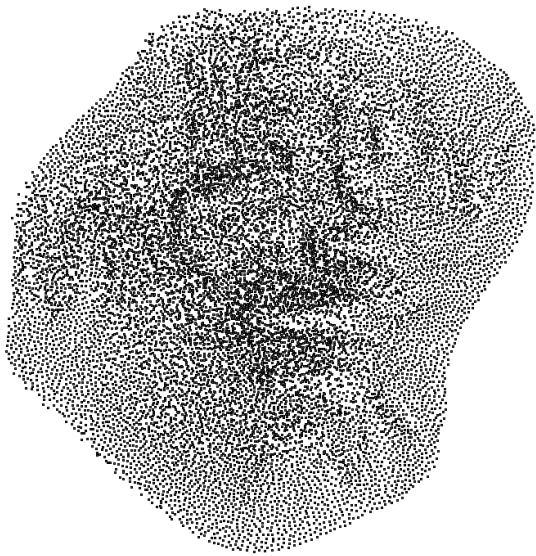
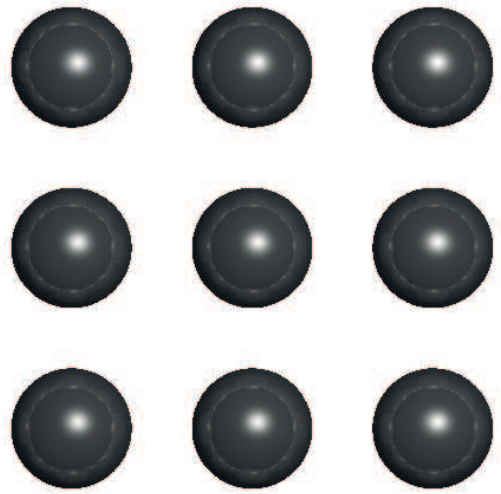


The Art of General-Purpose Computations on Graphics Processing Units



Boids



Ray Tracing

BrookGPU CUDA OpenCL

10th of January 2011
9th semester of
Software Engineering
Aalborg University

Søren Alsbjerg Hørup

Søren Andreas Juul

Henrik Holtegaard Larsen

Title:

The Art of GPGPU

Project period:

September 2nd, 2010 to
January 10th, 2011

Theme:

GPGPU Programming

Group:

d509a
Software Engineering

Authors:

Søren Alsbjerg Hørup
Søren Andreas Juul
Henrik Holtegaard Larsen

Supervisor:

Lone Leth Thomsen

Copies: 6

Total pages: 117

Appendices: 1

Project finished: 10th of January 2011

Aalborg University
Department of Computer Science
Selma Lagerlöfs Vej 300
9220 Aalborg
Telephone: (45)99409940
<http://www.cs.aau.dk>

Abstract:

In this report we document our analysis of General-Purpose computations on Graphics Processing Units (GPGPU) and the practical experience that we have gained throughout the process.

In the analysis chapter, we analyze the differences between GPUs and CPUs describe the architecture of the G80 chip, which is the chip on the Tesla cards available to us. Also we take an in-depth look at three GPGPU programming languages, CUDA, OpenCL and BrookGPU, where OpenCL and CUDA is supported by the the Tesla card, and we find several tools that can help with OpenCL and CUDA development.

In the development chapter, we implement two GPGPU powered applications, namely a ray tracer using CUDA and the Boids flocking algorithm using OpenCL and Brook+. Benchmarks are carried out, which are analyzed and discussed. We find that the GPU is indeed a powerful co-processor, but one must be able to program it correctly against several factors to obtain high performance.

Lastly, we compare the three GPGPU languages, that we found through the analysis, using a number of comparison criteria. Through the comparison we find that CUDA is the most expressive of the three, and is also the most mature, while OpenCL is quickly gaining popularity in the GPGPU field.

The content of this report is freely available, but publication is only permitted with explicit permission from the authors.

Preface

Style Guide

The following style is used throughout this report:

Citations are represented as a pair of angle-brackets containing a number. The number refers to a number in our bibliography. Used as *this section is based on [1]* means that the entire section is based on the mentioned source, unless other sources are explicitly stated.

A citation at the end of a sentence, but right before the full stop, means that the citation is used exactly for that sentence. A citation after a full stop means that the citation is applied to the whole paragraph, i.e. more than one sentence.

Citing a specific page, section and chapter, is done with "p.", "sec." and "chap." respectively, e.g. [1, p. 55], [1, chap. 2], etc.

Prerequisites

The intended readers of this report are students who have passed Software Engineering 8th semester or equivalent. A basic understanding of the C/C++ language is recommended, as many of the APIs discussed are written in C/C++ or a subset of it.

Terms

The following list introduces the most important terms used throughout this report.

GPU refers to Graphics Processing Unit, a specialized microprocessor that accelerates graphics computations.

Concurrency refers to computations done simultaneously, but can be interleaved.

Parallel refers to computations that are done simultaneously and in parallel.

Graphics card refers to a device which has a Graphics Processing Unit (GPU), memory and a host interface.

Discrete card refers to a graphics card which is not integrated into a system, but must be installed into an expansion slot, e.g. PCI, PCI-E, etc.

GPGPU refers to General-Purpose Computing on Graphics Processing Units, i.e. utilizing graphics cards for non-graphical computations.

Device refers to the graphics card containing the GPU.

Device code refers to code which is executed on a device.

Host refers to the host system containing the Central Processing Unit (CPU) and main memory.

Host code refers to code which is executed on the host.

Interactive framerates refers to the number of frames per second required by an application, while still being interactive, i.e. handle input from the users. We define this as at least 30 frames per second.

Rendering refers to the process of creating a 2D image from a 3D scene that can be displayed.

Enclosed CD

On the enclosed CD the source code developed during this project is available along with the benchmark results and this report in PDF format.

Thanks

We want to give thanks to our supervisor Lone Leth Thomsen, for guiding and helping us throughout this project. We would also like to thank Bent Thomsen for setting up the meeting with ConfiCore, Kim Guldstrand Larsen for giving us suggestions to our problem formulation, and Aage Sørensen who helped us acquire the needed hardware.

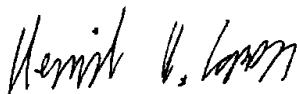
Signatures



.....
Søren Alsbjerg Hørup



.....
Søren Andreas Juul



.....
Henrik Holtegaard Larsen

Contents

1	Introduction	1
1.1	Motivation	1
1.1.1	Other Utilization of GPGPU	2
1.2	Problem Formulation	4
1.2.1	Practical Experience	5
1.2.2	Be Scientific	5
2	Analysis	6
2.1	What Is a GPU	6
2.2	Available Hardware	8
2.2.1	Problems with Tesla C870	8
2.3	G80 Architecture	9
2.3.1	Overview	9
2.3.2	Memory	11
2.3.3	Summary	12
2.4	Algorithms Suited for GPGPU Execution	12
2.4.1	Comparison of Performance between GPUs and CPUs . .	13
2.4.1.1	Discussion	15
2.4.1.2	Our Remarks	16
2.5	Programming GPUs for General Purposes	16
2.5.1	Stream Processing	17
2.6	GPU Languages	18
2.6.1	BrookGPU	18
2.6.1.1	ATI's Brook+	19
2.6.1.2	Example	20
2.6.2	CUDA	20
2.6.2.1	Architecture	21
2.6.2.2	Memory	21
2.6.2.3	Example	22
2.6.2.4	Compute Capability	24
2.6.3	OpenCL	25

2.6.3.1	Architecture	26
2.6.3.2	Framework	27
2.6.3.3	Example	28
2.6.4	Summary	29
2.7	Tool Support	30
2.7.1	CUDA-GDB	30
2.7.2	Nvidia Parallel Nsight	31
2.7.3	Nvidia Compute Visual Profiler	32
2.7.4	gDEBugger CL	32
2.7.5	CUDA Occupancy Calculator	33
2.7.6	Conclusion	33
2.8	Summary	34
3	Development	35
3.1	Boids Application	35
3.1.1	The Boids Model	35
3.1.1.1	Steering	36
3.1.1.2	Heading	37
3.1.1.3	Position	37
3.1.2	Optimizations	37
3.1.2.1	Sorting	39
3.1.2.2	Finding neighbors	40
3.1.2.3	Choice of optimization	42
3.1.3	Functional Requirements	42
3.1.4	OpenCL Implementation	42
3.1.4.1	Sort	43
3.1.4.2	BoidsSimple	43
3.1.4.3	Boids	44
3.1.5	Brook+ Implementation	45
3.1.5.1	Sort	45
3.1.5.2	BoidsSimple	47
3.1.5.3	Boids	47
3.1.6	C++ CPU Implementation	47
3.1.6.1	Sort	47
3.1.6.2	BoidsSimple	47
3.1.6.3	Boids	48
3.1.7	Summary	48
3.2	Ray Tracer Application	48
3.2.1	Ray Tracing Algorithm	48
3.2.2	Functional Requirements	49
3.2.3	CUDA Implementation	50
3.2.3.1	Work Partitioning (R+P)	50
3.2.3.2	Intersection Buffer (R)	50
3.2.3.3	Memory (P)	52
3.2.4	C++ Implementation	52
3.2.5	Summary	53
3.3	Summary	53

4	Benchmarks	54
4.1	Boids Benchmarks	54
4.1.1	Benchmark Setup	54
4.1.2	Initial Experiments	56
4.1.2.1	Results	57
4.1.3	Benchmark Results	58
4.1.4	Discussion	61
4.2	Ray Tracer Benchmarks	61
4.2.1	Benchmark Setup	62
4.2.2	Benchmark Results	64
4.2.3	Discussion	65
5	Comparison	66
5.1	Criteria for Comparison	66
5.1.1	Memory	67
5.1.2	Computation	67
5.1.3	Learnability	68
5.1.4	Concurrency	68
5.1.5	Support	69
5.2	Ratings	70
5.2.1	Memory	70
5.2.1.1	BrookGPU	70
5.2.1.2	CUDA	71
5.2.1.3	OpenCL	71
5.2.2	Computations	72
5.2.2.1	BrookGPU	72
5.2.2.2	CUDA	72
5.2.2.3	OpenCL	73
5.2.3	Learnability	73
5.2.3.1	BrookGPU	73
5.2.3.2	CUDA	74
5.2.3.3	OpenCL	75
5.2.4	Concurrency	76
5.2.4.1	BrookGPU	76
5.2.4.2	CUDA	77
5.2.4.3	OpenCL	78
5.2.5	Support	78
5.2.5.1	BrookGPU	79
5.2.5.2	CUDA	79
5.2.5.3	OpenCL	80
5.3	Summary	81
6	Epilogue	85
6.1	Conclusion	85
6.1.1	Problem Formulation	86
6.1.2	Practical experience	87
6.2	Discussion	88
6.2.1	Implementation Verification	88
6.2.2	Benchmarks	89
6.2.3	Be Scientific	90

6.2.4	Our Thoughts	90
6.3	Future Trends	91
6.3.1	GPGPU Programming is Becoming More Powerful	91
6.3.2	GPGPU Programming is Becoming Easier	91
6.3.3	GPU and CPUs are merging	92
6.3.4	GPGPU is Gaining Momentum	92
6.3.5	Summary	93
6.4	Future Development	93
Bibliography		101
A Nvidia G80		102
A.1	Components Details	102
A.2	Memory	104

Introduction

A GPU is a specialized microprocessor that accelerates graphics rendering, thus allowing interactive graphic-intensive applications on consumer-level computers, which would otherwise require a high performance and much more expensive CPU. [79]

Previous generations of GPUs, prior to 2000, used a fixed-function pipeline, meaning that the functionality of a GPU was fixed and one could only effect the behavior of the pipeline through parameters, e.g. one could affect how vertices are transformed by specifying a transformation matrix. [79, sec. 1.1.3][36, p. 96]

In the start of the 2000s, GPU manufacturers introduced programmable aspects to the GPU pipeline. This allowed more advanced graphical effects, such as bump-mapping, because developers were able to introduce small programs to the graphical pipeline, called "shaders", which could alter the results of the graphical computations. In 2002 the first GPUs that supported features such as, loops, branches, variables and floating point mathematics in pixel shaders were introduced, thus making GPUs more flexible and more like CPUs. [79, sec. 1.1.4][36, p. 98]

GPUs have continued to increase in flexibility and today's GPUs even allow general purpose computations, other than graphical computations, on the GPU. This is commonly referred to as General-purpose computing on graphics processing units (GPGPU) or, and to a lesser degree, GPU Computing. [20]

Examples of GPGPU applications include: neural network that utilizes the GPU to calculate the product of the neurons, computer vision algorithms that utilize the GPU to sample and analyze each separate pixel in real time, Fast Fourier Transform algorithms which is commonly used in encodings and decodings of audio and video, and many more. [36, p. 99]

1.1 Motivation

Our primary reason for looking at GPGPU programming is that we implemented a ray tracer for smartphones and CPUs as a part of our 8th semester project. Ray tracing, however, turned out to be a slow process using only the CPU. We have since seen several publications where different types of computations have

been accelerated using GPUs, we want to investigate whether we can create a faster ray tracer using GPUs. Another advantage of using a GPU over a CPU is that we might be capable of achieving comparable levels of performance using a cheaper systems with GPUs than CPUs.

Performance vs Cost With regards to performance vs cost, we look at the Floating point Operations Per Second (FLOPS) since the computations made in ray tracing is primary done on floating points. When looking at state of the art consumer level graphics cards and CPUs, we see that a Radeon HD 5970 graphics card with two ATI Cypress GPUs gives a theoretical peak double-precision floating-point performance of 925 gigaFLOPS, while an Intel Core i7 980 XE CPU has a max theoretical performance of 107.55 gigaFLOPS. Thus, the HD 5970 has nearly 10 times the computational power. [76]

According to `EDBPriser.dk` at the time of writing, the HD 5970 costs 3985kr (about 200 megaFLOPS/kr) compared to the 980 XE's price of 6860 kr (about 15 megaFLOPS/kr), making the CPU roughly 13 times more expensive with regards to double precision cost efficiency. Of course, theoretical performance should be taken with a grain of salt, since the actual performance also depends upon the implemented algorithm, e.g. algorithms requiring heavy synchronization may be bounded by memory access latency and not FLOPS.

8th Semester's ray tracer Implementation During the 8th semester we implemented a ray tracer [27], running on the CPU, and discovered that the rendering process required a lot of computational resources. We had an hypothesis that the rendering process could be carried out much faster using a GPU, instead of a CPU, because of the theoretical higher processing power of GPUs.

Looking at the literature with regards to GPGPU and ray tracing we see that several publications deal with the problem of creating an efficient GPU based ray tracer [17][63][62]; many of which are successful in achieving high frame rates on consumer level GPUs. Creating an implementation that is accelerated by a GPU requires a lot of work, and the implementation would not be compatible with the smartphone that we had access to at the time, thus requiring us to make two separate code bases which was outside the scope of the 8th semester project; we therefore want to investigate the potential of using GPGPU for ray tracing in this project.

1.1.1 Other Utilization of GPGPU

Looking beyond ray tracing on GPU hardware, we see that other algorithms have been implemented with GPU support with considerable increase in execution speed.

Crowd Simulation Crowd simulation is the process of simulating a larger group of actors, such as humans, animals, particles, etc. and their interaction, which leads to a form of collective behavior [75]. *Boids* is an algorithm that allows crowd simulation with regards to how animals exhibit flocking behavior [67], specifically, the Boids algorithm deals with the simulation of a herd of land

animals, a flock of birds or a school of fish; these behaviors are all captured within the Boids algorithm.

Several parallel CPU implementations of the Boids algorithm exist, such as [11] for multi-core CPUs and [72] for clusters, but some publications have also introduced GPU implementations of the Boids algorithm. [64] is one such publication, where each actor is stored in a two-dimensional data structure, that is partially sorted at each iteration, thereby allowing efficient spatial queries. Two implementations of the algorithm, one targeting the CPU and the other targeting the GPU, is evaluated. The CPU and GPU version is able to simulate one iteration of one million actors in 5394ms and 38ms respectively, on a Core 2 Quad 2.4GHz equipped with a Nvidia 8800 GTS graphics card. Thus, the GPU version is about 140 times faster than the CPU version.

Computer Vision Computer vision in science is the discipline concerned with the theory behind artificial systems that have the ability to extract information from images, such as photographs, video, scanners, etc. thus allowing computers to "see". [6, p xiii]

Several publications deal with computer vision. One such publication [1] from 2008, describes an open-source computer vision and image processing library for GPUs, called GpuCV. GpuCV was designed to be compatible with existing applications that utilizes OpenCV, a CPU based computer vision library, thus allowing transparent utilization of the GPU for existing OpenCV applications.

Benchmarking is done on a Intel Core2 Duo 2.13 Ghz CPU with 2GB of Random Access Memory (RAM) and a 1GB Nvidia GeForce GTX280 graphics card, the images used as input, ranging from 128x128 to 2048x2048 pixels in size. The results show that GPU support is beneficial when high resolution images are used as input, however, at lower resolutions the CPU is in most cases more efficient. At high resolutions however, GpuCV achieves speed ups in the range of 30 to 100 times that of the OpenCV implementation.

Sorting The problem of sorting has also been solved using GPUs, though with smaller factor speed ups compared to the applications presented above.

A publication [68] from 2008 suggests a hybrid sorting algorithm based upon quick-sort and merge-sort. Quick-sort is first used to divide the input list, that requires sorting, into L number of sublists, where all elements of sublist $i + 1$ are higher than the elements of i . L is the number of stream processors on the GPU, 128 for a GeForce 8800 GTS. Afterwards, a modified version of merge-sort is used to sort the L sublists.

This approach achieves nearly twice as fast sorting compared to other GPU sorting algorithms and is over 6 times faster than the Standard Template Library (STL) quick sort CPU implementation provided by Microsoft. Also, when utilizing two GPUs, sorting is 11 times faster compared to STL sort. Benchmarking was carried out using an Intel Core 2 Duo 2.66GHz and two GeForce 8800 GTS 512MB graphics cards. [68, sec. 7.1]

1.2 Problem Formulation

The aim of this semesters project is to gain a deeper understanding of GPGPU related theories and practices, which allows us to implement applications which can utilize the GPU for computational intensive operations. We have devised a problem formulation, presented below.

Our main problem is:

Learn the art of performing General-Purpose Computing on Graphics Processing Units, and the theories that relate to this subject

From the main problem we have derived the following questions we want to answer in this project:

- **How does the hardware architecture of a GPU look like, compared to a CPU?**

In order to efficiently utilize the GPUs, knowledge of the hardware architecture might be required

- *We have two Tesla C870 graphics cards available, as is described in Section 2.2. What does the hardware architecture of a Tesla C870 look like?*

- **What are the characteristics of well suited problems for GPU execution?**

Not all problems, or parts thereof, may be suited for GPU execution. We wish to identify which type of problems are suited, and which are not

- *What are GPUs currently used for other than graphical applications?*
- *What are the pros and cons of GPUs compared to CPUs?*

- **Which GPGPU APIs/Languages exist, and how does one use them?**

To better understand how GPGPU programming is done, it is wise to look at the tools of the trade

- *What are the pros and cons of the different APIs/Languages?*
- *Do any of the languages support high level of abstractions, i.e. are any of the languages high level programming languages?*
- *How can a valid comparison of the different languages be carried out?*

- **Which tools currently exist to help developers with GPGPU programming?**

Tools, such as debugging tools, may make the developer more efficient at implementing programs that make use of GPGPU

- *Which debugging tools are available?*
- *Which testing tools are available, e.g. Unit testing?*
- *What about profiling tools?*

- **Is it possible to utilizing multiple GPUs in parallel for GPGPU purposes? using technologies such as SLI and CrossFire**

Using several GPUs in parallel may increase performance, as more work can be done in parallel

- *Which problems arise when using multiple GPUs?*

1.2.1 Practical Experience

Furthermore we wish to gain practical experiences with GPGPU by implementing a ray tracer and the Boids algorithm on a GPU. The purpose of choosing two algorithms is to cover a broader area of problems. The Boids implementation will focus on how to implement data structures and the problems this may impose.

The ray tracer implementation will use a naive algorithm and instead focus on improving this algorithm by using the theories from the analysis, e.g. which types of memory should be used where.

1.2.2 Be Scientific

We must be scientific in our approach, which means that we must be able to reason about our choices, e.g. of a specific algorithm. Also, the sources we cite must be scientific publications, i.e. peer reviewed published papers, or at least some form of recognized documentation, such as an official specification or company website for a particular product.

In this chapter, we will first cover what a GPU is and how it differs from a CPU, and then look at the GPU hardware we have available and how we got it working. Afterwards, we will analyze the G80 architecture, which is the architecture used by the Tesla C870 GPU, and look at which problems can be solved using GPUs. Lastly, we will look at which programming languages programmers can use to program GPGPUs, and which GPGPU tools are currently available.

2.1 What Is a GPU

GPUs were originally designed for graphics computations and then started changing to allow more and more general types of computations, they have a very different designs than CPUs. In order to develop a program that can run efficiently on a GPU, we have to understand these differences.

The main difference between a CPU and a GPU is that a CPU has a few faster cores, while the GPU has many slower cores.

”If you were plowing a field, which would you rather use?... Two strong oxen or 1024 chickens?” [13]

The reason for CPUs having few but faster cores and GPUs have many slower cores is that they have different goals. The CPU focuses on low latency, i.e. making a single instruction execute as fast as possible. The GPU on the other hand focuses on delivering as high throughput as possible, i.e. the time it takes to complete a single instruction is less important than the time it takes to complete all the instructions of a given task.

The difference can be illustrated by an example of the addition of two vectors each containing 2000 elements. Assuming that we have a CPU that can execute an add instruction in 1ms completing the task would take $1 \times 2000 = 2000$ ms. On the other hand we might have a GPU that takes 100ms to execute one add instruction, but can execute 200 add instructions in parallel thereby only taking $\frac{2000}{200} \times 100 = 1000$ ms to complete the task.

This has led CPU architects to keep the CPU’s Arithmetic Logic Unit (ALU) busy by using large caches, using prefetching to minimize the latency of memory

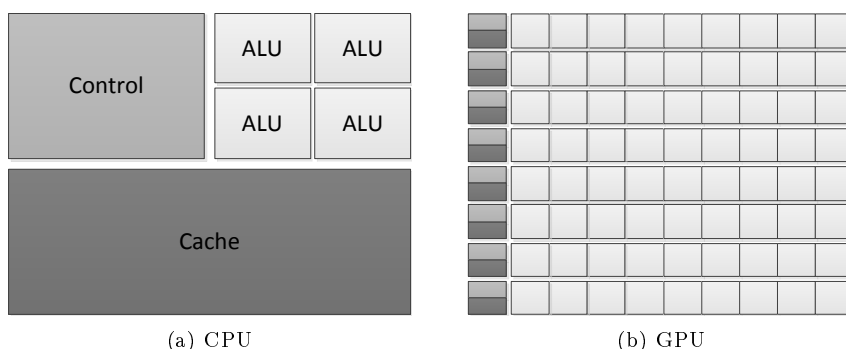


Figure 2.1: Difference between a CPU and a GPU [31]

access, and use complicated control logic such as branch prediction and out of order execution, thus taking advantage of cycles which would otherwise have been wasted due to memory access latency. All of this cache and control logic takes up a relatively large portion of the chip, compared to the space taken by the ALUs. This is shown on Figure 2.1(a).

On the other hand, GPUs are designed as a very parallel architecture with many simple cores, containing very simple control logic and small caches. GPUs also uses Single Instruction, Multiple Data (SIMD) to share a single control unit between multiple ALU [41] as shown in Figure 2.1(b). The use of SIMD means that more ALUs can be fitted on to a chip, but this also means that branching becomes slow. The reason branching becomes slow is that a single control unit can only follow one branch at a time. All ALUs not taking a given branch must wait for the ALU, which took the branch, to exit the branch before executing their own branch. Thus, if we have 8 ALUs per control unit, we have a worst case performance of $1/8^{\text{th}}$ the performance of a program without any branches.

To keep the ALU busy during memory access without the use of advanced control and large caches, the GPU instead rely on latency hiding, that is, having many lightweight threads ready to execute, such that the GPU can start running one of these when another thread is waiting for memory access.

These two design philosophies have led to a big difference in the theoretical peak performance of GPU and CPU. The performance is usually measured in FLOPS, as shown in Figure 2.2. The GPU has about a 10 times more compute power than a CPU and the difference appears to be growing. In recent years, CPU has hit a limit in the increase in sequential execution speed due to power consumption [71]. This has led CPU designers from the uni-processor design to multi-core design, i.e. multiple cores per die, which in turn means that the processing power of CPUs increases every year due to more cores, and not due to increased clock-frequency.

GPUs usually comes on a discrete card with its own RAM. Therefore, it is much easier for hardware manufactures to move to new RAM technologies or make a wider memory busses since it does not have to remain backward compatible with old hardware [31]. This means that GPUs usually have a much higher memory bandwidth than the CPUs. As shown in Figure 2.2, a GPU can have access to RAM that has above 10 times the bandwidth of the RAM that a CPU has access to.

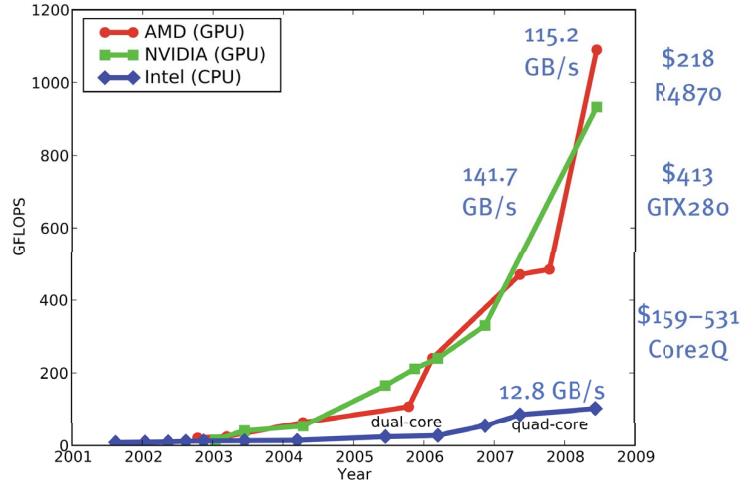


Figure 2.2: Performance comparison of GPU and CPU [31][61]

2.2 Available Hardware

We have a workstation with two Nvidia C870 Tesla cards at our disposal. These cards support Compute Unified Device Architecture (CUDA) of compute capability 1.0, which is the first version of CUDA released, and have a theoretical performance of 518.4 gigaFLOPS with a memory bandwidth of 76.8 GB/sec. Compute capability version defines which CUDA features are supported, e.g. compute capability 1.1 introduces atomic operations on integers in global memory. The compute capability depends on the GPU architecture and can be seen as an abstraction of the features of the architecture. The fact that the Tesla C870 only support 1.0 means that the programs we wish to run on these GPUs, can only use the features available in compute capability 1.0. Also, any optimizations must be based on this compute capability's specific requirements, e.g. memory access requirements. The architecture of the Tesla C870 is analyzed further in Appendix A, but we will give the most important details in Section 2.3.

Beside the Tesla cards, we have two developer computers with Nvidia graphics cards: Quadro FX880M, supporting compute capability 1.2 and a Quadro 140M, supporting compute capability 1.1, and we have a Radeon HD 2900XT from ATI with a theoretical performance of 475 gigaFLOPS with a memory bandwidth of 106 GB/s [23].

2.2.1 Problems with Tesla C870

The two Tesla C870 cards are installed in a LENOVO ThinkStation D10 6427H6G with a Intel Xeon E5420 Quad core CPU, with a theoretical performance of 40 gigaFLOPS [28], and 4GB PC2-5300 RAM running Windows 7 enterprise 64-bit. Since the Tesla cards do not have any display output, and since they take up both of the Peripheral Component Interconnect Express (PCIe) X16 slots on the motherboard, an old ATI Mach64 VT2 Peripheral Component Interconnect (PCI) graphics card was initially installed in the system.

However, after trying different drivers and searching for similar problems on the Internet, we found the installation guide for the Tesla C1060, which states that the Tesla C1060 only works with other Nvidia graphics cards that are compatible with the same drivers as the Tesla C1060, when running Windows.[49] This was not mentioned in any of the official documentation for the Tesla C870 that we were able to find [47][48][46], but since the C1060 is a newer card compared to the C870, we assumed that a Nvidia graphics card is also required with the C870, and thus the root of our problems. After discovering this, we installed a Quadro FX 1400 card which used the same driver as the Tesla cards. This setup worked as intended, but the Quadro FX 1400 took up one of the two available PCIe 16x slots. The Tesla cards also uses PCIe 16x thus only one Tesla C870 could be used.

We therefore acquired a GeForce 8400GS with 512MB RAM that uses the PCIe X1 interface. The system however was unable to register this card when it was inserted. We assume this is due to the motherboard for some reason not supporting PCIe X1 graphics cards even after updating to the newest Basic Input Output System (BIOS) version. At the time of writing, the BIOS version is 2XKT31A. Also, the graphics card was tested in another system where it worked perfectly, i.e. it was not faulty.

Having concluded that the motherboard did not support PCIe x1 graphics cards, we found a GeForce 8400GS with 256MB RAM that uses the older PCI interface. Using this graphics card, we managed to get both Tesla C870 cards running using Windows.

In total this process took us about a month, with work on and off. A lot of time was spend on other tasks, as we were waiting for new hardware.

2.3 G80 Architecture

Because Tesla C870 graphics cards implements the G80 architecture, this sections covers the most important details of the G80 architecture. The architecture is described in much more detail in Appendix A.

Nvidia is somewhat secretive about the details of their chips, therefore the following is based on information from people working for Nvidia found ind [35] and [54] supplemented with details found in [7] and [69].

2.3.1 Overview

In Figure 2.3 we show the high level components of the G80 and how it interacts with the other components outside the GPU. At the top of the Figure 2.3 is the host system, with the host CPU and system memory that can communicate with each other through a bridge, this is usually the north bridge but might also be integrated in the CPU itself.

As shown on Figure 2.3, all communication such as issuing instructions or copying memory between the host system and the GPU, goes through the **host interface**, this is usually the systems PCIe bus. When GPGPU work is assigned the GPU the **compute work distribution** takes care of splitting the work in to smaller groups which are then assigned to one of the 8 **Texture Processor Cluster** (TPC), that performs the actual computations. The TPC usually requires access to the **Dynamic Random Access Memory** (DRAM) on the

graphics card, and can access it through the **interconnection network** on the chip, that connects each of the TPC to each of the 6 DRAM banks on the graphics card. Each of these banks has its own **level 2 cache**.

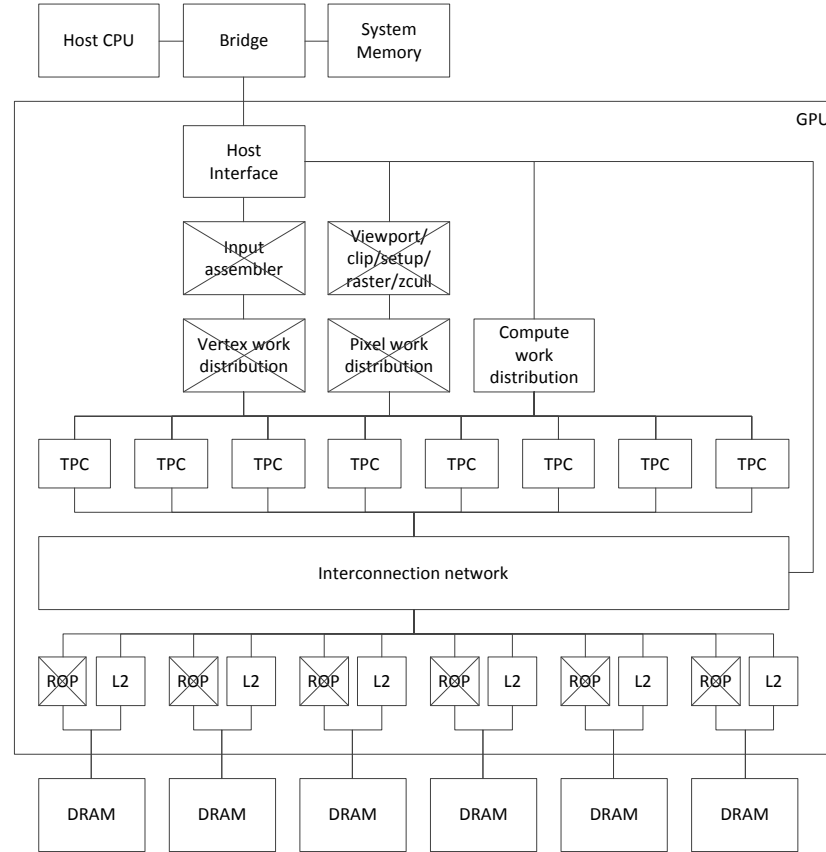


Figure 2.3: The architecture of the G80 GPU. The crossed out components are those that are of no interest to GPGPU. [35]

As shown in Figure 2.4, each TPC has a number of subcomponents, it has a single **SM Controller (SMC)** which controls two **Streaming Multiprocessor (SM)** and a **texture unit**. All communication between the SMs and the **texture unit** goes through the SMC, the SMC also decides which SM can access the **texture unit** at any given time. The **texture unit** is in charge of texture memory access and has a small read only **L1 cache**.

