandwidth-test application in the CUDA SDK where different sizes of memory are allocated and transferred between the GPU and CPU.

For testing the performance of streams in a CUDA application, we modified the lookup table (GPU-L) and standard (GPU-S) AES implementations from chapter 4 to use streams. We tested the implementations on a GTX 280 GPU (compute capability 1.3) as it was the GPU that gave the best performance in our tests in chapter 4. To see if there was any variation we test the different implementations with 512, 1024 and 2048 thread blocks. As each thread block encrypts 256 threads * 16 bytes, the number of thread blocks determine the amount of memory used. For concurrent execution between the GPU and CPU to occur, the application must use multiple streams otherwise no streams can be overlapped. Therefore we have tested with 2, 4, 6, 7 and 8 streams in our applications. We did not want to use an unnecessary large amount of pinned memory when testing, so we limited our tests to 8 streams. Each stream contains memory copy operations and a kernel launch, similar to the pattern seen in listing 7.1. We compare the stream implementation to a similar implementation that does not use streams (multi launch), and an approach with a single launch using a larger memory size (single). The implementations are illustrated in figure 7.1. All the implementations perform AES encryption on the same data, but the stream and multi launch implementations work on a smaller data size on each launch, while the single launch performs the encryption in one launch.

number of streams	GPU-L % speedup increase	GPU-S % speedup increase
2	6.6	2.1
4	13.1	5.3
6	13.8	6.0
7	14.6	6.1
8	15.1	7.1

Table 7.1: *The average increase in the performance using streams compared to the single launch implementation*

7.3 Results

The results from our pinned and pageable memory test are plotted in figure 7.2. From the figure, we can see that pinned memory has a higher throughput than pageable memory when using the same allocation size. The difference between the GPUs in the benchmark is the PCI express bus the GPUs use for communication with the CPU. Therefore, we have plotted the results from one PCI express 1.x and 2.0 GPU. PCI express 1.x can achieve a data rate of 250 MB/s per lane, compared to 500 MB/s in PCI express 2.0. The GPUs use a 16x channel, meaning they can transfer data across 16 lanes giving a theoretical data rate of 4000 MB/s and 8000 MB/s. The theoretical data rate is not achievable as the data rate is limited by the system memory.

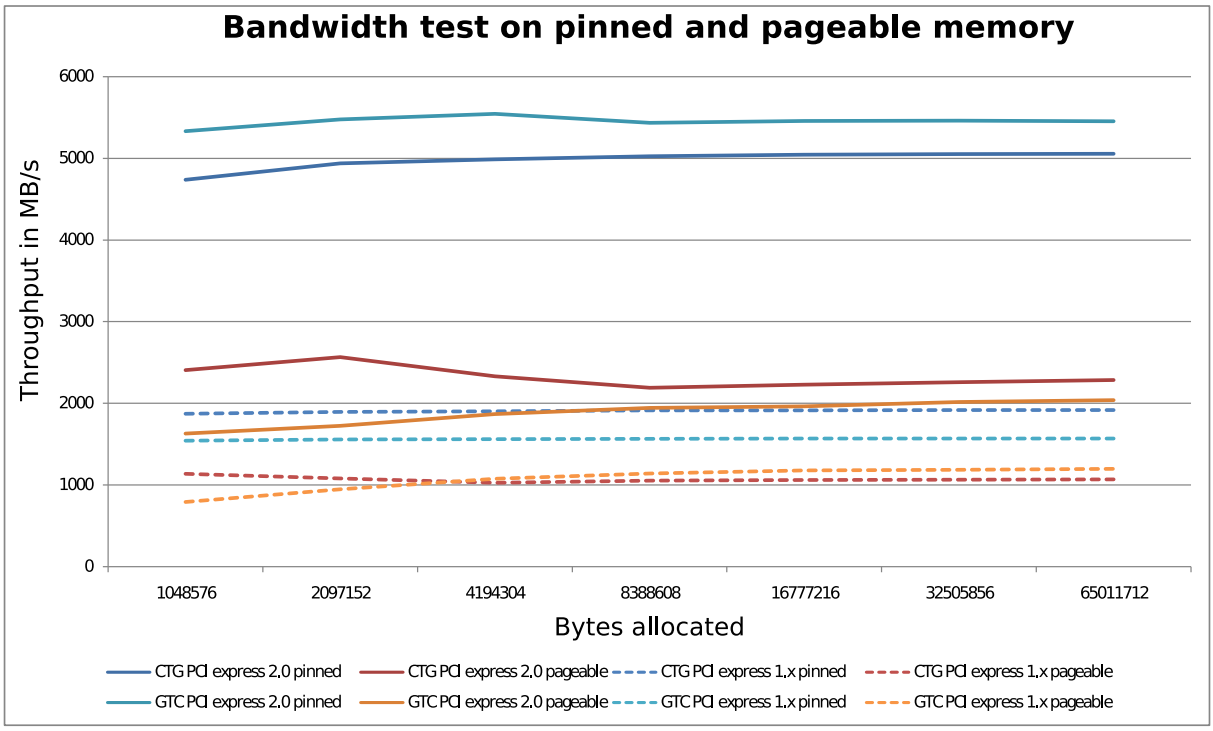


Figure 7.2: *Throughput of pageable and pinned memory on CPU to GPU (CTG) and GPU to CPU memory transfers (GTC).*

The performance of the implementations using 8 streams is shown in figure 7.3. The results from the other number of streams are listed in in table 7.1. From the figure, we see streams increase the speed of the GPU-L implementation, while table 7.1 indicate that the GPU-S does not experience the same speed-up. We believe this is caused by the slower runtime of the GPU-S kernel. To achieve an increase in the performance with concurrent execution, the amount of memory being copied must correspond to the kernel execution time. The time used for memory transfers in both implementations is the same when executing with similar parameters. As the runtime of the kernels differ, we believe this determines the amount of execution overlap that occurs.

The multi launch implementation is slower on GPU-S than GPU-L. An explanation for this can be that the GPU-S kernel has a lower occupancy on the GPU with the execution configuration we test with. The single launch implementation launches a higher number of thread blocks in one launch than the multi launch giving higher probability of increased occupancy. Higher occupancy means more thread blocks per SM, giving a larger amount of warps to schedule for the SIMT unit making it easier to hide memory latency.

