TU/e EINDHOVEN UNIVERSITY OF TECHNOLOGY

Eindhoven University of Technology

MASTER

Improving CUDA's compiler through the visualization of decoded GPU binaries

Nugteren, C.

Award date:
2009

Link to publication

Disclaimer
This document contains a student thesis (bachelor's or master's), as authored by a student at Eindhoven University of Technology. Student theses are made available in the TU/e repository upon obtaining the required degree. The grade received is not published on the document as presented in the repository. The required complexity or quality of research of student theses may vary by program, and the required minimum study period may vary in duration.

General rights
Copyright and moral rights for the publications made accessible in the public portal are retained by the authors and/or other copyright owners and it is a condition of accessing publications that users recognise and abide by the legal requirements associated with these rights.

- Users may download and print one copy of any publication from the public portal for the purpose of private study or research.
- You may not further distribute the material or use it for any profit-making activity or commercial gain

Improving CUDA's Compiler through the Visualization of Decoded GPU Binaries

Cedric Nugteren

Electronic Systems group
Faculty of Electrical Engineering
Eindhoven University of Technology

August 25, 2009

Master thesis under supervision of H. Corporaal and B. Mesman
Contact: mail@cedricnugteren.nl

Contents

1 Introduction **1**

1.1 Motivation 1

1.2 Problem statement 2

1.3 Thesis outline 2

2 Background **5**

2.1 The GPU as a general purpose processor 5

2.1.1 A brief history of GPU architectures 6

2.1.2 An introduction to GPGPU programming 6

2.2 The CUDA thread model 7

2.3 The CUDA compilation flow 8

2.4 G80's hardware architecture 10

2.4.1 G80's processor hierarchy 10

2.4.2 G80's memory hierarchy 11

2.5 Mapping threads onto multiprocessors 12

3 Motion estimation on a GPU **15**

3.1 The block matching algorithm 15

3.2 Comparing two different mappings 17

3.2.1 The first mapping approach 17

3.2.2 The second mapping approach 20

3.3 Describing optimization steps 23

3.4 Benchmarking the block matching algorithm 25

4 Visualizing GPU binaries **27**

4.1 Automating the mapping and optimization process 27

4.2 Analyzing an existing decoder 29

4.3 The design of CUDAvis 30

4.4 Evaluation of CUDAvis 31

5 Compiler improvements **35**

5.1 Non-linear register allocation 35

5.1.1 Current register allocation techniques 35

5.1.2 Proposed technique for register allocation 36

5.1.3 Analysis of non-linear register allocation 38
5.2 Value recalculation 39
5.2.1 Block matching as an example 39
5.2.2 When to apply value recalculation 40
5.2.3 An algorithm for value recalculation 41
5.2.4 Results of value recalculation 43
5.3 Efficient instruction re-ordering 44
5.3.1 Block matching as an example 44
5.3.2 Automatically applying instruction re-ordering 46
5.4 Using the shared memory as a register file 46
5.5 Removing redundant instructions 47
5.5.1 Multiple registers for the same value: an example . . . 47
5.5.2 Multiple instructions for one calculation: an example . 48
5.5.3 Evaluation of redundant instruction removal 48
5.6 Evaluation of the improvements 49

6 Functionality changes 51
6.1 From the texture cache to the shared memory 51
6.2 Scratchpad access for the texture cache 52

7 Conclusions 55
7.1 Related work 55
7.2 Further work 56
7.3 Summary 57

Chapter 1

Introduction

Graphical processing units are a hot topic in computer architecture design. Section 1.1 shows why and provides a motivation for the selection of graphical processing units as a subject in this work. However, programmers have to spend a lot of effort to efficiently map algorithms on such an architecture. Therefore, section 1.2 specifies the problem statement. This work adresses the problem, after introducing the reader to the architecture and the involved programming environment. A full overview of this work is outlined in section 1.3.

1.1 Motivation

As todays graphical processing units (GPUs) provide a higher raw computational potential than traditional general purpose CPUs, using GPUs for general purpose tasks becomes increasingly common [4]. In the past, GPUs have been used mainly for the acceleration of 3D-games. However, in the last few years, GPUs where used to provide hardware acceleration for high definition video decoding [1] and for physics calculations within 3D-games [9]. Adding to the popularity of GPUs for general purpose tasks is the in 2008 introduced hardware acceleration support for Adobe Photoshop [18] and Mathworks' Matlab [7], offloading computation tasks from the CPU.

GPUs are a typical example of the current shift within processor design. Traditional processor design was driven by frequency scaling, while nowadays, a trend towards hardware multithreading and computationally dense SIMD[1] architectures can be observed [14]. In other words, the shift replaces one complex high frequency processor by multiple simple parallel processors. The GPU is an example of an architecture consisting of multiple SIMD processors supporting hardware multithreading. High-end GPUs typically outperform traditional CPUs by a factor of 50 when measuring in raw computation power (FLOPS) [21].

[1]Single Instruction, Multiple Data

1.2 Problem statement

Along with the trend towards parallel processors comes a programming model shift. Sequential programming languages are replaced by parallel programming languages that expose the programmer to SIMD-style parallelism. For GPUs, NVIDIA introduced CUDA[2] as a parallel programming environment for general purpose applications. With CUDA, the programmer can exploit the parallel architecture of the GPU and accelerate general purpose applications.

Although CUDA is specifically designed for general purpose GPU programming, the mapping process of an algorithm onto a GPU remains non-trivial. Although general purpose GPU programming has developed fast in the last three years, the path to optimal hardware usage is still long. With the current hardware and compilation flow, programmers have to spend several weeks to reach a near-optimal hardware usage for their algorithms. For a motion estimation algorithm, different mappings show different advantages and disadvantages. The programmer has to select an appropriate mapping, depending on the nature of the algorithm. Then, a number of algorithm specific and general optimizations have to applied, all resulting in significant speed-ups.

With CUDA, parallelization of a sequential algorithm is performed quite easily, although in order to achieve optimal hardware usage, the programmer is required to have thorough knowledge of the hardware, the programming language and the target architecture. Achieving optimal hardware usage is non-trivial with different memories, caches and shared register files all mappable by the programmer.

Although it is clear that GPUs provide a strong platform for algorithm mapping, the burden on programmers to map their algorithms efficiently onto the GPU is too high. Therefore, the objective of this work is to reduce the burden on programmers, resulting in more hardware efficient mappings and a shorter development time.

1.3 Thesis outline

This document is organized as follows. First of all, chapter 2 provides background information on the CUDA environment and the GPU hardware. The chapter first introduces the concept of general purpose GPU programming. Then, the reader is introduced to the thread model defined by the CUDA environment, followed by an overview of the CUDA compilation flow. Also, the hardware is introduced from both a processing and a memory perspective. Finally, the coupling between the CUDA environment and the GPU

[2]CUDA is an acronym for Compute Unified Device Architecture

hardware is evaluated, in order to obtain insight in the mapping process of an algorithm onto a GPU.

Then, motion estimation is chosen as an example general purpose application and mapped onto the GPU. An example of motion estimation is the block matching algorithm, which is introduced in chapter 3. This block matching algorithm is mapped onto a GPU using two distinctive mappings, which are then compared for scalability, ease of implementation and performance. Concluding the chapter, benchmarks on two different GPUs are presented.

Since a part of the CUDA compiler is not open-source and no specifications are given, exact behaviour is unknown. This part of the CUDA tool-chain is the compiler *ptxas*, translating from a virtual instruction set to a GPU binary. The resulting GPU binary is unreadable. In chapter 4, the GPU binary is decoded using a new tool, introduced in the same chapter. This tool gives the programmer feedback by decoding and visualizing the GPU binary produced by ptxas. Additionally, the tool can also be used to understand the behaviour of ptxas.

As the tool from chapter 4 can be used to understand ptxas' behaviour, it can be used to propose improvements to the compiler. In chapter 5, a number of improvements are presented and discussed. Among these improvements are non-linear register allocation, improved value recalculation, instruction re-ordering, register spilling to shared memory and redundant instruction removal. Also using the tool from chapter 4, two weak points are found in the hardware, which are presented in chapter 6.

Finally, in chapter 7, conclusions on the research on ptxas' behaviour for CUDA will be presented, along with further work and a summary.

Chapter 2

Background

As the reader might have no or partial knowledge of GPU architectures and the CUDA programming environment, background information on CUDA as well as on the corresponding hardware is provided. Before introducing the hardware or the programming environment, a brief history along with a general introduction is provided in section 2.1, serving as an introduction to general purpose GPU programming. To program GPUs for general purpose applications, the CUDA environment is used. This environment consists of among others a thread model - introduced in section 2.2 - and a compilation flow consisting of several tools, which is introduced in section 2.3. To be able to understand design choices, programming behaviour and flaws found while using the CUDA environment, a GPU architecture is introduced. Section 2.4 shows both the different processing elements and the different memory elements of a G80 GPU. Finally, section 2.5 shows the relation between the CUDA environment and the introduced hardware.

2.1 The GPU as a general purpose processor

The last few years show an increase of interest in general purpose GPU programming. Since the introduction of the term, general purpose GPU programming has exponentially increased. Now, with GPU vendors introducing dedicated hardware for general purpose GPU programming, the interest has not stopped growing. To give an introduction to the concept of using a GPU as a general purpose processor, a brief history of GPU architectures is given in section 2.1.1, followed by the concepts of general purpose GPU programming in section 2.1.2. This section shows the most important techniques to perform general purpose GPU programming, along with their properties.

2.1.1 A brief history of GPU architectures

The modern 3D graphics processing unit (GPU) architecture as it exists nowadays evolved from a fixed-function graphics pipeline in 1999 into a programmable parallel processor. Programmability of GPUs started emerging around 2001 with the introduction of programmable vertex processors [14]. Later, the addition of programmable pixel processors added to the programmability of GPUs. Because of the need for increasing processing power in 3D-games, GPU vendors added an increasing number of vertex and pixel processors onto their GPUs. A unified programmable processor was developed [14], containing processors suitable for both vertex and pixel operations. The shift towards programmable GPUs was motivated by the unbalanced ratio between vertex and pixel operations in 3D-scenes. Although dedicated vertex or pixel processors are more efficient and faster individually, a GPU using a programmable processor was shown to be more efficient and faster due to the dynamic ratio of vertex and pixel processors, yielding an ideal ratio depending on the 3D-scene to be rendered.

2.1.2 An introduction to GPGPU programming

The introduction of GPUs using a unified processor architecture denoted the start of general purpose GPU (GPGPU) programming. Since the new GPU architectures come with a high grade of programmability, the GPU is now suitable for a wide range of applications, denoted as general purpose - non 3D-rendering - applications. Among these applications are computer vision, signal processing, finance, physics and streaming media [17].

Although GPGPU programming attempts were made prior to the unification of the GPU architecture, they were not successful. Due to the lack of a proper GPGPU programming language, OpenGL was used. OpenGL is a graphics rendering API, used for 3D-rendering. Although OpenGL gave access to the GPU, it required from the programmer large amounts of effort and knowledge to effectively program the GPU [15]. In order to overcome these problems, NVIDIA - as the first hardware vendor - altered their GPU design to make room for flexible and programmer friendly GPGPU programming. Since GPUs are particularly suited for parallel processing applications, GPGPU programming only benefits from data-intensive applications, while traditional CPUs are more suited for applications consisting of high control-intensive algorithms. Therefore, in GPGPU programming, the GPU and the CPU are required to work together, dividing an application for an efficient use of both devices.

The GPGPU programming language released by NVIDIA is part of the CUDA environment. It consists, apart from the programming language *C for CUDA*, of a toolset, a thread model and special GPGPU hardware. The CUDA programming language allows the programmer to specify which part

of the algorithm should be executed on the host - the CPU - and which part on the device - the GPU. Programmers can order data transfers from host memory to device memory and vice versa.

NVIDIA's direct competitor ATI followed with their own GPGPU programming language Close To Metal, which was succeeded by StreamSDK. Since both NVIDIA and ATI introduced their own environment, the Khronos group - holder of the OpenGL specification - started working on an open GPGPU programming language, suitable for all vendors' GPUs. In the beginning of 2009, the Khronos group released their OpenCL[1] specification, meant to unify GPGPU programming in one open-source language [16]. As of now, OpenCL lacks programmer support, but it may grow in the future with Apple integrating OpenCL in their newest operating system, due for release mid 2009 [16]. In this work, CUDA is chosen as a GPGPU programming environment, since it is the most widely used and best documented environment.