As shown in Figure 2.5 each SM consists of a number of subcomponents. It has an **instruction cache** and a **multithreaded instruction unit** that controls the eight **Streaming Processor (SP)** and two **Special Function Unit (SFU)**. The SP executes simple floating point and integer operations and the SFU performs more advanced functions such as fast approximations of sin and cos. In addition to this, the SM has a read only **constant cache** and a **read/write shared memory**.

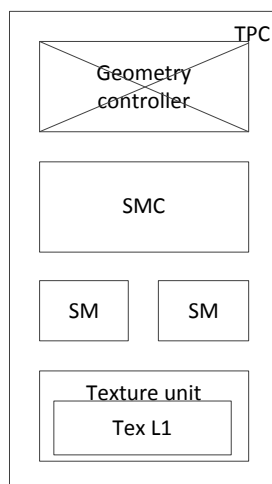


Figure 2.4: Components of the TPC on the G80 the component that are crossed out is of no interest to GPGPU. [35]

2.3.2 Memory

The Tesla C870 features a number of different memory types, where some are located on-chip and others are located off-chip.

- 8192 32 bit registers per SM
- 16KB shared memory per SM
- 8KB constant cache per SM
- 8KB L1 texture cache per TPC
- 1536MB off chip DRAM

These memory types are described in more detail in Section 2.6.2 and in Section A.2.

Memory access to DRAM is much slower than memory access to registers, typically 400 to 800 clock cycles. To avoid stalls when performing computations that require DRAM access, the program should feature many thousands of threads such that it can switch between threads when one of them stalls. This means that the memory access latency is hidden away by having many threads.

As an example, consider a program which performs one DRAM access per 10 instructions that are executed. Each performed instruction takes four clock cycles, and assuming that the DRAM latency is 600 clock cycles, we require 480 threads to hide this latency, since $(480/32) \times 10 \times 4 = 600$, where 32 is the number of threads that goes into a warp. A warp is basically a collection of 32 threads that execute the same program on different data concurrently and is the smallest schedulable unit on the GPU. Warps are explained in more detail in Section A.1.

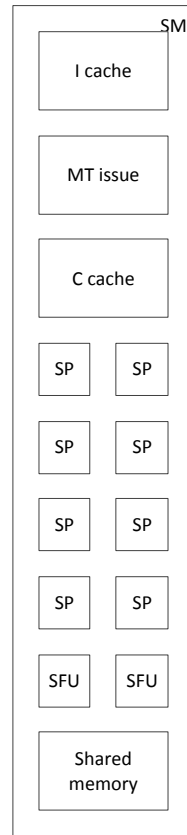


Figure 2.5: Components of the SM on the G80 [35]

2.3.3 Summary

During this section we have analysed and described the G80 chip, e.g. which components it has, which memory types are available and how they are best accessed. This may be useful when optimizing code for this architecture.

2.4 Algorithms Suited for GPGPU Execution

As described in Chapter 1, the primary usage of GPUs are the rendering of real time 3D graphics, such as in games, CAD applications, media players, etc. This means that GPGPU tasks have to exhibit many of the same characteristics as real time 3D graphics to obtain high performance. The aim of this section is to gain an overview of what these characteristics are, and how these characteristics influence the solutions used on the GPU compared to the solution used on the CPU.

Some of the things that characterize real time 3D graphics are:

- The computational requirements are very large. Each frame contains millions of pixels and each pixel requires hundreds of operations to calculate

its color. With multiple frames calculated each second real time 3D graphics requires many billions of calculations each second. [60]

- There are large opportunities for parallelism. Execution of programmable vertices and fragment/pixel shaders can easily be parallelized since they have no side effects. Thus, there are often millions of vertices and fragments in each frame, that can be processed in parallel. [60]
- There are opportunities for instruction stream sharing. Shaders are often executed on vertices and fragments that are close to each other. This locality leads to shaders often following the same control flow and thus are well suited for a SIMD style of execution. [16]
- Throughput is more important than latency. There is a large difference between the speed of the human visual system and the speed of modern processors. The human visual system, i.e. the processing of images in humans through eyes and the brain, works on a millisecond scale while processors work on a nanosecond scale, thus the time individual operations take is unimportant. Therefore the graphics pipeline is often quite deep, hundreds or thousands of cycles. [60]
- Well suited for read only caches that capture spatial locality. Graphics rendering involves few write operations, usual the only write operations performed by the graphics pipeline is outputting the final colors of the pixels. Instead, graphics involves a lot of read operations with a high level of spatial locality, e.g. a shader which applies texturing to one of the pixels of an object reads in data from one or more textures stored in memory, and there are a large chance that texturing neighboring pixels will involve accessing texture data close by. [16]
- Memory bandwidth is more important than memory latency. Graphics are well suited for wide memory busses since memory accesses often exhibit a high degree of spatial locality that allow to coalesce to fewer accesses. Data required by each stage in graphics pipeline are well suited for pre-fetching, thereby reducing the cost of high latency memory access. Additionally, the parallel nature of graphics rendering allows for latency hiding as was described in Section 2.1. [16]

2.4.1 Comparison of Performance between GPUs and CPUs

Several types of problems have been solved with a substantial speed up using GPUs instead of CPUs, as we described in Section 1.1. This has lead to claims such as GPUs being superior to CPU in compute intensive applications. Scientific publications that support this claim are however hard to find, since not many publications deal with comparing CPU performance with GPU performance, other than comparing the theoretical performance or use a unoptimized single threaded version of the application on the CPU.

We have however found a publication [34] from 2010 by the Intel's Throughput Computing Lab and Architecture Group that deals with the issue of comparing the performance of several applications executing on a multi-core CPU and on a Nvidia GPU. [34] tries to void studies claiming 10x - 1000x speed up

of using GPUs for data intensive applications, and concludes based upon benchmarks done on a Core i7 960 CPU and a GTX280 GPU, that the GPU achieves a speed up of only 2.5x on average. This might be due to other researchers GPU performance analysis have neglected CPU optimizations and concentrated on the GPU implementation.

The GPU implementations achieve only 2.5x normalized speed up on average, even though the theoretical performance of the CPU is only 102.4 gigaFLOPS compared to the GPU's 933.1 gigaFLOPS. The 622 of the 933.1 gigaFLOPS are available when using a fused multiply-add operation, and by using a multiply operation on the special function unit. Thus, the GPU is theoretically 9 times faster than the CPU but it cannot be expected that all applications can achieve this speedup.

The benchmarks in [34] are carried out using 14 algorithms implemented and optimized for both platforms. Four of the algorithms are described below, with emphasis on how they perform on the GPU compared to the CPU, and why the performance speed ups are as it is. The four algorithms were chosen because they are known to us. The description of the algorithms and the implementation optimizations are based upon [34].

Radix Sort is a multi-pass algorithm that sorts one digit of the input each pass, from least to most significant digit. Due to each pass induces sorting, i.e. data rearrangement, several memory scatter and gather operations are carried out. The best implementation on the CPU is by utilizing the large cache, thus performing the scatter oriented rearrangements within this cache and thereby avoiding the costly memory operations into main memory.

Since the GPU has a much less cache memory, as described in Section 2.3.1, in the form of shared memory, the Radix sort is rewritten using a technique called split, such that it requires less memory for each pass but with the trade-off of using more scalar operations. Also, since GPU are best suited for read only operations that capture spacial-locality, Radix Sort is not well suited for the GPU because of its use of memory scatter operations, i.e. operations that write to seemingly unordered locations in memory.

This is also evident on the performed benchmark, where Radix Sort performs worse on the GPU than on the CPU, only a factor 0.76 of CPU performance is achieved.

Volumetric Ray Casting is a technique used to visualize 3D datasets such as data gathered from medical CT scanners. In Volumetric Ray Casting, rays are traced through the 3D dataset which in the end generates a pixel based upon the opacity, color, etc. of the dataset.

Tracing several rays in a SIMD like pattern is challenging, since the rays may access incoherent memory, e.g. rays can be scattered in several different directions when tracing through the dataset. This reduces performance, since incoherent memory access on GPUs are generally more expensive compared to CPUs.

Volumetric Ray Casting is however a very compute intensive application, and the GPU implementation achieves a 1.8 speed up compared to the CPU.

Fast Fourier Transform is a standard signal processing algorithm used to convert signals from the time domain to the frequency domain, and vice versa. That is, instead of representing the signal as a signal amplitude change over time, it is instead represented as how much signal lies within each frequency band over a range of frequencies.

Fast Fourier Transform (FFT) improves the Discrete Fourier Transform algorithm from $O(n^2)$ to $O(n * \log n)$ operations, which are mainly composed of multiply-adds floating point arithmetic operations. However, the data access patterns of FFT are non-trivial, making FFT hard to parallelize efficiently in a SIMD like pattern.

However, moving FFT from the CPU to the GPU yields a performance speed up of 3.0. This speed up can be explained by the extensive use of multiply-adds, which the GPU does well.

Collision Detection among convex objects, i.e. any two points within the object can be joined by a line segment that lies within the object, is commonly used in physics based simulators and computer games. A large fraction of the execution-time is spent on vertex computations, such as determining the closest vertex of a given position and direction, making collision detection on convex objects compute bound.

Also, data parallelism can be exploited, because the vertices and edges that make up an object are independent from other objects' vertices and edges, and can therefore be processed in parallel. This maps well to the SIMD architecture of the GPU.

In addition, texture memory can be used to speed up memory reads on the GPU. These optimizations make the GPU perform 14.9 times better than the equivalent CPU implementation and is thus a good fit for GPU execution.

2.4.1.1 Discussion

Looking at the benchmark results from [34] we see that speed ups ranging from 0.5 to 14.9 were achieved on the GTX280 GPU compared to the Core i7-960 CPU, resulting in an average performance increase of a factor 2.5. This performance increase is smaller than what has previously been achieved with GPUs, and with similar benchmark specifications, and the following is a discussion on why this might be the case.

According to [34], one reason that some applications run much better on the CPU is due to the CPUs huge cache, compared to that of the GPU. Recall from Section 2.1, that the GPU relies on latency hiding by having hundreds of threads ready to be executed, however, this is not always possible depending on the application, e.g. due to synchronization constraints, or simply because the application requires too much register- or shared memory, thereby rendering it impossible to execute enough threads concurrently to achieve efficient latency hiding. The CPU does not suffer as much from this problem, since the CPU caches can in many instances hold large portions of the working set and the CPU program is normally not bound as much by register or memory constraints.

Looking at the results from this publication compared to other publications that deal with performance increase using GPUs, we only see one application that achieves above the 10x speed up mark, namely the collision detection algorithm with its 14.9 speed up, while other authors claim 100-1000x speed up.

[34] speculate that the reason for the speed up seen in prior papers, is mainly due to the CPU implementations being non-optimized versions, and in many cases single-threaded applications. Five references are made to publications that perform similar optimizations on CPU code, and the results are that the CPU and GPU implementations are much closer with regards to performance, thereby supporting this view. Also, some of the CPU implementations are executed on low-end CPUs, while their GPU counterparts are executed on high-end GPUs, thereby increasing the performance gap significantly.

Nvidia has given a rebuttal, in a blog post [50], to the results presented above claiming that the [34] is an attempt to promote the CPU, that the code that was run on the GPU was in unoptimized form, and that speed ups from 100x-300x have been seen by hundreds of developers. Also according to Nvidia, another contributing factor is that [34] compares the Core i7 960 (released October 09) with the GTX280 (released June 08), and Nvidia believes that using a newer GPU, with a newer and more advanced architecture, would yield much better performance.

The blog post also cites 10 publications which achieve 100x-300x speed up using GPUs. Looking at the publication claiming the highest speed up, 300x, on a Monte Carlo simulation [15] using a Nvidia 8800GT GPU compared to a Xeon Processor running 1.86GHz, we see that the GPU implementation uses single precision fast math, i.e. faster but less accurate single precision floating point operations, while the CPU implementation uses accurate double precision operations [15, p. 10]. Also, the CPU implementation is a C++ implementation called tMCimg, from 2004, which is apparently not being maintained [18]. In addition, the 300x speed up is only achieved when using non-atomic operations, which due to race conditions, can give incorrect results [15, p. 11]. When using atomic operations and normal floating point math, the speed up around 40x, which is significantly lower. Taking into account that double precision arithmetic on the CPU might be slower than single precision, e.g. due to the increased amount of memory usage, the 40x speed up might be reduced further to 20x speed up.

2.4.1.2 Our Remarks

Based upon our analysis of the results presented in this section, we find that the 300x speed up is questionable at best, but we also agree that even a 10-40x speed up is significant. Also, newer GPUs such as the ones using the Fermi architecture are much better at performing GPGPU than their predecessors, e.g. due to cheaper unconcealed memory access, and especially at performing double precision arithmetics.

According to the information and discussion above, we should not expect that the GPU implementations of the ray tracer and Boids application are able to utilize the Tesla card to its maximum.

2.5 Programming GPUs for General Purposes

With the move from the fixed-function pipeline, to a programmable one, programmers are able to write their own programs that are executed on the GPU in the form of shader programs.

When programming GPUs for general purposes using shader languages, such as the High Level Shader Language (HLSL) or the OpenGL Shading Language (GLSL), one has to transform the problem into the graphical domain before solving it, i.e. one has to represent the general problem as a rendering problem.

Mapping steps frequently include [24, chap. 1]:

1. The copying of input data from the host to a portion of texture memory on the device.
2. Writing a pixel shader that computes some function based upon each pixel in the texture.
3. Setting up an off-screen buffer in which to store the result.
4. Invokes the pixel shader by rendering a 3D primitive to the off-screen buffer.
5. Copies the resulting pixels from the off-screen buffer back to the host.

Although these steps allows GPGPU programming, they do so in a graphical context. Instead, one may utilize higher level languages, such as CUDA, Open Computing Language (OpenCL) and BrookGPU, which completely hides the graphical context, and instead provides the programmer with an abstraction.

2.5.1 Stream Processing

The following is based upon [29].

The Stream Processing paradigm advocates that input data is gathered into so called “Streams”, in which each element can be operated on by one or more “kernels” in parallel. A kernel is a program that takes as input a stream and produces a stream as output with the computed result. A stream is a set of data.

This style of programming is frequently referred to as “gather, operate, and scatter”, since source data is gathered into streams, then operated upon by a kernel containing several operations, and then scattered back into memory.

Stream Processing not only allows a kernel to operate on multiple elements in parallel, due to each element being independent of the other, but also allows efficient utilization of memory because of the producer-consumer nature of Stream Processing. The producers’ input and output data can in many cases be coalesced, which in turn will improve performance since only one transaction is required.

Larger granularity of pipelining can be performed when working on streams, by performing pipelining on the different stages rather than at instruction level. An example of this pipelining could be the Direct3D 9 graphics pipeline as shown in Figure 2.6.

The input to the graphics pipeline is a stream of Vertex Data and Primitive Data, this data is given to the first stage of the pipeline, which is the Tessellation stage where higher-order primitives are converted to vertices, thereby producing a new stream of vertices.

This stream of vertices are then passed as input to the Vertex Processing stage which transforms the vertices, e.g. using a rotation matrix, thereby crating a new stream of rotated vertexes.

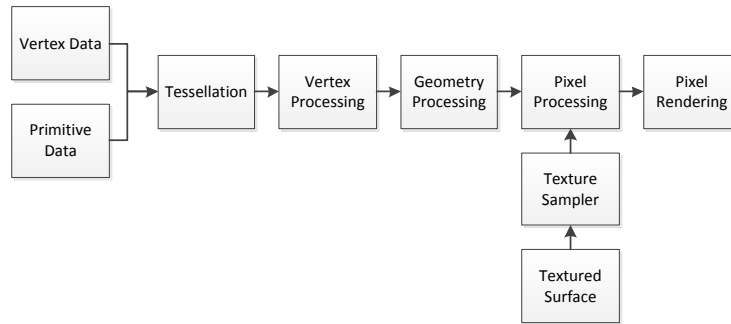


Figure 2.6: The Direct3D 9 Graphics Pipeline [38]

These are passed to the Geometry Processing stage, where they are rasterized thereby producing a stream of fragment; in DirectX terminology fragments are called pixels.

The fragment stream is then passed to the Pixel Processing stage where texturing is performed using the Texture Sampler to read the texture color for a given fragment.

The Pixel Processing stage produces a stream of fragments, which are passed to the Pixel Rendering stage where alpha blending and depth testing is performed, before the final stream of fragments are produced which can be shown on the screen.

2.6 GPU Languages

This section will introduce three GPGPU languages: BrookGPU, which is based upon the stream processing paradigm, CUDA, which is a GPGPU language and architecture developed by Nvidia, and OpenCL, which is an open GPGPU specification very similar to CUDA.

2.6.1 BrookGPU

This section is based on [12].

BrookGPU is a programming language and runtime system that allows developers to utilize the GPU as a streaming co-processor, using the stream processing programming paradigm.

The Brook language extends ANSI C with data-parallel constructs such as *streams* and *kernels*, which allows the developer to define streams of data and write compute kernels, which can be transmitted to and carried out on the GPU.

The BrookGPU architecture consist of The Brook Compiler (BRCC) and the Brook RunTime library (BRT). BRCC takes as input .br files that contain a mix of C and Brook code, and translates these with the help of BRT into C++ source files which are later compiled into an executable. The BRT is a class library that provides a generic and device independent interface for the BRCC, but with a device dependent implementation. [9]

Two well know implementations of BRT exists, one utilizes the Open Graphics Library (OpenGL) 1.3 or newer Application Programming Interface (API)

as backend, while the other utilizes the Direct3D 9 or newer API as backend. All GPUs that support at least one of the two APIs should be compliant with Brook and thus allow developers to utilize such GPUs for GPGPU purposes. [8]

2.6.1.1 ATI's Brook+

Even though Brook supports GPGPU programming using Direct3D and OpenGL backends, it does so through a “graphical” context, although abstracted away by the runtime library, it utilizes a graphics API that is optimized for drawing graphics and not optimized for GPGPU.

Brook+ is ATI's extension to the Brook language that allows developers to specifically write GPGPU code targeted at ATI graphics cards; in Brook+, the BRT implementation targets ATI's Compute Abstraction Layer (CAL). CAL is a device driver for stream processors, specifically ATI GPUs, which allows stream programming closer to the GPU hardware, without using a graphics API. [4, sec. 1.1]

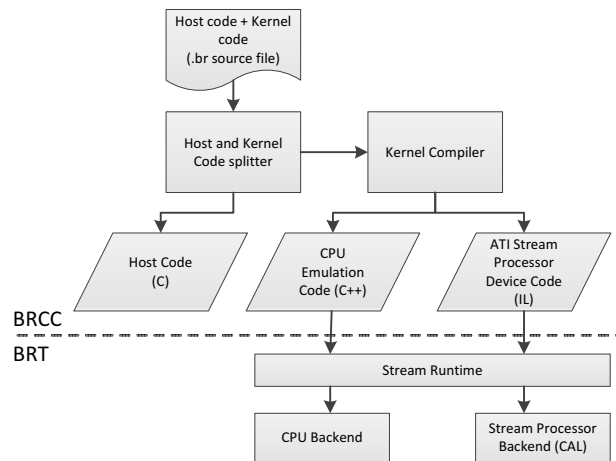


Figure 2.7: The architecture of ATI's Brook+, showing how Brook source code is compiled for use on CAL enabled devices. [4, p. 23]

As depicted on Figure 2.7, BRCC first splits the Brook program into the host code and the kernel code. The kernel code is compiled into either C++ code, which is later compiled to run on the host, or into ATI's Intermediate Language (IL), which is later compiled to run on the device. Obviously, to harness the power of ATI GPUs, one would target IL code; CPU code is used primarily as a reference implementation to ease kernel debugging, since debugging tools for the GPU exist [4, sec. 2.2.4].

IL code is passed to the BRT, where it is later compiled by the CAL compiler for the specific GPU architecture. This allows IL code to run on many different ATI GPUs, as long as the instruction-set is supported by the CAL compiler. Also, IL code and device-specific code can be optimized by hand, thereby potentially improving kernel performance. [4, sec. 3.1.2.2]

2.6.1.2 Example

Codeexample 2.1 shows a simple Brook program which sums two newly initialized matrices together.

```
1  #include <stdio.h>
3  //kernel definition, runs on the GPU
   kernel void sum (float a<>, float b<>, out float c<>)
5  {
   c = a+b;
7  }
   //host code, runs on the CPU
9  int main(int argc, char** argv)
   {
11     int i,j;
        //2D stream declarations of 10x10:
13     float a<10,10>;
        float b<10,10>;
15     float c<10,10>;
        //2D arrays declarations of 10x10
17     float input_a[10][10];
        float input_b[10][10];
19     float output_c[10][10];
        //initialize the 2D arrays
21     for(i=0; i < 10; i++)
        {
23         for(j=0; j < 10; j++)
        {
25             input_a[i][j] = (float)i;
             input_b[i][j] = (float)j;
27         }
        }
29     //copy the 2D arrays to the GPU
        streamRead(a, input_a);
31     streamRead(b, input_b);
        //call the kernel
33     sum(a, b, c);
        //get the result from the GPU
35     streamWrite(c, output_c);
   }
```

Codeexample 2.1: Sum.br: Sums together two matrices, a and b, and stores the result in c. [4]

The sum kernel takes as input, two 2D streams and returns, as output, a 2D stream containing the matrix summation. The operations are done in parallel on the GPU.

The main function allocates three 2D streams and three 2D arrays that make up the matrices. It then initializes the matrices, copies them to the GPU using the streams as references and schedules the sum kernel to the GPU.

Lastly, the result of the computation is copied from the GPU to the host program, by copying the content of the output stream to the output_c array on the host.

2.6.2 CUDA

This section is based on [54].

CUDA is general purpose parallel computing architecture developed by Nvidia to provides a unified way of performing GPGPU on Nvidia GPUs. As such CUDA supports a number of different languages and APIs those officially supported by Nvidia are: CUDA C, OpenCL, DirectCompute and CUDA Fortran. [55, sec. 1.2]. CUDA C and CUDA Fortran are Nvidia's own languages that provides a small number of extensions to the C and Fortran languages, respectively, that enables programmers to use the GPU.

In this section we will focus on CUDA C since C appears to be a much more popular language than Fortran [32]. CUDA comes in a number of different compute capabilities that describes the different features supported in CUDA. In the following, we will assume a compute capability of 1.0, as this is supported by the Tesla cards at our disposal as described in Section 2.2. A further description of the different compute capabilities is given in Section 2.6.2.4.

2.6.2.1 Architecture

CUDA C enables developers to write kernels that run on the GPU. In CUDA C, a kernel is executed by a number of concurrent threads. These threads are organized into a one-dimensional, two-dimensional, or three-dimensional thread blocks. As stated in Section A.1, a thread block is equivalent to Cooperative Thread Array (CAT)s in the G80 architecture. Thread blocks are further organized in a grid of thread blocks, which can be one-dimensional or two-dimensional [55, sec. 2.2].

As a consequence of this, the programmer first need to define the size of the thread blocks and the size of the grid when launching a kernel on the GPU. Each thread executes the same kernel, which defines the work to be done. From the kernel, each thread can be identified by its thread- and block id, allowing it to work on different pieces of data based upon its id. Kernels can only access data stored in memory on the graphics card, which means that the programmer needs to manage all memory on the host and device, and data needs to be copied between the two using host functions provided by CUDA.

If multiple CUDA capable GPUs are present in the system, these can all be utilized. This however requires that the programmer writes an explicit multi GPU capable application, i.e. an application that distributes work between more devices and not only one. CUDA also imposes that each device gets its own CPU thread; this must also be handled by the programmer. [55, sec. 3.2.3]

2.6.2.2 Memory

As shown in Figure 2.8, there are a number of different read and write memories in CUDA C that are shared on the thread, thread block and grid level. These memories have different performance characteristics and must in most cases be managed by the programmer.

Local Each thread can use up to 16KB of memory, called local memory, that is only accessible within the given thread. This memory is relatively slow as it is stored in DRAM on the graphics card and performs as described in Section A.2, i.e. global memory access have latency of 400-600 cycles. Use of local memory is manage by the CUDA C compiler and is primary used for large structures

that would use too much register space, e.g. arrays containing many elements, or if kernel consumes more registers than available on the GPU.

Shared Threads in a thread block have access to 16KB of memory, called shared memory, which is shared between the threads in the block. Shared memory is fast on chip memory as described in Section A.2. Since shared memory is fast, it is often used by programmers as a cache to increase performance when working with data stored in slower memory.

Global All threads both in the same grid and between grids have access to a large shared memory space, called global memory, that resides in DRAM on the graphics card. The size of the global memory is limited by the amount of DRAM on the graphics card. Performance of global memory is affected by the access pattern of programs, e.g. un-coalesced memory access results in much worse performance. This is described in much more detail Section A.2.

Read only In addition to these types of memories, there are also two types of read only memory areas that are shared between all threads.

The first of these is constant memory, which is limited to a size of 64K. Constant memory is optimized for broadcasting a memory read to multiple threads and is cached on chip. The performance is described further in Section A.2.

The second type of read only memory is called texture memory and is cached on chip and optimized for 2D spatial locality [54, sec. 5.3.2.5]. A kernel can at most use 128 texture memories, in addition to this limitation, texture memory is also limited in the maximum number of elements depending on whether it is a 1D, 2D or 3D texture.

A 1D texture can be at most 8192 or 2^{27} elements wide, depending on how it is allocated. A 2D texture can contain 65536×32768 elements and 3D texture can contain $2048 \times 2048 \times 2048$ elements. The performance of texture memory is described in greater detail in Section A.2.

2.6.2.3 Example

Codeexample 2.2 shows an example of CUDA code, which adds two vectors of 1000000 elements. Line 1 to 6 is the kernel that is to run on the GPU. All kernels in CUDA C must be denoted by the `__global__` key word, but otherwise look like a normal C function.

Line 3, the element of the vectors a given thread is to work on is calculated based on the thread id, block ids and block size.

Line 4 and 5, it is checked if the element to work on is within the size of the vectors, if it is, the two elements are added together.

Line 7 to 38 denotes the code which is executed on the host. Line 9 indicates the number of elements in the vectors. Line 10 to 13 allocates host memory for the vectors. Line 14 to 18 fills the vectors with values. Line 19 to 24 allocates device memory for the two input vectors and the output vector on the graphics card. Line 25 and 26 copies the input vectors from the host to the device. Line 27 and 28 calculates the number of thread blocks required in a grid with a given number of threads in a thread block, in this case 256, to process all elements in the vectors. Line 29 launches the kernel with the previously calculated number

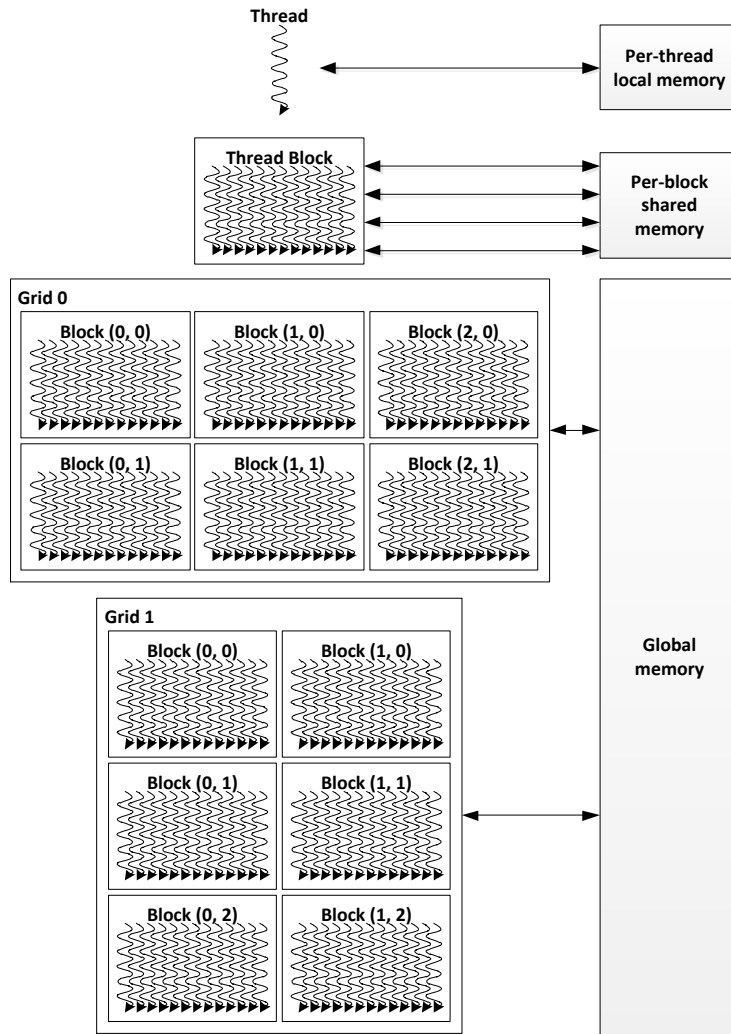


Figure 2.8: Shows the memory access model on CUDA. [54]

of blocks per grid and 256 threads per block. Line 30 copy the result of running the kernel from device memory back to host memory. Line 31 to 36 frees all the allocated memory both on device and host.

```

__global__ void addVector(float* A, float* B, float* C, int
    elements)
2 {
    int i = blockDim.x * blockIdx.x + threadIdx.x;
4     if (i < elements)
        C[i] = A[i] + B[i];
6 }
int main()
8 {
    int elements = 1000000;
10    size_t size = elements * sizeof(float)
    float* host_A = (float*)malloc(size);
12    float* host_B = (float*)malloc(size);

```

```
14   float* host_C = (float*) malloc(size);
    for(int i = 0; i < elements; i++)
    {
16       host_A[i] = i;
       host_B[i] = i*2;
18   }
    float* device_A;
20   cudaMalloc(&device_A, size);
    float* device_B;
22   cudaMalloc(&device_B, size);
    float* device_C;
24   cudaMalloc(&device_C, size);
    cudaMemcpy(device_A, host_A, size, cudaMemcpyHostToDevice);
26   cudaMemcpy(device_B, host_B, size, cudaMemcpyHostToDevice);
    int threadsPerBlock = 256;
28   int blocksPerGrid = (size + threadsPerBlock - 1) / threadsPerBlock;
    addVector<<blocksPerGrid, threadsPerBlock>>(device_A, device_B,
        device_C, elements);
30   cudaMemcpy(host_C, device_C, size, cudaMemcpyDeviceToHost);
    cudaFree(device_A);
32   cudaFree(device_B);
    cudaFree(device_C);
34   free(host_A);
    free(host_B);
36   free(host_C);
    return 0;
38 }
```

Codeexample 2.2: Example code, which sums two vectors together. This is a modified version of the example found on page 22 in [54]

2.6.2.4 Compute Capability

The compute capability of Nvidia GPUs consists of a major and a minor revision number. The major revision number denotes the overall core architecture, while the minor revision number denotes an incremental improvement on the overall core architecture, e.g. such as the addition of a new feature. Compatibility of GPGPU programs written targeting different compute capabilities are guaranteed on the binary level, but only from minor revision number to the next. Compatibility is not guaranteed between major revisions, i.e. compatibility between 1.0 and 1.3 is guaranteed, but 1.3 to 2.0 is not.