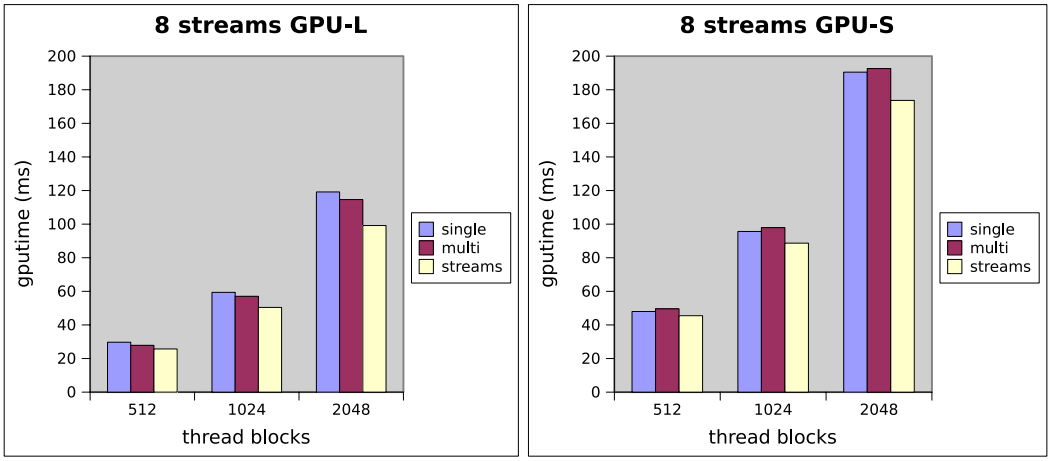


Figure 7.3: *Illustration of the runtime using streams compared to other approaches.*

In figure 7.4, we illustrate how a kernel with different run-times affects the performance when using streams. We measure the runtime of the kernel as a job size, to easier adapt the workload of the kernel for testing how it affects the concurrent execution. The job size is adjustable to adjust the runtime of the kernel to test how it affects concurrent execution. The number of thread blocks launched determines the size of memory being copied as it means more data needs to be processed. Figure 7.4 illustrates how certain job sizes influence how much overlap is achieved when using the same memory size in the memory transfers. As the kernel runtime increases, the size of memory being copied will become negligible, as the total runtime is only affected by

the runtime of the kernel. This is challenging for the developer, as there is at present time no way to determine the extent of overlapping that can be achieved between a kernel and a given size of memory without trial and error.

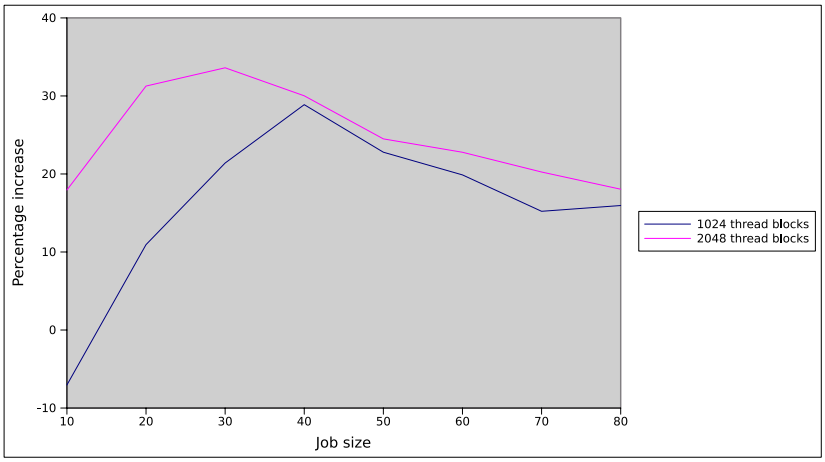


Figure 7.4: *Illustration on how the runtime of the kernel affects overlap in execution.*

7.4 Lessons learned

Our results show that streams can increase the performance of a CUDA application. The performance increase is determined by how much of the execution can overlap. From our results, we see that the amount of overlap that occurs is determined by how well the runtime of the kernel corresponds to the size of memory being transferred. In applications transferring small data sizes, streams may not achieve an increased speedup. Therefore, it is important for the developer to test a different number of streams, and memory sizes, to achieve the best possible speedup. The increase in the performance using streams is not just caused by the overlapping execution, but the asynchronous functions require the use of pinned memory. We have seen that pinned memory has a higher bandwidth due to the memory mapping done on the GPU, but using pinned memory reduces the total system memory available for other applications. This is a restriction for applications using large amounts of memory, which could potentially make a system unstable due to the lack of memory.

7.5 Summary

In this chapter, we have investigated how concurrent execution between the CPU and GPU through streams benefit the application, and how the performance is affected by the use of pinned system memory. The results show that the use of streams can improve the runtime of an application, when part of a memory transfer from the CPU can overlap with kernel execution on the GPU. The amount of overlap that occurs is determined by how well the runtime of the kernel corresponds to the size of memory being transferred. Through the use of streams, we have achieved concurrent execution between the CPU and GPU to increase the performance of our AES implementations from chapter 4.

Chapter 8

Discussion

In this chapter, we discuss our experience using the CUDA framework for developing applications to run on the GPU. We focus on how the developer should do optimisation, the tools available to assist in developing and how debugging is performed. We discuss our experiences with concurrent execution on the GPU, and explain what are thoughts are regarding future solutions for scheduling on the GPU. To summarise the discussion, we look at what is in store in future CUDA releases and the GPGPU research area.

8.1 Developing with CUDA

Creating algorithms that run in parallel can be challenging due to factors like bugs caused by race conditions. Communication and synchronisation between different subtasks are typical issues in algorithmic design that are challenging when the goal is to obtain good performance. Programming APIs like OpenMP [50] have been developed to assist in developing applications that can utilise and scale to multiple processor cores on modern CPUs. OpenMP is an API that supports multi-platform shared memory parallel programming in C/C++ and fortran [50], and is intended to give developers an interface to create scalable parallel applications. Like CUDA, OpenMP aims to scale to the number of cores available. CUDA and OpenMP share the same challenges in designing efficient parallel algorithms. CUDA uses a different memory model and thread model that might increase complexity of the development process for the programmer. To make it easier for the developer to overcome these challenges, CUDA offers the use of the C programming language, the possibility to explicit use the different memory spaces and easier integration with existing x86 code.