2.2 The CUDA thread model

The CUDA environment consists of a thread model, introduced in order to ensure that the hardware can benefit from its multi-threaded possibilities, allowing the programmer to parallelize sequential code. An overview of the thread model can be seen in figure 2.1 and is defined as follows:

kernel A kernel is a small program which is typically executed a large number of times on different data - also known as SPMD (single program, multiple data) [14]. In CUDA, the kernel is executed on the device, on which only one kernel can be active at any given time.

thread A thread is an instance of a kernel. In CUDA, the number of threads one kernel instances can be tens to hundreds of thousands.

threadblock Each thread belongs to a threadblock. Within one threadblock, threads can synchronize with each other using a barrier. Also, within one threadblock, one piece of shared memory can be used among all the threads it consists of for data re-use or communication between threads [14].

grid Each threadblock belongs to a grid. Together, all threadblocks in a grid form the complete execution of a kernel. Within a grid, no communication or synchronisation is possible, except within individual threadblocks [14].

[1]OpenCL is an acronym for Open Computing Language

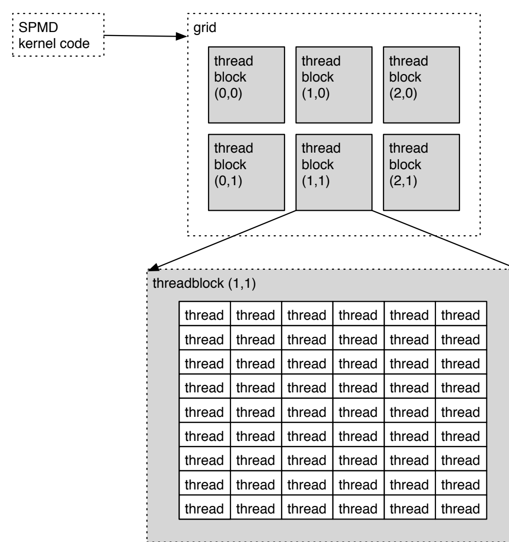

Figure 2.1: The CUDA thread model: an example layout

In figure 2.1, threadblocks and threads are both organised in a two-dimensional way inside their grid and threadblock respectively. Threadblocks and threads can also be organised in a one- or three-dimensional way, reflecting the data structure of the application. Within the CUDA programming language, the programmer can specify the layout and the number of threads and threadblocks.

Threads can be characterised as programs that fetch their instructions from the same binary, but can take distinct control paths depending on identifiers which are pre-set and stored in special registers [14]. These identifiers hold different values for each thread and correspond to both the coordinates of a thread in their threadblock and the coordinates of their threadblock in the grid. The formation of the threads and the mapping on the hardware is discussed in section 2.5.

2.3 The CUDA compilation flow

This section introduces the CUDA compilation flow along with its two proprietary compilers. First, before compilation, code is split up into a device part and a host part by a front-end called *cudafe*. The device part - the kernel - is ran through *nvcc*, compiling high level code into a virtual instruction set - the PTX[2] intermediate format. The resulting PTX code is ran through the second compiler, ptxas, which produces a GPU binary [20]. Since the host part is executed on the CPU, a standard C compiler is used.

[2]PTX is an acronym for Parallel Thread eXecution

The compilation flow is shown in figure 2.2.

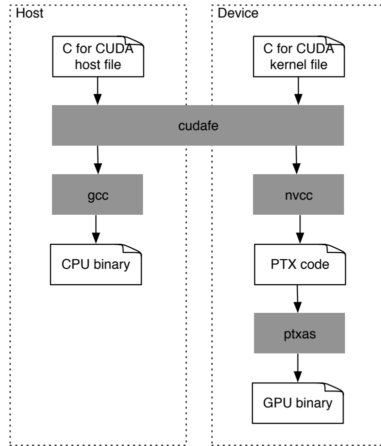

Figure 2.2: A simplified compilation flow

Please note that the compilation flow in figure 2.2 is an abstraction of the actual (more complicated) compilation flow. For the research purposes of this work, the abstraction of the compilation flow provides sufficient information. Research is focussed on the trajectory from CUDA code through nvcc and ptxas, resulting in a GPU binary. The two different compilers involved are:

nvcc The compiler nvcc, based on the open-source compiler Open64, performs the largest part of the compilation. It should be noted that newer GPUs introduce slightly different hardware specifications and a slightly different instruction set. Therefore, the intermediate PTX code that nvcc produces is chosen to be hardware independent, still being able to run on any CUDA enabled GPU [20]. Because of the introduced hardware independency, several compiler tasks are not performed by nvcc. Thus, PTX code can be seen as a virtual instruction set, targeting current and future hardware architectures. PTX is completely specified by NVIDIA [19].

ptxas The second compiler performs hardware specific compilation. As of 2009, four different hardware specifications exist [19]. The compiler can compile a GPU binary for any of these target specifications. In the case that a different hardware architecture is used, ptxas can also be executed at run-time as a just-in-time compiler. To be able to do so, a compiled program by ptxas includes PTX code in case of a change of the target hardware architecture [14].

Among the tasks of ptxas are register allocation and instruction reordering. Since PTX code assumes an infinite number of registers

available and the number of available registers can change with different architectures, ptxas needs to perform register allocation. In order to perform register allocation, ptxas can also perform instruction reordering.

2.4 G80's hardware architecture

In order to introduce the reader to the hardware architecture, a short overview of the layout of both memories (in section 2.4.2) and processing elements (in section 2.4.1) is presented. As a reference, the G80 layout is taken. This architecture is found on GPU boards from the GeForce 8 series as well as various Quadro and Tesla based boards [17]. Newer CUDA-enabled GPUs have a comparable layout, but have different specifications, such as memory sizes, number of processing elements and so on.

2.4.1 G80's processor hierarchy

First, the architecture is presented from a processing point of view. The device contains a number of texture processor clusters (TPC), ranging in the order from one (for entry-level hardware) up until sixteen (for the current high-end hardware) [17]. In G80 architecture, each TPC contains two streaming multiprocessors (SM), which in turn each contain eight processing elements (PE) and two special function units (SFU) [14]. A visualization[3] is shown in figure 2.3.

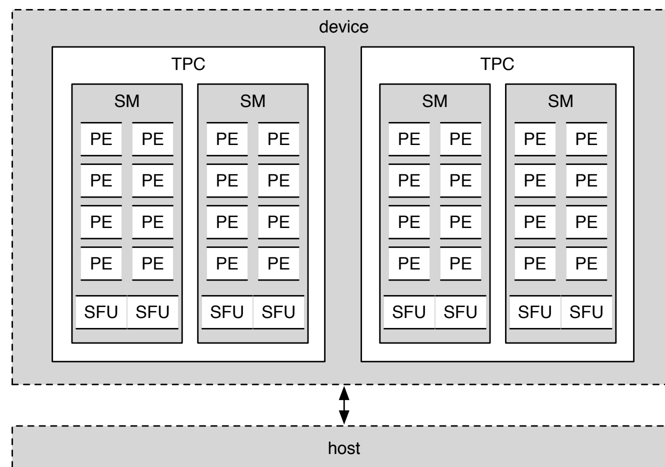

Figure 2.3: Processor hierarchy

[3]Please note that this visualization is an abstraction of the actual hardware, in this case featuring four SMs

An SM can be seen as an SIMD processor, containing PEs and SFUs - the main computation units for CUDA. PEs consist of scalar multiply-add (MAD) units, while the SFUs are used for transcendental functions, but also contain four floating-point multipliers each [14]. Apart from the processing elements seen in figure 2.3, the G80 architecture contains fixed-function logic specific for the graphics pipeline, such as a geometry controller and a raster operation processor. These processors are not used by CUDA and are therefore not shown in figure 2.3.

2.4.2 G80's memory hierarchy

From a memory point of view, each TPC contains a texture cache. Within each TPC, the two SMs each contain a register file, a shared memory and a constant cache. A visualization[4] is shown in figure 2.4, together with the off-chip memory.

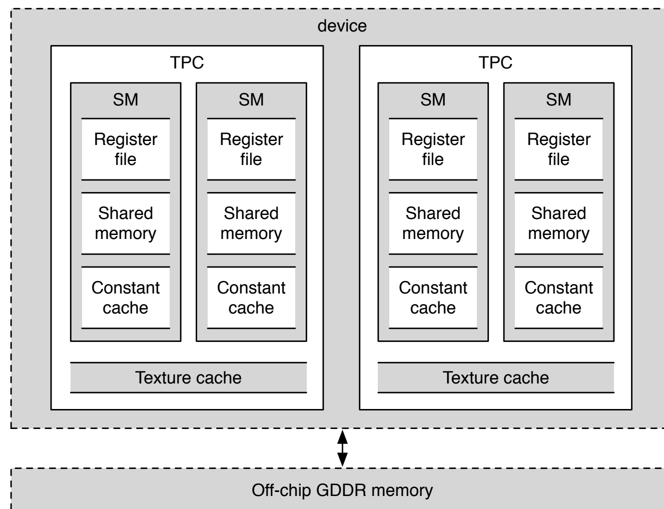

Figure 2.4: Memory hierarchy

In CUDA, these memories are divided into different parts, each with their own specific CUDA name. Firstly, the off-chip memory contains the global memory, the texture memory and the constant memory. From an SM point of view, the global and texture memories support reads and writes, while the constant memory only supports reads. Secondly, the cache is divided into a texture cache - corresponding with the texture memory - and a constant cache - corresponding with the constant memory. These caches are read-only from an SM point of view. The caching is transparant for

[4]The G80 consists of other memory elements, but are left out for simplification reasons

programmers. Lastly, the shared memory supports reads and writes, as does the register file.

An overview of the scope, the typical size and an estimate of the access latency[5] is presented in the table 2.1.

CUDA name	Scope	Location	Typical size	Access latency
Register file	Thread	SM	8K entries	register speed
Shared memory	Threadblock	SM	16KB	register speed
Constant cache	Program	SM	8KB	register speed
Texture cache	Program	TPC	16KB	register speed
Global memory	Program	Off-chip	0-1GB	200-600 clock cycles
Texture memory	Program	Off-chip	0-1GB	200-600 clock cycles
Constant memory	Program	Off-chip	64KB	200-600 clock cycles

Table 2.1: Overview of the different memories

2.5 Mapping threads onto multiprocessors

Now that the thread model and the hardware are discussed, it is time to understand how they cooperate. In other words: how are threadblocks and threads mapped onto the different processing elements of the GPU? To answer this question, the concept *warp* is introduced. A warp is defined by NVIDIA as a group of at most 32 threads, which start together at the same program address but are otherwise free to branch and execute independently [21].

On execution of a kernel, a warp is started onto a streaming multiprocessor. Because a warp typically contains 32 threads and a multiprocessor contains 8 processing elements, it will take 4 clock cycles to execute the first instruction[6] of each thread within the warp [13]. Then, the first instruction of the second warp is executed, taking another 4 clock cycles. When all warps finished their first instruction, the first warp starts executing the second instruction. This process goes on until a load instruction is encountered that needs to wait for the off-chip memory. At this point, the multiprocessor does not schedule this warp anymore. When a warp receives its data from the off-chip memory, it is enabled for scheduling again. In the case that all warps are waiting for memory transfers, the multiprocessor is idle.

Besides scheduling in groups of warps, the thread model from section 2.2 also introduces constraints on the scheduling possibilities. As mentioned before, threadblocks can synchronize and use a shared memory. Therefore,

[5] The speed depends on various factors, such as the bus usage

[6] The throughput is one warp per 4 cycles, the latency is higher due to a multiple stage pipeline

one entire threadblock must be scheduled onto one streaming multiprocessor. The number of threadblocks that fit onto one multiprocessor depends on several factors:

- Firstly, the register file has to be shared among all threads in all threadblocks that are scheduled onto the multiprocessor. In G80 hardware, the register file contains 8192 entries. Depending on the number of registers needed per thread, a maximum amount of threadblocks can be scheduled on an SM.
- Secondly, the shared memory needs to be shared among a threadblock. This shared memory should however be divided over the number of threadblocks active on the SM. Depending on the shared memory usage of a threadblock, a maximum amount of threadblocks can be scheduled on an SM.
- Lastly, there are hardware limits of a maximum of 768 threads or 8 threadblocks per SM. If the previously mentioned limits are not met, either 768 threads or 8 threadblocks will be scheduled on an SM, depending on which limit is reached first.

A group of warps sharing one set of shared memory and synchronizing using barriers is called a concurrent thread array (CTA). Warps inside a CTA are assumed to execute in parallel, apart from synchronization barriers set by programmers [6]. The reader might notice that a CTA has the same properties as a threadblock - synchronization and shared memory requirements - and can therefore be stated as equal. The concept of a threadblock is seen from a programmers perspective, while a CTA is seen from a hardware perspective. Otherwise, the concepts are the same. Therefore, one group of warps in a CTA has to be executed onto one multiprocessor. An overview of warps and CTAs can be found in figure 2.5, which is comparable to figure 2.1.