Parallel Thread Execution (PTX) code is an intermediate assembly language used by CUDA. At compile time, the CUDA application is compiled to PTX code, which is later compiled at runtime to native code on the device. PTX allows CUDA programs to use higher compute compatibility, than was available when the CUDA program was written, and can therefore insure forward compatibility of CUDA programs. Thus, PTX circumvents the restriction of binary compatibility between major revisions thus allowing better cross-platform support. [55, sec. 3.1.4]

The following is a short list of features introduced after compute capability 1.0, which is the compute capability that the Tesla cards support. For a more in depth look, see [55].

- 1.1 Added the possibility of performing atomic operations on 32-bit integers in global memory, this has among other things been used to implement dynamic memory allocation from within a kernel[26].

- 1.2 Added the possibility of performing atomic operations on 64-bit integers in global memory.
- 1.2 Added the possibility of performing atomic operations on 32-bit integers in shared memory.
- 1.2 Number of registers per SM increased from 8K to 16K often allowing more threads to run concurrently thus allowing for better latency hiding.
- 1.2 Better coalescing of non-sequential and misaligned accesses to DRAM.
- 1.3 Added support for double precision floating point numbers.
- 2.0 Number of registers per SM increased from 16K to 32K often allowing more threads to run concurrently thus allowing for better latency hiding.
- 2.0 Amount of shared memory per SM increased from 16K to 48K.
- 2.0 Part of the shared memory can be used as L1 cache, reducing accesses latency to DRAM.
- 2.0 Added the possibility of performing atomic operations on 32-bit floating points in global and shared memory.
- 2.0 Added the possibility of using function pointers to `_device_` function, i.e. functions that can be called within kernels.
- 2.0 Added support for recursion on `_device_` functions.
- 2.1 Can issue two instruction instead of just one at each instruction issue time.

The list shows that there have been a lot of changes since the release of the first CUDA enabled graphics cards: the GeForce 8800 GTX and GeForce 8800GTS released November 8th 2006 [77]. This relative fast change with six different compute capabilities in just over 4 years shows that CUDA is in a rapid development, constantly adding new features and trying to make it easier, and more powerful, to develop faster running CUDA programs. This however also means that it can be hard to optimize programs since the hardware is constantly changing and programmers have to think about what compute capability they want to target, e.g. if programmers wish to support compute capability 1.0 devices, they must avoid using atomic operations on integers.

Since the Tesla C870 only supports a compute capability of 1.0, we must avoid using features found in newer compute capabilities.

2.6.3 OpenCL

OpenCL is an open standard for parallel programming across CPU's, GPU's and other processors such as Accelerators. The aim of OpenCL is to provide efficient access to different processing platforms, be it CPU's, GPUs, etc. produced by different vendors such as Intel, AMD, Nvidia, etc, while at the same time provide software developers with a highly portable and well specified execution, memory and programming models. [30, sec. 1]

2.6.3.1 Architecture

OpenCL consists of an API, used by the host application to coordinate parallel computations across heterogeneous processors, and a cross-platform programming language based upon ISO C99, which is used to write kernels. In addition, OpenCL provides a runtime system and a compiler, that takes care of things like allocating GPU memory, sending queued work to the GPU, compiling kernels, etc. [30, sec. 3]

The core architecture of OpenCL can be divided into four models: the Platform, Memory, Execution and Programming model.

Platform Model As shown on Figure 2.9, the OpenCL platform model consist of one host connected to one or more compute devices, where each device is composed of one or more Compute Unit (CU)(s). Thus, if the system contains multiple GPUs, multiple devices will be present which can be utilized by the application. A CU contains one or more Processing Element (PE)(s), and each PE functions either as a SIMD unit, thus executing a stream of instructions in lockstep, or as a Single Program, Multiple Data (SPMD) unit, where each PE maintains its own program counter. Also, multiple OpenCL platforms can be provided by different compute device vendors, such as AMD, Nvidia, Intel, etc, and platforms can have different OpenCL versions, e.g. 1.0 or 1.1. [30, sec. 3.1]

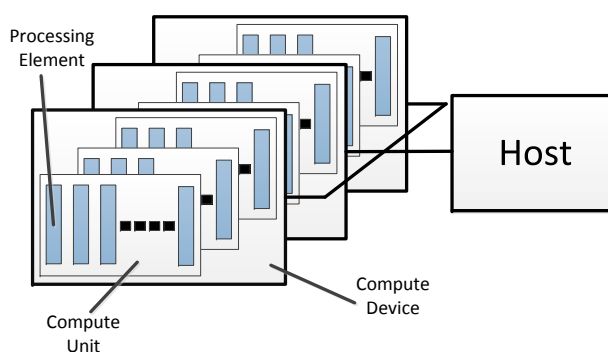


Figure 2.9: The Platform Model of OpenCL. [30, p. 19]

Execution Model OpenCL programs can be divided into two parts, the kernel programs and the host program. Kernels execute on one or more compute devices, such as a GPU or CPU, and are submitted for execution by the host program running on the host processor. Before a kernel is submitted, an N-dimensional index space (NDRange) is defined, which specifies the total number of work-items. A work-item is one instance of a kernel that executes the same code but may follow different execution path and operate on different data elements; thus, work-items are similar to CUDA threads, which were described in Section 2.6.2. Work-items are grouped into work-groups, similar to thread-blocks in CUDA. [30, sec. 3.2]

Memory Model OpenCL defines four distinct memory regions accessible by kernels: Global Memory, Constant Memory, Local Memory and Private Memory. The memory model is very similar to the memory model used by CUDA. Global memory can be read and written to by both host code and kernel code; however, only host code may allocate global memory. Constant memory is read and writable by host code, but read-only by kernels. Local memory, i.e. shared-memory using CUDA terms, is read and writable by all work-items in a work-group, and each work-group has its own local memory which other work-groups, and the host, have no read/write access to. In addition, each work-item has read/write access to its own chunk of private memory. [30, sec. 3.3]

OpenCL employ a relaxed memory consistency model which means that memory state between work-items are not guaranteed, except for local memory between work-items, which is consistent when, and only when, at a barrier, i.e. when all work-items have synchronized. This rule also applies to consistency of global memory within a work-group, however, there are no memory consistency guarantees between work-groups. [30, sec. 3.3.1]

Programming Model OpenCL supports two programming models: the Data Parallel Programming Model and the Task Parallel Programming Model.

The Data Parallel Programming model defines a computation as a sequence of instructions operating on multiple elements in memory. The index space defines the work-items, i.e. how many work-items per dimension, that are executed on the OpenCL device. OpenCL uses a relaxed version of the data parallel programming model, meaning that a strict one-to-one mapping between work-items and the data elements are not required, as is the case with stream processing languages such as BrookGPU, as was described in Section 2.6.1. Also, OpenCL supports implicit and explicit grouping of work-items into work-group, i.e. the programmer can either let OpenCL manage the size of work-groups, or explicitly define the number of work-items in work-groups which might improve performance. [30, sec. 3.4.1]

The Task Parallel Programming Model allows the execution of a single instance of a kernel, independent of any index space, on a CU. This is equivalent to executing a kernel with a work-group containing only one work-item. This programming model can be used to express parallelism using vector data types implemented by the device, enqueue tasks and executing native kernels, such as a CUDA kernel. [30, sec. 3.4.2]

2.6.3.2 Framework

The OpenCL standard also defines an OpenCL framework that is used by application developers to take advantage of OpenCL enabled platforms. The OpenCL framework defines three components: the OpenCL Platform layer, the OpenCL runtime and the OpenCL Compiler. [30, sec. 3.5]

The *OpenCL Platform layer* component is used by the host program to discover OpenCL enabled devices. This component also allows query operations on the devices, to determine their capabilities, and allows the creating of OpenCL contexts; an OpenCL context is used by the OpenCL runtime for managing allocated memory and kernels, among other things [30, sec. 4.3].

The *OpenCL Runtime* allows the host program to enqueue commands for a device, such as the execution of a kernel, initiation of a memory write, initiation

of a memory read, and the allocation and deallocation of memory. A typical usage scenario of the runtime component is: allocate device memory, copy data to the device, execute a kernel, copy back the result and release the allocated device memory. [30, sec. 5.1-5.2]

The *OpenCL Compiler* takes as input OpenCL kernels written in the OpenCL C programming language. The OpenCL C programming language is based upon a subset of C99 with added extensions, such as built-in scalar, vector data types and arithmetic operations on these [30, sec. 6]. The OpenCL compiler allows both online and offline compilation of kernels, i.e. the kernel source is compiled at runtime when needed, or the kernel source can be compiled to object code and afterwards loaded by the host application [30, sec. 5.4].

2.6.3.3 Example

Codeexample 2.3 shows the same matrix summation algorithm implemented in OpenCL; note that the OpenCL kernel code is defined as an inline string in the example, thus using the OpenCL compiler in on-line mode.

```
#include <CL/cl.h>
2 #define MAXPLATFORMS 1
  #define MAXDEVICES 1
4 const char* programSource =
  "__kernel void MatrixSum (__global float* a, __global float* b,
6   __global float* c)"
  "{"
    "unsigned int address = get_global_id(0) + get_global_id(1)*
      get_global_size(0);"
8   "c[address]=a[address]+b[address];"
  "}";
10
12 const int dataSize = 8192;
  float vectorA[dataSize][dataSize];
  float vectorB[dataSize][dataSize];
14 float vectorC[dataSize][dataSize];
  int main(int argc, char **argv)
16 {
    cl_int error;
18    //first get the available platforms, ATI, CUDA, etc.
    cl_platform_id platformIDs[MAXPLATFORMS]; // max one platform
    cl_uint sizePlatformIDs;
20    clGetPlatformIDs(MAXPLATFORMS, platformIDs, &sizePlatformIDs);
    cl_platform_id selectedPlatformID = platformIDs[0];
22    //we assume that we have only one platform
    //get the devices associated with the platform
    cl_device_id devices[MAXDEVICES];
24    cl_uint sizeDevices;
    //get the number of GPU devices:
26    error = clGetDeviceIDs(selectedPlatformID, CL_DEVICE_TYPE_GPU,
    sizeDevices, devices, &sizeDevices);
    //create an OpenCL context and command queue
30    cl_context context = clCreateContext(NULL, sizeDevices, devices,
    NULL, NULL, &error);
    cl_device_id selectedDevice = devices[0];
32    cl_command_queue commandQueue = clCreateCommandQueue(context,
    selectedDevice, 0, &error);
    //compile the OpenCL program:
34    cl_program program = clCreateProgramWithSource(context, 6,
    programSource, NULL, &error);
    error = clBuildProgram(program, 0, 0, 0, 0, 0);
```

```

36 //create a handle to the newly compiled kernel program:
   cl_kernel kernel = clCreateKernel(program, "MatrixSum", &error);
38 //construct data:
   for(int y = 0; y < dataSize; y++)
40 {
       for(int x=0; x < dataSize;x++)
42     {
         vectorA[y][x] = (float)x;
44         vectorB[y][x] = (float)y;
       }
46   }
   //allocate source GPU memory using CPU memory as source, and
   //allocate GPU mem for output
48   cl_mem gpuVectorA = clCreateBuffer(context,CL_MEM_READ_ONLY |
   CL_MEM_COPY_HOST_PTR,sizeof(float)*dataSize*dataSize,vectorA
   ,&error);
   cl_mem gpuVectorB = clCreateBuffer(context,CL_MEM_READ_ONLY |
   CL_MEM_COPY_HOST_PTR,sizeof(float)*dataSize*dataSize,vectorB
   ,&error);
50   cl_mem gpuVectorC = clCreateBuffer(context,CL_MEM_WRITE_ONLY,
   sizeof(float)*dataSize*dataSize,NULL,&error);
   //set up the input arguments for the kernel
52   clSetKernelArg(kernel,0,sizeof(cl_mem),&gpuVectorA);
   clSetKernelArg(kernel,1,sizeof(cl_mem),&gpuVectorB);
54   clSetKernelArg(kernel,2,sizeof(cl_mem),&gpuVectorC);
   //launch the kernel on the GPU
56   int blockSize = 16;
   size_t workSize[2] = {dataSize,dataSize};
58   size_t localSize[2] = {blockSize,blockSize};
   error = clEnqueueNDRangeKernel(commandQueue,kernel,2,NULL,
   workSize,localSize,0,NULL,NULL);
60   //block until finished.
   clEnqueueReadBuffer(commandQueue,gpuVectorC,CL_TRUE,0,sizeof(
   float)*dataSize*dataSize,vectorC,NULL,NULL,NULL);
62   clFinish(commandQueue);
}

```

Codeexample 2.3: Sum.c: Sums together two matrices, a and b, and stores the result in c.

2.6.4 Summary

In this section, we covered BrookGPU, which provide GPGPU abstraction on top of graphics APIs such as DirectX and OpenGL, but also have vendor specific backends, such as ATI Cal.

CUDA was also covered, which introduced a more powerful GPGPU architecture and language extensions for Nvidia GPUs. CUDA provides its own language, CUDA C, which is an extension of C thereby allowing kernel invocations.

Lastly, OpenCL was covered, which has many of the same features that CUDA does. OpenCL is however an open specification, thus allowing much better cross platform support, as several vendors may implement this specification.

Also, we implemented an example in each of the three languages thereby gaining our first GPGPU experience.

2.7 Tool Support

Several tools exist that can help with CPU programming, such as debuggers, Integrated Development Environment (IDE)s, unit testing, performance profilers, etc. These types of tools helps with regards to programming productivity, and we wish to investigate if such tools are available when doing GPGPU programming.

We will primarily look for performance profilers, IDEs and debuggers, because this is the tools that we frequently use when doing programming for the CPU.

Through the investigation, we have found standalone debuggers: CUDA-GDB and gDEDebugger CL, a standalone profiler: Compute Visual Profiler, an extension to Microsoft Visual Studio (VS): Parallel Nsight and a occupancy calculator. These are described in more detail below.

2.7.1 CUDA-GDB

The GNU Project Debugger (GDB) is an open-source debugger that is frequently found available in GNU like operating systems, such as Linux, but also runs on Windows. The debugger supports many programming languages, such as Ada, C, C++, and many more, and it supports different instructions-sets, such as x86 and ARM. The debugger does not come with any GUI and is command-line only. However, several front-ends have been developed; the majority of the front-ends are plugins for existing IDEs, such as Netbeans, Eclipse, VS and more. [70][78]

CUDA-GDB is an extension to the x86/x64 version of GDB, specifically version 6.6. It provide an all-in-one debugging environment, for both host code and CUDA code, on actual hardware in real-time. CUDA-GDB is only supported on devices of compute capability 1.1 or later, meaning that it is not possible to use CUDA-GDB with our Tesla cards, due to them being 1.0. CUDA-GDB is only supported on Linux at the time of writing, though a MacOS preview vesion is available also. [42]

We do however have access to two 1.1 capable GPUs in our developer laptops. Albeit much weaker than the Tesla cards, these allow us to debug our applications before running them on the Tesla powered workstation.

CUDA-GDB provides seamless native hardware debugging, by allowing breakpoints in both kernel and host code, and by allowing fine grained stepping. Thus, when the program is about to execute a kernel, control is handed to the developer who is then able to step through kernel code at warp level, i.e. each time the developer issues a *next* command all 32 threads in the warp are advanced to the next instruction. Breakpoints in kernel code are also supported. [43, p. 5]

In addition to stepping using commands such as *next*, commands such as *print* can be used to display the content of memory on the device. The content of any variable can be shown, including variables that are allocated in GPU memory regions such as shared, local and global memory. Also, special CUDA runtime variables such as *threadId*, that contains the id of the current thread, can be shown. [43, p. 6]

CUDA-GDB allows switching between different devices, SMs and warps, when using multiple devices in an application. This allows the developer to

switch to different devices in real-time, and monitor the state of these. In addition, the developer can switch directly to the different warps thereby allowing breakpoints and monitoring of variables for that particular warp. With the introduction of version 3.0, CUDA-GDB now allows switching between CUDA blocks and CUDA threads, thus giving a better one-to-one mapping between the code and execution, and allows easier debugging. [43, p. 7]

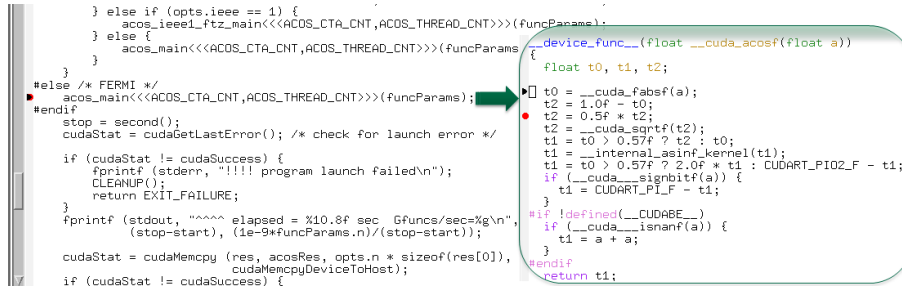


Figure 2.10: Screenshot showing parallel-source debugging through Emacs using CUDA-GDB [51]

As with GDB, CUDA-GDB does not feature its own GUI but allows seamless integrations with several editors and IDEs. Specifically, the Emacs editor has extensive support for CUDA-GDB and allows parallel-source debugging, meaning that one can debug both host and device code in the same window. This is depicted in Figure 2.10, where the program counter reaches the `acos_main` kernel and debugging control is transferred into the kernel source.

Programs being debugged in CUDA-GDB are limited to one GPU, i.e. multi-GPU programs are not supported. In addition, using the debugger on a machine with only one GPU requires that the window manager, such as X11, is not running. This can be circumvented by using two GPUs, i.e. one for the Window system and the other for the application, or using remote desktop tools such as VNC. Lastly, debugging kernels that utilize texture memory is not currently supported. [43, p. 18]

2.7.2 Nvidia Parallel Nsight

This section is based upon [59][58].

Nsight is a plugin for VS that provides a development environment for massively parallel computing targeting both CPUs and GPUs. Nsight comes in two versions, a free standard version and a paid professional version. A trial version of the professional edition is also available.

Both versions seamlessly integrate with the development environment of VS 2008 and 2010, and provide debugging support for CUDA and Microsoft Direct-Compute. As with CUDA-GDB, Nsight allows developers to set breakpoints, examine the memory content of variables and switch between threads. In addition, Nsight allows shader debugging and profiling in DirectX 10 and 11 graphic development, thereby helping graphics developers find bugs and optimize the shader code for better performance. In addition to local debugging, Nsight allows remote debugging of CUDA programs, i.e. the application being debugged

is running on a remote computer.

The professional edition supports CUDA and OpenCL kernel profiling, which can help find bottlenecks in the kernel code and help with optimization. Also, CUDA debugging is extended with data breakpoints, e.g. break whenever variable X is written to, and Tesla Compute Cluster support, i.e. a cluster of computers with Tesla cards.

As with CUDA-GDB, Nsight requires a GPU with at least compute capability of 1.1, thus the Tesla is not supported. Also, two GPUs are required when performing local debugging since the window manager occupies one of the GPUs; this is however not a requirement when performing remote debugging.

2.7.3 Nvidia Compute Visual Profiler

The following is based upon [57] and the documentation accessible through the tool.

Nvidia Compute Visual Profiler is a cross-platform performance profiling tool that gives developers a visual representation of the resource consumption of their GPGPU application, thus allowing identification of potential performance bottlenecks. The profiler supports both CUDA and OpenCL applications, and runs on both Mac, Windows and Linux.

Features including but not limited to:

- Showing the execution time of kernels.
- Showing the amount of register and private memory usage for each thread.
- Showing the ratio of active warps of total warps per SMs. A ratio below 1.0 indicates that not all active warps are scheduled to the SM, which can be caused by the total amount of warps requires too many registers to fit onto a single SM.
- Showing the amount of branches in a kernel, including how many divergent branches were taken. A divergent branch is when a thread in a warp follows a different execution path and can reduce performance, since threads not following a execution path gets disabled.
- Showing the number of texture cache hits or misses.
- Showing the amount of global memory read and writes, and how many were coalesced. Recall that non-coalesced memory access reduces performance.

2.7.4 gDEBugger CL

The following is based upon [66].

While the previous tools do have some limited form of OpenCL support, most of the debugging features are CUDA only. Graphic Remedy aims to rectify this with the upcoming debugger, profiler and memory analysis tool called gDEBugger CL.

gDEBugger CL provides Graphical User Interface (GUI) support for editing kernels, examining OpenCL memory buffers using graphical representation, among other things. gDEBugger CL includes similar features found in Nvidia Parallel Nsight, where some are listed below:

- Locate performance bottlenecks using the built in profiler.
- Edit and continue OpenCL kernels on the fly, i.e. while the OpenCL application is running.
- Set conditional breakpoints on OpenCL errors, breakpoints on function calls, etc.
- Monitor the OpenCL memory consumption and view the content of memory buffers as an image or as raw data.

gDEDebugger CL is set for release ultimo 2010, a free beta version is available but requires that one joins the beta program provided by Graphic Remedy.

2.7.5 CUDA Occupancy Calculator

The following is based upon [44].

CUDA Occupancy Calculator allows developers to compute the occupancy of a given kernel, on a specific compute capable device version, by entering information about the kernel into a Excel document. Occupancy is the ratio of active warps to the maximum number of warps supported by one SM, where an occupancy of 1.0 means that the SM is executing the maximum possible number of concurrent threads.

Given a CUDA kernel program, the CUDA compiler can output the number of registers required by an instance of that kernel program. This register count can be entered into the CUDA Occupancy Calculator, along with how much shared memory per thread-block is used by the program and the block size, i.e. how many threads there are per block, this results in an occupancy ratio telling how efficiently the SMs are utilized.

The tool supports different compute capable devices, from 1.0 to 2.0, and it can draw graphs that show the capacity of these devices. Also, given the information above, the tool can output what the limiting factors are, e.g. limited by register count, shared memory, etc.

2.7.6 Conclusion

Throughout the analysis we have looked at different tools to help with GPGPU based development. We did not find any tools related to testing, such as integration- and unit testing, and most tools that we did find was somewhat limited compared to the ones for CPU based development.

The debuggers that we have found all require two GPUs, one for the Window manager and the other for actual debugging. Also, the debuggers require at least compute capability of 1.1, where the Tesla cards only support 1.0, making it impossible to debug directly on the Tesla powered machine. We do however have access to two developer laptops, that both feature a GPU with compute capability of at least 1.1. We are unable to efficiently utilize these GPUs for debugging, since we have to shut down the Window manager when doing so.

Based upon the information above, we find that the Nvidia Compute Visual Profiler and the CUDA Occupancy Calculator are the most useful. The reason being that the debuggers requires two GPUs and a profiler allows us to profile our GPGPU applications, thus making it easier to reason about the bottlenecks,

and fix these. The Occupancy Calculator allows us to reason about how well our applications utilizes the GPU and how we can increase occupancy by decreasing shared memory or register usage.

2.8 Summary

In Section 2.2 we looked at the GPU hardware that we have available, namely the Tesla C870 graphics cards which supports compute capability 1.0. Getting the Tesla cards working was not as clear cut as we had hoped. We got it working after acquiring a GeForce 8400GS for the older PCI interface.

In Section 2.1 we looked at what a GPU is and how it differs from the CPU. We found that a GPU has many weak cores, compared to the CPU which has few strong cores, and that the GPU is dependent upon latency hiding by interleaving threads waiting for memory with thousands threads waiting to be executed. Also, we looked at a concrete architecture, namely the G80 architecture which supports compute capability 1.0, and found that this architecture is composed of SMs, which are the main component of interests when performing GPGPU and which contains SPs and SFUs, where the SP performs the common arithmetic operations and SFU performs the special functions such as sinus and cosines. Also, we found that the G80 architecture has different memories, where some are located on-chip thus improving performance.

In Section 2.4 we looked at which problems are suited for GPGPU execution. We found that they must be highly parallel and allows for many thousands of threads. Also, the computational requirements must be high, as the GPU is best suited for compute intensive tasks. Additionally, we found a publication from Intel stating that GPUs are not as great for general purposes as many researchers say they are. But even so, they concluded that the particular GPU was on average 2.5x faster than the CPU.

In Section 2.5 we analyzed three programming languages for GPGPU programming. We found that BrookGPU supported different backends, such as DirectX and OpenGL, while also supporting vendor specific backends, such as ATI Cal. We found that CUDA was developed by Nvidia, and that CUDA is only supported on Nvidia GPUs. OpenCL is however an open specification, and can therefore be implemented by many different vendors. Additionally, we found that CUDA and OpenCL's programming model introduced several types of memory types, which must be exploited explicitly by the programmer to increase performance, and that both CUDA and OpenCL supports multiple GPUs.

In Section 2.7 we found several tools which can help with GPGPU programming. Sadly, the debuggers only supports higher compute capable devices, higher than our Tesla's compute capability 1.0. We did however find that the Occupancy Calculator and the Nvidia Compute Visual Profiler both supports compute capability 1.0 and thus we can use these.

To gain GPGPU programming experience, we will implement two applications, the Boids application and the ray tracer application. Also, the applications will be used to assess how easy or how hard it is to do GPGPU programming using the platforms that we have available, i.e. using Brook+, CUDA and OpenCL.

For the first application, we have chosen to implement the Boids algorithm that was briefly introduced in Section 1.1, specifically, the optimization technique presented in the paper [64]. This optimization is interesting because it utilizes a two-dimensional structure that allows efficient spatial queries among actors, and because it has been successfully implemented in CUDA allowing one million actors at interactive frame-rates; the algorithm will be explained in closer detail in Section 3.1. The Boids application will be implemented in OpenCL, Brook+ and C++ targeting x86, such that we can compare the implementations and how they perform.

For the second application, we have chosen to implement a ray tracer similar to the one we developed on 8th semester, because we have experience in doing ray tracing and we know that ray tracing is a parallizable compute intensive application. The ray tracer will be implemented using CUDA. The primary aim of the GPU ray tracer is to improve performance through GPU specific optimizations, such as using shared memory, texture memory, optimizing block sizes, etc.

3.1 Boids Application

The aim of this section is to cover the Boids application. We will first explain the overall Boids algorithm that is the core of this application, including some optimizations. Afterwards, we will cover the functional requirements of this application, i.e. the features, and then cover the GPU and CPU implementations.

3.1.1 The Boids Model

The following is based upon [65].

Boids is a computer model that models the behavior of animal flock motions, such as fish schools or bird flocks. It was devised in 1986 by Craig Reynolds and

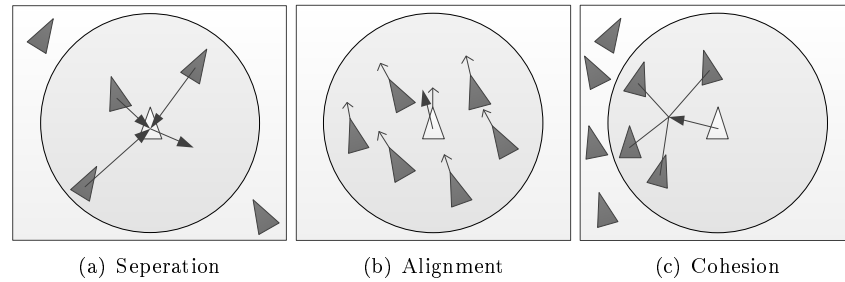


Figure 3.1: The three steering behaviors

introduces three simple steering behaviors, which describe how each individual actor maneuvers, based on the position and direction of its flock-mates.

The three steering behaviors are Separation, Alignment and Cohesion as shown on Figure 3.1. Common to all steering behaviors is that each actor is only influenced by other actors in a certain neighborhood around itself. The neighborhood of an actor is given by a distance, sometimes also referred to as radius, from the center of the actor and the direction of the actor. One way to think about the neighborhood is as a way of limiting the perception of an actor, e.g. if a fish is swimming in murky water it can only see a limited distance and in a limited angle in front of it. For simplicity we will always assume that actors have 360 degrees vision thus we can ignore the direction when taking about neighborhood.

An actor has a position and heading vector. The heading vector denotes the current heading of the actor, i.e. in which direction the actor is traveling and its speed. Each time the actors are updated a steering vector is calculated. The steering vector denotes in which direction the actor is currently steering, i.e. is the actor turning to the left or the right, and how fast is the actor turning. The computation of each these vectors are described below.

3.1.1.1 Steering

The steering vector is calculated based on the three steering behaviors shown in Figure 3.1.

Separation The separation steering behavior influences the actor such that it tries to steer away from local flock-mates, thereby reducing the density of the flock.

The arrow shows the steering direction in which the actor should turn to avoid crowding. The steering direction can be found by computing a vector from the position of each neighbor within the actors neighborhood to the position of the actor. Afterwards, all vectors are divided twice by the distance to the particular flock-mate, and summed into one vector which is the separation vector shown on Figure 3.1(a). The first division finds the unit vector, while the second division makes sure that flock-mates close by have a greater influence on the separation vector, i.e. the actor will avoid local flock-mates more than distant flock-mates.

Alignment The alignment steering behavior influences the actor's heading based upon the heading of other actors within its neighborhood. An alignment vector, shown on Figure 3.1(b) by a thick arrow, determines the steering direction of the actor with relation to the flock-mates' heading. The alignment vector can be computed by taking the average of the other flock-mates' headings, represented by the thin arrows in Figure 3.1(b).

Cohesion The cohesion steering behavior influences the actor such that it tries to stay coherent with the flock-mates within its neighborhood. As shown on Figure 3.1(c), a cohesion vector can be computed based upon the average position of the flock-mates.

Combination The three steering behaviors can be combined by summing the separation, alignment and cohesion vectors; this will produce the flocking behavior. Also, weights can be added to each behavior, which influences how much a single behavior will dominate the overall flocking.

3.1.1.2 Heading

In each iteration of the Boids algorithm the heading vector of each actor is updated by adding the steering vector to the current heading vector of the actor. To limit the acceleration and turning speed of the actors the length of the steering vector can be limited before adding it to the heading, e.g. by normalizing the steering vector. The top speed of actors can be limited by putting an upper limit on the length of the heading vector e.g. by normalizing the heading vector.

3.1.1.3 Position

The position of the actors is updated at each iteration by adding the heading vector of an actor to the current position vector of the actor.

3.1.2 Optimizations

Recall that the behavior of each actor depends upon the position and heading of its flock-mates within a certain neighborhood. The naive implementation of this is $O(n^2)$, where n is the number of actors, because each actor must look at all other actors and determine if each of them is within the neighborhood, and if so, use it to calculate the steering behaviors.

This can be substantially improved if we use a spatial data structure such as a quad tree and index the actors into such a tree; this is shown on Figure 3.2. A quad tree is a two dimensional data structure where each node in the quad tree is subdivided into four quadrants. Thus, each node has four child nodes, one for each quadrant. This allows faster range searches, because only child nodes that are intersected by the search area needs to be checked.

A quad tree however requires that we are able to split and merge nodes when actors change position. This requires some form of dynamic memory management, i.e. splitting a node requires sub-nodes to be allocated by a malloc like function. Dynamic memory allocation is not supported by our GPU [45], utilizing a quad tree therefore requires that we either pre-allocate enough

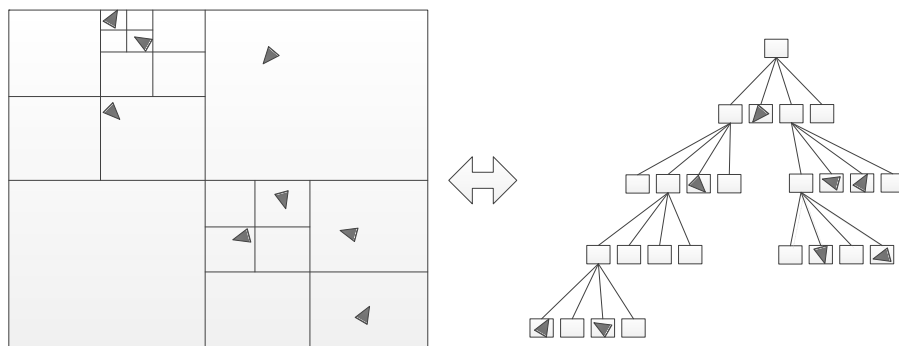


Figure 3.2: A quad-tree containing all actors. The figure on the left shows how the actor positions are contained within quadrants. The figure on the right shows the actual tree representation.

memory beforehand, or that we move the tree update functions from the GPU to the CPU. Pre-allocating enough memory beforehand requires that we implement some form of atomicity on the memory, to avoid race conditions between threads, otherwise multiple threads might try to use the same portion of the pre-allocated memory. Our GPU however does not support atomic operations, and we are therefore stuck with moving the tree-update function to the CPU or serializing the tree updating on the GPU.