8.1.1 Optimising code

Developers have their own preference with regard to how they prefer to develop applications, and is also dependent on what kind of application being developed. There is no wrong or right, so it is difficult to give any clear patterns for what is best to do. During our work with the CUDA framework developing applications like AES and performing optimisations, we have learned how to use the framework and what challenges the developers face when doing so. Designing a good parallel algorithm is challenging, but in CUDA you have the added difficulty of using the right memory space in the right manner. There are many ways to optimise the code when it comes to the use of the different memory spaces available. According to Donald Knuth: "We should forget about small efficiencies, say about 97% of the time: premature optimisation is the root all evil", which we believe is a valid statement for CUDA developers. Due to the different thread hierarchy and the memory model in CUDA, it is important to get an implementation working before starting to optimise memory accesses. If optimisations affects the design of the algorithm, the code may end up being difficult to understand, maintain and debug.

Optimisations in CUDA become easier when the developer has a lot of experience developing applications with the framework. For beginners, the programming model may be challenging, and it is smarter to focus on getting the algorithm working before doing optimisations. A step by step approach is to get a working implementation using a familiar memory space like global memory, before altering access patterns in steps to test if it improves performance. In our opinion, a good way for unexperienced programmers to start optimising is by only using global memory, as it resembles memory accesses seen in x86 applications and can be copied back to the CPU for debugging. The memory space is also available across thread blocks, limiting the use of unfamiliar memory spaces like shared memory.

When working with parallel algorithms, the design can be challenging to understand. In a simple design, it is easier to optimise the code and check the correctness of the algorithm. A benefit by optimising in steps from a simple base, is that it lets the developer concentrate on keeping a clean design, making adjustments to memory access patterns and placement step by step.

It is important to note that even tough an algorithm is not optimised, it should be designed so that optimisation of access patterns can be altered without having to rewrite the whole algorithm. This means that a stepwise approach is recommended for all developers, but for the novice programmers re-designing the algorithm might be nec-

essary during the process of learning how to use the framework. Through experience with the framework and architecture, we believe the developer will be better equipped to do faster optimisations in the future, like placement of data in the various memory spaces, how much workload a kernel should compute and in experimenting with parameters to see how performance changes using different thread hierarchies etc. The latter is relevant for experienced CUDA developers as well, as there is a lack of tools to assist in this process. But through experience, the basic steps for optimisations become shorter and easier.

8.1.2 The memory spaces

The explicit use of the different memory spaces increases the flexibility of the framework, but also gives the developer added options when designing the algorithm. For many developers the increased number of memory spaces can be off-putting. In some cases data may be very suitable to be placed in for example texture memory, but as it is unfamiliar for the developer, it is placed in global memory. This may limit the potential performance of the application, depending on how much data and the access pattern used in global memory. The texture memory space is perhaps the most challenging to use for a developer who has no experience with graphics APIs. It is very efficient for read-only data, and could also be used as a alternative to reading data from global memory to shared memory due to the dynamic setup the texture API uses. As textures are fetched in an optimised fashion by the hardware, it relaxes the restrictions to limit latency unlike global memory that has strict requirements in access patterns. This is a feature many developers can benefit from when struggling in obtaining an efficient access pattern on global memory. However, as we have had no previous experience working with textures in programming APIs like OpenGL [13], we found the usage of memory space challenging to use. From reading posts on various CUDA forums we see other users with the same opinion. Even if we found the texture memory space challenging to use, we think it is important that the developer has access to the memory space as it can provide very good performance if used correctly. The GPU is designed to access memory like this, and with perhaps added support in the API to use the data in a more general manner, it could be easier to use.

As for the other memory spaces we have found them easy to use, apart from the mentioned challenges in optimising memory accesses. The main problem for the developer is mapping the application to use the right memory space for the right computation. In general the developer should try to limit the use of global memory, and do the com-

putation in shared memory using a right access pattern. As an alternative, the cached memory spaces can be used to read-data if it is limited in size.

8.1.3 Tools to analyse code

As CUDA is mostly used for offloading computation to gain performance and free the CPU of operations, the performance of the application is usually a large issue. In combination with optimising the code, the developer will need to profile and test how the code is running. For easier testing, it is wise to parameterise the application to test different execution configurations to see how the occupancy of the GPU changes, and how much shared memory is used. The NVIDIA visual profiler [46] and occupancy calculator [31] are tools that give useful information on how the code runs. When it comes to tools available for development in CUDA, we agree with Ryoo et al. [51] in their assessment that CUDA is in need for better tools and compilers to allow programmers to experiment with their code to see the performance affects. A scenario where programmers could specify different organisation of their algorithms and tools to detect inefficient access patterns would reduce optimisation efforts considerably. Boyer et al. [47] have developed a tool to detect race conditions and shared memory bank conflicts. This is an example of a very helpful tool that we have seen being developed. In their paper, they mention a prototype to assist in detecting inefficient access patterns on global memory. As good as the visual profiler is to tell about the number of accesses that are coalesced or uncoalesced, it gives no indication of where in the code the mentioned accesses occur.

8.1.4 Debugging

Debugging is something that is done in most application development processes. With CUDA this is no exception. For new CUDA developers there might be the added challenge of working with so many parallel threads and an unfamiliar memory model. The debugging tools available are the emulation mode and the CUDA debugger CUDA-GDB. The emulation mode can find index out of bounds errors, and the same kind of memory access errors seen in x86 applications. It is however, limited in emulating the parallel hierarchy of threads run on the GPU and memory spaces like texture memory. As the threads in the emulation mode are run in sequential order, it cannot generate faults like race conditions. During the development phase of our applications, we hardly ever used the emulation mode as it would in most cases give us no idea on what

was wrong with our code.

The debugger lets the developer switch between execution of the different threads, thread blocks and step through individual warps. We believe that the release of a debugger like CUDA-GDB that is based on GDB is a great step in the right direction to ease debugging of CUDA applications. Unfortunately for us, the debugger first came out of beta recently, leaving us with little time to experiment with the debugger.

8.2 Concurrent CUDA applications on the GPU

We have seen in our investigation of CUDA that concurrent accesses to the GPU reduces performance of applications. This creates a challenge in systems where multiple applications want to use the GPU concurrently. At the moment, CUDA scales well for parallelisation on a single GPU, or if multiple applications use multiple GPUs to avoid fighting for the same resource. As CUDA applications increase in popularity, systems with a single GPU unit may want to use applications offloading processing to the GPU simultaneously.

What is missing?

On CUDA enabled NVIDIA GPUs, applications switch between two different visual computing architectures/modes; graphics and computing processing architecture [11] as shown in figure 8.1. The graphics mode is used in graphical applications and code using programming APIs like OpenGL, while the computation mode is used in CUDA applications. A shader thread dispatch logic in addition to setup and raster units is used in the graphics mode aimed at graphic applications [11], while CUDA uses the computation mode. The GPU is able to run both CUDA applications and graphics rendering by time slicing between the different modes. The modes are available to the low level device driver and can be changed with little overhead through micro-operations to the GPU.