From all the above mentioned concepts and models, it is important to realize the impact of the chosen number of threads and threadblocks. First of all, the number of threads inside a threadblock should at least be 32 for a warp to be completely filled and all processing elements to be used. Secondly, a threadblock has preferably a size dividable by 32, so that no semi-filled warps have to be formed. But lastly - the most important - the number of threads should be as big as possible. Because the G80 hardware and the CUDA environment are designed to switch to another warp whenever a warp has to wait for memory access, memory request times can be completely or partially hidden - if enough warps are available to switch to. The memory operation intensity together with the computation intensity can decide a sufficient amount of warps to ensure completely hidden memory request times, which can be analytically computed [13]. However, increasing the

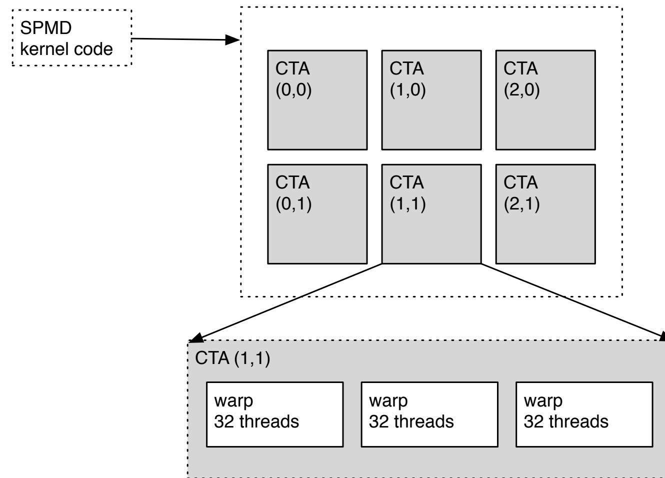

Figure 2.5: Warps and CTAs

number of warps - and thus threads - will always help to hide this latency, independent of the structure of the program.

GPUs are a typical example of many-thread architectures. Although the G80 architecture implements a texture cache, it is not big enough for most applications to hide their memory latencies [10]. Instead - to hide memory latencies - a high number of threads is scheduled onto the hardware. Apart from many-thread architectures such as the GPU, a second type of multiprocessor architecture exists: a many-core architecture [10]. Many-core architectures exploit large caches to minimize off-chip memory latencies, in contrary to the scheduling of many threads. An example of such a many-core architecture is Intel's Larrabee [22].

Chapter 3

Motion estimation on a GPU

In order to evaluate the mapping and optimization process a CUDA programmer experiences, a non-trivial algorithm is mapped onto the GPU. To do so, a motion estimation algorithm is chosen as an example application. To completely understand the results, knowledge of the algorithm is required. Therefore, section 3.1 introduces an algorithm to perform motion estimation, known as the block matching algorithm. This block matching algorithm is mapped onto the GPU using two different mappings. The mappings are then compared for scalability, ease of implementation and performance. The two different mappings are introduced and discussed in section 3.2. For each of these mappings, several general optimization steps can be taken, which are given in section 3.3. The results from both mappings and optimizations are shown through benchmarks in section 3.4, concluding the mapping and optimization process experiences.

3.1 The block matching algorithm

Motion estimation is a technique to improve the perceived quality of video streams. The technique is used in the video compression standards such as MPEG-4 (consisting among others of DivX, H.264 and Blu-ray) and MPEG-2. With motion estimation, a new video frame is calculated based on the motion information available in two subsequent target frames. This process results in a higher frame-rate, resulting in a better viewer experience. One technique to implement motion estimation is block matching [5]. This section introduces the algorithm.

For block matching, the two source frames are divided into small blocks, with typical values of 8 times 8 or 16 times 16 pixels. Then, for each block in the first frame, the algorithm searches for a similar block in the second frame. Lastly, the block from the original frame is set in the new intermediate frame at a position in between the matched blocks of the two target frames. This way, motion is interpolated by means of a matching process between two

blocks. In more detail, three different steps can be distinguished for each block - called the reference block:

1. First of all, the reference block from the first frame must be compared with a number of candidate blocks from the second frame. Typically, this is done within a window, limiting the number of comparisons. For example, within a window of 32 times 32 pixels a total of 256 different 16 times 16 blocks must be compared. The comparison itself is done using a sum of absolute difference (SAD) technique[1], which will be introduced in this section.

2. The result of the comparisons denotes the similarity to the reference block. All obtained results need to be sorted and the best candidate needs to be found - the winning block. In the case of a SAD comparison, the candidate with the lowest SAD value is the winning block.

3. Finally, the motion vector can be calculated and, with it, the resulting intermediate frame.

The calculation of the SAD value between a reference and a candidate block is performed as follows. For every pixel in the reference block, the value of the corresponding pixel[2] in the candidate block is subtracted from it. Then, the absolute value is taken. In other words: the absolute difference is taken between two pixel values. Lastly, all these absolute differences are summed up, to obtain the sum of absolute difference (SAD) value of a block. The SAD value can be seen as an error value - if the SAD value equals zero, the blocks are exactly the same.

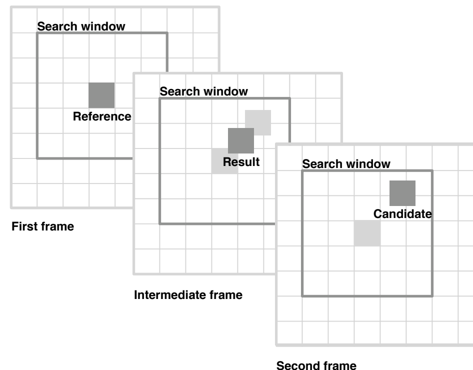

Figure 3.1: Motion estimation using block matching

[1] Other techniques are less used, but exist, such as mean squared error (MSE) or normalized cross-correlation function (NCCF) [5]

[2] In this case, a pixel is represented by its luminance value [5]

In figure 3.1, the block matching algorithm is depicted. In the first frame, a reference block and a search window are chosen. Then, the second frame - the rightmost in the figure - compares different candidates within the window. In the end, a winning block is chosen and a new block is set in the intermediate frame, as seen in the middle of figure 3.1.

Different variants of the block matching algorithm exist, reducing the number of arithmetic operations, but adding control operations [5]. Variants of the explained full-search are among others cross-search, spiral-search and N-step-search [5]. In this work, full-search is chosen, as it contains the smallest amount of control. The reason to choose an algorithm with the smallest amount of control is twofold. Firstly, the algorithm is less complicated, which makes it more suitable as an example algorithm. Secondly, algorithms with no or a small control structure map more efficiently on a GPU. The reader should keep in mind that any CPU implementation would perform significantly better using an alternative implementation, but for comparability reasons, full-search is used in both the GPU and the CPU implementations.

3.2 Comparing two different mappings

Because the mapping process of the block matching algorithm onto a GPU is not trivial, two different mapping approaches are considered. First, in section 3.2.1, the most intuitive - one kernel - approach is taken. But, as seen in section 3.1, the algorithm is dividable in three separate steps. Each step is then mapped onto the hardware as an individual kernel, resulting in the second approach, shown in 3.2.2.

3.2.1 The first mapping approach

The first mapping approach uses one kernel. As described in section 2.2, a division into threadblocks and threads has to be made. However, this division is not trivial and has an impact on performance, as threads within threadblocks can share a fast local memory and use a synchronization barrier. A mapping as depicted in figure 3.2 is chosen, which is explained and justified in this section.

First, the complete frame is divided into blocks with a size equal to the chosen reference block size, for example 16 times 16 pixels. These blocks are mapped 1 on 1 onto threadblocks. So, the number of threadblocks is equal to the number of reference blocks in the image. For a sample 720p[3] image, this division is depicted in figure 3.2. Since each threadblock now represents all the processing involved with one reference block, the details

[3] 1280 times 720 pixels

for each reference block can be described according to the three steps from section 3.1:

1. First, the reference block needs to be compared with each candidate block in the second frame. In order to do so, a number of threads is instantiated within the threadblock, equal to the number of candidate blocks available. The task of each thread is to calculate one SAD value, i.e., to compare one candidate block with the reference block.

2. After the first step is complete, the winning candidate needs to be found. The result of the previous step consists of a number of SAD values, equal to the number of threads in the threadblock. In order to find the winning candidate, the smallest SAD value needs to be found. This involves a comparison between all SAD values, which can be efficiently parallelized using an inverse tree structure, which is known as parallel reduction [11]. As the number of comparisons to be made at the start of the reduction is half the number of candidate blocks, half of the threads are completely idle during the whole operation, while other threads are partially idle due to the organization of parallel reduction.

3. When the winning candidate is known, pixels can be written to the intermediate frame. Ideally, one thread should write one pixel - the more threads, the better the latency is hidden. Because the number of threads in a threadblock cannot change within one kernel, it is equal to the number of candidate blocks and not to the number of pixels in a reference block. With smart search window and reference block size choices, this might lead to the ideal case, in which each thread can write one pixel. However, if one of the two parameters is set differently, threads have to write zero pixels or more than one pixel, depending on the parameters. If the parameters are unknown during design time, this fact will add a significant control overhead.

In pseudo-code, the complete algorithm as described and seen in figure 3.2 can be summarized as seen in listing 3.1.

As discussed before, within a threadblock, a fast shared memory is available. For the SAD comparison the reference block is used for every thread, while the candidate blocks are partially overlapping each other. A typical reference block can fit easily in the shared memory, leaving space to schedule other threadblocks with their reference blocks onto the same SM. In this way, the reference block is stored once in fast local memory and can be shared among each thread. In order to do so, each thread loads zero or more pixels from the reference block into the shared memory. Because the number of threads can differ from the number of pixels in a reference block due to parameters for the algorithm, control has to be added in the case of unfixed parameters at design time.

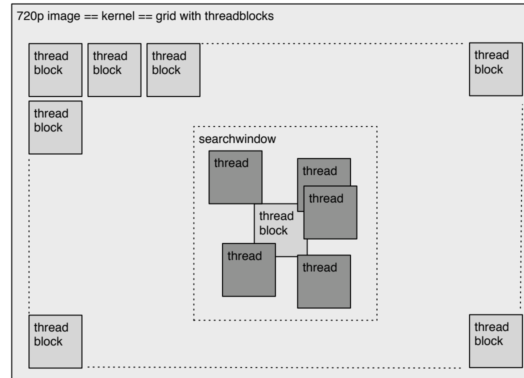

Figure 3.2: The first mapping approach

Listing 3.1: The first mapping in pseudo-code

```
for all threadblocks
    for all threads
        SADresults = SAD(referenceBlock, candidateBlock)
    end
    winningBlock = parallelReduction(SADresults)
    for all threads
        writeData(winningBlock)
    end
end
```

In the second step of the block matching algorithm, the shared memory is used again, this time to communicate all the resulting SAD values and to do the comparisons. Apart from the shared memory, grouping threads into threadblocks makes synchronization between threads possible. As seen from the algorithm, between each of the three steps a synchronization barrier is needed, since the algorithm must completely finish each step before being able to proceed to the next step.

The presented mapping approach has some drawbacks. Because the kernel contains three non-trivial steps, the kernel is relatively big. The CUDA programming guide suggests to use a lot small threads rather than a few bigger threads [21]. Although a single thread is quite big, the threadblock-size could still reach 256 (for typical block matching parameters) and the number of threadblocks could well go over 10000. Apart from the relatively large kernel, the mapping has several other drawbacks: it introduces dependencies between the hardware mapping constraints and the algorithm parameters:

- The number of threads within a threadblock is equal to the number of

candidate blocks - which is related to the window size. So a different window size selection leads to different performance characteristics. It can even lead to unsolvable constraints, as the maximum number of threads within a threadblock equals 512 and a typical window size already yields 256 threads.

- The size of a reference block is a parameter for the algorithm. However, a large reference block implies a lot of work per thread, whereas a smaller reference block will lead to smaller threads, which effects performance. For example, if the reference block is chosen to be very small, it makes no sense to load all the pixels of a reference block in the shared memory. The amount of control to divide this over all threads outweighs the gain of data re-using. The other way around shows that, if the reference block is very big, it consumes a lot of space in the shared memory, reducing the amount of threadblocks onto an SM to one or even to zero.

3.2.2 The second mapping approach

The second mapping approach is designed to reduce the size of the big kernel found in the first mapping. Using a smaller kernel means instantiating more threads - as the same amount of computation needs to be done - better hiding memory access latencies, as explained in section 2.5. A smaller kernel implies smaller threads - using fewer resources (registers and shared memory) - which enables the scheduler to allow more threads onto a single SM.