Due to these problems, we have looked at another alternative. [64] proposes to use a fine-grain grid to reduce the complexity of finding the actors in a neighborhood in crowding algorithms such as Boids.

The idea is to place each actor into a two-dimensional matrix, or three-dimensional matrix when working in 3D, and sort according to the relative position of each actor. This is shown on Figure 3.3, where each actor on the left is stored into the matrix on the right. The matrix on the right is sorted in such a way that actors appearing from the top-left have lower x,y coordinates than the actors appearing at the bottom-right.

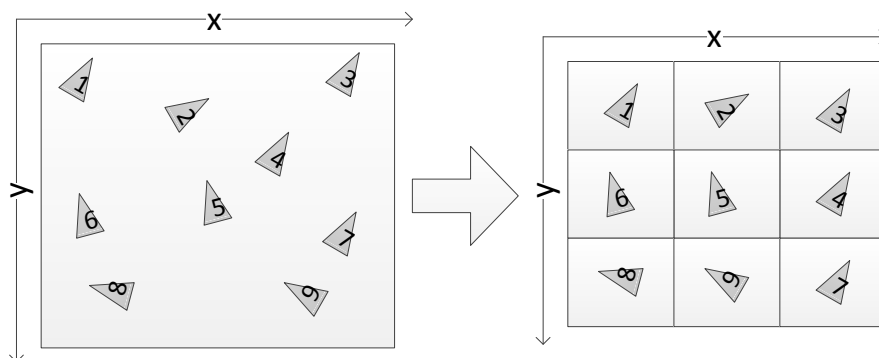


Figure 3.3: The actors in a two dimensional world are mapped to the matrix according to their relative position to each other

Using this approach the problem of finding the actor in a neighborhood can be split in two parts on which involves sorting the actors in the matrix and one that finds the actors in a neighborhood using the sorted matrix.

3.1.2.1 Sorting

Every time an actor moves the matrix has to be sorted, this can in the general case be done in $O(\log(n) * n)$ time. [64] argues that even though actors change position, the matrix is still close to being sorted because of the spatial locality of the actors, i.e. an actor moving will in most cases end up in the cell right next to it.

Therefore [64] suggests to use a partial sorting algorithm, i.e. an algorithm that sorts at most k iterations, thereby producing a matrix which is k iterations closer to being sorted. Thus, according to [64], running a partial sort on the matrix will in most cases produce a sorted matrix. The authors however do not provide any evidence for this claim, other than stating that no artifacts were encountered during their experiments.

The sorting algorithm used to perform the partial sort is an odd-even transposition sort that runs one iteration. The odd-even transposition sort is closely related to bubble-sort. The way the odd-even transposition sort performs the partial sort on the x and y dimensions, is by first comparing an element on an even row with the element on the next row and swapping them if required, and then comparing an element on an odd row with the element on the next row and swapping them if required. When this has been done on all rows the algorithm does the same on columns this is shown on Figure 3.4. Thus a single iteration of the odd-even transposition sort does not guarantee that the resulting matrix is sorted, only that it is closer to being sorted.

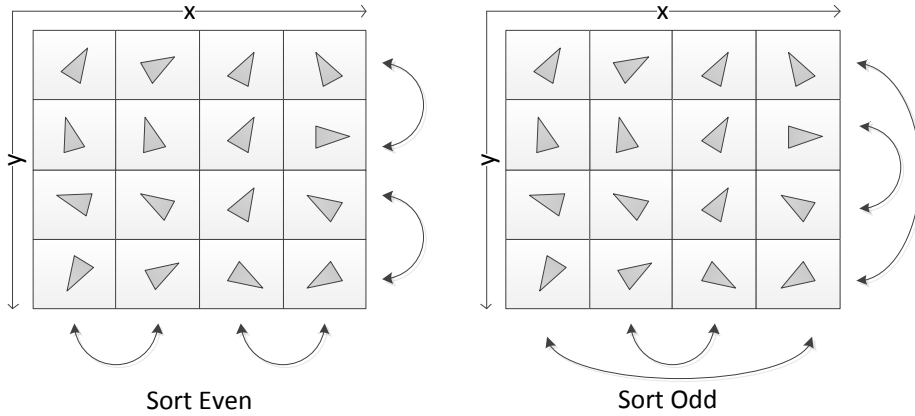


Figure 3.4: The partial sorting algorithm is based upon bubble-sort. The x and y dimensions are sorted by first comparing and swapping even elements, and afterwards odd elements.

Odd-even transposition sort can however still be used to perform a full sort, like bubble-sort, if the odd-even transposition sort is set to only terminate whenever no new swaps are detected in an iteration, this means that only few iterations are required when the matrix is close to being sorted. Using a full sort when the matrix is close to being sorted, should therefore not impact

performance much, since the sorting algorithm would terminate after only a few iterations.

3.1.2.2 Finding neighbors

There are number of different ways of finding out which actors are within a given neighborhood. Bellow we Will describe the approaches we have investigated.

Extended Moore Neighborhood [64] proposes an approach for finding the actors in a neighborhood in $O(1)$ time by using an extended Moore neighborhood instead of using a circle with a given radius used in [65]. Finding all flock-mates within a given square with width w , the algorithm simply returns all actors within $\frac{w}{2}$ cells of the given actor. Thus, if $w = 2$, the 8 immediate actors are returned, if $w = 4$, the immediate 24 actors are returned, see Figure 3.5, and so on. While this approach might prove realistic "enough", i.e. that it gives results indistinguishable from using a radius, it does not follow the Boids model entirely.

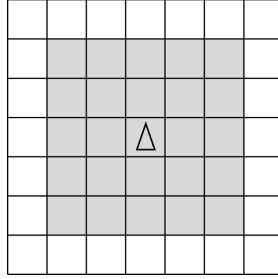


Figure 3.5: Illustrates Extended Moore Neighborhood with $w = 2$

Rings On The Water A solution to finding all the actors within a neighborhood based on distance would be to start by checking all the immediate neighbors of the actor. If one of the actors is within the radius then check the next square of neighbors, and so on, like rings om the water, until non of the actors on the periphery of the square is within the radius.

This, however, will increase the needed computations, as a full square will be tested, even though just one flock-mate was within radius in the previous square.

The worst-case complexity of this is $O(n)$, and happens when the size of the square reaches a size such that all flock-mates are within it, before all actors within the radius is reached. However, the worst-case performance is rarely observed due to the separation behavior of the actors, i.e. actors try to keep some distance to other actors.

To implement this we only need an additional integer of memory to hold the size of the square thus it should not impose any great memory limitations.

Pruned Neighbor Search Finding all flock-mates within the actor's radius can be done by iteratively checking the distance to each flock-mate in neighboring cells, and their neighbors. This is shown on Figure 3.6, where the white

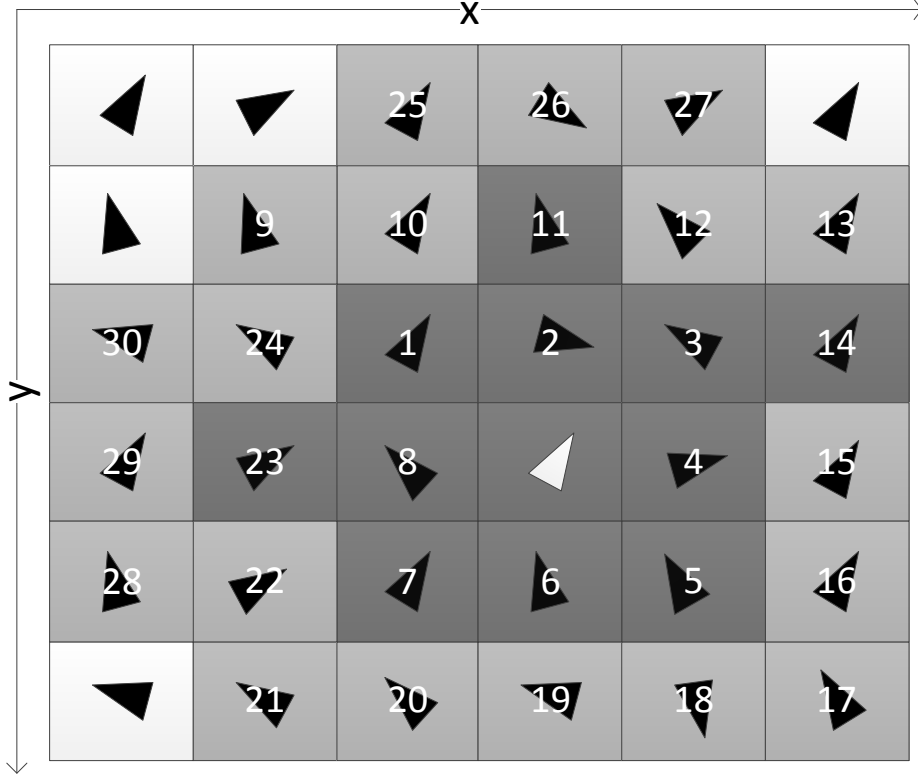


Figure 3.6: The white actor performs a spatial search, thereby finding all flock-mates within its radius

actor performs a search and finds all flock-mates that are within its radius. The numbers in Figure 3.6 indicate the order in which the check to see if an actors is within the neighborhood is performed. The actors that are part of the neighborhood is indicated by a dark background and the actors there are checked is indicated by a lighter background.

The white actor first finds its immediate neighbors, these are checked and deemed within its radius. Afterwards, their neighbors are checked, where three are found to be within the radius of the white actor. The neighbors of these three are then checked, but none are within the radius of the white actor and the algorithm can thus terminate.

The worst-case complexity of this is $O(n)$ in this case but only happens when all flock-mates are within the radius of the actor.

This algorithm requires that a list of actors are generated upon termination, which means that we have to preallocate an n sized array that can store these actors, where n is the maximum number of actors that can be stored. This is however a local list, i.e. no synchronization or atomic operations are required, thus we can simply allocate an n sized array in private memory and we do not have to worry about race conditions.

3.1.2.3 Choice of optimization

Recall that the Quad tree optimization requires that we can dynamically allocate memory, this is not possible on the Tesla C870 GPU. Instead, we choose to implement the algorithm proposed in [64], since they achieve good results; about a million actors at interactive frame rates. We will implement a partial and a full version of the sorting, and the pruned neighbor search algorithm the we propose in Section 3.1.2.2 in addition to the extended Moore neighborhood that [64] utilizes. The reason being that we are able to compare the simpler version of Boids to the more complex version, and see how this performs on the CPU and GPU.

We chose to implement the pruned neighbor search rather than the rings on the water approach as it limits the needed number of computations, and requires a more complex data structure, which is the purpose of this implementation.

3.1.3 Functional Requirements

The functional requirements of the Boids application are as follows:

- All actors exist in 2D space, i.e. each actor has an x and y coordinate
- Must be able to render and display the actors
- Actors must implement Separate, Alignment and Cohesion as defined above.
- In addition, actors must steer towards the mouse position, thus allowing simple interaction between the user and actors
- Must work with at least 1 million actors

In the end, we will have an interactive application that allows several actors flocking around the common aim of getting close to the mouse.

3.1.4 OpenCL Implementation

Recall from Chapter 3 that we have decided to implement the Boids Application using OpenCL and Brook+ for the GPU. This section will cover the implementation details that we find interesting, with regards to the OpenCL implementation. We have chosen to only support a single GPU, since this makes for a better comparison with the Brook+ implementation, due to we only have one GPU with Brook+ support.

The Boids application can be divided into three steps. 1) The sorting of the matrix, i.e. sorting all the actors with regards to their relative position, 2) the actual Boids algorithm, i.e. actors compute their flocking behavior based upon the position and heading of close by flock-mates, and 3) the neighbor search, i.e. finding the flock-mates within a given neighborhood.

We have separated these steps into four kernels: the `SortPartial` kernel, the `Sort` kernel, the `BoidsSimple` kernel and the `Boids` kernel.

3.1.4.1 Sort

The sort kernels takes as input a $N \times N$ matrix containing all n actors, and sorts these in-place using the odd-even transposition sort algorithm introduced in Section 3.1.2.1, i.e. by swapping even and odd rows, and then swapping even and odd columns.

Two versions of the Sort kernel have been implemented, one making a partial sort and one performing a full sort. The two version are nearly identical, except that the partial version always terminates after one iteration thus performing a maximum of $O(n)$ swaps, while the full version terminates only when no new swaps are needed, which in the worst case is $O(n^2)$.

Sorting is carried out by N threads, where $N = 1024$ when $n = 1,048,576$ actors. The N threads performs the element swapping on the N rows and the N columns. To avoid race conditions, a barrier is placed between the swapping of elements row-wise and swapping of elements column-wise. Codeexample 3.1 shows a stripped down version of the Sort kernel. The thread id is used to indicate what row and column that the thread will work on.

```

2  _kernel void Sort(__global struct Actor* actors)
3  {
4      //get the id of the column or row
5      int id = get_global_id(0);
6      bool done = false;
7      while(!done)
8      {
9          done = true;
10         //perform even and odd swapping of rows-wise
11         //elements on row id in the actor matrix
12         if(elements were swapped)
13             done = false; //not in the partial version
14     }
15     barrier(CLK_GLOBAL_MEM_FENCE);
16     done = false;
17     while(!done)
18     {
19         done = true;
20         //perform even and odd swapping of column-wise
21         //elements on column id in the actor matrix
22         if(elements were swapped)
23             done = false; //not in the partial version
24     }
25 }
```

Codeexample 3.1: Stripped down version of the Sort kernel

3.1.4.2 BoidsSimple

The BoidsSimple Kernel takes as input a $N \times N$ matrix of actors, a two dimensional float that represents the location in which the actors flocks to, and returns a new $N \times N$ matrix of actors which holds the new state of the actors.

The BoidsSimple kernel is a simpler version of the Boids algorithm, which does not use a *radius* but instead utilizes a fixed rectangular window in which local flock-mates are found, as described in Section 3.1.2.2.

One instance of the kernel is created for each actor, and each thread carries out the computations for exactly that actor, i.e. separation, alignment and co-

hesion behaviors. Since the actor's velocity and position changes are written to a new matrix, no race conditions are present.

3.1.4.3 Boids

The boids kernel takes the same input and returns the same output as the `BoidsSimple` kernel, and uses the same matrix data structure. The two kernels are thus very similar, except that the `Boids` kernel iterates through a list of flock-mates, which are all within a certain *radius*, instead of looking at a fixed number of flock-mates as the `BoidsSimple` kernel does.

The `Boids` kernel calls a `GetNeighbors` function, which takes as input the matrix containing the actors, a pointer to an integer array which is used as output, and an integer that denotes the maximum size of the integer array to avoid overflows. The `GetNeighbors` function finds all flock-mates who are within the actor's *radius*, unless the maximum is reached. If this maximum is chosen sufficiently large, this should not have a large impact on the result of the algorithm since the maximum will only rarely be reached, and even if the maximum is reached, the actors will still flock with many of its flock-mates. The flock-mates' matrix positions, who are within the radius, are stored in the integer array. Afterwards, the `GetNeighbors` function returns the number of flock-mates that were found.

As described in Section 3.1.2.2, all flock-mates within the *radius* of a given actor can be found by first checking all the immediate neighbors, check their neighbors, and so on. This can be implemented on the CPU by recursively adding flock-mates to a list if they are deemed within the radius of the actor, i.e. the function recursively calls itself on neighboring flock-mates if it is determined that they are within the radius of the given actor.

Recursion is however not supported in the OpenCL C language, meaning that the implementation must be recursion free. The recursion free version uses a *processed* pointer, which is moved whenever the neighbors of a flock-mate has been processed and saves the flock-mates within the *radius* in a list.

At the start of the algorithm *processed* = 0 and the neighboring flock-mates of the actor is added to a list, called *neighbors*, if the neighbors are within the *radius* of the actor. Afterwards, the algorithm iteratively adds the neighbors of the flock-mate located at the *processed* index, to the *neighbors* list if they are deemed within the *radius* of the actor. The *processed* pointer is incremented, and the next flock-mate in the *neighbors* list is processed in the same manner. This is done until the list is filled or all flock-mates in the list have been processed.

Each instance of the `Boids` kernel must call the `GetNeighbors` function, meaning that each thread must dynamically allocate memory to hold the result of the `GetNeighbors` function. Dynamic allocation of memory is currently not possible in OpenCL and we therefore have to preallocate enough memory to hold the result, prior to running the `Boids` kernel. "Enough" memory depends on the input, i.e. one million and one actors can potentially have one million neighbors, which in total is many more actors than we can handle due to memory constraints. For each thread, we preallocate an array in private memory of size 100 that can hold the flock-mates' indexes into the actor matrix. The size of 100 means that an actor can maximum "see" and act upon 100 of its flock mates. This size was chosen arbitrarily. Had we used recursion, we would still have a

limit on how many actors we could investigate due to stack limitations.

The choice of private memory stems from the fact that private memory is fast on-chip memory in OpenCL, compared to CUDA's local memory which is located off chip. However, after further investigation, we found that Nvidia's implementation of OpenCL puts private memory off chip [56] which might impact performance; but since we focus on data structures in this implementation, we have chosen not to optimize the usage of memory. This issue is discussed further in Section 4.1.4.

3.1.5 Brook+ Implementation

Here we will cover the implementation details of the Brook+ implementation. We will primary focus on the differences between the OpenCL implementation and the Brook+ implementation.

3.1.5.1 Sort

The biggest differences between the Brook+ and OpenCL implementations is in the sort kernels. Since the HD 2900XT graphics card does not support scatter operations [2] the sort kernels had to be rewritten to only perform gather operations. Since output streams are write only we have split the sort up in four kernels: One that sort on even rows, one that sort on odd rows, one that sort on even columns and one that sort on odd columns.

As an example, one of the sort kernels that performs partial sorting starting on even rows are given in Codeexample 3.2. First the kernel determines if it is working with an even or odd row. If it is working on an even row a check, to determine if there is a row after the row currently being processed, is performed to ensure that the bounds of the matrix is not exceeded. If the bound of the matrix would be exceeded the actor at the current position is written to the output stream.

If the bounds of the matrix is not exceeded, a comparison of the x-position of the actor at the current row and at the next row is performed. If the actor in the next row have a lower x-position, the two actors are not ordered correctly and the actor in the next row is written to the output stream, to perform the first half of the swap of the two actors. If the actor in the next row have a higher x-position, the actors are ordered correctly and the actor at the current position in the matrix is written to the output stream.

If the kernel is working on an odd row, a comparison between x-position of the actor at the current row and the x-position of the actor at the previous row is performed. If the actor in the previous row has a higher x-position the actors are not ordered correctly, and the actor from the previous row is written to output stream, thus performing the other half of the swap of the two actors. If the actor in the previous row has a lower x-position, the actors are ordered correctly and the actor at the current row is written to the output stream.

```

kernel void xEven(float4 inputActors[][], out float4 outputActors
    <>, int sizeX)
2 {
    int odd = (instance().x) % 2;
4     if(!odd)
    {
6         if(instance().x +1 < sizeX)

```

```
8      {
10          if (inputActors[instance().y][instance().x].x > inputActors[
12              instance().y][instance().x+1].x)
14              {
16                  outputActors = inputActors[instance().y][instance().x+1];
18              }
20          else
22              {
24                  outputActors = inputActors[instance().y][instance().x];
26              }
28          else
30              {
32                  outputActors = inputActors[instance().y][instance().x];
34              }
36      }
```

Codeexample 3.2: Example of a sort kernel in Brook+

To perform a partial sort, first the kernel that starts on even rows is called, followed by the one that starts on odd rows, then the one that starts on even columns and finally the one that starts on odd columns.

To perform a full sort each of the sort steps might need to be called multiple times. To determine if more iterations of sorting is required on either rows or columns an additional output stream is used. The kernels that sorts on even positions writes an 1 to this stream, each time an actor is written to a position that is different from the actors position in the input stream, and a 0 if no change has taken place.

The kernels that work on odd positions also output a stream of 1 and 0 in the same manner as the kernels that works on even positions, except that the odd kernels also takes the stream of ones and zeros produced by the even kernels as input, and produces a 1 if this stream contained a 1. Thus after calling an even kernel followed by the corresponding odd kernel we have a stream that contains a 1 in all the position where either of the kernels swapped an actor and a 0 in all other positions, i.e. if the stream contains only zeros the actors are fully sorted.

To avoid writing the content of the entire stream to host memory and checking each position for a 1 or a 0, to determine if the actors are fully sorted or the sort kernels needs to be called again, an additional kernel is used. This extra kernel is a reduction kernel that calculates the sum of all the values in the stream as shown in Codeexample 3.3 thus only one value needs to be written to host memory and if this value is different from 0 the sort kernels are called again.


```
1 reduce kernel void sum(float a<>, reduce float b)
  {
3   b += a;
  }
```

Codeexample 3.3: Reduction kernel there calculates the sum of all elements in the stream a

3.1.5.2 BoidsSimple

The Brook+ implementation of the BoidsSimple kernel is fairly similar to the OpenCL implementation the biggest difference is that the OpenCL implementation uses a structure of two float2's, a float2 is a two dimensional vector, to represent the position and direction of each actor, since the use of structures caused problems with the brook+ compiler. Instead we used a float4, a float4 is a four dimensional vector, where the x and y dimensions is used to represent the position and the z and w dimension to represent the direction of an actor.

3.1.5.3 Boids

We have chosen to not implement the pruned neighbor search since it would be very time consuming to implement without using scatter operations. Also it would require a lot of kernels to be called by the CPU, thus causing poor performance due to the overhead involved in each kernel call. The reason that many kernel calls are required is that output streams is write only, thus every time a kernel needs to read an actor from the list of actors in the neighborhood, that has been added by the same kernel call, a new kernel call is required.

3.1.6 C++ CPU Implementation

The C++ CPU implementation is very similar to the OpenCL implementation, as the C++ implementation is a back-port of the OpenCL implementation. To gain multi-threading support, we have used the Parallel Patterns Library. This library was made available with the release of Visual Studio 2010 and the C++ 0x standard.

3.1.6.1 Sort

The **Sort** kernel has been ported to C++ from OpenCL and the code is nearly identical. One exception is that the C++ **Sort** implementation was not made multi-threaded, since the patterns library did not appear to feature any barrier construct and thus only utilizes one core. This could however have been avoided by rewriting the **Sort** implementation, but this was not done. One must be aware of this when performing benchmarks.

3.1.6.2 BoidsSimple

This implementation is nearly a one-to-one port of the OpenCL version, with only a few differences. The primitive vector types, such as **float2**, are replaced with the host equivalent types, such as **cl_float2**.

Also, the OpenCL language directly supports arithmetic operations on vector types, e.g. `vector4 a = b + c` is supported, where `b` and `c` are also `vector4` types. This is not supported on the host, we therefore implemented simple helper functions such as the `Add` function, which takes as input two `cl_float2` and returns the vector sum of these.

Lastly, we have used the Parallel Patterns library to make `BoidsSimple` multi threaded, since no barrier constructs were required as was the case with the sorting. This is done by using the `parallel_for` construct, which makes a for loop run in parallel, thus processing the actors in parallel.

3.1.6.3 Boids

The `Boids` kernel has been back-ported in the same manner as the `BoidsSimple` kernel, and is therefore a nearly one-to-one port of the OpenCL version.

3.1.7 Summary

During this section we have described the Boids algorithm and possible optimizations. We have also described how we implemented the algorithm chosen in OpenCL, Brook+ and C++ and which problems was encountered during the implementation.

3.2 Ray Tracer Application

This section gives a short introduction of raytracing, followed by the functional requirements to our GPU Ray Tracer (GRT) and CPU Ray Tracer (CRT) implementation. Afterwards, we will describe the actual CUDA implementation, and shortly cover the CRT implementation.

The purpose of implementing the GRT is experimenting with techniques for optimization on the CUDA platform, specifically for the Tesla C870 as explained in Appendix A, rather than using data structures to achieve better performance.

3.2.1 Ray Tracing Algorithm

This section introduces the basic concepts of ray tracing and is based on the ray tracing analysis found in our 8th semester report [27, sec. 2.5.3].

Ray Tracing is a technique to render 3D geometry, and bases itself upon the way light interacts with objects in the real world. At least one ray is traced through each pixel of the image, known as primary rays. If a ray intersects an object, such as a polygon or a sphere, secondary rays are generated and traced. Three types of secondary rays are commonly found in ray tracer: reflection-, refraction- and shadow-rays.

Reflection rays are, as the name indicates, used to simulate reflection found on reflective materials such as a mirror or water.

Refraction rays are used to simulate the type of refraction commonly found when a photon passes through objects made of materials such as glass or water.

Shadow rays are traced between object intersections and light sources. If the Shadow ray intersects an object before reaching the light source, the light source is not applied.

All rays are illustrated in Figure 3.7.

Furthermore, a shading technique can be applied to the objects in the scene using Ray Tracing. Like the implementation in our 8th semester project, we will use Phong shading, which combines ambient, diffuse and specular lighting, as shown on Figure 3.8. [27, sec. 2.5.1]

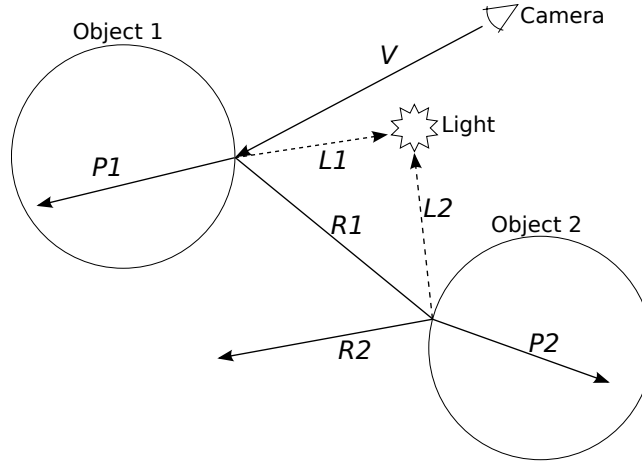


Figure 3.7: Shows the primary ray V and secondary rays: $R1, R2, P1, P2, L1, L2$. $L1$ and $L2$ are shadow rays. $P1$ and $P2$ are refraction rays. $R1$ and $R2$ are reflection rays. [27]

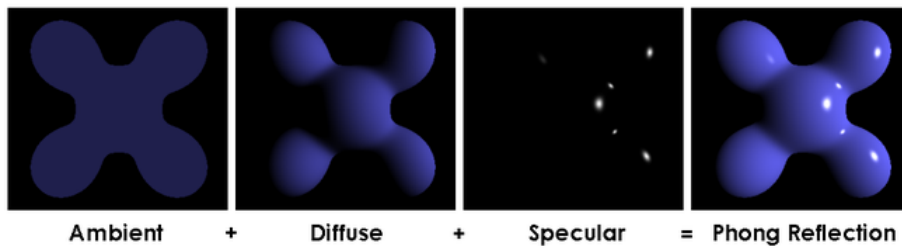


Figure 3.8: The Phong shading components. [80]

3.2.2 Functional Requirements

The functional requirements of the GRT application are:

- Trace primary.
- Trace secondary rays: reflection and shadow rays.
- Use spheres as primitives.
- Use point lights.
- Compute the resulting color of rays and lights.

- Display the resulting image.
- The camera position c is always $(0, 0, 0)$ and the camera direction vector is $(0, 0, 1)$.
- The field of view (FOV) of the camera can be changed.
- Looking from the cameras point of view, positive- x , y and z coordinates are right, up and forward, negative is the opposite, left, down and backwards.

3.2.3 CUDA Implementation

This section describes the parts of the implementation that have been specialized for the GPU, either for the purpose of performance or because CUDA requires it, e.g. because CUDA does not support recursion. Sections concerning performance is marked with a P in the headline and sections concerning CUDA requirements are marked with a R in the headline.

3.2.3.1 Work Partitioning (R+P)

The image plane, the image where a primary ray is shot from each pixel, is partitioned in such a way that one thread is allocated to rendering each pixels. The threads are grouped in 2D thread blocks, with pixels adjacent to each other, e.g. 8×8 or 16×16 pixels, and the thread blocks are grouped in a 2D grid block. In our implementation, we are limited by the number of available registers. We therefore choose a thread block size of 8×8 threads, as this gives better utilization of the SMs.

An alternative would have been creating a queue of primary and secondary rays to be traced and let all threads participate in the tracing. However, this will require that the queue is first build, then synchronized to the device, afterwards a kernel is started that traces the rays. The result of the trace would be new rays that are added to the queue, if any, and a partial color for the pixel. The process would be repeated until no new secondary rays were generated, which would result in the final rendered image.

Because of the many memory accesses required by this method, synchronization and kernel startups, this method imposes large overhead. However, this method also has the ability to sort rays, such that better cache locality can be achieved [19], with increased control flow complexity. Also, as explained in Section 1.2.1, the aim of this implementation is to optimize for CUDA, and not optimize the speed of the ray tracer by implementing data structures. We will therefore not use this method.

3.2.3.2 Intersection Buffer (R)

A common approach to ray tracing is with the use of recursion, however, CUDA does not support recursion on devices with lower compute capability than 2.0 as described in Section 2.6.2.4. Recall from Section 2.2 that the Tesla C870 only has compute capability 1.0. Instead, the ray tracer is implemented with an intersection buffer for each pixel, to store the data needed when computing the color, after tracing the rays. This is depicted on Figure 3.9. The intersection

buffer is kept in shared memory, as described later in the section. The elements depicted on Figure 3.9 is explained here:

- *Number of objects* - The number of rays traced
- *Origin x, y, z* - The point, for which the first ray has its origin
- *Object id* - Id of the object intersected by the ray
- *Intersection x, y, z* - Point of which the ray and the object with *object id* intersects

Note that each *Intersection* acts as origin for the succeeding *Intersection*, e.g. *ib* is the origin of *ia* as illustrated on Figure 3.9, except for the first *Intersection* which uses the *Origin* point as origin.

The ray tracer only traces up to a maximum of intersections, but the buffer needs to be able to contain this maximum, and is therefore statically sized to handle this, i.e. the buffer is allocated before the kernel is run. The number of intersections actually contained in the buffer is stored in the first cell *Number of objects*, as depicted on Figure 3.9.

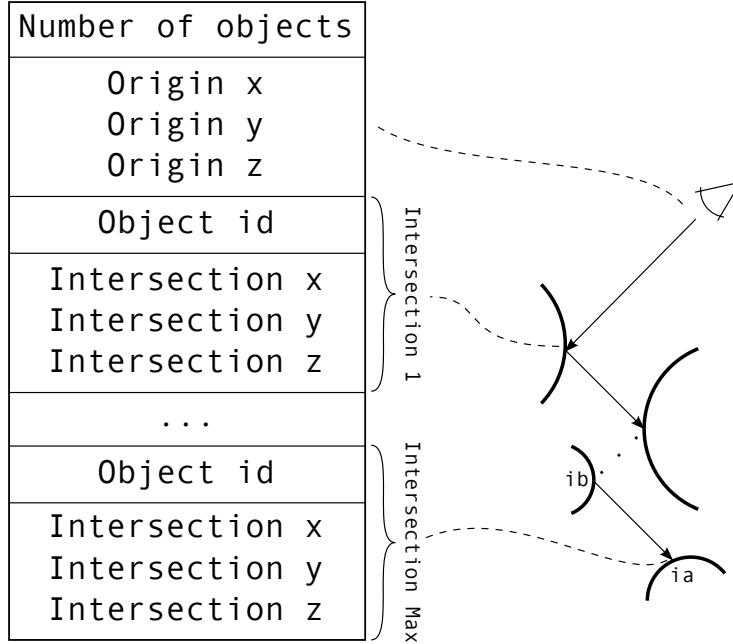


Figure 3.9: This figure depicts the structure of the intersection buffer of ray intersections

When the intersection buffer has been build, it can be iterated backwards in order to calculate the resulting color of the pixel. The *intersection* and *object id* data is used to determine which lights in the scene apply light to the point, i.e. which light rays are not intersected by other objects, what the color of object is, and the level of light reflected from an object.