We have talked about concurrency between CUDA applications, but not how CUDA applications can co-operate with typical graphics applications. A possibility for concurrent execution between the two modes would be to allocate a set of SMs for each mode. This would mean less processors available for parallel computation for each application, but it might improve the user experience when the applications running spend less time idling. Our experiences with running applications concurrently using

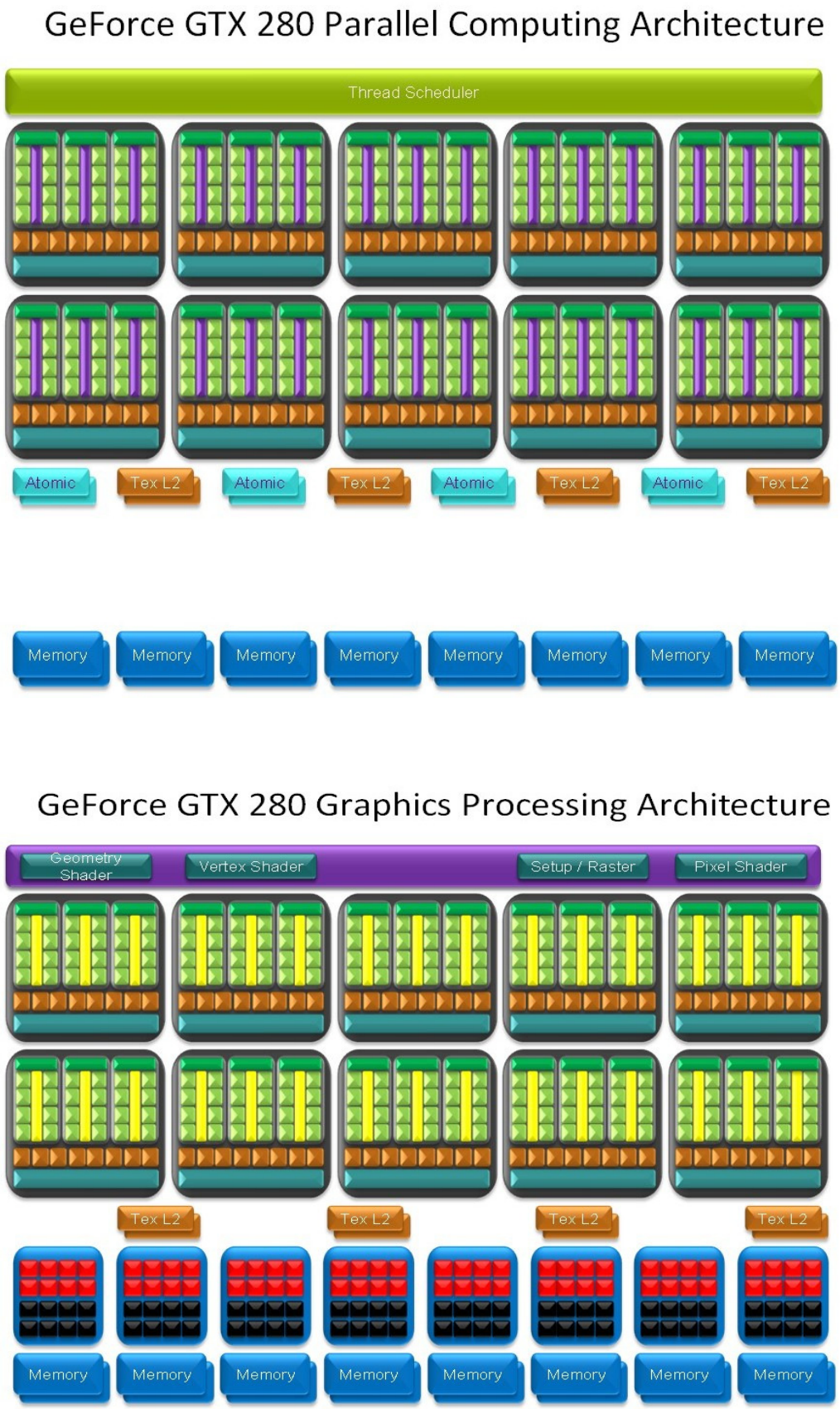


Figure 8.1: *The different architectural personalities of the GPU in CUDA [11].*

the different modes is that the time slicing decreases performance of the applications significantly, and that it could be worth while looking into the possibility of allocating SMs for each mode. This would require a dynamic device driver that was able to allocate SMs when applications needed the resources. The hardware design is also an issue when it comes to concurrent execution of the different architectural modes. As we are not sure about how the compute and graphics architectures are mapped to the GPU hardware, we do not know if it is possible to combine when running in parallel.

In our experiments with static scheduling in chapter 6, we had no control over what SM the thread blocks were assigned to. An alternative to having control of thread block assignment, could be to have a dynamic device driver using a queue system for threads blocks per application accessing the GPU. This would mean multiple applications could request computation concurrently, and the need for serialisation would be removed. By having the device driver dynamically change the number of SMs available for each application according to the number of applications running. If an application is run exclusively it would use all of the SMs, but if two applications are running concurrently, the number of SMs available for each application could be halved. From our experience, it would also be important to map the same application to the same set of SMs, to get the benefit of the cached memory spaces and avoid the cost of switching between applications on the SM.

We are not sure why NVIDIA have opted to not implement this feature, but it is most likely a restriction in the hardware design as the GPU is optimised to perform the same operation in parallel on all of the processing cores. Additionally, there is the added issue regarding the caching mechanisms we have mentioned earlier and in chapter 6, but we do not believe this would be the case if the same workload is assigned to the same SM.

CUDA applications scale well on different GPUs depending on the number of cores available. But for the application to scale over multiple GPUs, the developer has to specify this through the use of contexts. A context is used to create one thread per GPU that performs memory allocation and processing. At present time, there is no way for the developer to specify that the application should run on a GPU that is not occupied. The developer has to specify this through the driver API using `cudaSetDevice`. If no context is specified using the driver API, a context is created and executed on the default GPU of the system. There is, however, no functionality for the developer to specify in an application to use any GPU as long as it is not in use. When not using contexts, the runtime API creates a context for you and this is done on the default GPU of the system. This means that CUDA does not offer dynamic scaling across multiple

GPUs for multiple applications. In our opinion this should be done by the driver when contexts are created, as it is already possible to specify what GPU the context is created on.