In the first mapping approach, only one kernel exists, which is executed once. This kernel then instantiates threadblocks containing the individual threads. The second approach however, introduces three kernels, each of them executed a large number of times. It splits the complete image in different reference blocks, analogue to the first approach, but now assigns the execution of three kernels to each reference block. This is best seen in figure 3.3 and the following pseudo-code listing:

```
for all referenceBlocks
   kernel1()
   kernel2()
   kernel3()
end
```

The reader should note that after the execution of any kernel, all registers and shared memories are reset, so no communication is possible between different kernels, except through the off-chip memory. Also, kernels are guaranteed to execute in order, so the above pseudo-code listing is sequentially executed. It constraints the execution order to $kernel1, kernel2, kernel3$ for each reference block individually, completing a reference block before starting another. The mapping specified in section 3.2.1 parallelizes the

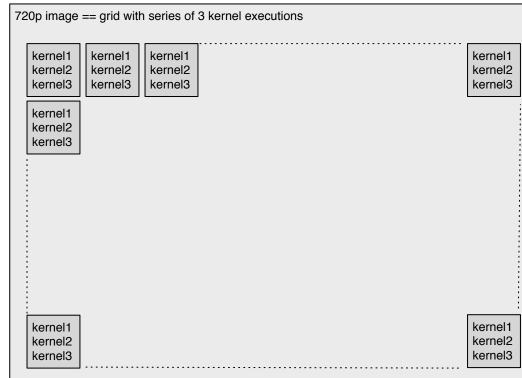

Figure 3.3: The second mapping approach

algorithm at the level of the reference blocks, introducing a possible out-of-order processing of reference blocks. The second mapping approach differs from this: parallelism is exploited within the processing involved of an individual reference block, with the mapping of reference blocks performed sequentially. Each step from section 3.1 now reflects one kernel:

kernel1 The first kernel calculates the SAD values for all candidate blocks. In contrast to the first approach, the calculation of the SAD values can now be divided into threadblocks instead of threads. Now, each SAD value is calculated by one threadblock, which is divided into threads according to the number of pixels in a candidate block. Each thread has a light task, calculating one absolute difference (AD) value between the reference block and its candidate block. When all threads calculated their AD value, the sum must be taken to obtain the SAD value. This is done using parallel reduction. The tasks of the first kernel are:

```
# Start of kernel1
for all threadBlocks
    for all threads
        AD(referenceBlock, candidateBlock)
    end
    SAD = parallelReduction(ADresults)
end
```

kernel2 The second kernel identifies the winning block, implemented using parallel reduction. Because parallel reduction does not benefit from more threads than half the number of SAD values, only one threadblock is used. The number of threads in the threadblock is limited - equal to half the number of threadblocks in the first kernel. The task of the second kernel in pseudo-code is:

```
# Start of kernel2 (one threadblock)
winningBlock = parallelReduction(SADresults)
```

kernel3 The third kernel also consists of one threadblock, with a number of threads equal to the pixels in a reference block. Each thread writes one value to the resulting intermediate frame:

```
# Start of kernel3 (one threadblock)
for all threads
    writeData(winningBlock)
end
```

The design of the three different kernels is depicted in figure 3.4. Because no fast communication is possible between any kernel, the SAD values resulting from the first kernel must be passed on to the second kernel through the off-chip memory. Then, the identification of the winning block calculated by the second kernel must be passed through the off-chip memory to be available for the last kernel.

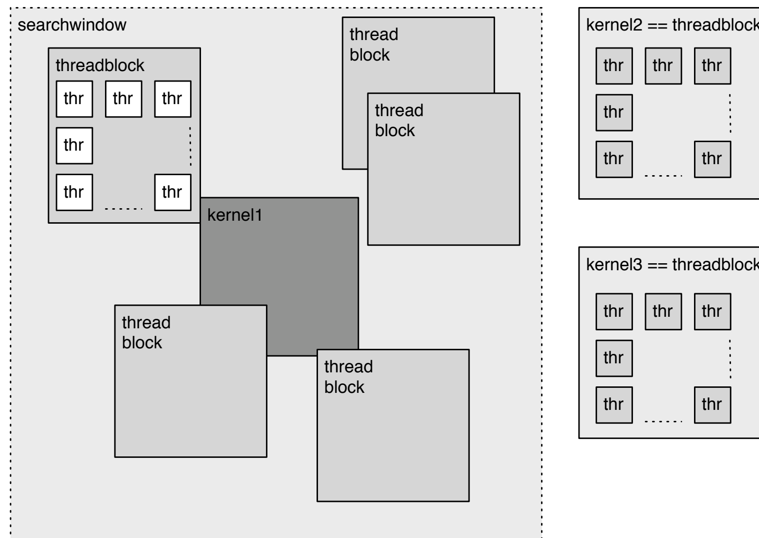

Figure 3.4: The second mapping approach's kernels

Although the mapping introduces additional memory accesses, a second parallel reduction and does not exploit parallelism at image-level, the mapping could still be competing with the first mapping, because of the use of smaller threads. The performance of the second mapping differs from the first mapping, mainly due to the following:

- Although the total number of threads is much higher than in the first mapping, the number of threads instantiated by a kernel in the second

mapping is comparable to the number of total threads instantiated in the first mapping. Since every execution of a kernel needs to be performed sequentially, the number of threads available at any time for scheduling is in the same order of magnitude in both approaches.

- In between the kernels of the second mapping, the off-chip memory is used to transfer the results from one kernel to another, which is not necessary in the first mapping. Additional memory accesses are necessary for the second mapping approach.
- The kernels in the second approach are smaller, allowing for more kernels to be scheduled onto an SM. This results in a larger pool of available warps, making room for better memory latency hiding.
- The second and the third kernel consist of only one threadblock each. This implies that, during the execution of these kernels, only one threadblock can be scheduled onto one SM. Depending on the specifications of the GPU, a number of SMs will always be idle. Even the SM that does perform work has a limited number of threads available for scheduling, possibly resulting in incomplete memory access hiding.
- The second mapping does not imply any data re-use within a threadblock, and is thus not able to benefit from the shared memory as the first mapping does.
- Both mappings introduce dependencies between the hardware mapping constraints and the algorithm parameters, but the dependencies are different. The second mapping approach implies that the number of pixels inside a reference block cannot be bigger than 512, but does not introduce dependencies for the amount of threadblocks or the window size.

3.3 Describing optimization steps

The mapping process of a sequential algorithm such as block matching onto a GPU does not exclusively involve the mapping of kernels, threadblocks and threads. Along with these mappings, mapping includes choices of memories, usage of special functions, accessing patterns for memories, restructuring of the algorithm and others. In this section, the most important improvements to the algorithm are briefly introduced:

Grouping pixels Typical RGB or YUV pixel values are represented by 8-bit values [5]. Since the hardware architecture works with 32-bit integers or floats, four pixel values can be grouped together into one 32-bit word. Using grouping, one memory read equals four pixel reads, stored in one register or shared memory place. This can save memory,

reduce memory loads, and does not introduces additional instructions, as vector datatypes are available within CUDA. Still, there is a drawback. Because each read now implies reading four pixel values, threads will have four times as much computational work to perform, reducing the total amount of threads.

Using the texture cache To use the texture cache, the image data needs to be binded to a texture. This process does not change the location of the data - it remains off-chip - but it changes name-wise from global memory to texture memory (see section 2.4.2). Then, for reads only, the texture cache will cache loaded data. Section 2.5 explains that in a typical case threads keep switching context. In most cases - as in the case of block matching - data re-use exists within threadblocks, but the benefit is minimal or non-existent as thousands of threads are executed in a TPC, all overwriting the 16KB of cache. Still, the texture cache is used, because it does not have any drawbacks. In a worst-case scenario, the algorithm will perform equally fast with or without caching.

Hardware specific functions The instruction set contains special instructions which would otherwise take multiple regular instructions. An example of this is the SAD function, which is implemented on the hardware as one arithmetic instruction with four operands (the result, the cumulative sum and the two target values) [19]. Using regular instructions, the SAD operation requires one add instruction (subtracting the two target values), one compare instruction (implementing the absolute value function) and two conditional add instructions (one adding and one subtracting the difference to the sum). Although the hardware SAD function reduces the number of instructions by a factor 4 for the part of the kernel containing SAD calculations, the speed-up cannot be calculated in this way, as instruction count may not be the bottleneck.

Using shared memory Since the shared memory can be accessed at register access speed, the shared memory can be exploited in different ways. As mentioned in section 3.2.1, the shared memory can be useful when data is re-used within threads in a threadblock. Also, the shared memory can be used (together with a synchronization barrier) to communicate data over threads, as is done for parallel reduction. Apart from this, complex calculations that yield the same result within a threadblock can be performed once and communicated through the shared memory. Finally, the shared memory can be used as an extension to the register file, providing more storage space.

Reducing control The sequential block matching algorithm contains con-

trol to minimize the amount of calculations needed. More advanced block matching algorithms, as described in section 3.1, introduce even more control, minimizing the amount of calculations. However, additional control can lead to worse performance on a GPU, as divergence within a warp can lead to the execution of both branches, not reducing the amount of calculations. In general, the more control is omitted, the faster a kernel executes [8]. Therefore, replacing control structures from the sequential algorithm with computations in the parallel algorithm, can improve performance in most cases.

Coalescing memory access The order in which the off-chip memory is accessed can be coalesced or not. Consider the case of threads accessing 4-byte data elements from an array located in the off-chip memory. If the index for the array increases by 4 whenever the thread ID increases by one, the memory accesses are coalesced and are handled efficiently by hardware, with a typical increase in performance of an order of magnitude over a non-coalesced memory access pattern [12].

3.4 Benchmarking the block matching algorithm

The performance[4] of the block matching algorithm mapped on a GPU is measured in different stages. First, a naive sequential implementation of the algorithm is executed on a 2009 mid-range CPU[5]. Then, both mappings from sections 3.2.1 and 3.2.2 are executed on a GPU in different stages of the optimization process. The optimization steps from section 3.3 are divided into three different stages:

Stage 1: Consists of a naive implementation extended with the usage of the texture cache, reduced control, the grouping of pixels and the usage of shared memory.

Stage 2: Stage 1 extended with the usage of additional shared memory and hardware specific functions.

Stage 3: The previous stage extended with coalesced memory access.

The results of the mentioned different stages for the two mappings combined with a naive CPU implementation are shown in table 3.1. Both the mean execution time and the speed-up[6] are shown, running on a G80 architecture with 16 PEs clocked at 800MHz. This GPU is commercially available as the GeForce 9400M and is positioned in 2008 as a low-power entry-level GPU.

[4] The algorithm is executed on a 720p image with a threadblock-size of 256
[5] Intel Core 2 Duo running at 2GHz
[6] Speed-up is compared to the CPU implementation

Platform and mapping	Optimization	Execution time	Speed-up
CPU	Naive	3.00s	1.0
GPU mapping 1	Stage 1	1.43s	2.1
GPU mapping 1	Stage 2	0.66s	4.5
GPU mapping 1	Stage 3	0.37s	8.1
GPU mapping 2	Stage 1	1.32s	2.3
GPU mapping 2	Stage 2	0.58s	5.2
GPU mapping 2	Stage 3	0.41s	7.3

Table 3.1: Benchmarking the block matching algorithm

Table 3.1 shows that the differences in performance between the two mappings are minimal. Therefore, it can be concluded that the differences presented in section 3.2.2 are insignificant and/or cancelling each other. To evaluate the scalability of both mappings, a second G80 GPU is introduced, containing 112 instead of 16 PEs. This GPU, the GeForce 8800GTS, runs at almost twice the clock frequency of the 9400M and is positioned in 2007 as a high-end GPU. The benchmarks show a speed-up between a factor 8 and 9 compared to the 9400M GPU for the various stages and mappings. This brings the speed-up compared to the CPU implementation ranging from a factor 20 (stage 1 optimizations) to 75 (stage 3 optimizations) for the first mapping approach. The second mapping shows a similar speed-up compared to the 9400M GPU.

Chapter 4

Visualizing GPU binaries

Since chapter 3 shows a long optimization and mapping process in order to achieve an efficient mapping, the question arises whether it is possible to automate parts of this process. Automating this can be done by improving the CUDA compilers, achieving a more efficient mapping automatically. However, since a GPU binary is unreadable and a part of the compiler is unspecified, compiler behaviour cannot be evaluated nor improved. To do so anyway, a GPU binary decoder needs to be developed. A motivation for the development of such a GPU binary visualizing tool is presented in section 4.1. By evaluating an existing decoder in section 4.2, the properties of GPU binaries are introduced to the reader. Then, with the development of a new decoding tool justified, a new tool is presented in section 4.3. The tool is positioned within the CUDA compilation flow and evaluated in section 4.4.

4.1 Automating the mapping and optimization process

In this section, the programming experiences with respect to the mapping of an algorithm onto a GPU using CUDA are evaluated, resulting in a motivation for the design of a GPU binary visualization tool. First of all, a typical programmers' timeline is given, modelled after the mapping of the block matching algorithm:

0. **Learning CUDA** When mapping the first algorithm, the programmer needs to learn the CUDA basics, including some background on GPU hardware.

1. **Creating a naive implementation** Assuming the target algorithm is suited for parallelization, the programmer needs to know little of the algorithm to create a naive implementation.

2. **Optimizing the algorithm** To create an optimized solution - comparable with stage 1 or 2 from section 3.4 - the programmer needs to know more about the algorithm, the GPU hardware and the CUDA thread model.

3. **Exploring different mappings** For a more efficient solution, the programmer might change from a naive mapping to another mapping, as is done with the block matching algorithm. To design an efficient mapping, the programmer might alter the algorithm itself.