As mentioned, the *Number of objects* data specifies how many rays have been traced, such that the color function only consider valid data in the buffer.

3.2.3.3 Memory (P)

Three types of memory have been used for the ray tracer: global, texture and shared.

Global memory is used for storing the resulting color of each pixel. The reason for using global memory is that the data is needed on the host and only global memory allows this, because constant and texture is read-only and private and shared is only accessible per thread and thread block, respectively. The global memory is only written to once per thread and only read on the host when the ray tracer has finished ray tracing a scene.

Texture memory is used for storing the needed data for objects and lights, e.g. position, radius, color and intensity. Texture memory is used because we do not need to update this data, that is, it adheres to the read-only requirement, and read-only memory can thus be used which is faster than global memory because it is cached. Although texture is read-only, animations can be done by moving objects on the host and copying the new data to the texture memory between each rendering, this is however not a functional requirement and therefore not implemented. We chose texture memory instead of constant memory, as constant is limited to 64KB, and texture memory can hold 2^{27} elements for a 1D texture.

Shared memory is used for the intersection buffer, described above, as it is faster than global. Local memory could have been used, but shared memory is faster since shared memory is on-chip where local memory is off-chip. The amount of shared memory available, i.e. 16KB for the Tesla C870, limits the number intersection that can be traced, as for each trace an intersection possibly needs to be stored.

If for example an 8×8 thread block is used, with the intersection buffer as described above, each thread will use up 16 bytes for *Number of objects* and *Origin*, leaving $\frac{16384 - 16 \cdot 8 \cdot 8 - 10}{8 \cdot 8} \approx 239$ bytes per thread for intersections. Note the -10 , these 10 bytes are used by the CUDA system. Each intersection takes up 16 bytes, thus $\frac{239}{16} \approx 14$ intersections are the theoretical maximum imposed by the thread block size. If more intersections are needed, the thread block size must be lowered, e.g. from a 8×8 to a 4×4 block size.

For comparisons sake, we have also implemented a ray tracer which uses global memory for objects and lights instead of texture memory, and private memory instead of shared memory for the intersection buffer. Apart from the changes of memory types, the source codes are equivalent, this will help us reason about the affect of or optimizations.

3.2.4 C++ Implementation

The code for the CRT is based on the C++ ray tracer we wrote for the 8th semester student project. This ray tracer was used on single core computers, and was therefore not multi-threaded.

To gain multi-threading support, we have used the Parallel Patterns Library that allows easy parallelization of loops using lambda expressions as we did with the C++ Boids implementation.

The screen is divided into N horizontal part, where N is the number of cores of the CPU, i.e. 4 cores for our quad processor. Each part is thereafter rendered in parallel on the CPU, and combined to form a complete image on the screen.

3.2.5 Summary

In this section we have described the basics of how ray tracing and phong shading can be performed, defined functional requirements of the ray tracer implementation should adhere to. We also described how the GRT was implemented, including alternatives and why they were not chosen, and given a brief description of how we have extended the CRT with multi-threading support. Benchmarks using the ray tracers will be performed in Section 4.2.

3.3 Summary

This chapter covered the development of the Boids application and the ray tracer application.

The Boids application was implemented in OpenCL and C++, and we chose the acceleration strategy proposed in [64], where all actors are indexed into a two dimensional matrix thereby allowing faster neighbor searches. With this acceleration structure, it should be possible to simulation at-least one million actors at interactive frame rates, i.e. at least 30 frames per second.

The ray tracer was implemented in CUDA targeting Nvidia GPUs. Shared memory was used to store the resulting intersections between the primary rays and spheres, and the intersections between the secondary rays and spheres, which should be faster compared to saving them in global or local memory. The spheres were kept in texture memory, not global memory, thus decreasing the bandwidth usage, since it is cached on chip.

Benchmarks will be carried out in Chapter 4 using both the ray tracer and the Boids application.

This chapter covers the benchmarks using the Boids application, which was described in Section 3.1, and ray tracer applications, which was described in Section 3.2.

The aim of this chapter is to get benchmarking data which says something about the performance of the GPU compared to the performance of the CPU, and which can be discussed.

4.1 Boids Benchmarks

We start by describing how the Boids application is configured for benchmarking and then describe the initial experiments, which is used to determine how to benchmark the system in a way that reflects the systems performance best. Afterwards we presents the results and discuss them.

4.1.1 Benchmark Setup

To determine the performance of the three implementations, i.e. the OpenCL, Brook+ and C++ CPU implementations, we will benchmark each implementation with 1024×1024 actors, roughly a million actors. This is the number of actors that [64] uses in their benchmarks, and this allows us to better reason about the performance compared to theirs.

We measure the performance of our implementations the same way as [64] does, by running each benchmark for 300 seconds and record how many iterations were calculated during this time. To increase consistency of the result, each benchmark is run 10 times and the average of these runs is calculated.

Recall that the functional requirements of the Boids implementations stated that the actors should follow the mouse, Section 3.1.3. During benchmarking however, the mouse will be disabled to avoid external influences, instead, the actors will flock around $x, y = (0, 0)$. In addition to disabling the mouse, the rendering of the actors will also be disabled, such that the rendering process does not influence the performance of the implementations.

The actors will initially be placed evenly in a grid, with a distance of 2 to each horizontal and vertical neighbor. This is to avoid worst-case performance

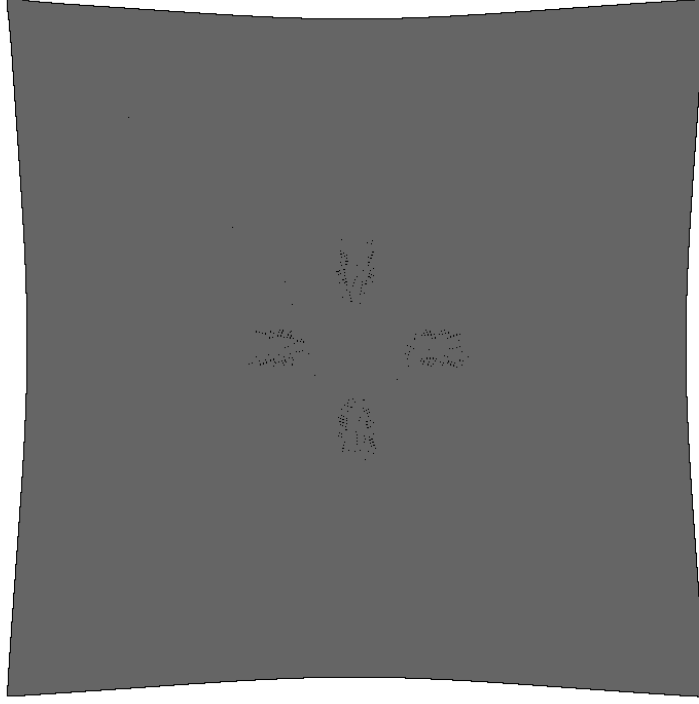


Figure 4.1: The Boids setup after a few iterations, containing roughly a million actors. The actors are flocking to the middle of the screen.

at the start of each benchmark, i.e. if all actors were placed at $x, y = (0, 0)$, all actors would be within each others radius resulting in all actors having to check all other actors. The setup is shown on Figure 4.1, where some iterations have passed. Note that each individual actor is had to see, since the setup contains roughly one million actors.

In addition to these requirements, we will perform benchmarks using a combination of the following configurations on both the CPU and GPU implementation:

- Benchmark using partial sort, in which only k iterations of the sort algorithm is done, as described in Section 3.1.2.1. In our case k is set to 1.
- Benchmark using full sort, which performs a full sort of the data structure as described in Section 3.1.2.1
- Benchmark using the *Pruned Neighbor Search* algorithm, in this chapter referred to as *complex*, described in Section 3.1.2.2, with *radius* = 2. See the left figure in Figure 4.2.
- Benchmark using the *Extended Moore Neighborhood* search algorithm, in this chapter referred to as *simple*, described in Section 3.1.2.2, with

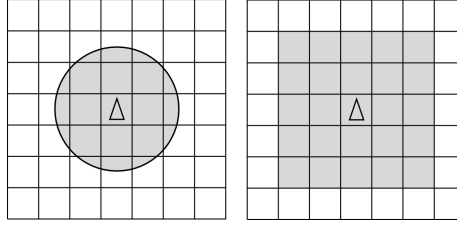


Figure 4.2: Illustrates Pruned Neighbor Search (complex) and Extended Moore Neighborhood (simple)

$radius = 2$, thus returning only the 24 closest neighbors. See the right figure in Figure 4.2.

The hardware used in the benchmarks is the LENOVO ThinkStation described in Section 2.2, which is equipped with a Tesla C870 for the OpenCL benchmarks, and a Radeon HD 2900 XT for the Brook+ benchmarks.

Since the proximity of the actors can influence the performance of the algorithm, we will also run a benchmark where the algorithm has run a number of iterations such that the performance has stabilized. We assume that actors coalesce or disperse from their original positions in the grid, and at some point achieve a relatively stable flock. To determine the number of iterations needed to achieve a stable flock, we will run an experiment for each configuration.

4.1.2 Initial Experiments

We expect the time it takes to calculate each iteration will increase as the actors flock closer around $x, y = (0, 0)$, because more and more actors comes inside each others radius. In order to find out when the performance of the algorithm begins to stabilize, i.e. when there is no longer any general increase or decrease in performance, we will execute the Boids algorithm for 2000 iterations, and measure the execution time for each iteration.

This approach is however problematic, since using the complex search algorithm causes the kernel to take more than 2 seconds to execute, which means that the Timeout Detection and Recovery (TDR) feature of Windows causes Windows to reinitialize the graphics drivers, thus restarting the GPU. Windows does this to ensure the responsiveness of the GUI. The number of seconds before the TDR kicks in can be changed by altering the `TdrDelay` registry key under `HKLM\System\CurrentControlSet\Control\GraphicsDrivers`. [39]

By simple trial and error we find that 300 is a good amount of time, thus ensuring that the TDR does not kick in during experimentation.

With regards to the partial sort and simple neighbor search algorithms, we do not expect much change in performance over time, because these algorithms always operate on a simple number of actors. On the other hand, we expect the full sort and the complex search algorithm to perform worse if the actors are situated close together.

We expect the full sort will perform worse because there is a greater possibility that an actor gets moved multiple positions in the matrix, thereby increasing the number of iterations to sort the matrix.

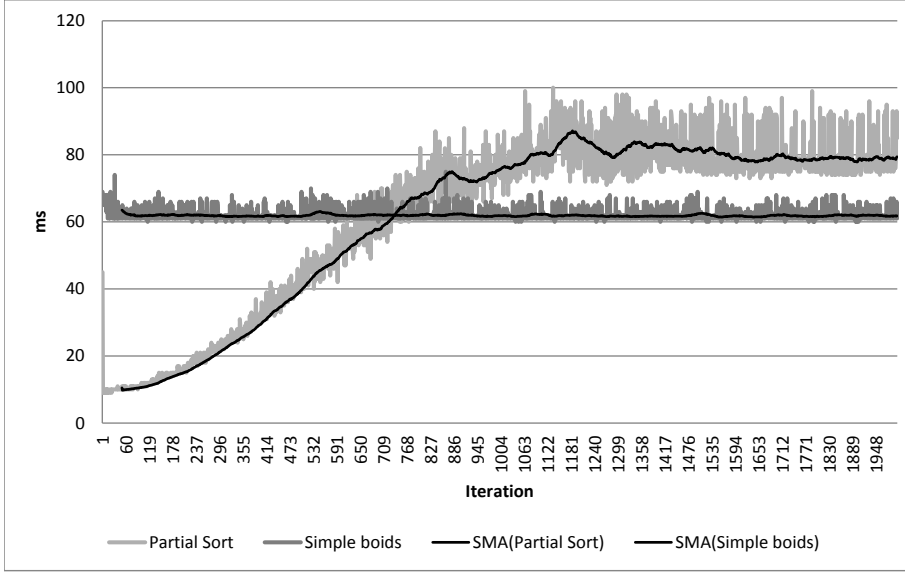


Figure 4.3: The time taken to compute a partial sort and Boids using simple neighbor search for a given iteration on the GPU

We expect the complex search algorithm will perform worse because each actor will have more flock-mates within its radius, which must be taken into consideration when calculating the steering behaviors, as was described in Section 3.1.1.

Recall that we perform these experiments to find out in which iteration the positioning of the actors has reached a sufficiently stable grouping, that is, there is no longer any general increase or decrease in performance. Since there will always be some difference in performance from one iteration to the next, we will be calculating a Simple Moving Average (SMA) over the past 50 iterations to help spot more general trends.

The experiments will be run using the OpenCL implementation because we expect that the other implementations will be affected by the grouping of actors the same way, due to all the implementations using the same algorithms.

Also, to make it easier to reason about the results, we will measure the sorting time and the Boids algorithm independently.

4.1.2.1 Results

As expected, the experiments show that the simple neighbor search algorithms are not affected by the positioning of the actors as seen both in Figure 4.3 and Figure 4.4 where it takes around 60ms in all iterations.

The partial sort appears to be somewhat affected by the positioning of the actors, this is most likely because actors more frequently change positions in the matrix when actors are closer together. Still, in the partial sort, actors are moved at most one position in the matrix thereby posing an upper limit to the number of operations performed by the algorithm. As shown in Figure 4.3, the limit appears to be reached after about 1000 iterations when the performance

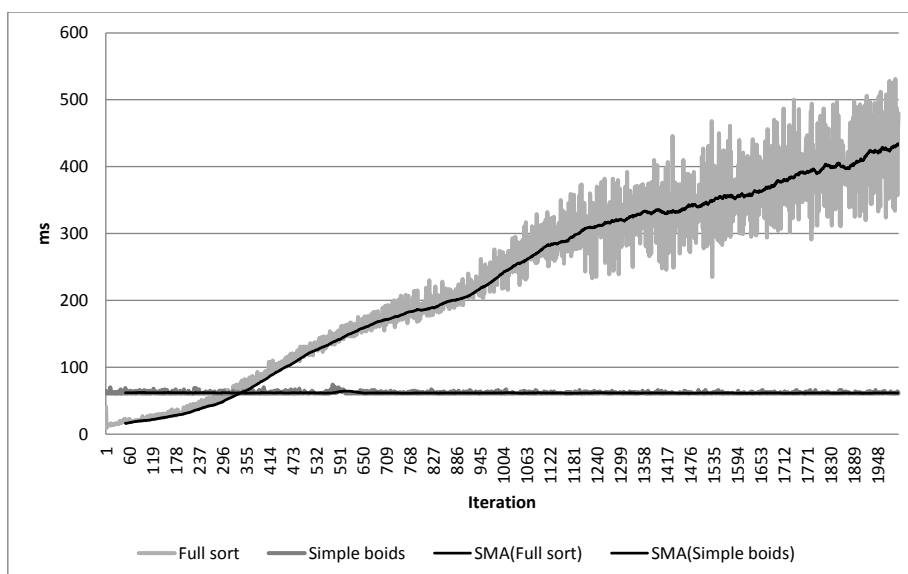


Figure 4.4: The time taken to compute a full sort and Boids using simple neighbor search for a given iteration on the GPU

starts to stabilize, this is also the case with the partial sort with complex search, even though hard to see on Figure 4.5.

As expected, the complex search algorithm is also affected by the positioning of the actors. The results of the experiments shown in Figure 4.6 and Figure 4.5 clearly show that the performance decreases over the first 1000 iterations. The execution time increases from just a couple of hundred ms to over 14000ms for a single iteration, but after 1000 iterations the performance decrease starts to stabilize.

The performance of the full sort decreases as we expected. This is shown on Figure 4.4 where the performance appears to decrease over all 2000 iterations. The full-sort in Figure 4.6 appears to stabilize after 1500 iterations, which was not the case on Figure 4.4. This might be because the complex search algorithm allows for better flocking due to its use of a radius, i.e. more actors can be considered when calculating the steering behaviors, thus creating a more ordered matrix which is easier to sort.

4.1.3 Benchmark Results

Based on the experiments, we have decided to start the benchmarks after iteration 1500, since we have a relatively stable flock of actors. After 1500 iterations all the algorithms appears to have stabilized, except the full sort. Figure 4.7 shows the number of iterations completed by the different implementations in the 300 seconds the benchmark was running.

Figure 4.8 shows the speed up achieved by the different implementations relative to the CPU implementation.

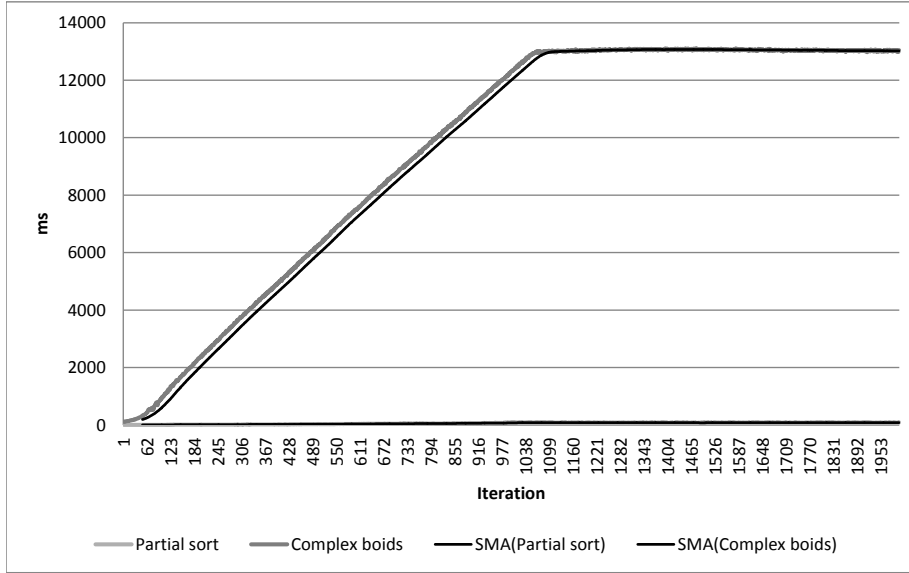


Figure 4.5: The time taken to compute a partial sort and Boids using complex search for a given iteration on the GPU

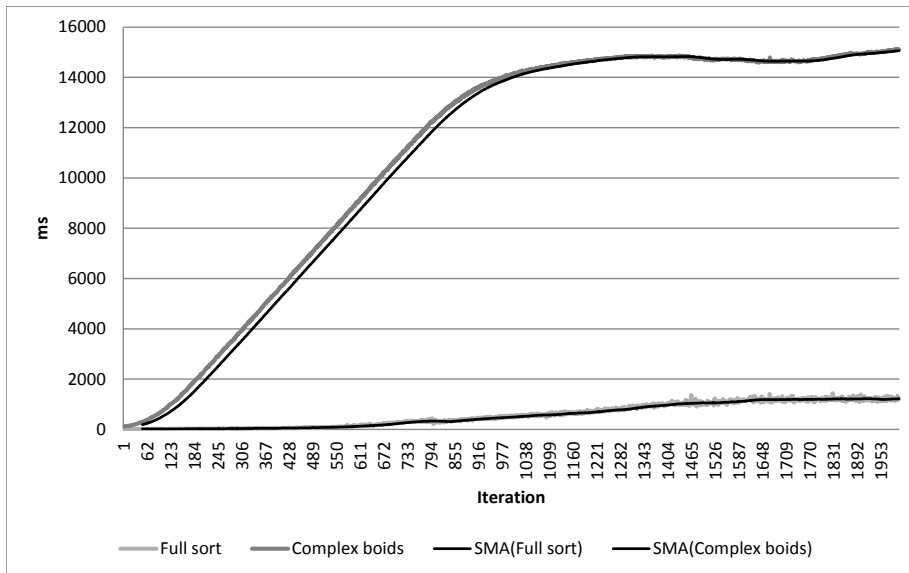


Figure 4.6: The time taken to compute a full sort and Boids using complex search for a given iteration on the GPU

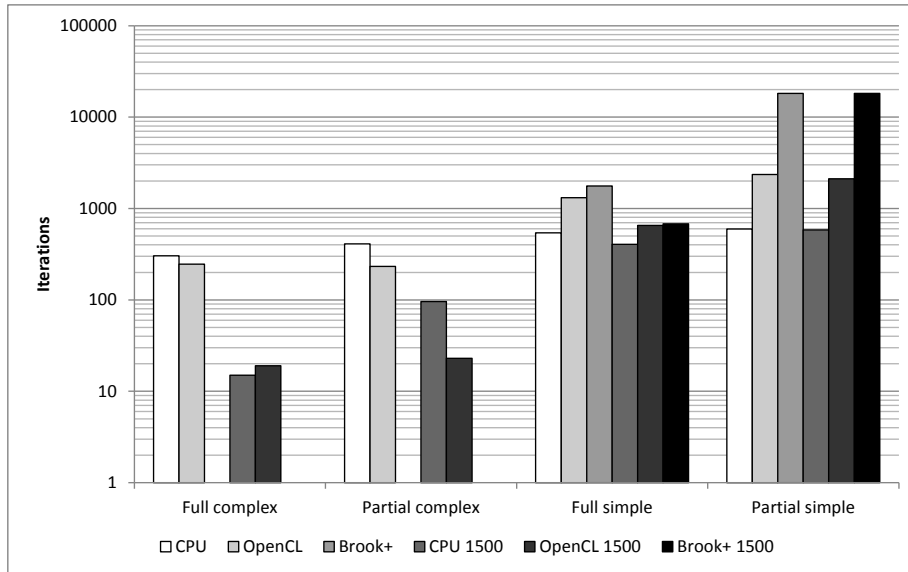


Figure 4.7: Benchmark results for the CPU and OpenCL implementations of the Boids algorithms. Simple refers to the use of the Extended Moore Neighborhood search algorithm while complex refers to the use of the Pruned Neighbor Search algorithm. Full refers to the use of a full sort while partial refers to the use of a partial sort. 1500 refers to a benchmark starting at iteration 1500.

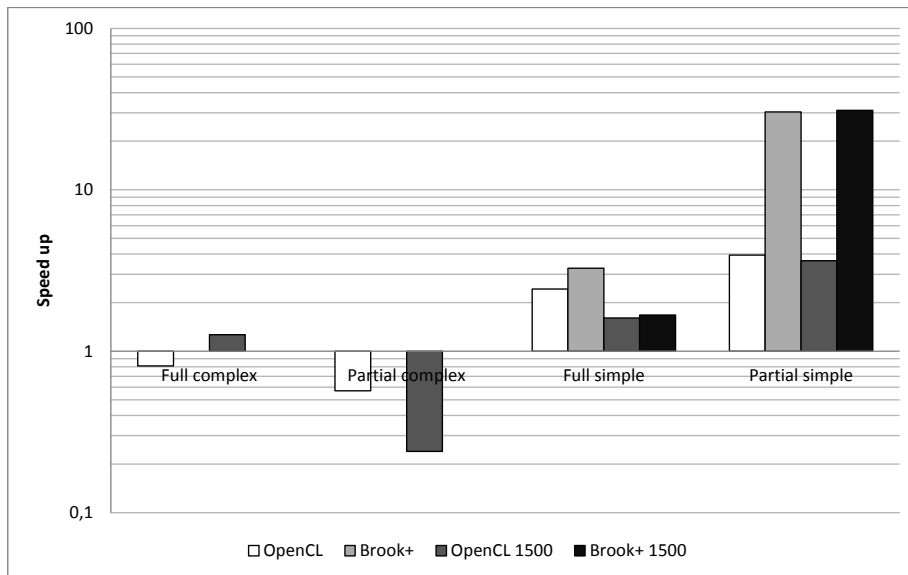


Figure 4.8: Speedup of the different Boids implementations relative to the CPU implementation. Simple refers to the use of the Extended Moore Neighborhood search algorithm while complex refers to the use of the Pruned Neighbor Search algorithm. Full refers to the use of a full sort while partial refers to the use of a partial sort. 1500 refers to a benchmark starting at iteration 1500.

4.1.4 Discussion

Figure 4.8 shows that the complex search algorithm in three out of four cases is slower on the GPU than the CPU. Only in the case where complex search is used in conjunction with a full sort and when run after 1500 iteration can the GPU outperform the CPU and even so only by a little. The reason that we do not see better performance from the GPU, even though it has a theoretical performance of 518.4 gigaFLOPS against the 40 gigaFLOPS of the CPU, is that the complex search does not fit well with the characteristic described in Section 2.4. More specifically it performs a lot of writes to memory when adding neighbors to the list and it performs scattered memory accesses leading to low memory throughput. Some of this should have been alleviated by using private memory which according to the OpenCL specification is located close to itsPE[30, p. 24], but Nvidia has chosen to implement private memory in DRAM, which is not close to the PE as we initially expected.

When looking at the simple neighbor search algorithm we see that the GPU outperforms the CPU in all benchmarks, however, we do not see anything close to the theoretical $\frac{518.4}{40} = 12.96$ speed up for the OpenCL implementations. This is most likely because we do not use any of the on-chip memory on the GPU, thus its performance is severely limited by the available memory throughput, which is already decreased by the many uncoalesced memory accesses performed when sorting.

For the Brook+ implementation we see a greater increase in performance than the OpenCL implementations, in fact the partial sort, simple neighbor implementation goes beyond the theoretical speed up of $\frac{475}{40} = 11.875$, with a speed up of around 30 and 23 for the benchmarks from iteration 0 and 1500 respectively.

One reason for this large speed up compared to the OpenCL implementation is that Brook+ automatically make use of the GPU on chip memory. We also suspect the use of a four dimensional vector to represent an actor helps increase performance, since this is a primitive implemented in hardware on the Radeon HD 2900 XT, where the Tesla C870 can only operate on scalar values. When comparing partial sort with fully sort implementations, we see that the Brook+ implementation has a much larger performance decrease compared to the CPU implementation, from a speed up of 30 to a speed up of around 3, and a speed up of 23 to a speed up of less than 2, than the OpenCL implementation. This may be caused by the Brook+ implementation has to copy data to the CPU for each iteration of the odd-even sort to check if the sorting is completed, and the lunch a new kernel if it is not. This process creates a considerable overhead, which the other implementations do not have.

4.2 Ray Tracer Benchmarks

This section will cover the benchmarks using the ray tracer application. We will first cover the setup that we will use, i.e. the scene and configuration of the ray tracer benchmark. Afterwards, the actual benchmarks are performed and the results are shown and discussed.

4.2.1 Benchmark Setup

To benchmark the GRT and CRT implementation, we have defined a scene setup. An alternative would have been to use a scene defined by other researcher, however our implementation only support spheres, thus complex objects cannot be rendered which means that our ray tracer cannot render such scenes.

Defining a scene allows us to reason about the performance of the GRT and the CRT which we developed through our 8th semesters project. The results are not affected by the complexity of the scene, since the scene is used by both ray tracers. The computations of the scene should be fairly static, since no animations are present, i.e. the execution time of one frame will not be much different from the last frame.

We will measure the performance of the implementations by running each benchmark for 300 seconds, and record how many frames were rendered in this time. To assure consistency of the result each, benchmark is run 10 times and the average of these runs are calculated.

The scene we have chosen for the benchmarks is defined as follows, and shown on Figure 4.9:

- 289 spheres with a radius of 10 are placed in a 17×17 grid, with a distance of 30 between each horizontal and vertical neighbor sphere. The upper left sphere in the grid is located at $(-240, -240, 1024)$, i.e. sphere i, j in the grid is located at $(-240 + 30 * i, -240 + 30 * j, 1024)$.
- 10 lights are placed on a straight line with an horizontal distance of 20 between them. The left most light is located at $(-90, 0, 512)$, i.e. light i on the line is located at $(-90 + i * 20, 0, 512)$.
- The resolution of the image is set to 512×512 pixels.
- All of the spheres has a reflective surface.
- A max of three reflection rays are traced per pixel.
- The field of view is $\pi/6 \approx 30^\circ$.

We have chosen 289 spheres since each sphere takes up 32 bytes of memory, i.e. three floats for coordinates, three floats for color, one float for diameter and one float for the reflection constant. This means that the 289 objects exceeds the available texture memory cache, which is 8KB at maximum [54, p. 148], the benchmark scene thus poses a challenge for the GPUs memory system.

We chose 10 lights instead of just one since this poses a greater computational challenge, thus our benchmark scene with 289 spheres o , and 10 lights l with a naive implementation will have to make 289 intersection checks for each primary ray, and 288 intersection tests per secondary ray, as secondary rays do not need to check with its own origin object. This means that for a scene with 512×512 pixels p , and a maximum of two secondary rays s , we need $p^2 \times ((o \times s + 1) - s) = 512^2 \times ((289 \times 3) - 2) = 226,754,560$ intersections for a worst case, i.e. all primary rays intersect an object and each secondary ray does as well. Furthermore, each found ray intersection needs to check if any object blocks the light sources, this results in $p^2 \times l \times (o - 1) \times (s + 1) = 512^2 \times 10 \times 288 \times 3 = 778,567,680$ intersection tests. The total number of intersection tests in the worst case is

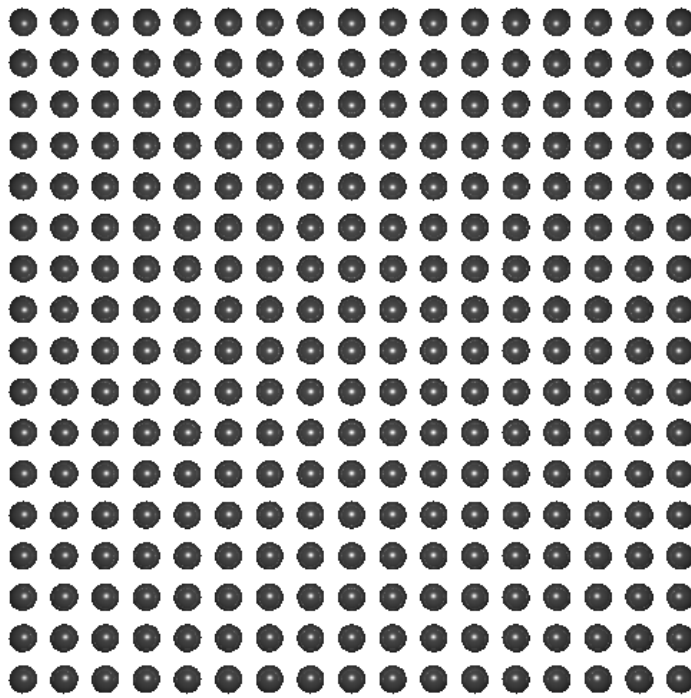


Figure 4.9: The scene used in the ray tracer benchmark.

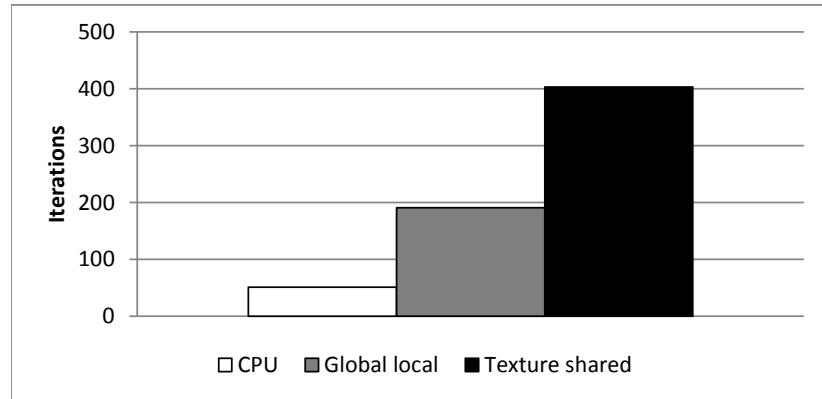


Figure 4.10: Benchmark results for the ray tracers

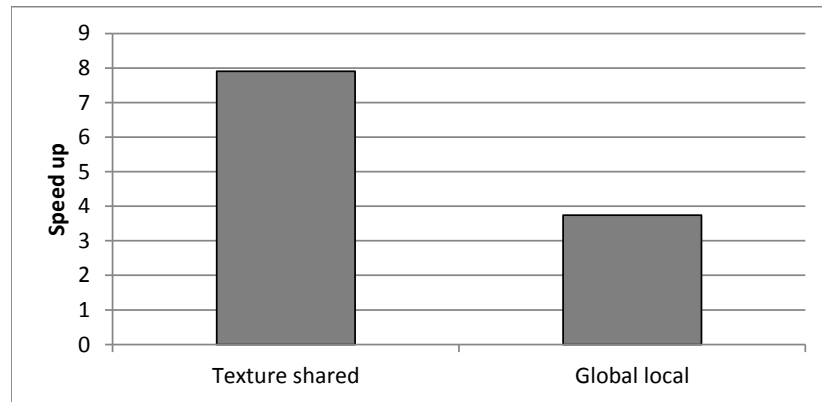


Figure 4.11: Speed up for the different ray tracer implementations relative to the CPU implementation

therefore 1,005,322,240. Initial tests of the benchmark scene shows that only about 60% of the primary rays intersects a sphere, this however still poses a significant work load.