Intel's future Larrabee GPU will use a complete different approach for a multi-core GPU. They also propose a more flexible programming model when it comes to scheduling of data and cores. By offering affinity of threads to a particular core, the developer can schedule the workloads like we have wanted to do with our static scheduler in chapter 6. Details regarding the execution and programming model is limited at this time, but we think it looks promising when it comes to low level freedom and core assignment of threads. They have revealed that they will offer a task scheduling API based on a light weight distributed task stealing scheduler by Blumofe et al. [52]. This enables software scheduling so systems can adjust their resource scheduling based on each workload's resource demands.

From the design of the Larrabee architecture, we believe the architecture looks promising from a GPGPU point of view and for developers like us who want to control the hardware resources to a larger extent. It will also assist in a scenario where several applications use the GPU for offloading of processing, as the developer has more control over the scheduling of the applications.

A future CUDA scheduler API, issues that needs considering

For the sake of the discussion, we can assume that CUDA offers low level support for scheduling thread blocks to a dedicated SM in their driver API. In this case, we could have more control over the thread blocks, and override the way thread blocks are assigned to SMs in todays framework. We do not suggest to change the SIMT architecture, as the way warps are created and scheduled are a good way to hide memory latency, but we wish to add the possibility for the developer to group thread blocks to an application. From our experience trying to schedule thread blocks, we have seen certain properties that are important to consider in an eventual low level scheduler. It should be noted that we do not have access to the CUDA source code, and therefore we do not know how todays scheduler works, but our experiments using the framework has made us interested in seeing how the framework can be improved.

First of all it is important that the two different modes/architectures are kept, as the GPU is still used mainly for graphics processing. If this was left as it is today, we can focus on scheduling the applications in the computation mode, as we know little about

how the architectural modes can be combined.

In our static scheduler, we saw that it was beneficial for the performance of the cached memory spaces, that thread blocks from the same application are processed on the same set of SMs. This is similar to how the CUDA driver schedules thread blocks in todays framework when a single application is executed. To be able to map thread blocks to an application, the driver will need to be able to process requests from different applications without serialising the access. In todays framework the thread blocks are queued before processing, we suggest to increase the functionality of the queue mechanism to be able to queue thread blocks from different applications. The developer could access the thread blocks from the queue, by using a context identifier to identify the different queues.

A combination of queues for an application and specific mapping of SMs to applications would make it possible for the scheduler to keep a track of applications requesting the GPU, and what SMs are available. The challenge for queues is to be able to adapt when new resources are requested, and be possible to adapt the SM allocation. We believe this is plausible to achieve, as the GPU changes between a graphics architecture and a computation architecture with the use of micro-operations.

The dynamic ability is also challenging when it comes to memory operations between the CPU and GPU. As different applications will want to transfer at different times. This is a challenge for the scheduler, as it will need to combine the memory transfers efficiently. However, the asynchronous capability of the GPU should assist in achieving this.

Another issue would be the optimisations we have discussed for a single application. If the applications are mapped to one SM, coalesced memory accesses should not be an issue as the requirements are per SM and not the whole GPU. The same goes for bank conflicts on shared memory.

In conclusion, the proposed changes to the API will make sure the developer can access thread blocks from different applications by using a context to address the thread blocks. If the driver makes sure the thread blocks of the context are executed on the same set of SMs, the cost of context switching on the cached memory spaces would be limited. This functionality would most likely not work with existing CUDA applications, as the developer will need to control this in the source code. Unless NVIDIA manage to integrate it in the framework directly.

8.3 Future developments with CUDA

CUDA has proved to be a successful framework, increasing GPGPU popularity as more applications have been implemented to use the GPU for offloading computation. At the same time, NVIDIA are developing their framework to support new features. NVIDIA has just released CUDA 2.2 [53] in beta with a set of new features offered to developers. Among these features is the ability to access system memory directly from the GPU. This can be done to copy data directly into shared memory and avoiding the latency of global memory, referred to as zero-copy. This feature is currently only available on NVIDIA MCP79x [53] chipsets. The MCP79x is a single-chip chipset integrating a GPU directly on the chipset. In addition, there is added support for 64-bit architectures in CUDA-GDB, and an updated visual profiler with added counters. Among the counters added is a memory bandwidth counter, making it easier to determine if an algorithm is memory or computation bound, and to easier see if the size of memory is optimal for use with streams.

We have mentioned in chapter 6, that NVIDIA have planned to offer support for preemption in a future release of CUDA. As of now, there are no details on how this functionality will be implemented, or how it can be used by the user. Context switching on the hundreds of running threads on the GPU is not an option due to the amount of memory it would require to save a full state. As thread blocks are queued up by the driver, it is more likely that the preemption will occur after execution of a set of threads blocks, or through the use of streams. These are the only parts of CUDA as of today that are independent and can overlap in execution. With thread blocks, they are executed concurrently on the GPU within an application.

CUDA has also been considered for other architectures like x86 multi-core CPUs. The MCUDA [54] framework is a working implementation based on CUDA used for mapping CUDA implementations to multi-core CPUs. According to Stratton et al., CUDA is an effective way of specifying data-parallel computation in programming model that is portable across a number of different parallel architectures. MCUDA offers source-to-source translation of CUDA to standard C that interfaces to a runtime library for parallel execution [54]. This shows that CUDA is not only in development for GPU computing, but also as a helpful architecture for efficient parallel execution.

We believe that CUDA has played, and will play, an important role in how the processing power of parallel architectures is used for applications suitable for parallel execution. There are still many issues related to parallel programming, and also a lot of manual labour needed to develop efficient applications. But as the frameworks in-

crease in popularity, we hope it will get easier to write parallel applications. These issues in combination of providing support for concurrent execution, must be resolved before CUDA is a framework suitable for most applications, but for high-performance applications we find it excellent.

Chapter 9

Conclusion

In this chapter, we summarise our work and present our main contributions. Additionally, we look at what improvements can be done in our work, and what we have not been able to test due to time constraints.

9.1 Summary and contribution

In the past years, there has been an increasing interest in the GPGPU field of research, and more applications have started using the GPU for parallel processing to increase the performance. In this thesis, we have investigated the performance potential of offloading processing to the GPU through NVIDIAs CUDA framework. We focus on investigating on how multiple CUDA applications in a system can access the GPU resource concurrently, and how to achieve good performance in a CUDA application running exclusively on the GPU. This was done by both adapting existing projects to the architecture and by writing our own applications from scratch.