4. **Full hardware usage** To make complete use of the hardware, an optimal mapping has to be designed, including optimizations for both memory and computation operations. In this step, the programmer might evaluate compiled code - optimizing the intermediate PTX code or the resulting GPU binary.

For block matching, performing steps 1, 2 and 3 yields a speed-up of a factor 75[1]. However, most programmers do not fulfill every above mentioned step [2]. Moreover, an estimation of typical programmer behaviour is shown in figure 4.1, along with estimations of typical speed-up factors on a high-end GPU compared to a naive CPU implementation[2]. Please note that the percentages in figure 4.1, representing the fraction of programmers choosing the corresponding path, are for illustration purposes only and do not represent actual data.

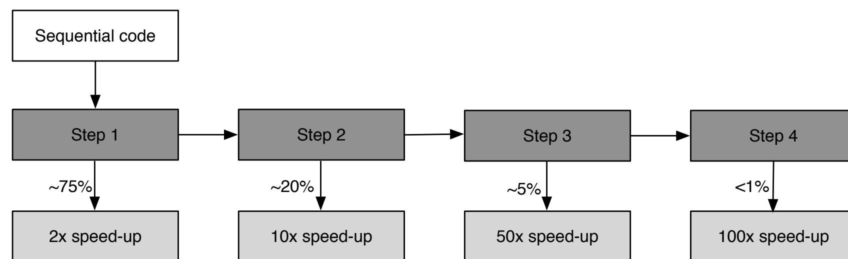

Figure 4.1: Behaviour of CUDA programmers

For an efficient mapping of any algorithm - including the block matching algorithm - the programmer needs to have knowledge of CUDA, the thread model, the GPU architecture and the algorithm itself. Apart from knowledge, in order to efficiently use the hardware, the programmer needs valuable time. As has been seen, parallel programming languages - such as CUDA - still demand knowledge and time from the programmer in order to efficiently use the hardware. To automate the mapping and optimization

[1]Speed-up compared to a naive CPU implementation using a 8800GTS GPU
[2]Estimations are based on the CUDA-zone showcases [17]

process, the CUDA compilation flow can be improved by either adding tools or modifying the compilers.

Currently, CUDA source-to-source optimizers exist [2]. These optimizers are positioned at the beginning of the CUDA compilation flow, as can be seen in figure 4.2. Shown in the same figure are adjustments to the nvcc compiler and PTX-to-PTX compilers, both areas currently being researched. However, the last part of the compilation flow - involving ptxas - is an area in which no research has been done. Since the source of ptxas is unavailable and the resulting GPU binary is unreadable, this part of the CUDA compilation flow is still a mystery. This chapter presents a decoder for GPU binaries, in order to propose improvements for ptxas, further automating the mapping and optimization process.

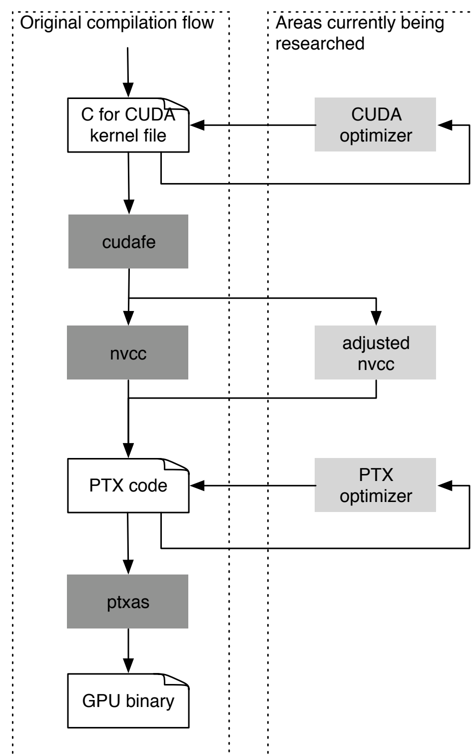

Figure 4.2: Automating the mapping and optimization process

4.2 Analyzing an existing decoder

To be able to analyze the ptxas compiler, the binary (which is in hexadecimal form) needs to be decoded into a readable, understandable language. In 2007, W.J. van der Laan created such a decoder, named *decuda*, decoding

GPU binaries into an assembly language close to PTX [23]. The decoder, designed through reverse engineering, has several drawbacks:

Design Since the designer of decuda started from an empty set of known instructions, the source code of the tool is chaotic. For every known rule, an additional if-then-else statement is introduced, expanding the source code while gathering more information and ending up with unreadable source-code.

Correctness Since decuda is solely based on reverse engineering, it cannot be proved that the decoding algorithm is correct. Since it is based on a finite set of test cases, it cannot be guaranteed that decuda decodes correctly in all test cases.

Usability Using decuda, information can only be extracted with understanding of the PTX virtual instruction set - or partly for any other assembly language. Then, in order to draw any conclusions, the obtained decoded instructions need to be analyzed and compared with the PTX or CUDA source code.

4.3 The design of CUDAvis

In order to overcome the mentioned drawbacks of decuda in section 4.2, a new decoder is introduced, named *CUDAvis*[3]. The design of CUDAvis can be divided into two stages - the decoding stage and the visualization stage. Both stages are depicted in figure 4.3. First, in the decoding stage, instructions are identified. From the GPU binary, individual instructions are passed on to the instruction-type decoder, which uses a look-up-table to identify the instruction-type and the operands' locations and types. Then, the instruction is passed on to the operands decoder, which uses rules from a database, able to modify the operands' locations and types found in the previous step. After all instructions are decoded, the branches and labels are identified. The resulting data is stored in assembly format and as a data structure.

The resulting data structure is accessed in the visualization stage. The instruction and operand visualizers create a visual interpretation for each instruction and operand in the binary. Then, branches and labels are visualized, followed by a register live range checking algorithm. This results in a full view of every register, indicating if it is occupied, written, read or unused. Finally, a bottleneck identifier highlights possible bottlenecks, giving feedback to the programmer.

[3]CUDAvis is short for *CUDA visualization tool*

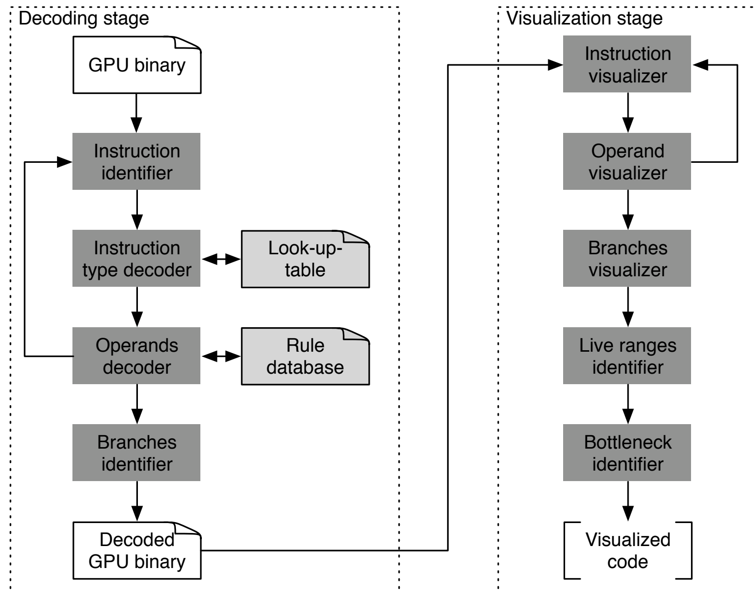

Figure 4.3: The design of CUDAvis

4.4 Evaluation of CUDAvis

The development of CUDAvis results in the removal of decuda's drawbacks:

Design Since decuda's result can be re-used in CUDAvis, the design can be made more elegant. Because the GPU includes a hardware instruction decoder, hardware exists that mimics the behaviour of decuda. This hardware is designed using general rules and look-up-tables instead of thousands of small rules. CUDAvis is designed with the idea to mimic the hardware instruction decoder, using look-up-tables and general rules. In the design of the tool, data is separated from control and saved in pre-defined structures. The result is readable, compact and intuitive code, leaving room for adjustments and expansions.

Correctness Since CUDAvis follows general guidelines and uses look-up-tables, the probability of correctness compared to decuda is increased. Still, full correctness cannot be guaranteed.

Usability After decoding, CUDAvis generates a visual impression of the resulting assembly code, identifying among others loop structures, basic blocks, memory accesses, register live ranges and bottlenecks.

CUDAvis is positioned within the CUDA compilation flow as seen in figure 4.4. The usage of CUDAvis is twofold:

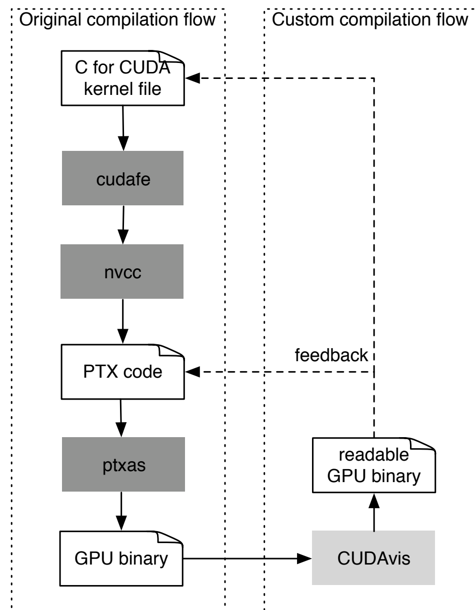

Figure 4.4: CUDA compilation flow extended with CUDAvis

- Firstly, CUDAvis is used by programmers to evaluate compiled code. Programmers can identify structures, register live ranges and bottlenecks in the code, which may not be present or visible in high-level CUDA or intermediate PTX code. Programmers can then alter and recompile high-level code, to remove bottlenecks or to avoid unwanted steps taken by any of the two compilers.

- Secondly, CUDAvis is used to evaluate the behaviour of ptxas. In chapter 5, a number of improvements to ptxas is proposed, resulting in more automated optimizations within the CUDA environment.

The results of both the decoding and the visualization stage are shown in figure 4.5, serving as an illustrating example.

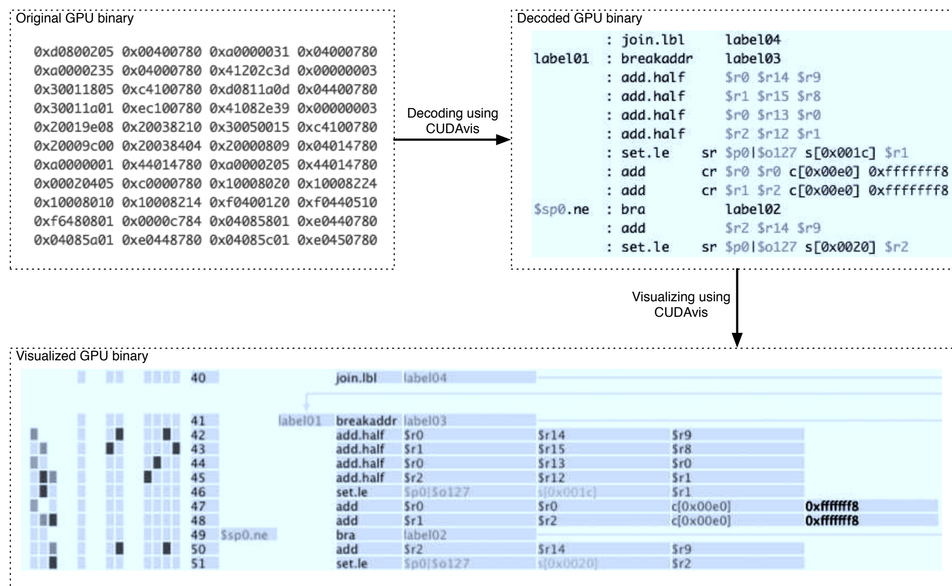

Figure 4.5: Example usage of CUDAvis

Chapter 5

Compiler improvements

With the tool introduced in chapter 4, the ptxas compiler can be evaluated and adjustments can be proposed. Since ptxas' source code is not available, all improvements are presented as algorithms using pseudo-code. This chapter uses the first mapping of the block matching algorithm as a running example. The different adjustments are introduced in different sections. First, in section 5.1, non-linear register allocation is discussed. Next, in section 5.2, the possibilities of value recalculation are discussed, along with, in section 5.3, a technique to more efficiently schedule instructions. Then, in section 5.4, the idea of using the shared memory as a register file is introduced. Finally, in section 5.5, redundant instructions are identified and removed. This chapter will conclude with an evaluation of all proposed improvements in section 5.6.

5.1 Non-linear register allocation

Currently, ptxas performs linear register allocation. That is, ptxas tries to minimize the register count. In subsection 5.1.1, the current register allocation technique is discussed. It is shown in subsection 5.1.2 - in which an alteration to the current technique is given - that a non-linear approach to register allocation can show an improvement to performance. The non-linear register allocation technique is presented as a proposed improvement for ptxas through pseudo-code and is analysed in subsection 5.1.3.