Recall from Section 3.2 that one of the goals of the GRT implementation was to experiment with the use of different memory types in CUDA to increase performance. To show the difference in performance caused by using different types of memory in CUDA, we will also include results where texture and shared memory is not used.

4.2.2 Benchmark Results

Figure 4.10 shows the Benchmark results for the different ray tracer implementations.

Figure 4.11 shows the speed up achieved by the different GPU implementations of the ray tracer compared to the CPU implementation.

4.2.3 Discussion

When looking at the results of the benchmarks in Section 4.2.2, we see that the GPU implementation which uses texture and shared memory, referred to as the optimized implementation, is about twice as fast as the GPU implementation which only uses global and local memory, referred to as the unoptimized implementation. The reason for this is that global and local memory resides in the relatively slow and high latency DRAM on the graphics card, while shared memory resides solely in much faster and lower latency on-chip memory, and the texture memory is cached on-chip thus avoiding some DRAM access.

In theory, the difference in latency between the different types of memory should not have a large impact on performance if there are enough threads running on the GPU, since the latency of memory access can be hidden. The number of threads that can run concurrently on the GPU is limited by the use of registers and shared memory, thus, to investigate the latency hiding ability of the two GPU-based implementations, we use the CUDA Occupancy Calculator described in Section 2.7.5.

Using the `-ptxas-options=-v` compiler option to determent the resources used by each implementation, we see that the unoptimized implementation uses 39 registers and 42 byte shared memory while the optimized implementation uses 32 registers and 4132 byte shared memory.

Using the CUDA Occupancy Calculator we see that both implementations can at maximum run 192 concurrent threads per SM, which is just 25% the maximum possible. Had the unoptimized version been able to run more threads concurrently, we expect there would have been less of a gap between the two implementations' performances, as it would have been able to better hide memory latency. Using a GPU with more register memory and with the same amount of shared memory, such as a GPU with compute capability 1.2, we might see that the unoptimized implementation is faster, since this implementation is limited by register used, while the texture and shared memory implementation is limited by use of shared memory, as described in Appendix A.

When comparing the GPU implementations to the CPU implementation, we see that neither of them achieves the theoretical speed up of $\frac{518.4}{40} = 12.96$ we would expect when looking and the theoretical performance of the CPU and GPU.

However the theoretical peak performance of the GPU is only reached when performing fused multiply-add operations on the SP and multiply operations on the SFU simultaneously, as described in Section A.1. We assume this is not always possible as it would require an application that only made use of multiply or add operations, performed twice as many multiply operations as add operations, and a compiler which produces optimal code.

If we instead assume that we can not perform fused operations, the GPU has a performance of $\frac{518.4}{3} \times 2 = 345.6$, thus we would only expect to see a speed up of $\frac{345.6}{40} = 8.64$ which is quit close to the speed up we actually see for the implementation using texture and shared memory.

This chapter covers the comparisons of the three GPGPU languages, including their architectures, that we found through the analysis in Section 2.6, and that we used during development of the ray tracer Application and the Boids Application as described in Chapter 3.

The aim of the comparisons is to determine the properties of the GPGPU languages and architectures, and formalize our understanding these properties.

The assessments are based on theory found in the analysis of the languages in Section 2.5, and the experiences obtained through the implementations in Chapter 3.

We start by defining the criteria we will be comparing each languages to, and afterwards perform the actual comparison using these criteria.

5.1 Criteria for Comparison

Like [10] and [14] we will set up criteria in order to compare GPGPU programming platforms. These criteria will help us ensure that the architectures and languages are compared on the same basis.

To simplify the comparison, we will merge the architecture and language comparison unless explicitly stated, e.g. when we talk about CUDA we mean both the architecture defined as CUDA and the CUDA C language, we thus introduce the term *platform*. A computing platform is defined as hardware and software framework that allows software to run [74].

[10] focuses on how languages perform with regards to teaching computer science students programming, and its criteria thus focuses on ease of learning, these criteria can help us assess and compare the learning curve of languages.

[14] focuses on comparing concurrent languages and lists properties which can be used to rank languages' functionality, these can likewise be used to compare GPGPU languages.

The criteria below are inspired by both [10] and [14], and based on the experience we gained through the development process in Chapter 3.

Each criterion is rated between two extremes and displayed on a figure, such as Figure 5.1. The thick line indicates the rating for the particular language.

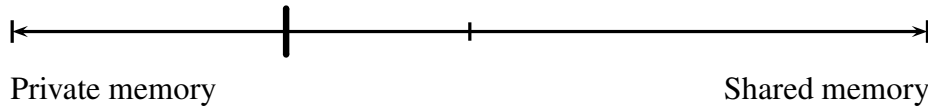


Figure 5.1: Example of criterion rating

5.1.1 Memory

The type- and management of memories influences how a program is written, e.g. a program written in a programming language that supports random memory allows the program to randomly access memory when necessary. The opposite situation is where only a small part or a single element of memory is accessible to each part of the program, e.g. as known from stream processing.

Memory types Shared memory allows easy thread communication and allows threads to work on the same set of data, thereby allowing very fine grained parallelism, although this can lead to race conditions. Private memory does not have the risk of race conditions, because memory is private to each thread, but the granularity that can be achieved is more coarse unless some form of message passing is used. Also, as no memory is shared, no locks are required.

Private memory Only private memory is available to threads in the kernel, thus no data can be shared between threads.

Shared memory Only shared memory is available and all memory is shared between all threads.

Memory management Automatic memory allocation and deallocation means that the programmer does not have to define what type of memory is allocated and how much is allocated and deallocated, which means that the compiler or runtime component makes these decisions for the programmer. In manual allocation and deallocation, the programmer has to explicitly define the type of memory and how much is allocated, and explicitly free up the memory when done.

Manual The programmer explicitly defines the data type, what type of memory, e.g. private or shared, and how much memory should be allocated. The programmer also manually frees up memory when done.

Automatic The compiler or runtime is in charge of allocating and freeing memory, and automatically figures out the memory type.

5.1.2 Computation

The computational ability of the architecture affects how problems can be solved, e.g. if loops are not allowed these must be unrolled at compile time or explicitly unrolled by the programmer.

Standard compliant If the platform is not standard compliant, the programmer needs to be aware of this and take this in to consideration, e.g. the programmer must know whether the architecture supports the IEEE float and

double precision float standards if the correctness of the application depends on these.

No standard compliant No standards are guaranteed.

Fully standard compliant All standards are followed.

Branching Branching allows the control flow to be altered, when certain conditions are fulfilled or by following pointers. This can make code more readable, as it can be segmented in functions and loops does not have to be unrolled, too much branching however might affect performance [54, E.2].

No branching No branching is supported.

Branching Full branching is supported.

5.1.3 Learnability

In this context we define learnability as how easy it is to learn the syntax, semantics and framework of the platform, and how easy it is to program the platform for a programmer who has no previous experience with GPGPU programming. Learnability is important for programmers new to a platform, but also for correctness, e.g. if the platform has special case rules, the programmer needs to know this to avoid the problems that may arise, for example when comparing Strings in java `.equals()` must be used, where primitives, such as `int`, uses `==` [10].

Abstraction Abstraction needs to be balanced, i.e. if the level is too low, it may become tedious and error prone, while too high level, might make it hard to learn and even inhibit the programmer in understanding what is actually going on on the hardware, thereby hindering optimization [10].

Low-level No details are abstracted away.

High-level All details are abstracted away.

5.1.4 Concurrency

Concurrency is what enables the speed gain on GPUs, and is thus an important part of GPGPU programming.

Determinism Execution order affects the way computations should be handled, that is, if no particular order is ensured the developer needs to setup barriers if threads depend on data computed by other threads. If the execution order is guaranteed, this may affect the overall speed as latency hiding can not be achieved to the same degree as described in Section 2.1.

Nondeterministic There is no guaranteed order of execution between threads, one thread may execute all steps before all other threads.

Deterministic The execution order of threads is known to the programmer.

Concurrency management Automatic concurrency relieves the programmer from the task of defining which parts of the source code should be run concurrently, however the compiler needs to be able to make sound decisions about this, and may not be able to optimize the code as much as a programmer could.

Manual concurrency The programmer explicitly programs the concurrent parts of the program.

Automatic concurrency The compiler automatically makes relevant parts of the program concurrent.

Fault restriction A fault restricted platform prevents the programmer from introducing concurrency faults, such as race conditions. Fault restriction increases the reliability of the program, but the programmer may feel inhibited in his expression power, as the programmer may not explicitly influence the behavior of the program. A very expressive model allows the programmer to express exactly the intended behavior, but also requires care in not introducing concurrency faults.

Expressive model The programmer can express exactly how the program should behave, but needs to take care not to introduce concurrency faults.

Fault restricted This model restricts the developer from introducing concurrency errors such as deadlocks, this may however restrain the programmer.

5.1.5 Support

Even though a platform might in theory provide programmers with the needed GPGPU functionality, the platform must also be well supported before programmers will use it, or can use the platform effectively. In the absence of any implementations of the platform, programmers will have little desire to use the platform for actual development. The same applies if the implementations, e.g. the compiler or tool-chain, are no longer being developed and therefore have a low maturity level.

Cross-platform If a programmer is to develop an application targeting computer architecture(s), the programmer must know how well the platform is supported on the computer architecture(s). A platform which has several implementations targeting different computer architectures has greater cross-platform support than a platform with few implementations, though several implementations may make it harder to optimize.

One implementation One implementation of the platform exists, thus programs developed for the platform have little cross-platform support.

Several implementations Several implementations of the platform exist, thus programs developed for the platform have great cross-platform support.

Maturity Even though implementations of a platform exist for several computer architectures, the implementations might be of low quality or have little to no documentation. Also, further development of the platform might have

ceased and be out of date. The maturity of the platform is therefore an important aspect for programmers.

Immature The platform has low level of maturity.

Mature The platform has high level of maturity.

5.2 Ratings

In this section we will rate all three platforms using the criteria found above.

5.2.1 Memory

This section rates BrookGPU, CUDA and OpenCL using the Memory types and Memory management criteria.

5.2.1.1 BrookGPU

BrookGPU is rated as follows:

Memory types Kernels in BrookGPU can have two types of parameters: input/output streams and constants. Each thread has access to only one position in a stream, e.g. thread number 11 can access element 11 in the input and the output streams. Constant memory, however, can be randomly read by all threads.

Due to the limitations of how memory can be accessed in BrookGPU, we argue that the memory model is more private than shared as showed on Figure 5.2.

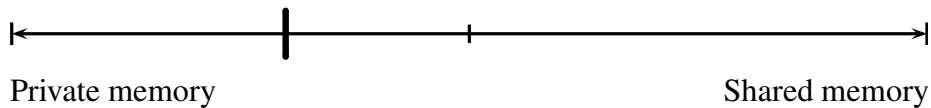


Figure 5.2: Brook memory type rating

Memory management Memory management in BrookGPU is very similar to C's memory management model, except when working on streams. The only difference for the programmer is that stream memory is defined using `<>`, as described in Section 2.6.1, while constant memory is not. We therefore rate BrookGPU's memory management somewhat manual, as shown on Figure 5.3, since one has to explicitly define the size, types, etc of the streams.

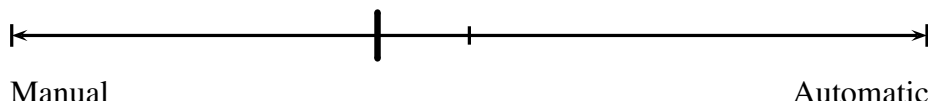


Figure 5.3: Brook memory management rating

5.2.1.2 CUDA

CUDA is rated as follows:

Memory types CUDA supports several variants of memory: global which all threads can access, shared which only threads in the same thread block can access and private which is only access to the individual thread, and two read only memories. We thus argue that CUDA should be rated more global than private Figure 5.4, as both types are available, but more global memory is available compared to private memory.

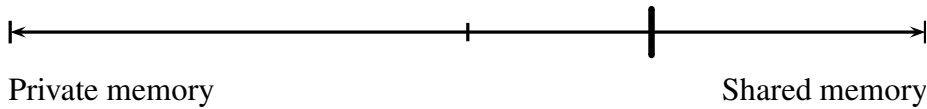


Figure 5.4: CUDA memory type rating

Memory management CUDA requires that all memory is explicitly allocated and deallocated, except for registers, local and shared memory. Local memory is known as private memory in OpenCL, and is used when all registers are used and for large data structures that can not fit in registers, as described in Section 2.6.2.2. Local memory can not be manually allocated, freed or defined when to be used, and is thus automatically managed. Shared memory is automatically managed, but must be preceded with the `__shared__` keyword, e.g. `__shared__ float foo[20];`. Because registers and local memory are automatically managed, while global, texture and constant memory are manually managed, we rate CUDA more manually managed than automatic managed, as shown on Figure 5.5.

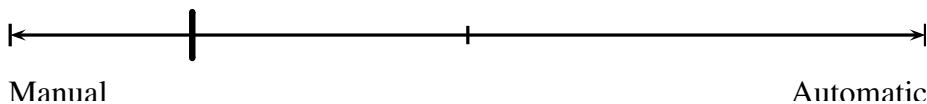


Figure 5.5: CUDA memory management rating

5.2.1.3 OpenCL

OpenCL is rated as follows:

Memory types As with CUDA, OpenCL supports both private and global memory type. Private is very much like CUDAs local memory, but is manually managed. Local memory, which is the same as shared memory in CUDA, is local to all threads in a workgroup. And global, which is the same as global memory in CUDA. We therefore rate OpenCL as CUDA, as shown on Figure 5.6.

Memory management OpenCL supports is very similar to CUDA, however private memory needs to explicitly defined by preceding the declaration with `__private__`, thus we rate OpenCL a bit more manual as shown on Figure 5.7.

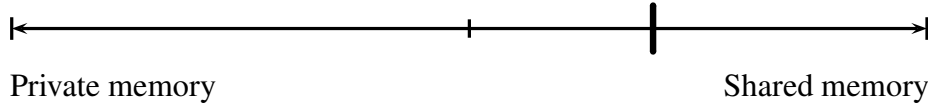


Figure 5.6: OpenCL memory type rating

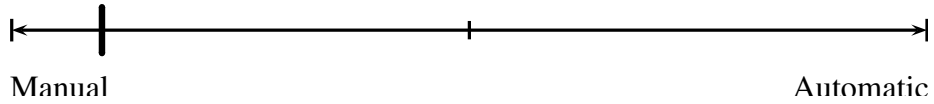


Figure 5.7: OpenCL memory management rating

5.2.2 Computations

This section rates BrookGPU, CUDA and OpenCL using the Standard compliant and Branching criteria.

5.2.2.1 BrookGPU

BrookGPU is rated as follows:

Standard compliant The backends of BrookGPU, e.g. DirectX, OpenGL, CAL, etc. define what standards are used when performing floating point arithmetic. Thus, BrookGPU does not guarantee any standards as shown on Figure 5.8.

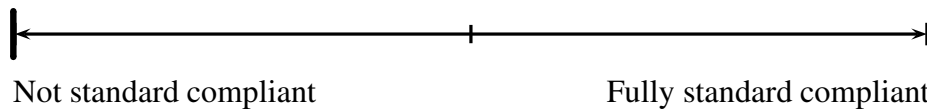


Figure 5.8: Brook standard compliant rating

Branching BrookGPU supports branching only if the underlying backend supports it, e.g. branching is not supported in pixel shaders prior to 3.0. Brook+ does however support branching, as does newer pixel shader versions, i.e. 3.0+. In addition, function calls are not supported in BrookGPU, meaning that all functions must be inlined and recursion is therefore not supported. Based upon this information, we rate it as between full branching and branching as shown on Figure 5.9.

5.2.2.2 CUDA

CUDA is rated as follows:

Standard compliant CUDA does not fully adhere to the IEEE standard, however this is only a problem for floats, as described in Appendix A. Thus we rate standard compliance as shown on Figure 5.10.

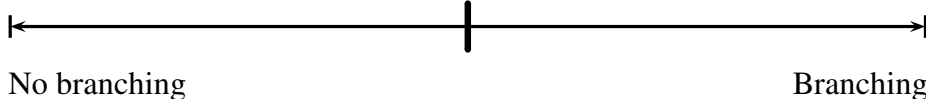


Figure 5.9: Brook branching rating

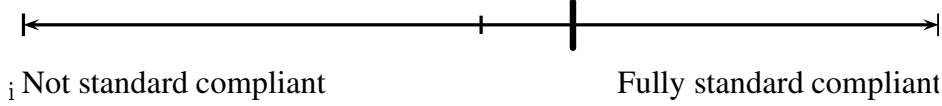


Figure 5.10: CUDA standard compliant rating

Branching CUDA supports full branching in loops, if-statements and function calls, though branching can reduce execution speed, as only one branch can be followed at a time, as described in Appendix A. Furthermore, recursing is not supported. Based on the drawback to branching and no recursion, we rate CUDA as having close to full branching support as shown on Figure 5.11.

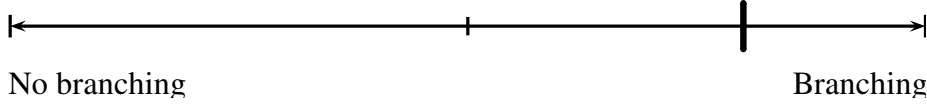


Figure 5.11: CUDA branching rating

5.2.2.3 OpenCL

OpenCL is rated as follows:

Standard compliant OpenCL allows the programmer to specify certain flags when compiling an OpenCL kernel, e.g. using fast math thereby sacrificing precision. This also allows the programmer to specify whether a standard must be followed, thereby influencing correctness. OpenCL defines which standards should be followed, however not all IEEE 754 is followed to the letter [22, 9.3.9], thus we rate OpenCL a little less than fully standard compliant as shown on Figure 5.12.

Branching As with CUDA, OpenCL supports branching except for function calls, i.e. functions are inlined, and no recursion. We therefore rate OpenCL as CUDA, this is shown on Figure 5.13.

5.2.3 Learnability

This section rates BrookGPU, CUDA and OpenCL using the Abstraction criteria.

5.2.3.1 BrookGPU

BrookGPU is rated as follows:

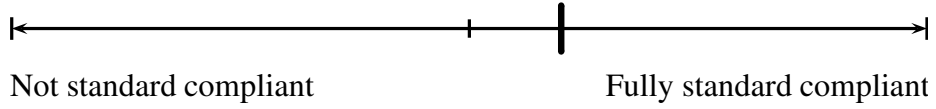


Figure 5.12: OpenCL standard compliant rating

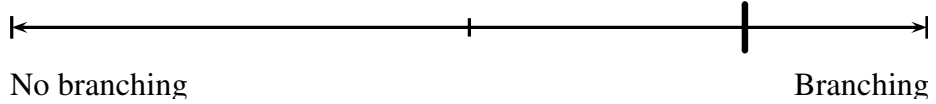


Figure 5.13: OpenCL branching rating

Abstraction BrookGPU comes with C functions that can be used to operate streams and kernels, e.g. `streamRead(s, data);` fills the stream `s` with the data contained in `data`. In addition, arithmetic and vector functions are provided to kernels. Also, the Brook language is based upon C, with some extensions that allow programmers to declare streams and kernels, e.g. `float s<10,10>;` declares a two dimensional stream of floats.

Kernels in Brook looks very much like normal functions, as seen in Codeexample 5.1. The angle brackets denote the streams, while the `reduce` keyword denotes that this kernel is a reduction kernel, i.e. a kernel that reduces a stream to a smaller stream or simply a scalar value.

```

2 void reduce sum(float a<>, reduce float result<>)
3 {
4     result = result + a;
5 }

```

Codeexample 5.1: A reduction kernel, that reduces the 'a' stream and stores the result in the 'result' stream

Assuming the programmer knows the C language, we estimate that learning the Brook language is not very difficult, since the language extensions are few.

Based on this information we rate the BrookGPU as somewhat high-level, as shown on Figure 5.14.

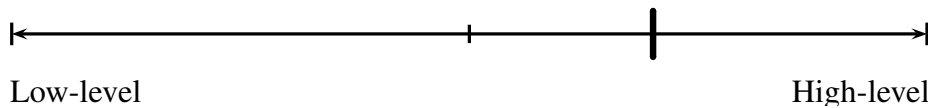


Figure 5.14: Brook abstraction rating

5.2.3.2 CUDA

CUDA is rated as follows:

Abstraction High-level commands exist, such as `cudaMallocPitch` and `cuMemAllocPitch`, which allocates padded memory, i.e. memory which has been aligned by padding extra bytes to the data. These two functions abstract away the task of padding memory in a way appropriate to the hardware executing the program. [54, sec.

5.3.2.1.2][52]

There exist both a high-level and low-level API for allocating texture memory [54, sec. 3.2.4.3]. However, CUDA does not abstract away things such as how many threads should be executed or memory allocation for all types of memory, thus we rate it less high-level than BrookGPU, as seen on Figure 5.15.

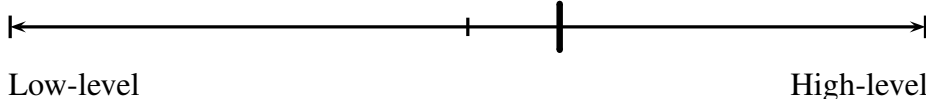


Figure 5.15: CUDA abstraction rating

5.2.3.3 OpenCL

OpenCL is rated as follows:

Abstraction The OpenCL framework provides access to GPGPU functionality on Nvidia and ATI GPUs, and among others, by providing a low level C API that can be used to issue memory reads, writes, start kernels, query devices, etc.

Any programming language that can interact with C libraries can be used as host language, e.g. C++, C#, Java, Python, and many more. This means that programmers do not have to learn a specific language to write the host part of the OpenCL application. The programmers still need to learn the API and how the framework is used. In addition, OpenCL introduces different memory types, such as private, local and global memory. Exploiting these memory types in the right situation is in many cases crucial to achieve good performance, as is the case with CUDA. Managing these types of memory is an extra burden on the programmer, and must be learned.

OpenCL has many similarities with CUDA, but the way OpenCL launches kernels is much more low level compared to that of the CUDA runtime. In OpenCL, a kernel is launched as shown in Codeexample 5.2, where the `clSetKernelArg` function is called for each argument passed to the kernel, and the `clEnqueueNDRangeKernel` function is called which executes the kernel. This is in contrast to the CUDA runtime, which uses a language extension to make kernel scheduling much cleaner, as previously seen on Codeexample 2.2. The programming overhead of using this low-level kernel invoking makes the platform more complex, as more functions need to be known by the programmer, we thus rate OpenCL more low-level than CUDA as shown on Figure 5.16.

```

...
2  clSetKernelArg( kernel , 0 , sizeof( cl_mem ), &gpuVectorA );
   clSetKernelArg( kernel , 1 , sizeof( cl_mem ), &gpuVectorB );
4  clSetKernelArg( kernel , 2 , sizeof( cl_mem ), &gpuVectorC );
   int blockSize = 16;
6  size_t workSize[2] = { dataSize , dataSize };
   size_t localSize[2] = { blockSize , blockSize };
8  error = clEnqueueNDRangeKernel( commandQueue , kernel , 2 , NULL ,
                                   workSize , localSize , 0 , NULL , NULL );
...
10 }
```

Codeexample 5.2: Launching a kernel in OpenCL requires many lines of code

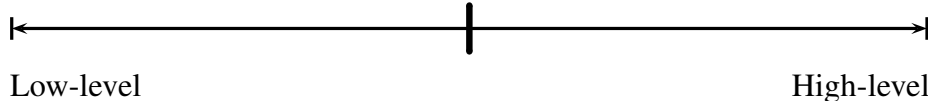


Figure 5.16: OpenCL abstraction rating

5.2.4 Concurrency

This section rates BrookGPU, CUDA and OpenCL using the Determinism, Concurrency management and Fault restriction criteria.

5.2.4.1 BrookGPU

BrookGPU is rated as follows:

Determinism Execution order of threads is not known to the programmer. However, all instances of a kernel will always finish executing before the next kernel is started. We thus rate BrookGPU as between deterministic and non-deterministic as shown on Figure 5.17.

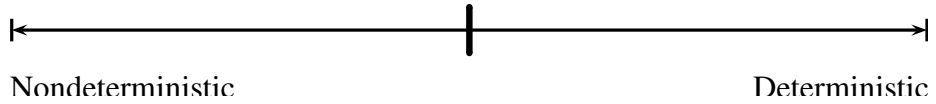


Figure 5.17: BrookGPU determinism rating

Concurrency management BrookGPU automatically parallelizes the stream processing on the GPU, without the programmer stating it explicitly and the parallelization only depends on the sizes of the stream. Thus, BrookGPU provides almost automatic concurrency as shown on Figure 5.18.

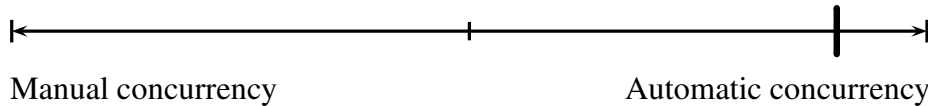


Figure 5.18: BrookGPU concurrency management rating

Fault restriction Locking and synchronization are not supported in BrookGPU, because of its stream like behavior not supporting random access to streams, i.e. two threads cannot access the same position in streams. The only exception is when writing reduction kernels, this must be explicitly stated by the developer, i.e. that the kernel is a reduction kernel by using the **reduce** keyword.

Deadlocks and live-locks can therefore not occur, and we rate BrookGPU as fault restricted as shown on Figure 5.19, since the expressiveness of BrookGPU is very limited, i.e. random access between threads, synchronization and locking mechanisms are not supported.

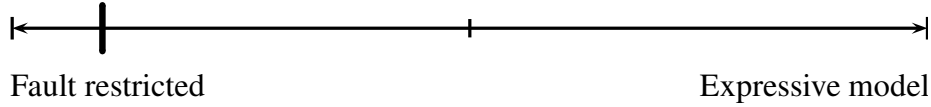


Figure 5.19: BrookGPU fault restriction rating

5.2.4.2 CUDA

CUDA is rated as follows:

Determinism On CUDA all threads are executed nondeterministically to hide memory latency, thus we cannot assume at any time that any of the other threads are done or at a certain point in the code, except when using `_syncthreads()` which acts as barrier but only for threads in the same thread block. Furthermore, all instances of a kernel can be forced to be executed before running the next kernel. Based on this we rate determinism less than fully nondeterministic as shown on Figure 5.20.

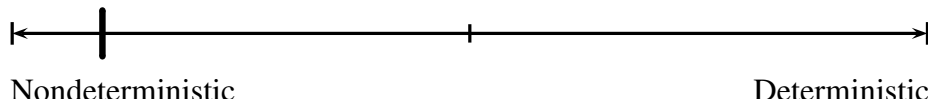


Figure 5.20: CUDA determinism rating

Concurrency management CUDA requires that the programmer explicitly defines kernels that should be executed on the device, however, the device takes charge of spawning new threads and interleaving them to hide memory latency. The developer only specifies how many threads should be in a thread block, and how many thread blocks should be in a block grid. We therefore rate CUDA as somewhat manual, as shown on Figure 5.21.

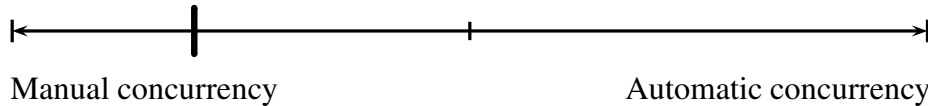


Figure 5.21: CUDA concurrency management

Fault restriction CUDA does not have locks, however, CUDA supports atomic operations, thus locks can be implemented by the programmer if needed. CUDA does not make any restriction and the programmer has to handle all risks in the program, e.g. race conditions on global and shared memory. We therefore rate CUDA as somewhat expressive, as shown on Figure 5.22.

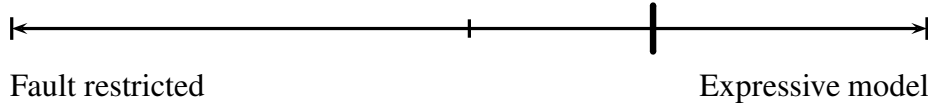


Figure 5.22: CUDA fault restriction rating

5.2.4.3 OpenCL

OpenCL is rated as follows:

Determinism As with CUDA and BrookGPU, only the execution order of kernels can be deterministic. Thus, the execution order of threads is very non-deterministic, though having barriers. Execution order of kernels does not have to be, we thus rate OpenCL as CUDA. Figure 5.23.

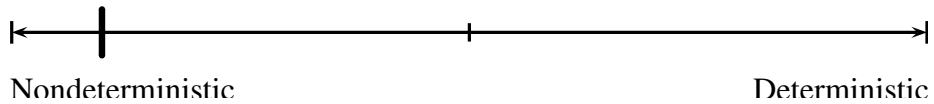


Figure 5.23: OpenCL determinism rating

Concurrency management As with CUDA, OpenCL requires that the programmer defines the number of work-items (threads) that are to be executed on the device. However, OpenCL does not require that the programmer defines the number of work-groups (thread blocks), and can optionally leave this decision to the underlying implementation, e.g. an Nvidia OpenCL implementation might use a work-group size of 8x8 work-items while another implementation might use a size of 2x2. This means that OpenCL can be regarded as having more automatic concurrency compared to CUDA and we thus rate OpenCL as shown on Figure 5.24.

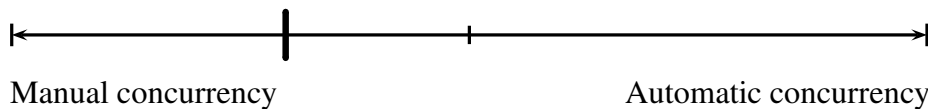


Figure 5.24: OpenCL determinism rating

Fault restriction OpenCL allows threads to access the same data in global memory, thus race conditions can occur. These must be explicitly taken care of, e.g. by making sure that no race conditions can occur by using the Atomics extension provided by newer version of OpenCL, or by making avoiding this behavior. We thus rate OpenCL the same as CUDA as shown on Figure 5.25.

5.2.5 Support

This section rates BrookGPU, CUDA and OpenCL using the Cross-platform and Maturity criteria.

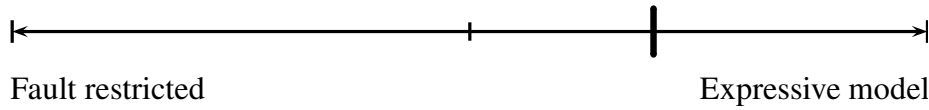


Figure 5.25: OpenCL fault restriction rating

5.2.5.1 BrookGPU

BrookGPU is rated as follows:

Cross-platform BrookGPU has several implementations, some using DirectX 9, some using OpenGL and some even using a proprietary instruction format such as CAL. This makes BrookGPU very cross-platform as shown on Figure 5.26.

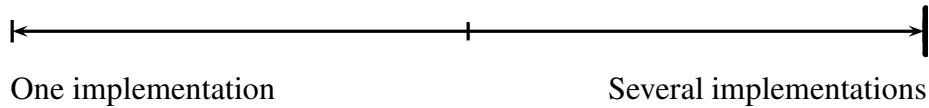


Figure 5.26: OpenCL cross-platform rating

Maturity BrookGPU has more or less been abandoned in favor of OpenCL. The original implementation, dubbed BrookGPU, is only available as source code and has to be compiled manually for programmers to use. Brook+ from ATI is however still available, but still only in an beta release, and newer versions have not been released since Marts 2009 [3]. We therefore rate BrookGPU as very immature as shown on Figure 5.27, due to BrookGPU being abandoned.