During our investigations, we examined GPU architectures and programming frameworks available with focus on NVIDIA and the CUDA framework. We decided to focus on NVIDIAs CUDA framework as we believe that CUDA is the most promising framework due to the quality of the documentation, the large development community and the number of applications developed using the framework.

To experience the challenges connected with developing applications using CUDA, we ported two AES implementations using different approaches. One approach was to use code based on the AES standard [38], focusing on simplicity rather than efficiency. The other approach was to use an optimised version of the algorithm for a

single x86 core using pre-defined lookup tables. The use of lookup tables creates a difference in the characteristics of the implementations. Because of the lookup tables the algorithm becomes memory access bound, while the standard implementation is computation bound. Both implementations performed better on the GPU than the implementations running on the CPU. The memory limited the efficiency of our lookup table implementation on the GPU, due to inefficient access patterns. While the standard implementation experienced a larger increase in performance than the lookup table implementation as it is not memory access bound.

In addition to gaining performance, we learned the importance of optimising memory accesses and the placement of data on the GPU. Each memory space is designed for a specific use, and has different requirements on how to achieve the most efficient access pattern for memory. We have explained how to obtain efficient memory transactions by following the requirements of the different compute capabilities, and seen how the compute capability can affect performance. The results show the importance of adapting the algorithm to suit the right access pattern, and that the cached memory spaces in combination with the on-chip shared memory should be used for computation. Additional optimisations can be done with concurrent execution between the CPU and GPU through CUDA's asynchronous API called streams. By using streams we gained up to an increase in performance in our AES lookup table implementation. In our work with streams, we have shown that for the execution of the CPU and GPU to overlap efficiently, the size of memory being transferred by the CPU needs to correspond to the runtime of the kernel.

As CUDA applications increase in popularity there is a higher probability that applications will issue processing requests to the GPU concurrently. We have investigated how multiple requests for GPU are handled by the device driver in todays framework, and what impact this has on performance. Our findings show that in concurrent executing CUDA processes on the CPU, the processing offloaded to the GPU is exclusive for one application, meaning the accesses are serialised on a first come first serve basis. While the GPU accesses are serialised, the CPU can execute other CUDA applications, but there is no support for preemption or time slicing between CUDA applications when executed on the GPU. When serialised, we see a penalty in performance depending on how many CUDA applications are competing for the resources, and how the CPU parts of the CUDA applications are scheduled by the CPU. In most cases, we see a added runtime of the applications equal to the number of kernels that are executed while the application awaits GPU time.

We tried to limit the performance affects of serialisation by creating a static scheduler

that combines the workload from two kernels into one. The scheduler tries to allocate a set of SMs to the different applications running concurrently on the GPU. Our attempt was unsuccessful due to the limited control the developer has of scheduling workloads to different processing units, but our work gave us knowledge of issues regarding how context switching on the GPU might affect performance.

Our work has shown us the important of optimising code to improve performance of applications, and what challenges developers face when creating or running applications on a GPU. We have gained insight in memory access patterns, how the memory spaces fit applications patterns, what tools are available and how the CUDA framework works when using multiple CUDA applications in a system.

9.2 Future work

During our research, we have looked at a number of issues and challenges. Unfortunately, we did not find time to answer all questions. We would have liked to potentially have increased the throughput of our AES implementations by trying different approaches, and to find further optimisations to investigate. We have spent a lot of time examining optimisations on memory and concurrent execution between the CPU and GPU. However, there are issues regarding optimisations we would have liked to investigated further. Issues that were not tested are the cost of branching and loops in a kernel, which according to the CUDA programming guide are not optimal structures for parallel algorithms [1].

As we believe there is a lack of tools available to help assist in reducing the manual labour of optimising code, we would have liked to develop an application that can assist developers in finding inefficient accesses in global memory, and pinpoint where in the code this occurs.

When investigating how CUDA applications are executed on the CPU when accesses for the GPU are serialised, we did not have time to see how the use of streams would affect this. We mentioned in chapter 8 that we believe that concurrent computation on the GPU might be offered through streams, so it would have been interesting to test how it works in todays framework. Hopefully, CUDA will become more open in the future, as we struggled to have any form for control in assigning workloads to the processing units we wanted.

NVIDIA are continuously adding new features to their framework, with CUDA 2.2

they will offer zero-copy support to the GPU if using certain chipsets. We would have liked to adapt our applications to use this new functionality to see how much of a performance increase is given by avoiding the use global memory. CUDA 2.2 also adds an improved debugger that will benefit for all CUDA developers, as it makes it possible to debug code during runtime.

Finally, we would have liked to test OpenCL to see how our experiences map to the framework when a public compiler and runtime environment is made available. We discuss OpenCL in appendix A, where we focus on how it will affect the research area and CUDA.

Appendix A

Future directions: OpenCL

With the recent release of OpenCL 1.0, many have speculated about the future GPGPU programming frameworks. In this appendix we will present our view on the matter.

NVIDIA is a supporter of OpenCL and have made it clear they do not see CUDA as a competitor to OpenCL. The OpenCL framework provides a low-level hardware abstraction and a framework to support programming of heterogeneous architectures. At the present time there is no public compiler, runtime environment or applications implemented with OpenCL available, so we have not been able to test the framework and therefore base this discussion on the API specification [12].

By looking at the programming model in the API, we can see similarities to CUDA in hardware abstractions like the memory model and the thread hierarchy as pictured in figure A.1. The programmer specifies a thread hierarchy that can be grouped like with CUDA thread blocks, but in OpenCL, they are called work groups with threads called work items. The memory model is also similar where the developer has access to global, constant, local and private memory. Texture memory is not included, but it is a typical GPU specific memory abstraction that does not serve a propose on other architectures. However, it would be beneficial for CUDA developers if they could still use it in OpenCL as in CUDA.

By studying the programming API, we believe that the memory model and thread hierarchy in OpenCL resembles the organisation of CUDA. Therefore, we believe there will be little problem in porting applications from CUDA to OpenCL, if the developer wants a platform independent application that scales well on parallel architectures.