5.1.1 Current register allocation techniques

The main task of ptxas is to assign variables to locations in the register file. In PTX, variables are mapped to virtual registers. In other words, corresponding register names in PTX denote corresponding variables. However, in PTX, an infinite number of registers is assumed, and therefore, no register is re-used to store another value. In compiler theory, research for optimal

register allocation has been done extensively in the past [3]. Register allocation algorithms come in two flavors, one for fixed-size register allocation and one for minimum register allocation. In ptxas, the latter is used.

In G80 hardware, each streaming multiprocessor (SM) has a register file, consisting of 8192 registers, each able to store a 32-bit word. As mentioned before in section 2.4, these registers are divided over the threadblocks running on the SM. Then, for each threadblock, the registers are divided over the threads running in a threadblock. A maximum of 768 threads are allowed on one multiprocessor. Please keep in mind that, as stated in section 2.5, the more threads are running on a multiprocessor, the more the off-chip memory latencies can be hidden.

Let us look at an example, using G80 hardware and a threadblock-size of 256 threads. If one thread would use 11 registers, one threadblock uses 2816 registers. In this case, 512 threads (two threadblocks) can be scheduled onto one streaming multiprocessor, consuming 5632 registers. Three threadblocks will not fit, as they would use more than the available 8192 registers. Now, consider a second example in which one thread would use 16 registers. In this case, still two threadblocks (4096 registers each) can be scheduled onto one multiprocessor, using all the available registers.

From this example it can be seen that any register allocation optimization done on a 16-register-per-thread 256-threads-per-block program yielding 11 or more registers will not contribute to any speed-up. In fact, it could even be slower, as register allocation optimizations can introduce additional instructions. To summarize the conclusions drawn from the example, a table is drawn, showing hardware usage for different numbers of registers-per-thread for a threadblock-size of 256[1]. This is seen in table 5.1, which shows a non-linear number of threads per SM when evaluating against the number of registers per thread.

5.1.2 Proposed technique for register allocation

Since the unused registers from the examples in 5.1.1 and from table 5.1 are not used for any other purpose, it is not necessary to try to reduce the number of registers to a value as small as possible. As seen in the example, for a threadblock-size of 256 and a G80 architecture, it is only beneficial to reduce the number of registers per thread to a value of 10 (yielding 768 threads per *SM*), 16 (yielding 512 threads per *SM*) or to 32 (yielding 256 threads per *SM*). Thus, the register allocation algorithm should not be linear, but step-wise.

To implement this in ptxas, an algorithm needs to be found, valid in all cases. Define the number of registers available per streaming multiprocessor as *totalRegisters*, the threadblock-size as *threadblockSize* and the number of

[1] Please note that 256 is an example value. Any other threadblock-size value will yield a similar conclusion

Registers per thread	Threads per SM	Register usage	Register usage (%)
0-9	768	-	-
10	768	7680	94
11	512	5632	69
12	512	6144	75
13	512	6656	81
14	512	7168	88
15	512	7680	94
16	512	8192	100
17	256	4352	53
18-32	256	-	-

Table 5.1: Hardware usage for different register usages

registers per thread as $threadRegisters_n$ (in which n denotes the number of the boundary). Then, the formula

$$threadRegisters_n = \left\lfloor \frac{totalRegisters}{768 - n \cdot threadblockSize} \right\rfloor,$$

$$\text{with } 0 < n < \left\lfloor \frac{768}{threadblockSize} - 1 \right\rfloor$$

gives us the boundaries for register allocation. In the case of a threadblock-size of 256, the boundaries between 768 and 512 threads, 512 and 256 threads and between 256 and zero threads per *SM* can be found using

$$threadRegisters_0 = \left\lfloor \frac{totalRegisters}{768} \right\rfloor,$$

$$threadRegisters_1 = \left\lfloor \frac{totalRegisters}{512} \right\rfloor \text{ and}$$

$$threadRegisters_2 = \left\lfloor \frac{totalRegisters}{256} \right\rfloor.$$

In the last case, only one threadblock can be scheduled onto a streaming multiprocessor. If, however, more registers are used than available, register spilling (discussed in section 5.4) is used to ensure that the kernel can be scheduled onto the hardware.

Now, with these boundaries known, ptxas can extract the variables - from the architecture (*totalRegisters*) and from the CUDA code (*threadblockSize*) - and calculate the boundaries. The compiler tries to meet the first boundary ($threadRegisters_0$), then the second ($threadRegisters_1$) and so on. If none of these boundaries are met, the compiler should use a minimum register

Listing 5.1: Non-linear register allocation

```
threads = 768
finished = false
while finished == false
    result = fixedRegisterAssignment(boundary(threads))
    if result == succes
        finished = true
    else
        threads = threads - threadblockSize
    end
    if threads < threadblockSize
        minimumRegisterAssignment
        finished = true
    end
end
```

allocation and use register spilling for any register not meeting any boundary. In pseudo-code, the complete algorithm can be seen in listing 5.1.

In words, the algorithm can be described as follows. First, the boundaries are tested in reverse order. If any of the boundaries are met, the process stops. If none of the boundaries are met, a minimum register allocation algorithm is applied, using register spilling to ensure scheduling on the hardware. Using the presented algorithm, the amount of registers per thread will always be equal to one of the boundaries, resulting in optimal hardware usage. In the case of the G80 and a threadblock-size of 256, this means threads will either use 10, 16 or 32 registers.

Please note that the presented algorithm abstracts from some use-cases. Firstly, it must only be applied if register usage per thread is the limiting factor for thread scheduling. Also, in the case when the user specifies less than 768 threads, a slightly modified variant of the algorithm needs to be applied. Or, in the case of a limit caused by the shared memory usage of a thread, a similar algorithm needs to be applied, but now assigning registers depending on the usage of shared memory per threadblock.

5.1.3 Analysis of non-linear register allocation

Even though the proposed technique for non-linear register allocation seems promising, the actual speed-up might be small or non-existent. Since the original linear register allocation algorithm would find the smallest register allocation and the non-linear algorithm will have more freedom and thus use more registers, the number of threadblocks running on one multiprocessor will not increase. However, a relaxation of the register constraints may contribute to several different aspects:

Compile time Having weaker register constraints can reduce compile time. For just-in-time compilation, this can be a useful achievement, but in most cases reduction in compilation time is not regarded as a valuable improvement.

Instruction re-ordering Since the compiler has more available registers, it will also have more freedom with instruction re-ordering. To benefit from this, the compiler needs to know when to apply re-ordering, which will be discussed in section 5.3.

Value recalculation If the compiler knows it has to save a few registers to reach a boundary in the non-linear algorithm, special techniques can be applied to use fewer registers, while introducing additional instructions. One such a solution is value recalculation, discussed in 5.2.

So, in general, it can be concluded that the use of a non-linear register allocation algorithm benefits from other compiler techniques, which are currently not implemented in ptxas, but are proposed in this chapter.

5.2 Value recalculation

Currently, ptxas does not provide the ability to use value recalculation as a register-saving technique. However, value recalculation can save registers during bottlenecks, recalculating discarded register values whenever needed. In subsection 5.2.1, an example is taken to show that ptxas does not perform value recalculation in cases where it intuitively should. Then, in subsections 5.2.2 and 5.2.3, an algorithm is proposed to perform value recalculation. The results of this improvement will be shown in subsection 5.2.4.

5.2.1 Block matching as an example

Let us take the example of the block matching algorithm as described in section 3. After visualizing the kernel code using the tool CUDAvis from section 4.3, it can be observed that four variables are kept alive during the entire kernel and are only used sporadically. These variables are grouped in two types:

- The variables *threadIdx.x* and *threadIdx.y* denote the position of a thread within a threadblock (in this case in two dimensions), used by CUDA to distinguish threads from each other. These variables are unique for each thread and are stored in a special register file. The compiled block matching kernel converts the two variables to another format (using the *cvt* instruction), and stores them in a regular register. This is done in the first two lines of the kernel. Then, they are

read in lines[2] 4, 5, 6, 44, 45, 123, 124, 128, 131, 312 and 314. At the moments *threadIdx.x* and *threadIdx.y* are read, there is no register bottleneck. However, in between, the variables are kept alive, with register bottlenecks at lines 23 and 90 of the kernel. Recalculation is not performed by ptxas, but only takes four instructions (two variables, two bottlenecks) and save two registers per thread, allowing for a potential scheduling of more threads on the SM.

- The variables *blockIdx.x* and *blockIdx.y* are analogue to the thread identifiers, but represent the coordinates of a threadblock in the grid. These variables are equal for all threads in a block, and are stored in a special shared memory. In the block matching kernel, they are both multiplied by a constant and stored in regular registers in the first lines of the compiled code. Then, they are read in lines 8, 11, 42, 43, 50, 313 and 318. The same conclusions can be drawn as for the thread identifiers - they are kept alive during the register bottlenecks in the code, but require only one instruction each to recalculate.

The example shows potential for register saving through value recalculation. Still, the conclusions from section 5.1 must be kept in mind: value recalculation is only interesting if a boundary is crossed, otherwise it only introduces additional instructions, resulting in poorer performance.

5.2.2 When to apply value recalculation

In section 5.2.1 it is suggested that value recalculation is useful to apply for the example case. Thus, an algorithm needs to be found to be able to determine if a value should be kept alive or recalculated. Intuitively, this algorithm compares the drawback (additional instructions) to the potential gain (fewer registers). The potential gain can be expressed as the crossing of a boundary using the non-linear register allocation technique. Crossing one boundary can introduce up to twice as many threads, leading to a speed-up of a factor 2 in an ideal case. This speed-up can be reached only if the bottleneck of the kernel is determined by memory latency, and the number of threads scheduled is not enough to hide this latency.

To decide for the recalculation of a value, the compiler needs to know certain properties of the algorithm, which are discussed below:

- Firstly, it needs to know when a boundary is crossed. Using non-linear register allocation, the compiler knows whenever the crossing of a boundary occurs. Only when a boundary can be crossed, value recalculation should be considered as an option.

[2] Please note that the line numbers of compiled kernel instructions are introduced for illustration purposes and do not refer to any actual code included in the document

- Secondly, the compiler needs to know where the register usage bottleneck is situated in the kernel. If the value that needs to be recalculated is used at the moment of the bottleneck, it should not be recalculated, because it would not save any registers. Value recalculation should only be applied if the register for the target value is kept alive during the time of the bottleneck, but not used. The compiler can calculate the bottlenecks for register usage and the accesses to the target value, so it can see if this condition holds.

- Then, the compiler needs to know the number of instructions required to recalculate the target value. This can be extracted from the code itself, and does not need to present any problem for the compiler. The type of instruction should be evaluated, introducing additional accesses to the off-chip memory will not likely lead to any performance increase.

- Lastly, the compiler needs to know how much the crossing of a boundary will contribute to better performance due to the scheduling of more threads on the hardware. However, this could be a problem. The compiler can analyse the structure of the code and do a prediction on how much the performance of the kernel depends on the number of threads. However, this can be fairly complicated and inaccurate.

To illustrate this, an example is taken. Say, a kernel consists of 300 instructions and a target variable is kept alive during the bottleneck of the register usage. In this example, the reduction of this one register will lead to the crossing of a boundary, but will also introduce 30 computation instructions[3]. Combined, this will introduce 10% more instructions. The compiler then has to estimate the potential gain and decide whether to perform value recalculation.

5.2.3 An algorithm for value recalculation

In 2009, S. Hong and H. Kim introduced an analytical model to completely describe the performance of the execution of a kernel on a GPU [13]. Though their result shows to be quite accurate, the model considers many more aspects than needed for the evaluation of the potential gain when introducing value recalculation. Therefore, a new algorithm is developed, satisfying the rules mentioned in section 5.2.2. First, a number of variables are defined:

- The variable $threadGain$ is defined as the additional number of threads to be scheduled whenever a boundary is crossed. This value corresponds to the threadblock-size: $threadGain = threadblockSize$.

[3] Computation instructions are considered instructions not accessing the off-chip memory, so a shared memory load for example is considered a computation instruction

- Secondly, the amount of warps needed to hide the memory access latencies is given by

$$memAccessHiding = \frac{memoryAccess}{4 \cdot (compOps + 1)},$$

with *memoryAccess* denoting the memory access time in clock cycles and *compOps* the number of computation operations prior to a memory access [4]. As seen in section 2.5, other warps execute instructions while the first warp waits for their memory accesses.

- With the minimal amount of threads known to hide the memory latency, a potential thread shortage (*threadShortage*) can be calculated as *currentThreads* subtracted by *memAccessHiding*. Note that the shortage can be negative, in which case the memory access latencies are completely hidden and the addition of more threads will not result in a speed-up.

- Lastly, *additonalOps* is introduced. This variable is defined as the number of additional computation instructions added to the kernel for the recalculation of a target variable. These instructions are assumed to be added after the memory load, but the presented algorithm can be adjusted in case of the additional instructions being scheduled before the memory load.