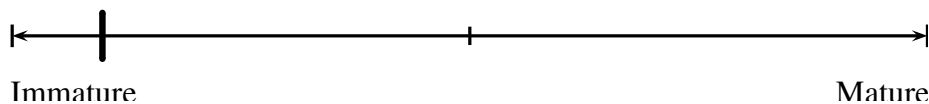


Figure 5.27: BrookGPU maturity rating

5.2.5.2 CUDA

CUDA is rated as follows:

Cross-platform CUDA has implemented Standard Development Kit (SDK)s for Windows, Linux and Mac, however, CUDA was created and is maintained by Nvidia, and only implemented on Nvidia graphics cards. We thus rate CUDA as only a little cross-platform as shown on Figure 5.28.

Maturity CUDA has been public available since 2007, and several updates and the highest compute capability is currently 2.1. CUDA comes with a range of tools, such as profilers and debuggers. We thus rate CUDA very mature, as shown on Figure 5.29.

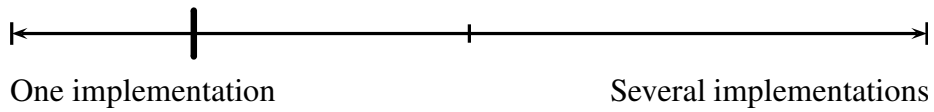


Figure 5.28: CUDA cross-platform rating

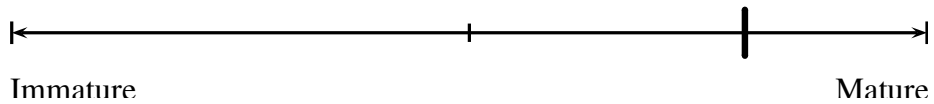


Figure 5.29: CUDA maturity rating

5.2.5.3 OpenCL

OpenCL is rated as follows:

Cross-platform OpenCL is an open specification, and implementations currently exist both for Nvidia, AMD GPUs and CPUs, making OpenCL applications very cross-platform. However, OpenCL requires architecture support for complex computations, such as branching, making OpenCL unfeasible for older generation of GPUs. We thus rate OpenCL as being less cross-platform than BrookGPU, but more so than CUDA, as shown on Figure 5.30.

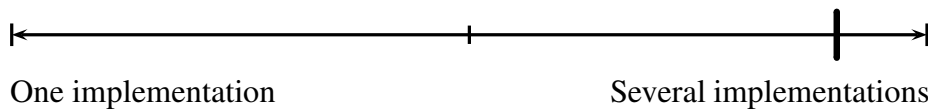


Figure 5.30: OpenCL cross-platform rating

Maturity We have used Nvidia's OpenCL implementation for CUDA enabled GPUs, while not as error prone as BrookGPU, we still find the CUDA implementation more stable. This is however getting much better, and we believe that many of the quirks that haunt the CUDA OpenCL implementation will dissipate over time. Furthermore, the first CUDA SDK was released early 2007 and the final specification of OpenCL, not implementation, was released late 2008. Based on this information we rate OpenCL less mature than CUDA, as shown on Figure 5.31.

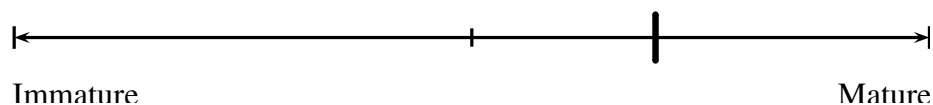


Figure 5.31: OpenCL maturity rating

5.3 Summary

To gain an overview of the platforms ratings, this section will briefly cover all the platform's ratings and compare them to each other. Lastly, we will end this chapter with an end rating of BrookGPU, CUDA and OpenCL.

Memory types CUDA and OpenCL supports private and shared memory types, and allow random access to these. BrookGPU is much more limited, and is thus much more private memory oriented. This is shown on Figure 5.32.

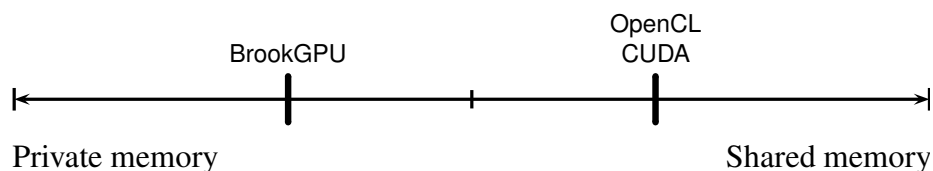


Figure 5.32: Comparison of CUDA, OpenCL and Brook memory type rating

Memory management All three platforms are rated as having some degree of manual memory management. BrookGPU is much simpler with regards to memory allocation and is thus rated less manual than OpenCL and CUDA. OpenCL allows programmers to specify private memory, while CUDA does not, thus OpenCL is rated more manual memory management than CUDA. This is shown on Figure 5.33.

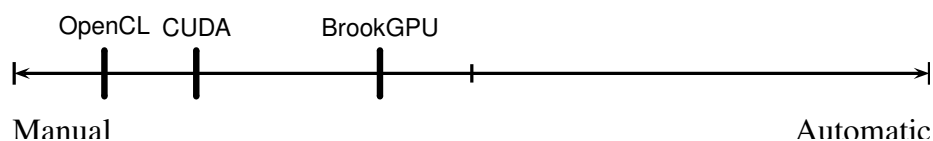


Figure 5.33: Comparison of CUDA, OpenCL and Brook memory management rating

Standard compliant None of the platforms are rated as being fully standard compliant with regards to IEEE. BrookGPU is much less standard compliant than OpenCL and CUDA, due to BrookGPU not defining which standards are followed, or not. This is shown on Figure 5.34.

Branching OpenCL and CUDA support the same level of branching, while BrookGPU is more limited in this regard. This is shown on Figure 5.35.

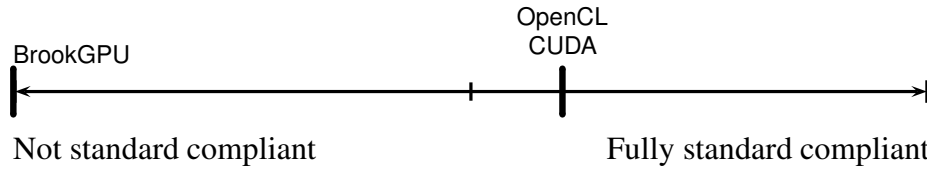


Figure 5.34: Comparison of CUDA, OpenCL and Brook standard compliant rating

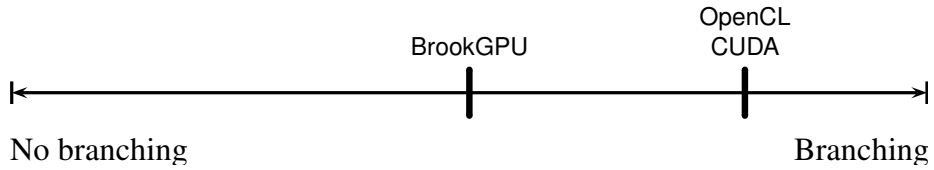


Figure 5.35: Comparison of CUDA, OpenCL and Brook branching rating

Abstraction BrookGPU is rated as being the platform having the highest abstraction, due to much of the low level functionality being abstracted away. OpenCL is more low level, since OpenCL’s API exposes functions that are abstracted away in both BrookGPU and CUDA, e.g. scheduling of a kernel is done using several low level functions. CUDA is rated as being in-between the two, since CUDA provides a language extension to the C language thereby writing kernels easier. This is shown on Figure 5.36.

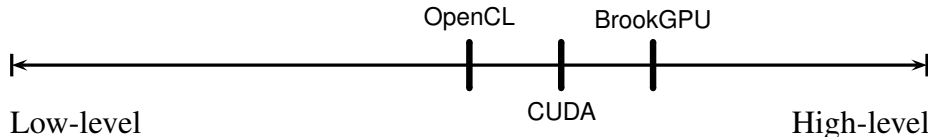


Figure 5.36: Comparison of CUDA, OpenCL and Brook abstraction rating

Determinism Due to BrookGPU being a simpler platform and BrookGPU not allowing for much random access, we rate BrookGPU as being balanced between nondeterministic and deterministic. CUDA and OpenCL are both rated as being very nondeterministic. This is shown on Figure 5.37.

Concurrency management BrookGPU leans more towards automatic concurrency than OpenCL and CUDA, due to BrookGPU being more abstract in nature. CUDA is much more manual, because the programmer is required to specify the number of threads and thread blocks used by the CUDA application. OpenCL is also manual, but leans more to the middle compared to CUDA, because OpenCL does not require that the programmer specifies the number of work-groups. This is shown on Figure 5.38.

Fault restriction Due to BrookGPU being very simplistic in expression power, fewer concurrency faults can be generated by the programmer compared to

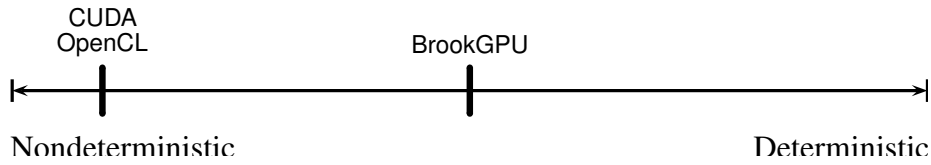


Figure 5.37: Comparison of CUDA, OpenCL and Brook determinism rating

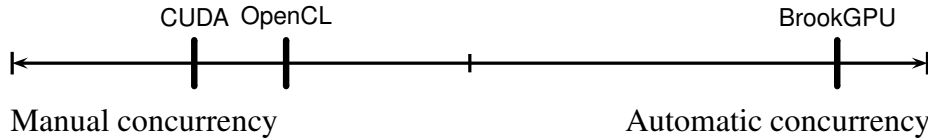


Figure 5.38: Comparison of CUDA, OpenCL and Brook standard compliant rating

OpenCL and CUDA, which support random access and synchronization mechanisms. CUDA and OpenCL are therefore much more expressive in nature, compared to BrookGPU. This is shown on Figure 5.39.

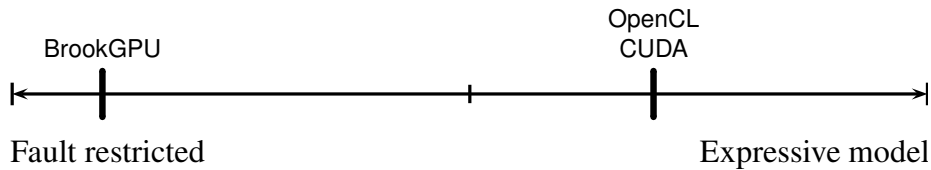


Figure 5.39: Comparison of CUDA, OpenCL and Brook fault restriction rating

Cross-platform BrookGPU has support for several types of backends, e.g. OpenGL, DirectX and CAL. This makes BrookGPU very cross-platform with regards to which GPUs can be used for GPGPU. CUDA on the other hand is only implemented on Nvidia graphics cards, thus CUDA is much less cross-platform. OpenCL is also very much cross-platform, due to the specification being open. OpenCL has higher requirements than BrookGPU, as does CUDA, making OpenCL unsuitable for GPUs that do not support random access to memory, such as scatter operations. This is shown on Figure 5.40.

Maturity CUDA is a very mature platform and is currently in use in many different applications. BrookGPU has been abandoned in favor of OpenCL, and not much support is available. OpenCL is gaining momentum in the industry, is supported by many different vendors and applications are emerging with OpenCL support. The Nvidia implementation of OpenCL is however not as mature as their CUDA implementation. This is shown on Figure 5.41.

BrookGPU vs OpenCL vs CUDA

BrookGPU Through the comparison, we see that the stream based model BrookGPU uses is simple compared to the two other platforms, e.g. BrookGPU

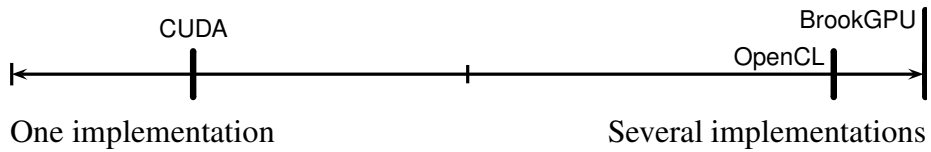


Figure 5.40: Comparison of CUDA, OpenCL and Brook cross-platform rating

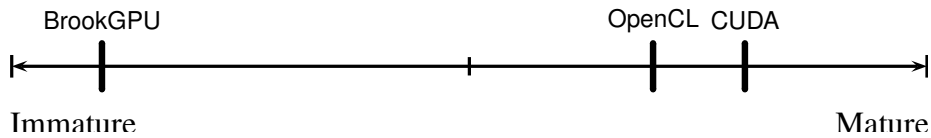


Figure 5.41: Comparison of CUDA, OpenCL and Brook maturity rating

mainly utilizes private memory and the stream model restricts the concurrent model in that only one thread can be run for each element in the stream.

OpenCL and CUDA OpenCL are CUDA very close with regard to the criteria we rate. On our scale, ranging from 0 to 10, they are rated with one in distance or at the same level, except for cross-platform, which is because OpenCL is an open standard and CUDA is a Nvidia-only platform.

This chapter concludes upon the project as a whole. First we will cover the conclusion by answering the questions that we put forth in the problem formulation. We will then give a discussion of the results that we achieved throughout the project. Lastly we will take a step back and look at where GPGPU programming is heading, and suggest projects for our next semester.

6.1 Conclusion

The aim of this project was to learn the art of GPGPU and the theories that relate to this subject. The primary motivation for this project was our 8th semester project, where we developed a ray tracer for the x86 and ARM CPUs. This ray tracer, however, turned out to run quit slowly on the CPU, and we speculated at that time that it might be possible to achieve a substantial speed up by using the GPU to do most of the computations. The reason that we thought we could achieve a speed up, was that GPUs have a much higher theoretical performance than equivalent generation of CPUs, especially when looking at performance per dollar.

In this project, we first performed an analysis of the difference between CPUs and GPUs, made an in depth analysis of G80 GPU architecture, and which problems are well suited for GPU execution. Furthermore we have looked at BrookGPU, OpenCL and CUDA and which tools are available when doing GPGPU programming.

In order to gain experience with GPGPU programming, we implemented two applications: Boids in Brook+ and OpenCL, and a ray tracer in CUDA. Both of these applications were benchmarked against CPU implementations, to determine if any speed up was gained. Lastly, we compared the languages to find the properties of the languages.

Recall that we asked several questions in Section 1.2, these will now be answered. Afterwards, we will talk about the practical experience that we have gained through this project.

6.1.1 Problem Formulation

As described in Section 1.2 we asked several questions regarding GPGPU programming and set forth to answer these by doing a broad analysis of GPGPU related theories and practices. In this section, we will answer the questions that we asked in Section 1.2

How does the hardware architecture of a GPU look like, compared to a CPU?

A GPU differentiate from the common x86 CPU in several ways: *a)* A GPU is more focused on throughput while CPUs are more focused on latency. *b)* A GPU has many relatively weak and simple cores while the CPU has less but much more complex and powerful cores. *c)* A GPU features several types of memory, such as private, shared, global and texture memory. The CPU typically only has main memory with L1 and L2 caches sitting in front. *d)* A GPU rely on latency hiding by having several thousands of threads ready to be executed whenever a high latency instruction is encountered. The CPU uses many transistors on latency reducing features, such as branch prediction and cache memory, which are traded away for more FLOPS on the GPU. *e)* A GPU often make use of SIMD to fit more ALUs on a die e.g. each SM on the G80 can be viewed as a 32 way SIMD processor. *f)* CPU source code is typically compiled to the native instruction set of the CPU, e.g. x86 instructions. This is seldom the case for GPUs, instead the kernels are compiled to intermediate assembly code, such as PTX, and just in time compiled at runtime.

What are the characteristics of well suited problems for GPU execution?

GPUs are primarily used to render 3D graphics. The problems that can be solved on the GPU therefore have to exhibit the same properties as 3D graphics problems, to be best suited for GPU execution. Some of these properties are: *a)* 3D rendering is a highly parallel problem, e.g. execution of shaders can be done in parallel. *b)* Memory access is very spatial, this is for example seen when accessing texture where the texture cache is optimized for spatial locality. *c)* Throughput is more important than latency, i.e. a problem that must be solved with low latency demands is not well suited for GPU execution. *d)* Memory latency is less important, but high throughput is important. The memory bandwidth of GPUs are much higher than their CPU counterparts, but so is the memory latency.

These properties were defined in Section 2.4, along with an analysis of the performance between an Nvidia GPU and an Intel CPU. The GPU was found to be 2.5 times faster on average, with only a small number of problems that exhibited better performance on the CPU. Even though the theoretical performance of the CPU was 102.4 gigaFLOPS and the theoretical performance of the GPU was 933.1 gigaFLOPS.

Which GPGPU APIs/Languages exist, and how does one use them?

We have analyzed and compared three general purpose platforms: CUDA, OpenCL and BrookGPU. We see that BrookGPU has a simple memory and concurrency model, but is also very restricted, where CUDA and OpenCL are much more advanced and expressive.

The analysis in Section 2.5 describes the features in each platform and shows examples of the code. In Chapter 5 we setup criteria for comparison of the

three platforms and compared them. We found that OpenCL and CUDA are very similar with regards to their memory model, abstraction and concurrency management. CUDA is however a more mature platform, while OpenCL is much more cross-platform capable.

Which tools currently exists to help developers with GPGPU programming? Throughout Section 2.7, we analyzed several tools which can help with GPGPU programming.

We found debuggers, such as CUDA-GDB and gDEDebugger CL, which allow debugging on kernel level in CUDA and OpenCL applications. Unfortunately, the Tesla C870 does not support debuggers because of its low compute capability of 1.0, but new generations of graphics cards with higher compute capability do support debugging.

Profiling tools were also found which can help finding bottlenecks in programs, by analyzing the resource usage of kernels and their memory access patterns. We were however unable to find any testing tools, i.e. unit testing and integration testing tools.

We were also unable to find any tools to help in the development of BrookGPU applications.

Is it possible to utilize multiple GPUs in parallel for GPGPU purposes? Using techniques such as SLI and CrossFire While analyzing CUDA and OpenCL in Section 2.6.2 and Section 2.6.3, we discovered that it is indeed possible to utilize multiple graphics cards for GPGPU purposes. However, automatic utilizing of multiple GPUs is currently not possible in CUDA or OpenCL, one has to explicitly write the application such that it can utilize multiple GPUs. This requires more work by the programmer, especially with regards to CUDA, which requires each device having its own CPU thread.

Implementations To gain experience with GPGPU programming we have implemented two applications: Boids and ray tracer. The Boids application was done in OpenCL and Brook+ and focused on data structure optimizations. Different versions of the Boids application were made, one based on the optimizations proposed in [64] which does not follow the Boids model entirely, while another was made which does. And finally, two implementations were made using a combination of the optimizations to allow for a comparison of performance.

The ray tracer was done in CUDA and focused on memory optimizations. Two version of the ray tracer were made and benchmarked, one which was optimized using texture and shared memory and one which only used global and local memory.

We also implemented a CPU C++ Boids application, and improved the ray tracer from our 8th semester project with multi-threading support.

6.1.2 Practical experience

We have gained practical experience due to our choice of implementing a ray tracer and Boids application, which we might not have gained if we had chosen only to perform an analysis of the GPGPU theories and practices.

We learned that a Nvidia Graphics Card is required to provide display output when using the Nvidia C870 Tesla card.

We gained practical experience in developing applications in CUDA C, OpenCL and Brook+, and by benchmarking our applications we gained experience in which optimizations work well on GPUs and which do not.

We have shown that some applications are not well suited for execution on GPUs. This is shown by the complex Boids applications which in most cases achieve worse performance on the GPU than the CPU. The speed ups range from 0.2 to 1.3 when comparing the OpenCL implementation to the CPU implementation.

On the other hand, we have shown that some application can achieve a substantial speed up on the GPU. The simple Boids application in the OpenCL implementation achieves a speed up of 4, while the Brook+ implementation achieves a speed up of around 30, in comparison to the CPU implementation.

We were also able to confirm that a ray tracer can be implemented on a GPU with a significant speed up compared to the CPU. In the optimized version, we saw a speed up of 8 which is close to the theoretical maximum of the GPU, when not using multiply-add. The unoptimized version achieved a speed up of 4, thus also showing that using on chip memory on the GPU can lead to a substantial increase in performance.

6.2 Discussion

In this section we will discuss the implementation verification, i.e. to which degree it was done, the benchmark results and give reason for abnormalities. Lastly we will consider the chosen references and for their validity or missing thereof.

6.2.1 Implementation Verification

We have implemented several algorithms on several platforms: Boids on OpenCL, Brook+ and in C++, a ray tracer in CUDA and used a C++ version from our previous project, which was configured to generate the same result. However we do not formally concern ourselves with verifying the correctness of these implementations. This poses a uncertainty for the results, as a wrongly implemented algorithm may perform better but yield the wrong result.

To increase our confidence in the implementations correctness we compared their output, e.g. by taking screenshots after k iterations and compare how closely the images resemble, in all cases we found these comparisons had sufficient resemblance, e.g. in the Boids applications the actors moved in the same patterns and exhibited the separation, alignment and cohesion properties. Furthermore we set up function requirements which should be followed when implementing the algorithm and benchmark setups which defined how the configuration of the algorithm should be, when running the implementation, such that this would be persistent across the implementations, allowing to argue for the validity of our results.

The benchmarks are carried out on both CPU implementations and GPU implementations, to evaluate the effectiveness of writing GPU-programs. However, the fairness of this comparison can be argued to be little, as we chose

problems that were well suited for GPU execution, i.e. adhere to the properties described in Section 2.4. Furthermore we used a lot of time optimizing the ray tracer for texture and shared memory, though this time usage can be explained by us being inexperienced in the platforms and thus requiring more development time. The CPU implementation of Boids was a backport of the OpenCL implementation, and thus not optimized in any way, apart from being partly parallelize. This may explain that some of the implementations gave a relatively large speed up on the GPU.

6.2.2 Benchmarks

As discussed in Chapter 4, we do not see the theoretical maximum speed up in all benchmarks. In some cases, the GPU implementation is even slower than the CPU implementation, and in other cases, we see several factors of speed up.

Boids Benchmark With regards to the Boids Section 4.1, we see that a performance increase has been achieved on both applications. The simple version of the Boids application achieved a speed up of 30, compared to the CPU, when using the Brook+ implementation. This might be due to the Brook+ version automatically uses texture memory. Also, the Brook+ implementation was executed on an ATI Radeon 2900XT card, which has higher memory bandwidth, thus improving the performance of the application. The speed-up of 30 is actually higher than the theoretical maximum. This might be due to the ATI Radeon 2900XT having higher memory bandwidth and that the CPU implementation is not be optimized, due to it being a back-port of the OpenCL version.

The OpenCL version achieved much less of a speed up on the Tesla C870, which might be due to the absence of any memory optimizations, due to uncoalesced memory access or because the occupancy rate of the kernels are too low to effectively carry out latency hiding.

With regards to the complex version of Boids, the GPU version generally achieve worse performance then the CPU, which might be due to private memory in Nvidia's OpenCL implementation is off-chip and that the pruned neighbor search algorithm has many uncoalesced memory accesses.

The Boids benchmarks are done on two different cards, as the Tesla card did not support Brook+. The Nvidia Tesla C870 for the OpenCL implementation and a ATI Radeon 2900XT for the Boids+ implementation. The two cards have different architecture, which make the results difficult to compare, e.g. the 2900XT has about 100 GB/sec where the C870 only has about 76 GB/s bandwidth. Instead we compare the theoretic achievable speed-up with the speed-up actually found in the implementations. This however relies on FLOPS and thus does not account for memory bandwidth.

Ray Tracer With regards to the ray tracer application, we see a speedup in both the unoptimized and optimized implementations. The optimized version is roughly twice as fast as the unoptimized version. this is due to the use of the texture memory cache, which decrease the bandwidth demands of the application, and the use of shared memory for the intersections, which is faster on-chip memory compared to the use of local memory.

6.2.3 Be Scientific

Recall from Section 1.2 that we must be scientific in this project, which means that our choices must be reasonable and that our sources must be scientific. We have achieved this by mainly relying on peer-reviewed publications and sources from companies, and organizations, which are the creator of the product, e.g. Nvidia is often used as source for sections concerning CUDA and for OpenCL the “Khronos OpenCL Working Group” is frequently cited, as they are the creators of the OpenCL standard. We argue that these sources are reliable as they either are peer-reviewed and thus checked for errors or is the creator of the product in question and therefore know how it is implemented and functions. A company may benefit from overstating a product’s abilities, therefore we are sceptical when concerned with praises of products given by the company producing the product. Furthermore we mainly use documents describing with hard facts, such as documentation of an API set. Sources are mainly used when reasoning about a choice, e.g. the choice of using texture and shared memory in the ray tracer was due to Nvidia stating that bandwidth and latency could be improved by using these.

We also have a few other sources, such as Wikipedia and PCWorld. Wikipedia has sparked a lot of dispute as some argue that because it is editable by all, it may lead to incorrect articles and allow us to change the article such that it suits the point we are trying to state. Others argue that it is, to some degree, trustworthy as it is peer-reviewed, although not necessarily in a scientific community. This has lead to a professor banning Wikipedia and Google as research sources for her students [73].

Some cases removes the possibility for using other sources, e.g. in Section 2.6.2.4 we cite the Wikipedia page about GeForce 8 ([77]), as we were unable to find the release date of the Nvidia 8800 GTS in any official Nvidia documents or scientific publications. The date is used to reason that CUDA has evolved a lot since the first release. We only use these sources like Wikipedia when the correctness is of less importance for the statement we are trying to give, e.g. CUDA has evolved a lot since the first release, even though Nvidia 8800 GTS was released years earlier than the cite stated.

We also cite forum posts, however these forum posts were made by a researcher at Nvidia and contained information about a Nvidia tool and dynamic memory allocation on their GPUs. We therefore argue that this information is trustworthy.

6.2.4 Our Thoughts

At the beginning of this project we had no prior experience with GPGPU, except a little experience with graphics programing in OpenGL and DirectX. Coming into the project, we had expectations that we could achieve very high speed ups, since we had seen publications claiming 100X - 1000X speed ups. However, as we progressed with the project we found that such claims were unrealistic, since they were often due to the use of unoptimized CPU implementations.

At the beginning of the project, we also thought that GPGPU would require much more graphics like programing than it does. It turns out that GPGPU is more high level then we expected, though it still requires the use of C, which might be scary for programmers used to only doing programming in more high

level languages, such as Java, Haskell and C#. We were also pleasantly surprised by the amount of tools available to help programmers, even though we would have liked to see more help in optimizing the code and testing tools.

Even though the speedups of 100X - 1000X have to some extent been dismissed, we still see a substantial speed up in some applications. Therefore, we still think that GPGPU is beneficial, especially when concerning performance vs cost.

6.3 Future Trends

This section looks at where the future trends of GPGPU programming are heading, by stating four trends that we have witnessed while doing this project, and citing sources which support these trends.

6.3.1 GPGPU Programming is Becoming More Powerful

Looking at the Tesla C870 GPU that we have access to, we see that it is limited with regards to which programming languages are supported, the abstraction these languages provide, and their expressive power. Looking beyond the Tesla we see that the architectures of new GPUs are getting more advanced, with increasing performance especially in the area of coalesced memory access, and new instructions that provides more expressive power to the programmer, e.g. programmers are able to utilize atomic operations on devices with compute capability 1.1.

Fermi Architecture [53]

Looking at the state of the art Nvidia Fermi architecture, used in graphics cards such as the high-end Tesla C2050 and mid-end GT420, we see that many new performance increasing features have been added, such as L1 and L2 cache support, support for concurrent kernels, two warp schedulers, etc.

In addition, the Fermi architecture supports PTX 2.0 which introduces several new features not found in PTX version 1.x, which is used in 1.x compute capable devices. New features include full IEEE 32-bit floating point precision, 64-bit addressing, better pointer support, function pointers, virtual functions as seen in C++, exception support, new/delete operators for dynamic memory management.

All these features allow more expressiveness compared to older generations of GPUs, and allow languages with increasing abstraction. Nvidia has already utilized many of these features in their new CUDA SDK version 3.2, where they added support for C++ but with a few limitations, e.g. virtual functions are not yet supported but will be added in the near future.

6.3.2 GPGPU Programming is Becoming Easier

The increasing interest of GPGPU programming has spawned numerous publications on this subject. Many of the publications deal with the question of how GPGPU programming can be done easier.

One such publication [33] from November 2010 proposes an OpenMP based programming interface called OpenMPC. This interface provides a high level

abstraction from the CUDA programming model. OpenMPC automatically optimizes the code and performs auto-tuning, i.e. figures out the thread block sizes, to increase performance. Programs written using this abstraction achieve 88% of the performance of their hand written counterparts, thus there is room for improvement.

Another publication [37] from September 2010 describes a language extension, called Nikola, which supports computations on arrays and is embedded into the Haskell language. A compiler backend is implemented targeting CUDA code, thus allowing array computations on the GPU. Nikola automatically marshals data to and from the GPU, automatically creates and manages memory, automatic loop parallelization, and more. Their benchmarks show that the performance overhead of Nikola is negligible, when using static compiling, and negligible on large data sets, when using runtime compilation.

6.3.3 GPU and CPUs are merging

Most of the GPGPU publications that we have seen, utilize some discrete graphics card, i.e. a standalone graphics card that is installed in the computer. The reason for this is that the GPUs on discrete graphics cards are much more powerful compared to their integrated counterparts, which have much less processing power and video memory. Intel and AMD are however in the process of merging the GPU with the CPU, by installing a GPU directly into the architecture of the CPU, thus increasing the processing power of the CPU.

The i5-540m CPU from Intel integrates a 45nm GPU on its chip, thereby allowing GPU accelerated video, such as full HD movie playback, and accelerated 3D graphics, such as in computer games and CAD applications. Albeit, the GPU is much weaker than some of the discrete solutions available, it is however much more powerful than many of the GPUs that are integrated on the chipset. Also, moving the GPU from the chipset, and discrete solution, to the CPU, allows for more efficient performance which leads to better battery life in notebooks, and physical space, which leads to thinner notebooks. Lastly, moving the GPU closer to the CPU, and thus closer to main memory, allows for much faster memory transfers between host and GPU, thus allowing faster synchronization between the CPU and GPU. Thus is very useful for some GPGPU applications, e.g. applications that issue many memory transfers, and kernel invocations between host and device. [40]

AMD have also come up with a CPU chip with integrated GPU, called AMD Fusion. AMD Fusion is an Accelerated Processing Unit (APU), which integrate the general purpose computation design of the CPU, with the more vector processing design of the GPU into one package. As with the i5, moving the GPU closer to the CPU and main memory leads to lower latency to and from memory. AMD provides OpenCL support for their APU, meaning that OpenCL applications can make use of this new architecture from the start. [5]

6.3.4 GPGPU is Gaining Momentum

It appears that GPGPU acceleration is a powerful tool for current and future applications. This is evident looking at how companies such as Intel and AMD are pushing to include GPU like components in their CPUs.

Most applications however do not utilize the potential that lies in GPU accelerating their computations. [5, 7] gives some examples of companies that are pushing GPU support onto their products. Adobe have implemented GPU support when decoding video streams in their Flash Player 10.1. This allows more efficient video playback on GPU enabled systems as the CPU power consumption is reduced, thereby extending battery life.

[5, 7] also mentions a recently started up company at Silicon Valley, whom are working on a software solution that can clean up video files using GPUs, thereby compensate for noise, pixilation, graininess, poor focus, and more.

A publication [25] from October 2010 presents a framework called Packet-Shader, which can utilize the GPU and CPU to perform high throughput packet switching in networks. Their results show that they achieve 28 Gbps of throughput using two consumer GPU and a Quad core CPU, compared to 6 Gbps using the CPU alone. Also, the packet generator that they use can maximum generate 28 Gbps worth of packets, meaning that their benchmark results are capped at 28 Gbps and that their solution might support higher throughput than can be measured using their equipment.