Although, we wonder if there will be a cost in performance due to the hardware abstraction OpenCL uses. As it is not as vendor specific as CUDA, developers may ask

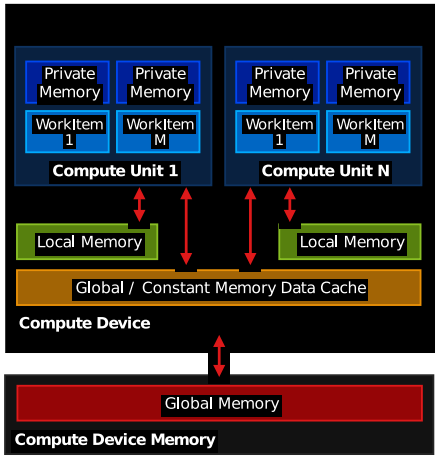


Figure A.1: *Conceptual OpenCL device architecture* [12].

the question if it will be able to obtain the same performance as CUDA code. This is something that will be interesting to look at when a public compiler is released. In the case of NVIDIA and OpenCL, it seems at present time that NVIDIA will compile code to PTX just like CUDA, which is a good start in achieving the same performance as they use the same assembler.

OpenCL will support native kernels for the different architectures, meaning that if a programmer has a very good implementation of an algorithm in CUDA, OpenCL supports the use of native kernels for the NVIDIA architecture. Native kernels are also supported for other architectures like the CBE [55]. This gives the developers a large degree of freedom, as optimised kernels for a specific architecture can be used in combination with a more generic OpenCL kernel that will make sure the code works on the different architectures.

As the same memory abstraction is used on all devices in OpenCL, there is a question on how the memory is physically mapped on the different architectures. Not all the architectures supported use the same memory model as for instance CUDA or CBE. This is the case in for instance the proposed Intel Larrabee architecture [25]. We believe that OpenCL will solve this by mapping the abstracted memory space to an available memory space on the architecture. This might not be as the developer intended, but this is one of the cost of scalability across devices.

In our work with CUDA, we have focused a lot on optimisation. It is not clear how this will work with OpenCL, when the same kernel will be used on the different devices. There are different ways of optimising access patterns on the CBE compared to the GPU. At present time, we do not know how OpenCL will manage this, but if the developer needs to write targeted kernels to reduce performance loss, there will be an

added amount of labour for the developer.

In our opinion, an another important advantage with the development of OpenCL is the added focus on GPGPU development. It will be easier to attract developers if they know that their code can be used on any GPU available, not limiting the application to say an NVIDIA GPU. By the increased number of devices supported, we think there will be more developers interested in using the framework. This also increases the possibility for more debugging tools available, tools for optimisation and of course the added number applications that will be developed. It will also make GPGPU development more future proof in that if a manufacturer suddenly releases a hardware device that can outperform other architectures, it will limit the amount of work needed when porting the application.

Appendix B

Source code and test results

Attached is a CD-ROM containing the source code used, and the benchmark results from our code. The content can also be found at the following address:

`http://www.ping.uio.no/~alexao/master/`

Bibliography

[1] NVIDIA. Nvidia CUDA compute unified device architecture. *http://www.nvidia.com/object/cuda_develop.html*, accessed August 2008.

[2] John D. Owens, David Luebke, Naga Govindaraju, Mark Harris, Jens Krüger, Aaron E. Lefohn, and Timothy J. Purcell. A survey of general-purpose computation on graphics hardware. *Computer Graphics Forum*, 26(1):80–113, 2007.

[3] University of Illinois at Urbana-Champaign. Ece 498 AL1 : Programming Massively Parallel Processors. *http://courses.ece.uiuc.edu/ece498/al1/Syllabus.html*, accessed September 2008.

[4] Mike Murphy. NVIDIA's Experience with Open64. *http://www.capsl.udel.edu/conferences/open64/2008/Papers/101.doc*, Accessed October 2008.

[5] Wikipedia. SubBytes step for AES. *http://en.wikipedia.org/wiki/Image:AES-SubBytes.svg*, accessed October 2008.

[6] Wikipedia. ShifRows step for AES. *http://en.wikipedia.org/wiki/Image:AES-ShiftRows.svg*, accesed October 2008.

[7] Wikipedia. MixColumns step for AES. *http://en.wikipedia.org/wiki/Image:AES-MixColumns.svg*, accessed October 2008.

[8] Wikipedia. AddRoundKey step for AES. *http://en.wikipedia.org/wiki/Image:AES-AddRoundKey.svg*, accessed October 2008.

[9] Wikipedia. Cipher Block Chaining (CBC) mode encryption. *http://en.wikipedia.org/wiki/Block_cipher_modes_of_operation#Cipher-block_chaining_.28CBC.29*, accessed October 2008.

[10] Wikipedia. Counter (CTR) mode encryption. *http://en.wikipedia.org/wiki/Image:Ctr_encryption.png*, accessed October 2008.

[11] NVIDIA. GeForce GTX 280. http://www.nvidia.com/docs/IO/55506/GeForce_GTX_200_GPU_Technical_Brief.pdf, accessed November 2008.

[12] Khronos. OpenCL overview. http://www.khronos.org/opencl, accessed Febuary 2009.

[13] Jackie Neider and Tom Davis. *OpenGL Programming Guide: The Official Guide to Learning OpenGL, Release 1*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1993.

[14] Microsoft. DirectX 10. http://www.microsoft.com/directx, Accessed October 2008.

[15] AMD. AMD Firestream SDK. http://ati.amd.com/technology/streamcomputing/stream-computing.pdf, accessed September 2008.

[16] M.L. Curry, A. Skjellum, H.L. Ward, and R. Brightwell. Accelerating reed-solomon coding in RAID systems with GPUs. *Parallel and Distributed Processing, 2008. IPDPS 2008. IEEE International Symposium on*, pages 1–6, April 2008.

[17] Leonel Sousa Gabriel Falcao and Vitor Silva. Massive parallel LDPC decoding on GPU. In *PPoPP '08: Proceedings of the 13th ACM SIGPLAN Symposium on Principles and practice of parallel programming*, pages 83–90, New York, NY, USA, 2008. ACM.

[18] Folding@Home. Folding@Home distributed computing. http://folding.stanford.edu/, accessed January 2009.

[19] Adobe. Premiere Pro CS3. http://www.adobe.com/products/premiere/, accessed September 2008.

[20] NVIDIA. GeForce3 Technical Briefs. http://www.nvidia.com/page/pg_20010529782175.html, accessed December 2008.

[21] OpenGL. OpenGL Shading Language. http://www.opengl.org/registry/doc/GLSLangSpec.Full.1.20.8.pdf, accessed september 2008.

[22] NVIDIA. NVIDIA - Developer Zone. http://news.developer.nvidia.com/2007/02/cuda_for_gpu_co.html, accessed January 2008.