Using the above notations, the algorithm can be presented. It is based on partitioning the kernel into blocks of (zero or more) computation instructions followed by exactly one memory access. With this partitioning, every block can be evaluated in terms of computation instructions compared to the number of threads. This way, a value recalculation can be justified or not. The algorithm is presented in pseudo-code in listing 5.2, using the introduced notations.

To illustrate the usage of the algorithm, an example is taken. Assume a threadblock-size of 256 ($threadGain = threadblockSize = 256$) and a possibility to perform value recalculation by using 2 additional operations ($additonalOps = 2$), which is said to cross a boundary. In an example block of instructions, *compOps* is equal to 14. This means that every warp has a total of 60 (*compOps* plus the memory operation, multiplied by four) cycles of work, before it has to wait for a memory access, which takes 600 cycles in this example case ($memoryAccess = 600$). Then, the minimum number of threads for memory access hiding is calculated, yielding 10 warps (or 320 threads). After subtracting this value from the actual number of threads, the shortage is known: 2 warps (or 64 threads) in the example.

[4] One instruction (computation or memory) is assumed to be executed in four clock cycles, as part of an SIMD warp [14]

Listing 5.2: Value recalculation

```
threadGain = threadblockSize
computation = 4 * (compOps + 1)
memAccessHiding = memoryAccess / computation
threadShortage = memAccessHiding * 32 - currentThreads
if (threadShortage > 0)
   speedUp = (min(threadGain, threadShortage)/32)*computation
   speedDown = threadGain * additonalOps * 4 / 32
   if (speedUp > speedDown)
      performValueRecalculation()
   end
end
```

Since this value is positive, adding more threads will result in a speed-up, which yields 120 ($\frac{64}{32} \cdot 4 \cdot (14 + 1)$) cycles per threadblock. This value is then evaluated against the drawback - the addition of instructions - which is equal to the added instructions multiplied by the number of threads in a threadblock: $2 \cdot 256 = 512$ instructions per threadblock. In a warp, 32 instructions execute in 4 cycles, yielding $\frac{512 \cdot 4}{32} = 64$ cycles per threadblock. Because the drawback (64) is smaller than the gain (120), value recalculation will be performed in this case.

It should be noted that the algorithm in listing 5.2 is only applicable to an algorithm consisting of zero or more computation instructions followed by one memory access. The algorithm can be extended to support multiple blocks of computations followed by memory accesses, so that it can draw a conclusion for the complete kernel whether value recalculation is useful. For a typical simple kernel, the above algorithm will suffice, but bigger algorithms may introduce scattered memory accesses and synchronization points. However, the nature of the algorithm remains the same.

5.2.4 Results of value recalculation

Along with non-linear register allocation, value recalculation can provide a speed-up in certain cases. With the proposed algorithm, value recalculation will not be applied when yielding negative effects on performance. With a slightly modified algorithm, the value recalculation algorithm can be applied to any kernel code as discussed in section 5.2.3.

For the block matching algorithm, value recalculation is proven to be useful. With the addition of 8 instructions, the four variables seen in section 5.2.1 can be recalculated twice. In this way, the four registers are free during all bottlenecks in the code, resulting in more threadblocks per SM. In the case of block matching, this results in a minor speed-up, due to already enough active threads executing on an SM at any given time.

Listing 5.3: Block matching as an example

```
texture[read{$tex1},index{$r1,$r2},write{$r1,$r2,$r3,$r4}]
texture[read{$tex2},index{$r5,$r6},write{$r5,$r6,$r7,$r8}]
shared[read{$r1},write(sharedMemory)]
shared[read{$r2},write(sharedMemory)]
shared[read{$r3},write(sharedMemory)]
shared[read{$r4},write(sharedMemory)]
shared[read{$r5},write(sharedMemory)]
shared[read{$r6},write(sharedMemory)]
shared[read{$r7},write(sharedMemory)]
shared[read{$r8},write(sharedMemory)]
```

5.3 Efficient instruction re-ordering

Currently, efficient instruction re-ordering is not performed by ptxas. In subsection 5.3.1, an example is taken to motivate for instruction re-ordering. Then, subsection 5.3.2 proposes to automatically perform re-ordering.

5.3.1 Block matching as an example

When analyzing the block matching algorithm with CUDAvis, the pattern seen in listing 5.3 can be identified a number of times. In words, the first two instructions write 8 values into the register file, which are read in the next 8 instructions. After these instructions, the registers are not used again, since the data is stored in the shared memory. In table 5.2, the register usage can be seen. The table shows if a register is empty ('-'), live ('l'), read ('r'), written ('w') or both read and written ('a'). The register status is evaluated after each instruction in the code pattern shown in listing 5.3.

	1	2	3	4	5	6	7	8
1	a	a	w	w	l	l	-	-
2	l	l	l	l	a	a	w	w
4	r	l	l	l	l	l	l	l
5	-	r	l	l	l	l	l	l
6	-	-	r	l	l	l	l	l
7	-	-	-	r	l	l	l	l
8	-	-	-	-	r	l	l	l
9	-	-	-	-	-	r	l	l
10	-	-	-	-	-	-	r	l
11	-	-	-	-	-	-	-	r

Table 5.2: Register occupation in the block matching example

Listing 5.4: Block matching as an example (2)

```
texture[read{$tex1},index{$r1,$r2},write{$r1,$r2,$r3,$r4}]
shared[read{$r1},write(sharedMemory)]
shared[read{$r2},write(sharedMemory)]
shared[read{$r3},write(sharedMemory)]
shared[read{$r4},write(sharedMemory)]
texture[read{$tex2},index{$r5,$r6},write{$r1,$r2,$r3,$r4}]
shared[read{$r1},write(sharedMemory)]
shared[read{$r2},write(sharedMemory)]
shared[read{$r3},write(sharedMemory)]
shared[read{$r4},write(sharedMemory)]
```

As can be seen from table 5.2, the peak register usage equals 8. However, listing 5.3 can be changed into the pattern as seen in listing 5.4. Then, the register usage changes at the bottleneck from 8 to 6 registers, resulting in occupations as shown in table 5.3.

	1	2	3	4	5	6
1	a	a	w	w	l	l
2	r	l	l	l	l	l
3	-	r	l	l	l	l
4	-	-	r	l	l	l
5	-	-	-	r	l	l
6	w	w	w	w	r	r
7	r	l	l	l	-	-
8	-	r	l	l	-	-
9	-	-	r	l	-	-
10	-	-	-	r	-	-

Table 5.3: Register occupation after modification

Changing the order of the instructions does not only alter the number of registers needed, it also affects performance due to different scheduling for the texture load instructions. Since the hardware switches to other warps when it encounters a high latency memory operation a high number of threads are needed to hide the memory latency. However, if threads have additional computation instructions in between high latency operations, fewer threads are needed to hide the memory latency. With the scheduling of computation instructions in between memory loads - as done in the example - less threads are needed to hide the memory latency. This results in a higher performance in the cases when the memory latency is not entirely hidden.

5.3.2 Automatically applying instruction re-ordering

Although instruction re-ordering can provide two clear advantages - a smaller register usage and better memory latency hiding - the example from section 5.3.1 must be performed by hand, as ptxas does not perform it automatically. The design of an instruction re-ordering algorithm for register minimization is not presented in this work, since it has been thoroughly analyzed by others and is widely used in compilers. Examples can be found in any compiler theory handbook and can be implemented in ptxas with little adjustment.

5.4 Using the shared memory as a register file

As mentioned before in sections 2.4 and 5.1, register spilling to the off-chip memory is a technique used by ptxas to ensure a valid mapping for threads with heavy register usage. To fit at least one threadblock onto an SM, register spilling to the off-chip memory is performed as soon as the available registers are all used. Since the off-chip memory is relatively slow, kernels that need register spilling are underperforming, forcing programmers to redesign their kernels to fit onto the register files. Since a fast shared memory is available on-chip, register spilling can instead be performed onto the shared memory. A simple algorithm is presented in this section.

The algorithm first calculates the amount of registers to be spilled per threadblock (*spillingReg*) as

$$spillingReg = threadReg \cdot threadsblockSize - totalReg,$$

in which *totalReg* is an architecture constant and *threadReg* and *threadblockSize* are algorithm specific. Then, if the calculated value is positive, the amount of unused shared memory *freeShared* is calculated as

$$freeShared = totalShared - usedShared,$$

in which $totalShared$ is an architecture constant and $usedShared$ is algorithm specific. If the amount of free shared memory space is enough to fit all spilled registers for the scheduled threadblock, registers spilling is performed onto the shared memory. Else, register spilling can be performed to the off-chip memory and possibly partly to the shared memory, as long as there is space available. The presented algorithm is listed in pseudo-code in listing 5.5. Since shared memory cannot be accessed in all instructions as register values can, spilled registers are moved to the register file at points where they are needed, analogue to the off-chip register spilling technique currently used in ptxas.

Since the shared memory access times are comparable to register access times, the only drawback of register spilling is the addition of instructions to move spilled registers from or to the shared memory. The new register

Listing 5.5: Register spilling onto the shared memory

```
spillingReg = threadReg * threadblockSize - totalReg
if (spillingReg > 0)
   freeShared = totalShared - usedShared
   if (freeShared > spillingReg)
      performSpillingShared()
   else
      performSpillingSharedPartly(freeShared)
      performSpillingOffChip()
   end
end
```

spilling technique removes high latency off-chip reads and writes, enabling previous inefficient mappings to run efficiently on the GPU, extending the possibilities of different mappings.

5.5 Removing redundant instructions

Evaluation of GPU binaries shows several occurrences of redundant instructions. The first two subsections of this section present examples to illustrate the possibilities of redundant instruction removal, while the third subsection presents and evaluates a technique to implement this in ptxas.

5.5.1 Multiple registers for the same value: an example

In the block matching algorithm, the following (simplified) pattern can be identified in high level CUDA code a number of times:

```
value3 = texture1[value1 + value2]
value4 = texture2[value1 + value2]
value5 = texture3[value1 + value2]
```

In which *texture*1, *texture*2 and *texture*3 denote three different arrays in texture memory and *value*1, *value*2, *value*3, *value*4 and *value*5 are register values. Because the compiler identifies the re-use of the index value for the three different arrays, it calculates it only once. The following pseudo-code listing is obtained from the compiled binary, found using CUDAvis:

```
add[read{value1},read{value2},write{$r0}]
move[read{$r0},write{$r1}]
move[read{$r0},write{$r2}]
texture[read{$tex1},index{$r0},write{$r0}]
texture[read{$tex2},index{$r1},write{$r1}]
texture[read{$tex3},index{$r2},write{$r2}]
```

As can be seen, the compiler ensures that the index value is pre-calculated, but then copies it to both $\$r1$ and $\$r2$, storing it a total of three times.

Each of the texture reads now read from a different register with the same contents. This is obviously a waste of resources and can be changed into the following listing without adjusting the behaviour of the code:

```
add[read{value1},read{value2},write{$r2}]
texture[read{$tex1},index{$r2},write{$r0}]
texture[read{$tex2},index{$r2},write{$r1}]
texture[read{$tex3},index{$r2},write{$r2}]
```

In this case, the total number of instructions is reduced. There are also cases in which the removal of redundant copy instructions results in a reduction of the number of used registers.

5.5.2 Multiple instructions for one calculation: an example

The second example shows redundant computation instructions. Using CUDAvis, the following lines of GPU binary code are decoded:

```
shiftleft[read{$r0},write{$r0},amount{3}]
shiftright[read{$r0},write{$r0},amount{1}]
```

In words, a value is first shifted left by 3 - multiplied by 8 - and then shifted right by 1 - divided by 2. Obviously, this equals a shift left of 2 - a multiplication by 4. Therefore, the above listing can be changed without altering the behaviour into:

```
shiftleft[read{$r0},write{$r0},amount{2}]
```

5.5.3 Evaluation of redundant instruction removal

From the two examples, it can be seen that GPU binaries contain redundant instructions. These two types of redundant instructions can be removed using the ptxas compiler:

- Removing multiple registers with the same data requires a pass over all move instructions. Move instructions with two registers as operands are target for removal. However, it appears that, from thorough binary code analysis, the texture read function seen in section 5.5.1 has restrictions. Apparently, the index value has to be the same as the target register[5]. This justifies the register movement, which can therefore not be eliminated as done in the example. The question arises if this is an instruction set restriction or a hardware restriction. In the first case, the instruction set can be adjusted, enabling the removal of redundant instructions.

  Apart from the move instructions occurring in the example, move functions between two registers also occur without being followed by a

[5] If there are multiple index values, the index values must overlap the first target registers

texture read. These move functions might possibly be introduced to overcome certain restrictions, in which case they are not redundant. More knowledge on the instruction set and the hardware would enable alterations to the instruction set, enabling the removal of redundant move instructions.

- The second example requires a pass over all computation instructions, tracking each value and identifying redundant computations. Of course, the compiler must then check if the merging of two instructions does not lead to erroneous data due to instructions using the intermediate data.