Amazon announced in November that they now provide clusters of GPUs in their compute cloud. This allows organizations to offload their compute intensive operations to the Amazon cloud. They provide a subscription based services and on-demand service, i.e. pay by the hour with no obligations. [21]

6.3.5 Summary

We have seen that GPGPU programming is becoming a more powerful tool with the advent of new GPU architectures and support for more languages, such as C++. Also, even though GPGPU programming is becoming more powerful, we have also witnessed that GPGPU programming is becoming easier since new architectures have more memory, and much more forgiving with regard to uncoalesced memory access, and new abstractions have been introduced thus making GPGPU programming higher level. Also, we have seen the trend that GPUs and CPUs are merging into a APU, with decreased latency between the GPU and the CPU components compared to current technologies, and we believe that at some point this will be the norm. Big players such as Amazon now have GPU powered compute clusters that can be rented by developers for a fee, and several papers have been published on the context of GPGPU, we therefore argue that GPGPU is currently gaining momentum and not only in the scientific community.

6.4 Future Development

In this section we will end this project by talking about some of the ideas that we have for a 10th semester project.

Automatic GPU Parallelization Many programmers today are proficient in a high level programming language, such as C#, with garbage collection support and advanced exception handling. We argue that it is hard for these programmers to utilize the power of the GPU, since they are required to write

and invoke kernels on the GPU, while managing the different memory types explicitly.

CUDA and OpenCL bindings exist for C#, but these bindings provide a very thin abstraction on-top of CUDA, i.e. programmers still have to think in kernels and memory types. Instead, we propose an extension to C# that automatically parallelizes parts of the C# code, and moves these parts to the GPU. Thus, the programmer just has to annotate which part of the code they wish to move to the GPU, the low level details such as allocating and copying memory, selecting the best thread-block size, writing the kernel, invoking the kernel, etc. are handled by the underlying system.

Testing Support When performing our analysis of GPGPU tools we did not come across any tools that relate to unit testing and integration testing. Testing is an important part of software development, especially when dealing with critical systems, and this is currently lacking for GPGPU applications on the device side, i.e. unit testing of kernels is not possible.

We propose a tool that can perform unit and integration testing of kernels, and that can automatically create stubs and mock objects when required. Also, this tool would have some form of IDE support, e.g. Visual Studio 2010 using parallel Nsight.

Performance Analysis Tool CUDA and OpenCL both have profilers available that can be used to measure the performance of kernels. The profiler is however quite limited with regards to giving tips on how to improve the performance of a given kernel. Currently, programmers can run the profiler on an OpenCL or CUDA program and get information on how well the program used the resources of the GPU, but not exactly where the resources were used or which bottlenecks are present.

We propose a tool that given an OpenCL or CUDA kernel, is able to measure exactly where the bottlenecks of the code are and give tips on how to improve execution performance. We claim such a tool will help programmers, such as us, write more efficient kernels.

Bibliography

- [1] Yannick Allusse, Patrick Horain, Ankit Agarwal, and Cindula Saipriyadarshan. Gpucv: A gpu-accelerated framework for image processing and computer vision. In *ISVC '08: Proceedings of the 4th International Symposium on Advances in Visual Computing, Part II*, pages 430–439, Berlin, Heidelberg, 2008. Springer-Verlag.
- [2] AMD. ATI Stream SDK v1.4-beta System Requirements. <http://developer.amd.com/archive/gpu/ATIStreamSDKv1.4Beta/pages/ATIStreamSystemRequirements.aspx#cards>. Last seen: 22th of December 2010.
- [3] AMD. Ati stream software development kit (sdk) v1.4-beta. <http://developer.amd.com/archive/gpu/ATIStreamSDKv1.4Beta/Pages/default.aspx#five>. Last seen online 14th of December 2010.
- [4] AMD. *User Guide - ATI Stream Computing*. AMD, april 2009.
- [5] AMD. Amd fusion family of apus: Enabling a superior,immersive pc experience. http://sites.amd.com/us/Documents/48423B_fusion_whitepaper_WEB.pdf, 2010. [Online; accessed 16-December-2010].
- [6] Dana H. Ballard and Christopher M. Brown. *Computer Vision*. Prentice Hall, 1982.
- [7] Daniel J. Bernstein, Tien-Ren Chen, Chen-Mou Cheng, Tanja Lange, and Bo-Yin Yang. Ecm on graphics cards. In *EUROCRYPT '09: Proceedings of the 28th Annual International Conference on Advances in Cryptology*, pages 483–501, Berlin, Heidelberg, 2009. Springer-Verlag.
- [8] BrookGPU. BrookGPU: Getting Started. <http://graphics.stanford.edu/projects/brookgpu/start.html>. Last seen: 27th September 2010.
- [9] BrookGPU. BrookGPU System Architecture. <http://graphics.stanford.edu/projects/brookgpu/arch.html>. Last seen: 26th September 2010.

- [10] Benjamin M. Brosgol. A comparison of ada and java as a foundation teaching language. *Ada Lett.*, XVIII(5):12–38, 1998.
- [11] Niel Brown. Boids simulation: Part 5. <http://chplib.wordpress.com/2009/09/16/boids-simulation-part-5/>, 2009. [Online; accessed 24-December-2010].
- [12] Ian Buck, Tim Foley, Daniel Horn, Jeremy Sugerman, Kayvon Fatahalian, Mike Houston, and Pat Hanrahan. Brook for gpus: stream computing on graphics hardware. *ACM Trans. Graph.*, 23(3):777–786, 2004.
- [13] Seymour Cray. http://en.wikiquote.org/wiki/Seymour_Cray. His response when asked his opinion on clusters.
- [14] Birthe Damberg and Anders Mørk Hansen. *A Study in Concurrency*. Aalborg University, 2007.
- [15] Qianqian Fang and David A. Boas. Monte carlo simulation of photon migration in 3d turbid media accelerated by graphics processing units. *Opt. Express*, 17(22):20178–20190, Oct 2009.
- [16] Kayvon Fatahalian and Mike Houston. A closer look at gpus. *Commun. ACM*, 51:50–57, October 2008.
- [17] Tim Foley and Jeremy Sugerman. Kd-tree acceleration structures for a gpu raytracer. In *HWWS '05: Proceedings of the ACM SIGGRAPH/EUROGRAPHICS conference on Graphics hardware*, pages 15–22, New York, NY, USA, 2005. ACM.
- [18] Center for Biomedical Imaging Harvard. Monte carlo photon transport. <http://www.nmr.mgh.harvard.edu/DOT/resources/tmcimg/>, 2004. [Online; accessed 30-November-2010].
- [19] Kirill Garanzha and Charles Loop. Fast ray sorting and breadth-first packet traversal for gpu ray tracing. *Computer Graphics Forum*, 29(2):289–298, 2010.
- [20] GPGPU.org. About GPGPU.org. <http://http://gpgpu.org/about>. Last seen: 20th october 2010.
- [21] GPGPU.org. Amazon announces gpus for cloud computing. <http://gpgpu.org/2010/11/22/amazon-ec2-gpu>, 2010. [Online; accessed 29-November-2010].
- [22] Khronos OpenCL Working Group. *The OpenCL Specification, version 1.1, Document revision: 36*. Khronos, 2010.
- [23] GURU3D. Jetway Radeon HD 2900 XT Crossfire review . <http://www.guru3d.com/article/jetway-radeon-hd-2900-xt-crossfire-review/> 2. Last seen: 3th of January 2011.
- [24] Dominik Göddeke. GPGPU::Basic Math Tutorial. <http://www.mathematik.uni-dortmund.de/~goeddeke/gpgpu/tutorial.html>. Last seen: 20th September 2010.

- [25] Sangjin Han, Keon Jang, KyoungSoo Park, and Sue Moon. Packetshader: a gpu-accelerated software router. In *Proceedings of the ACM SIGCOMM 2010 conference on SIGCOMM*, SIGCOMM '10, pages 195–206, New York, NY, USA, 2010. ACM.
- [26] Chuntao Hong, Dehao Chen, Wenguang Chen, Weimin Zheng, and Haibo Lin. Mapcg: writing parallel program portable between cpu and gpu. In *Proceedings of the 19th international conference on Parallel architectures and compilation techniques*, PACT '10, pages 217–226, New York, NY, USA, 2010. ACM.
- [27] Søren Alsbjerg Hørup, Søren Andreas Juul, Benjamin Krogh, and Henrik Holtegaard Larsen. *Distributed Rendering for Mobile Devices - Using Ray Tracing*. Aalborg University, 2010.
- [28] Intel. Intel xeon processor. <http://www.intel.com/support/processors/xeon/sb/CS-020863.htm>, 2010. [Online; accessed 29-November-2010].
- [29] Mendel Rosenblum Jayanth Gummaraju. Stream Processing in General-Purpose Processors. <http://userweb.cs.utexas.edu/users/skeckler/wild04/Paper14.pdf>. Last seen: 20th of September 2010.
- [30] Khronos OpenCL Working Group. *The OpenCL Specification, version 1.0.29*, 8 December 2008.
- [31] David B. Kirk and Wen-mei W. Hwu. *Programming Massively Parallel Processors: A Hands-on Approach*. Morgan Kaufmann, 1 edition, February 2010.
- [32] langpop. Programming language popularity. <http://langpop.com/>. [Online; accessed 27-November-2010].
- [33] Seyong Lee and Rudolf Eigenmann. Openmpc: Extended openmp programming and tuning for gpus. In *Proceedings of the 2010 ACM/IEEE International Conference for High Performance Computing, Networking, Storage and Analysis*, SC '10, pages 1–11, Washington, DC, USA, 2010. IEEE Computer Society.
- [34] Victor W. Lee, Changkyu Kim, Jatin Chhugani, Michael Deisher, Daehyun Kim, Anthony D. Nguyen, Nadathur Satish, Mikhail Smelyanskiy, Srinivas Chennupaty, Per Hammarlund, Ronak Singhal, and Pradeep Dubey. Debunking the 100x gpu vs. cpu myth: an evaluation of throughput computing on cpu and gpu. *SIGARCH Comput. Archit. News*, 38:451–460, June 2010.
- [35] Erik Lindholm, John Nickolls, Stuart Oberman, and John Montrym. Nvidia tesla: A unified graphics and computing architecture. *IEEE Micro*, 28:39–55, 2008.
- [36] David Luebke and Greg Humphreys. How gpus work. *Computer*, 40(2):96–100, 2007.
- [37] Geoffrey Mainland and Greg Morrisett. Nikola: embedding compiled gpu functions in haskell. In *Proceedings of the third ACM Haskell symposium on Haskell*, Haskell '10, pages 67–78, New York, NY, USA, 2010. ACM.

- [38] Microsoft. Direct3D Architecture (Direct3D 9). [http://msdn.microsoft.com/en-us/library/bb219679\(VS.85\).aspx](http://msdn.microsoft.com/en-us/library/bb219679(VS.85).aspx). Last seen: 1th of December 2010.
- [39] Microsoft. Timeout detection and recovery of gpus through wddm. http://www.microsoft.com/whdc/device/display/wddm_timeout.msp. [Online; accessed 13-December-2010].
- [40] PCWorld Nate Ralph. Intel's 2010 'arrandale' laptop cpus: Core i5-540m impressions. http://www.pcworld.com/article/185857/intels_2010_arrandale_laptop_cpus_core_i5540m_impressions.html, 2010. [Online; accessed 16-December-2010].
- [41] John Nickolls, Ian Buck, Michael Garland, and Kevin Skadron. Scalable parallel programming with cuda. *Queue*, 6(2):40–53, 2008.
- [42] Nvidia. CUDA-GDB. <http://developer.nvidia.com/object/cuda-gdb.html>. Last seen: 14th of November 2010.
- [43] Nvidia. CUDA-GDB (NVIDIA CUDA Debugger). http://developer.download.nvidia.com/compute/cuda/3_0/toolkit/docs/CUDA_GDB_v3-0.pdf. Last seen: 14th of November 2010.
- [44] Nvidia. CUDA Occupancy Calculator. <http://forums.nvidia.com/index.php?showtopic=31279>. Last seen: 17th of November 2010.
- [45] Nvidia. Dynamic Global Memory Allocation ? faulty documentation - malloc within Kerne. <http://forums.nvidia.com/index.php?showtopic=189389>. Last seen: 30th of December 2010.
- [46] Nvidia. Gpu computing technical brief version 1.0.0. http://www.nvidia.com/docs/I0/43395/Compute_Tech_Brief_v1-0-0_final__Dec07.pdf, 2007. [Online; accessed 29-November-2010].
- [47] Nvidia. Nvidia tesla gpu computing solutions for hpc. http://www.nvidia.com/docs/I0/43395/tesla_product_overview_dec.pdf, 2007. [Online; accessed 29-November-2010].
- [48] Nvidia. Nvidia tesla c870 gpu computing processor board. http://www.nvidia.com/docs/I0/43395/C870-BoardSpec_BD-03399-001_v04.pdf, 2008. [Online; accessed 29-November-2010].
- [49] Nvidia. Tesla c1060 installation guide. http://www.nvidia.com/docs/I0/56484/NV_C1060_UserManual_Guide_FINAL.pdf, 2008. [Online; accessed 29-November-2010].
- [50] Nvidia. Blog: Nvidia ntersect - “gpus are only up to 14 times faster than cpus” says intel. <http://blogs.nvidia.com/ntersect/2010/06/gpus-are-only-up-to-14-times-faster-than-cpus-says-intel.html>, 2010. [Online; accessed 30-November-2010].
- [51] Nvidia. Cuda-gdb with emacs. <http://developer.download.nvidia.com/pix/CUDA/cuda-gdb.png>, 2010. [Online; accessed 14-November-2010].

- [52] Nvidia. cudamallocpitch. http://developer.download.nvidia.com/compute/cuda/2_2/toolkit/docs/online/group__CUDA__MEMORY_g80d689bc903792f906e49be4a0b6d8db.html, 2010. [Online; accessed 30-November-2010].
- [53] Nvidia. Next generation compute architecture. http://www.nvidia.com/content/PDF/fermi_white_papers/NVIDIA_Fermi_Compute_Architecture_Whitepaper.pdf, 2010. [Online; accessed 16-December-2010].
- [54] Nvidia. *NVIDIA CUDA C Programming Guide (Version 3.1.1)*. Nvidia, 2010.
- [55] Nvidia. *NVIDIA CUDA C Programming Guide (Version 3.2)*. Nvidia, 2010.
- [56] Nvidia. Nvidia opencl best practices guide. http://www.nvidia.com/content/cudazone/CUDABrowser/downloads/papers/NVIDIA_OpenCL_BestPracticesGuide.pdf, 2010. [Online; accessed 3-December-2010].
- [57] Nvidia. Nvidia visual profiler. <http://developer.nvidia.com/object/visual-profiler.html>, 2010. [Online; accessed 16-November-2010].
- [58] Nvidia. Parallel nsight. <http://www.nvidia.com/object/parallel-nsight.html>, 2010. [Online; accessed 14-November-2010].
- [59] Nvidia. Parallel nsight - features. <http://www.nvidia.com/object/parallel-nsight-features.html>, 2010. [Online; accessed 14-November-2010].
- [60] J.D. Owens, M. Houston, D. Luebke, S. Green, J.E. Stone, and J.C. Phillips. Gpu computing. *Proceedings of the IEEE*, 96(5):879–899, May 2008.
- [61] John Owens. Eec 277: Graphics architecture. <https://smartsite.ucdavis.edu/access/content/group/41cdf6c8-0223-40c9-a69a-1543d7ea2575/lectures/1-intro.pdf>. Last accessed: 3th of january, 2011.
- [62] Jacopo Pantaleoni, Luca Fascione, Martin Hill, and Timo Aila. Pantaray: fast ray-traced occlusion caching of massive scenes. In *SIGGRAPH '10: ACM SIGGRAPH 2010 papers*, pages 1–10, New York, NY, USA, 2010. ACM.
- [63] Steven G. Parker, James Bigler, Andreas Dietrich, Heiko Friedrich, Jared Hoberock, David Luebke, David McAllister, Morgan McGuire, Keith Morley, Austin Robison, and Martin Stich. Optix: a general purpose ray tracing engine. In *SIGGRAPH '10: ACM SIGGRAPH 2010 papers*, pages 1–13, New York, NY, USA, 2010. ACM.
- [64] Erick Baptista Passos, Mark Joselli, Marcelo Zamith, Esteban Walter Gonzalez Clua, Anselmo Montenegro, Aura Conci, and Bruno Feijo. A bidimensional data structure and spatial optimization for supermassive crowd simulation on gpu. *Comput. Entertain.*, 7(4):1–15, 2009.

- [65] Craig Reynolds. Boids - background and update. <http://www.red3d.com/cwr/boids/>, 2010. [Online; accessed 26-October-2010].
- [66] Graphic Remedy. gDEBugger CL. <http://http://www.gremedy.com/gDEBuggerCL.php>. Last seen: 17th of November 2010.
- [67] Craig W. Reynolds. Flocks, herds and schools: A distributed behavioral model. In *SIGGRAPH '87: Proceedings of the 14th annual conference on Computer graphics and interactive techniques*, pages 25–34, New York, NY, USA, 1987. ACM.
- [68] Erik Sintorn and Ulf Assarsson. Fast parallel gpu-sorting using a hybrid algorithm. *J. Parallel Distrib. Comput.*, 68(10):1381–1388, 2008.
- [69] Rys Sommefeldt. NVIDIA G80: Architecture and GPU Analysis. <http://www.beyond3d.com/content/reviews/1/>. Last seen: 14th of October 2010.
- [70] Open source Community. GDB: The GNU Project Debugger. <http://www.gnu.org/software/gdb/>. Last seen: 14th of November 2010.
- [71] David Tarditi, Sidd Puri, and Jose Oglesby. Accelerator: using data parallelism to program gpus for general-purpose uses. In *ASPLOS-XII: Proceedings of the 12th international conference on Architectural support for programming languages and operating systems*, pages 325–335, New York, NY, USA, 2006. ACM.
- [72] Kevin M. Greenan Thomas E. Portegys. Managing flocking objects with an octree spanning a parallel message-passing computer cluster. <http://www.acs.ilstu.edu/faculty/portegys/research/pmtree-PDPTA03.pdf>, 2003. [Online; accessed 24-December-2010].
- [73] Education Editor Times Online, Alexandra Freen. White bread for young minds, says university of brighton professor. http://technology.timesonline.co.uk/tol/news/tech_and_web/the_web/article3182091.ece. Last seen online 20th of December 2010.
- [74] Wikipedia. Computing platform — wikipedia, the free encyclopedia. http://en.wikipedia.org/w/index.php?title=Computing_platform&oldid=401695834, 2010. [Online; accessed 13-December-2010].
- [75] Wikipedia. Crowd simulation — wikipedia, the free encyclopedia. http://en.wikipedia.org/w/index.php?title=Crowd_simulation&oldid=385547491, 2010. [Online; accessed 20-October-2010].
- [76] Wikipedia. Flops — wikipedia, the free encyclopedia. <http://en.wikipedia.org/w/index.php?title=FLOPS&oldid=390775620>, 2010. [Online; accessed 20-October-2010].
- [77] Wikipedia. Geforce 8 series — wikipedia, the free encyclopedia. http://en.wikipedia.org/w/index.php?title=GeForce_8_Series&oldid=401029719, 2010. [Online; accessed 8-December-2010].

- [78] Wikipedia. Gnu debugger — wikipedia, the free encyclopedia. http://en.wikipedia.org/w/index.php?title=GNU_Debugger&oldid=396109819, 2010. [Online; accessed 14-November-2010].
- [79] Wikipedia. Graphics processing unit — wikipedia, the free encyclopedia. http://en.wikipedia.org/w/index.php?title=Graphics_processing_unit&oldid=391476688, 2010. [Online; accessed 20-October-2010].
- [80] Wikipedia. Phong shading — wikipedia, the free encyclopedia. http://en.wikipedia.org/w/index.php?title=Phong_shading&oldid=395725735, 2010. [Online; accessed 23-November-2010].



Nvidia G80

As we have access to two Tesla C870 cards which are based on the G80 GPU architecture, we will give a description of the parts of this architecture that are of interest for GPGPU. This can help reason about performance of CUDA and OpenCL applications on the G80 architecture.

We have previously given an overview of the components that make up the G80 chip in Section 2.3. Here we will give a more detailed description of each of these, such that we can better reason about why GPGPU applications behaves as they does.

Nvidia is somewhat secretive about the details of their chips. The following is therefore based on information from people working for Nvidia, found in [35] and [54], supplemented with details found in [7] and [69].

A.1 Components Details

As mentioned in Section 2.3, the **host interface** on the G80 is usually the PCIe bus, more specifically the high bandwidth 16 lanes PCIe slot. PCIe v 1.0 gives a bandwidth of 4 GB/s between the G80 and host system. This is relatively low compared to the 25.6GB/s between an Intel Core i7-980X CPU and main memory, this means that the **host interface** can easily become a bottleneck if programmers do not take this into consideration, i.e. by reducing or batching data transfers between the GPU and the CPU.

Work given to the GPU is split into a number of CATs. A CAT is equivalent to a thread block in CUDA, or a work group in OpenCL. A CAT is a collection of 1 to 512 threads, that can execute the same program concurrently. When work is received by the GPU, the **compute work distribution**, running at 600MHz, assigns a CAT to be executed on a SM in a round-robin fashion.

When a SM has sufficient resources to execute a CAT, the SMC creates the CAT and assigns a thread ID (TID) to each thread in the CAT.

The **multithreaded instruction unit** of the SM executes threads in what Nvidia calls warps. A warp is a group of 32 parallel threads running the same program. Each SM manages a pool of 24 warps. Each SM can thus have up to 768 concurrent threads.

Table A.1: Operations performed by the SP

Operation	Data type	Operations per clock cycle
add	32bit floating point	1
multiply	32bit floating point	1
multiply-add	32bit floating point	1
add	32bit integer	1
logical operation	32bit integer	1
bit shift	32bit integer	1
compare	32bit integer	1
multiply	24bit integer	1
type conversion	all	1

At each nstruction issue time, the **multithreaded instruction unit** selects a warp which is ready to execute and issues its next instruction, thus allowing for latency hiding when some of the warps are waiting for memory. The **multithreaded instruction unit** runs at half the speed of the SP and SFU, i.e. $\frac{1350MHz}{2} = 675MHz$. It does this since the SP and SFU take multiple cycles to execute an instruction of all threads in a warp.

The reason that **multithreaded instruction unit** can afford to potentially execute an instruction on a new warp at every instruction issue time, is that there is zero cost context switch between warps. This is achieved by keeping the context of each warp in register and shared memory in the SM, i.e. the context is not flushed to DRAM.

The SM has 8192 32 bit registers, 16KB of **shared memory** and a 8KB **constant cache**, these are shared between warps. Thus, to have the maximum number of threads running on a SM at a time, thereby achieving the best possible latency hiding, each thread can only use $\frac{8192}{768} \approx 10$ registers. 768 must be evenly divisible by the size of the CATs, the size of the CATs must be evenly divisible by 32 and each CAT must use no more than $\frac{16384}{n}$ B of shared memory where n is the number of CATs, e.g. if the size of a CAT is 256 threads, note that $\frac{768}{256} = 3$ and $\frac{256}{32} = 8$ and are thus evenly divisible, maximum latency hiding can be obtained by using fewer than 10 registers and less than $\frac{16384}{3} \approx 5461$ B of shared memory, where 16384 is the amount of shared memory on each SM.

Each thread in a warp must execute the same program, i.e. start from the same program address. They are however still allowed to branch independently of each other. This is achieved by deactivating the threads that did not take a given branch, leading to a decrease in performance. Thus, a SM can be regarded as a 32 way SIMD processor. Note that threads in different warps may branch without any decrease in performance.

The SP and SFU, both running at 1350MHz, execute the actual instructions on the threads within a warp. Each SP can execute one of the operations in Table A.1.

All operations take 1 clock cycle in Table A.1, thus all of these operations take 4 clocks cycles to complete for the entire warp, since there are eight SPs per SM, and each warp has 32 threads, i.e. $\frac{32}{4} = 8$ cycles. The two SFUs can execute the operations in Table A.2.

The SFUs can perform a multiplication on all threads in a warp, in only 4

Table A.2: Operations performed by the SFU

Operation	Data type	Operations per clock cycle
multiply	32bit floating point	4
reciprocal	32bit floating point	1
reciprocal square root	32bit floating point	1
base 2 logarithm	32bit floating point	1
base 2 exponential	32bit floating point	1
sine	32bit floating point	1
cosine	32bit floating point	1

clock cycles, while the other operation takes 16 clock cycles. Because the **SP** and **SFU** can execute independently, a single **SM** has a theoretical peak performance of two **F**loating point **O**peration (**FLOP**) per **SP**, in the form of a multiply-add instruction, and four **FLOP** per **SFU**, in the form of multiplications. Combined, the total theoretical performance is: $(2 \times 8 + 4 \times 2) \times 1350 = 32400 = 32.4$ **gigaFLOPS** per **sm**.

Note that the G80 does not support double precision floating points [54, C.1.2], and that the implementation of single precision floating points is not fully IEEE 754 compliant. This means that some computations might give different results on the GPU, compared to the CPU. Some of the most important differences are listed below, for a full list see [54, G.2].

“Denormalized numbers are not supported and are converted to zero, underflows are flushed to zero, in multiply-add the mantissa of the intermediate multiplication is truncated, division is implemented using the reciprocal in a non standard compliant way, square root is implemented using the reciprocal square root in a non-standard-compliant way and for addition and multiplication only round to nearest even and round towards zero are supported.” [54, G.2]

A.2 Memory

There are a number of different memory types on the Tesla C870 some on the G80 chip and one off chip, these are listed below:

- 8192 32 bit registers per **SM**
- 16KB shared memory per **SM**
- 8KB constant cache per **SM**
- 8KB L1 texture cache per **TPC**
- 21KB L2 cache per **DRAM** memory bank
- 1536MB off chip **DRAM**

As mentioned, a **SM** constantly switches execution between warps that are ready to execute the next instruction. The most common reason for a warp not being able to execute its next instruction, is that the input operands for that instruction are not available yet. This means that the latency of the different types of memory can have a big impact on performance. [54, sec. 5.2.3]

Latency can even exist when all input operands are in the registers. This happens if an instruction is dependent on the result of the previous instruction, and thus have to wait for that instruction to finish executing. Since it typically takes about 22 clock cycles for an instruction to move through the execution pipeline, and since it takes 4 cycles to execute one instruction for all 32 threads in a warp, we need at least $22/4 \approx 6$ warps to hide the latency, thus achieving full efficiency.

Access to DRAM is much slower than registers, typically 400 to 800 clock cycles. The amount of warps required to hide this latency depends on the code being executed, i.e. if the program has a high number of instructions which do not require access to DRAM, fewer warps are required compared to a program with a lower number of instructions that do require access to DRAM. As an example, assuming that we have a program that performs 10 instructions per DRAM access, each instruction takes 4 clock cycles, and that DRAM latency is 600 clock cycles, we need at least 15 warps to hide this latency, since $15 \times 10 \times 4 = 600$.

As previously mentioned, the DRAM on the G80 is split in to 6 banks on the Tesla C870 card. Each bank has 256MB of GDDR3 memory clocked at 800MHz and is connected to the G80 using 64 pins. Since GDDR3 can transfer 4bits per pin in two clock cycles, this gives a maximum bandwidth per bank of $4 \times 64 \times \frac{800}{2} = 102400$ 102.4 Gbit or $\frac{102400}{8} = 12800 = 12.8$ GB/s thus the total bandwidth on the card is $12800 \times 6 = 76800 = 76.8$ GB/s.

All DRAM access is performed as either 32, 64 or 128 byte memory transactions and these transactions must be naturally aligned aligned, i.e. offsets in memory must be a multiple of the transaction size of the architecture.

Each thread in a warp can access different memory addresses, which means that a lot of bandwidth is potentially wasted on data that is not used. To avoid this, memory access can be coalesced into fewer memory transactions, this is done by first splitting the threads of a warp into two half-warps, each having 16 threads. Coalescing within each of these half-warps happens if the following requirements are met:

- The threads of the half-warp must access a 4, 8 or 16 byte word
- if the size of the word is 4 bytes all 16 words must be in the same 64 byte segment
- if the size of the word is 8 bytes all 16 words must be in the same 128 byte segment
- if the size of the word is 16 bytes all 8 words must be in the same 128 byte segment and the other 8 words must be in the next 128 byte segment
- the n 'th thread in the half-warp must access the n 'th word

Thus, accessing 4 and 8 byte words is translated to one memory transaction, and access to 16 byte words is translated to two transactions. If these requirements are not met, 16 transactions of 32 byte are performed for each half warp, resulting in worse performance.

To illustrate the effect of different memory access patterns assume all 32 threads in a warp accessing 4 byte words in the sequential and aligned pattern

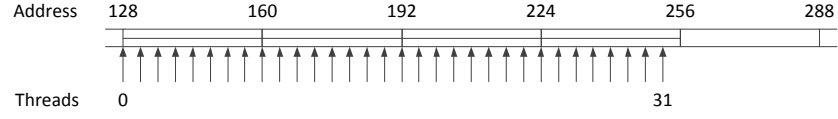


Figure A.1: Aligned and sequential memory access. [54, figure G-1]

shown in Figure A.1. This pattern results in one 64 byte memory transaction at address 128 and on 64 byte memory transaction at address 192.

If two threads in the first and second half-warp accesses 4 byte words in a non sequential manner, as shown in Figure A.2, it requires a total of 32 transactions of 32 byte i.e. 8 transactions at address 128, 8 transactions at address 160, 8 transactions at address 192 and 8 transactions at address 224, thereby wasting bandwidth and reducing performance.

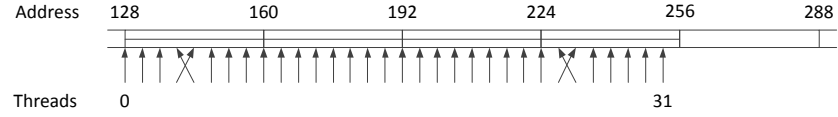


Figure A.2: Non sequential memory access. [54, figure G-1]

If memory is accessed in a misaligned manner as shown in Figure A.3, 32 memory transactions of 32 byte are used as follows: 7 transactions at address 128, 8 transactions at address 160, 8 transactions at address 192, 8 transactions at address 224 and 1 transaction at address 246.

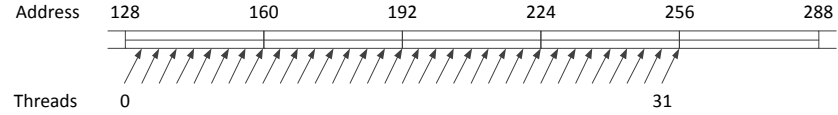


Figure A.3: Misaligned memory access. [54, figure G-1]

The **constant cache** can be used to speed up access to DRAM. To do this, the memory must be read-only and only 64KB in size. As with normal DRAM access, the warp is split in to two half warps. Within each half warp, a request is split up to a request per address in the original request. Each of the requests are then serviced by the **constant cache**, in the case of a cache hit, and as a normal DRAM access in the case of a cache miss. Thus, the **constant cache** can decrease DRAM bandwidth usage by avoiding memory access for already cached addresses and by avoiding multiple threads reading the same address. On the other hand, it can also decrease performance when accessing many different memory addresses, since multiple memory accesses is created for what could have been a single memory access for normal DRAM.

Another way of speeding up DRAM access is with the use of the **texture cache**, which is designed to capture 2D spatial locality, i.e. threads in the same warp which access addresses close together will achieve better performance. A cache hit is designed to reduce the DRAM bandwidth demands, but not the latency of DRAM access. So applications with a memory access pattern that does not fit well for coalescing of normal DRAM, or with the use **constant**

`cache`, can potentially get better performance by using the `texture cache`. The `texture cache` is however not kept coherent with DRAM so it should only be used for read only memory.

Each of the `SM` has 16KB of `shared memory`, which can be shared between threads running on the `SM`. The `shared memory` is split into 16 banks, each of these banks has a bandwidth of 32bits every two clock cycles so the performance depends on the division of memory access between banks. Access to `shared memory` is split into two half warps, thus threads from each half warp can not access a bank simultaneously. To reduce conflicts between banks, memory is split between banks in 32-bit words, thus if an application accesses subsequent elements in an array of 32-bit words there are no bank conflicts and all threads in a half-warp can be serviced in two clock cycles. On the other hand, subsequent elements in an array of 8-bit words generates 4 bank conflicts and thus takes 8 clock cycles.