[23] AMD & ATI. Close to metal open source project. http://sourceforge.net/projects/amdctm/, accessed October 2008.

[24] Stanford University. BrookGPU. http://graphics.stanford.edu/projects/brookgpu/, accessed September 2008.

[25] Larry Seiler, Doug Carmean, Eric Sprangle, Tom Forsyth, Michael Abrash, Pradeep Dubey, Stephen Junkins, Adam Lake, Jeremy Sugerman, Robert Cavin, Roger Espasa, Ed Grochowski, Toni Juan, and Pat Hanrahan. Larrabee: a many-core x86 architecture for visual computing. *ACM Transactions on Graphics*, 27(3):1–15, 2008.

[26] Blaise Barney. POSIX Thread Programming. `https://computing.llnl.gov/tutorials/pthreads/`, accessed November 2008.

[27] NVIDIA. NVIDIA GeForce 8800 GT. `http://www.nvidia.com/object/geforce_8800gt.html`, Accessed November 2008.

[28] Advanced Micro Devices (AMD). AMD FireStream 9170: Industry's First GPU with Double-Precision Floating Point. `http://ati.amd.com/products/streamprocessor/specs.html`, accessed September 2008.

[29] Advanced Micro Devices (AMD). ATI Radeon™ HD 4800 Series - GPU Specifications. `http://ati.amd.com/uk/products/radeonhd4800/specs.html`, accessed September 2008.

[30] Anand Lal Shimpi & Derek Wilson. Intel's larrabee architecture disclosure: A calculated first move. *http://www.anandtech.com/cpuchipsets/intel/showdoc.aspx?i=3367*, Accessed November 2008.

[31] NVIDIA. CUDA GPU Occupancy Calculator. `http://developer.download.nvidia.com/compute/cuda/CUDA_Occupancy_calculator.xls`, accessed September 2008.

[32] NVIDIA. The CUDA compiler driver NVCC. `http://www.nvidia.com/object/io_1213955090354.html`, 2008.

[33] Computer Architecture and Parellel Systems Laboratory (CAPSL). Open64 - The open research compiler. *http://www.open64.net/*, accessed September 2008.

[34] NVIDIA. PTX Parallel Thread Execution ISA version 1.2. *http://www.nvidia.com/object/io_1213955209837.html*, accessed November 2008.

[35] GNU. The GNU Compiler Collection. `http://gcc.gnu.org/`, accessed September 2008.

[36] GNU. GDB: The GNU Project Debugger. *http://sourceware.org/gdb/*, accessed January 2009.

[37] NVIDIA. CUDA-GDB: The NVIDIA CUDA Debugger. *http://developer.download.nvidia.com/compute/cuda/2_1/cudagdb/CUDA_GDB_User_Manual.pdf*, accessed January 2009.

[38] Federal Information Processing Standard FIPS 197. Announcing the ADVANCED ENCRYPTION STANDARD (AES). *National Institute of Standards and Technology (NIST)*, 2001.

[39] Claude E. Shannon. Communication Theory of Secrecy Systems. *Bell System Technical Journal, vol.28-4, page 656–715, 1949*, 1949.

[40] Takeshi Yamanouchi. AES Encryption and Decryption on the GPU. *http://http.developer.nvidia.com/GPUGems3/gpugems3_pref01.html*, accessed October 2008.

[41] Svetlin A. Manavski. CUDA COMPATIBLE GPU AS AN EFFICIENT HARDWARE ACCELOERATOR FOR AES CRYPTOGRAPHY. *IEEE International Conference on Signal Processing and Communications (ICSPC 2007), 24-27 November 2007, Dubai, United Arab Emirates*, 2007.

[42] Niyaz PK. Advanced Encryption Standard (AES) Implementation in C/C++ with comments. *http://www.hoozi.com/Articles/AESEncryption.htm*, accessed October 2008.

[43] Philip J. Erdelsky. Rijndael Encryption Algorithm. *http://www.efgh.com/software/rijndael.htm*, accessed March 2008.

[44] Håvard Espeland. Investigation of parallel programming on heterogeneous multiprocessors. Master's thesis, University of Oslo, Norway, August 2008.

[45] NVIDIA. Geforce 8800. *http://www.nvidia.com/page/geforce_8800.html*, accessed October 2008.

[46] NVIDIA. NVIDIA Visual Profiler. *http://developer.download.nvidia.com/compute/cuda/2.0-Beta2/docs/CudaVisualProfiler_README_1.0_13June08_Linux.pdf*, accessed November 2008.

[47] Westley Weimer Michael Boyer, Kevin Skadron. Automated Dynamic Analysis of CUDA Programs. *Third Workshop on Software Tools for MultiCore Systems (STMCS)*, 2008.

[48] Elemental Technologies. RapiHD - badaboom. *http://www.elementaltechnologies.com/products.php?id=5*, accessed December 2008.

[49] Kenan Rahmani. A proposal for GPU scheduler to optimize large matrix multiplication performance. *http://www.cs.purdue.edu/research/ugrad/docs/f08_Rahmani.pdf*, accessed March 2009.

[50] OpenMP. The OpenMP API specification for parallel programming. *http://openmp.org*, accessed April 2009.

[51] Shane Ryoo, Christopher I. Rodrigues, Sara S. Baghsorkhi, Sam S. Stone, David B. Kirk, and Wen-mei W. Hwu. Optimization principles and application performance evaluation of a multithreaded gpu using cuda. In *PPoPP '08: Proceedings of the 13th ACM SIGPLAN Symposium on Principles and practice of parallel programming*, pages 73–82, New York, NY, USA, 2008. ACM.

[52] Robert D. Blumofe, Christopher F. Joerg, Bradley C. Kuszmaul, Charles E. Leiserson, Keith H. Randall, and Yuli Zhou. Cilk: an efficient multithreaded runtime system. In *PPOPP '95: Proceedings of the fifth ACM SIGPLAN symposium on Principles and practice of parallel programming*, pages 207–216, New York, NY, USA, 1995. ACM.

[53] NVIDIA and Dr. Dobb's portal. CUDA 2.2 beta released. *http://www.ddj.com/linux-open-source/216403490*, accessed April 2009.

[54] John A. Stratton, Sam S. Stone, and Wen-mei W. Hwu. MCUDA: An Efficient Implementation of CUDA Kernels on Multi-cores. *Center for Reliable and High-Performance Computing*, 2008.

[55] IBM. The CELL project at IBM Research. *http://www.research.ibm.com/cell/*, accessed September 2008.