5.6 Evaluation of the improvements

With all five improvements evaluated individually, it can be stated that the modification of ptxas with the proposed improvements has a potential to yield significant improvement. Some individual improvements show a potential for more than a factor of 2 speed-up, given the right circumstances. Together, the speed-up of all improvements can be noticeable on certain algorithms. If ptxas would be improved as shown in this chapter, all existing algorithms could benefit from a set of potential improvements at no cost, reducing development time and/or yielding better performance.

However, since the improvements are not actually implemented in the compiler, no numbers are known on the potential of the improvements, only guesses can be made. To obtain any numbers on the actual benefits, either NVIDIA has to adjust its compiler, or a ptxas clone with the proposed improvements has to be designed from scratch.

Chapter 6

Functionality changes

With the tool CUDAvis from section 4, a few weak points are found in the CUDA environment which are not able to be solved by adjusting the toolchain. This section proposes functionality improvements for the hardware of NVIDIA's G80 family and alike. First, in section 6.1, a way to transfer memory from the texture cache to the shared memory is proposed. Secondly, in section 6.2, a second (non-cache) mode for the texture cache is introduced as a hardware improvement.

6.1 From the texture cache to the shared memory

In the block matching algorithm, both the shared memory and the texture cache are used to improve performance, as has been discussed in section 3.3. A significant amount of variables read from texture memory (either hitting or missing in the cache) are stored directly into the shared memory. However, when evaluating the decoded GPU binary using CUDAvis, such a load from texture memory is translated into multiple instructions. The reason for this lies in the limitations of the texture reading instruction: the result can only be stored in one (8/16/32-bit read) or more (vector read) registers. In order to store the result into the shared memory, the intermediate values have to be read from the register file and stored in the shared memory.

Apart from the additional instruction cost, this limitation requires a set of registers to be available. In the block matching algorithm, three texture vectors are read, each requiring 4 registers. These 12 registers are read once and not used after - the values are stored directly into the shared memory. During this texture read, a total of 12 registers are needed, yielding the algorithm's bottleneck of register usage. The high level CUDA code and the decoded binary for one such vector read are both listed:

```
# CUDA-code
shared[x][y] = tex2D(texture, x, y);
```

```
# Decoded binary (PTX assembly)
tex.load {$r0,$r1,$r2,$r3} $tex1 {$r0,$r1}
mov.shrd s[0x002c] $r0
mov.shrd s[0x002d] $r1
mov.shrd s[0x002e] $r2
mov.shrd s[0x002f] $r3
```

As can be seen, the texture load stores its result into registers $\$r0$, $\$r1$, $\$r2$ and $\$r3$, using the two-dimensional index values $\$r0$ and $\$r1$. The intermediate registers are then directly read in order to be saved onto the shared memory in successive memory locations. In order to reduce the above example to fewer instructions and reducing the register usage, the following instruction is proposed:

```
tex.load s[0x002c] $tex1 {$r0,$r1}
```

This instruction reads the two index values from the register file and uses them to read the correct data from the texture memory. It then accesses the shared memory at the correct location to write the loaded data on successive locations.

In order to implement this proposed instruction, both the hardware and the compiler need to be adjusted. Since the exact hardware architecture is unknown, a proper implementation can only be guessed at. Because instructions saving to shared memory exist and instructions reading from the texture cache exists, it might be sufficient to add this instruction type to the hardware instruction decoder, enabling the new instruction. Then, no changes have to be made to the SM or PE architecture.

6.2 Scratchpad access for the texture cache

As seen in section 2.4.2, two SMs share a 16KB texture cache. For 32-bit words, each SM has on average access to 2048 words. With multiple threadblocks executing on an SM and with hundreds of threads context switching to hide memory latency, cache contents include data from numerous threads. The texture cache can be used efficiently by the programmer in two ways:

- The programmer specifies two reads of the same variable in one thread, assuming the second read will result in a cache hit. However, even if the two reads are directly following each other in the programmers code, a large number of threads (from the same and from other threadblocks) could be using the texture cache in the meantime. Therefore, the programmer must thoroughly evaluate the scheduling of his threads on the hardware, before assuming a cache hit.
- Secondly, the programmer can read a value in one thread and assume a cache hit in another thread. The programmer has to consider other threadblocks using the cache in the meantime. Also, threads from the

same threadblock that are executing different parts of their code might overwrite these values.

Because of the limited size of the cache and the concurrent execution of multiple threads, texture cache benefits in CUDA are limited. Only with a good analysis, the texture cache can be used efficiently.

To improve performance, the texture cache can be given a second mode. Apart from the cache mode already implemented, the user could be able to enable through CUDA a second scratchpad mode. This mode disables the cache functionality and enables an on-chip memory comparable to the shared memory. However, since the texture cache is read-only from a thread perspective, the newly created scratchpad memory will be read-only as well. A potential usage could be along the lines of a controlled texture cache:

1. First, the programmer specifies which part of the off-chip memory is the most re-used.
2. Then, on execution of the program, the host copies that data onto all texture caches.
3. On execution of a thread, this data can be read from fast on-chip memory.

To enable such a second mode, no major hardware changes have to be made. Since the memory and the busses are already available, only the caching mechanism needs to be extended with a scratchpad access mechanism. In order to use the newly created possibilities, the instruction set and the CUDA API have to be adjusted.

For applications that do not benefit from the texture cache - such as the block matching algorithm - the second mode can be used, leading to a potential performance gain. For other applications, the original cache mode can still be used, resulting in no performance loss. The only drawback is the addition of little hardware, almost negligible compared to the addition of a new 16KB scratchpad memory next to the texture cache. However, as seen in section 2.5, a GPU like architecture does not benefit from a cache as much as a traditional multi-core architecture. Instead, a GPU hides its memory latencies by scheduling more threads. Guz et. al. show a negligible speed-up when using a cache in a many-thread machine such as a GPU [10].

Chapter 7

Conclusions

To conclude the research performed in this work, the related work is summarized in section 7.1. With the related work in mind, the uniqueness of the research is shown. Then, possibilities for further work are given in section 7.2. These sections together give the background - the starting knowledge - and a vision of the future, positioning the work in its context. The work itself is then summarized in section 7.3, highlighting the most important conclusions and achievements.

7.1 Related work

As mentioned before, GPUs can accelerate algorithms or part of algorithms, yielding up to orders of magnitude more performance than traditional CPU architectures. However, in order to fully utilize the hardware, programmers are left with the non-trivial task of mapping and going through optimization procedures. To reduce the needed programming effort and to increase the hardware utilization, the CUDA compilation flow can be adjusted.

One such adjustment to the CUDA compilation flow is the addition of CUDA source-to-source compilers. S. Baghsorkhi et. al. present such a source-to-source compiler, named CUDA-lite [2]. With CUDA-lite, performance can be increased by a factor 2 to 17, depending on the level of optimization already applied and the algorithm. However, this source-to-source compiler requires annotations in the source code, giving pointers to CUDA-lite for potential optimization steps. For an automated optimizer requiring no knowledge nor effort from the programmer, annotations as necessary by CUDA-lite should be omitted.

Other automatic optimization and mapping efforts are performed as part of the design of a simulator or a translator. An example of this is Ocelot [6], a translator from a GPU to a Cell architecture at a PTX level. In the work performed by G. Diamos et. al., several GPU concepts are mapped onto a Cell architecture automatically. Since the target architecture is a

Cell processor, no specific GPU optimizations are performed, although some optimizations and mapping techniques are valid for both architectures.

Another way to reduce programming effort and to increase hardware utilization is to modify the current CUDA compilers. As of now, no modifications to either compilers are known to be made. However, the work of W.J. van der Laan shows interest in this area, as he reverse engineered GPU object code [23]. With the results of Van der Laan's research, a new tool is introduced in this work, showing improvements on correctness and usability over the original tool. Many conclusions in this work are based on Van der Laan's research.

Although automated mapping and optimization for GPUs is an area not widely researched, AMD and Intel have shown interest in a hybrid CPU/GPU architecture, performing automated distribution of work. Similar to this is Intel's Larrabee architecture [22], showing a multiprocessor architecture including simple CPUs and wide vector units. Larrabee is compatible with the x86 instruction set, mapping automatically the work onto the different processing elements available.

7.2 Further work

To shorten GPGPU programming time and to achieve better hardware usage, several abstractions need to be made from the programming and hardware models. Programmers will require a fully automated memory, thread and threadblock mapping. Also, general and algorithm specific optimizations have to be done automatically. Although the road to such an abstraction is long, this work proposes a number of improvements to the existing compiler in section 5, setting a step in the right direction. This work identifies a number of weaknesses in one specific area of the compilation flow, indicating a much larger opportunity for improvement to both the compilation flow and the hardware.

With GPUs performing orders of magnitude better in several applications compared to traditional CPU architectures, it is worth considering the effort to work towards this adjusted compilation flow and hardware, creating a more efficient mapping process for general purpose algorithms onto GPUs. In the future, this could lead to a heterogeneous hybrid multiprocessor, containing both a CPU and a GPU part. An automated compiler could take sequential code, mapping tasks to either the GPU or the CPU part of the hybrid processor, resulting in efficient hardware usage without additional programmer requirements.

7.3 Summary

The last few years show a shift towards multi-threaded processor architectures, among with the rise of GPGPU programming. The shift replaces single thread machines with massively parallel machines, running thousands of threads. The G80 architecture, programmable in CUDA, is designed to hide memory access latency through context switching. For GPGPU programming, the G80 architecture contains a fast specialized on-chip shared memory.

With CUDA, mapping an algorithm - such as block matching - onto a GPU is a fairly simple task. However, after the first naive mapping, the programmer must follow a long trajectory of alterations and optimizations. The algorithm benefits from all capabilities of the hardware only after this trajectory is completed. This is shown in the mapping process of the block matching algorithm onto a GPU. Two different mappings and a number of optimizations have been explored, resulting in speed-ups orders of magnitude higher than the initial naive implementation.

From the mapping and optimization process of the block matching algorithm, it is concluded that programmers have to follow a trajectory possibly subject to partial automation by the CUDA tool-chain. Since the first compiler, nvcc, is already thoroughly evaluated, the second compiler, ptxas, is analyzed. Since the behaviour of ptxas is unknown, a decoder for GPU binaries has been developed. This decoder, known as CUDAvis, provides visualization functionality to give feedback to the programmer. Additionally, CUDAvis has been used to analyze ptxas and to present improvements.

With CUDAvis, a number of possible adjustments to ptxas are identified and presented:

Non-linear register allocation It has been shown that register allocation on a GPU must be performed stepwise. Applying non-linear register allocation yields significant improvements when other register allocation techniques are implemented in the compiler. Reducing the register count makes room for more scheduled threads, resulting in better memory latency hiding. An algorithm to implement non-linear register assignment has been presented.

Value recalculation The recalculation of register values not used at bottlenecks in the kernel yields a lower register count per thread. However, it introduces a number of additional computational instructions, depending on the value to be recalculated. In the presented algorithm, the drawbacks are compared to the potential performance gain.

Instruction re-ordering Through an example, it has been shown that instruction re-ordering can reduce register count. Automated instruction

re-ordering for register optimization is implemented in many modern compilers, but currently lacks in ptxas.

Efficient register spilling Register spilling is currently performed onto the off-chip memory. Since this reduces performance significantly, kernels that use too many registers are typically adjusted to fit onto the register file anyhow. However, when the shared memory is not entirely used, register spilling can be performed onto the shared memory. An algorithm to do so has been presented.

Redundant instruction removal Two types of redundancy have been identified by examples. First, register duplication occurs in the GPU binary. Secondly, multiple computation instructions that can be done in one instruction occur as well. An evaluation of both types of redundancy has been provided.

Also, from the mapping of the block matching algorithm and with the use of CUDAvis, two hardware improvements are identified:

Texture cache to shared memory loads Altering the instruction set enables instructions writing directly to the shared memory from the texture cache, an operation common in the block matching algorithm. Currently, this is done in two steps, using registers as intermediate storage. The presented hardware change reduces the instruction count and the register usage.

The texture cache as a scratchpad memory In a lot of CUDA kernels, the texture cache is used inefficiently. With little hardware addition, the texture cache can be transformed into a scratchpad memory, enabling content control for programmers.

Bibliography

[1] PureVideo: Digital home theater video quality for mainstream PCs with GeForce 6 and 7 GPUs. Technical report, 2005.

[2] Sara Baghsorkhi, Melvin Lathara, and Wen mei Hwu. CUDA-lite: Reducing GPU programming complexity. 5335:1–15, 2008.

[3] Preston Briggs. *Register allocation via graph coloring*. PhD thesis, Houston, TX, USA, 1992.

[4] Shuai Che, Michael Boyer, Jiayuan Meng, David Tarjan, Jeremy W. Sheaffer, and Kevin Skadron. A performance study of general-purpose applications on graphics processors using CUDA. *J. Parallel Distrib. Comput.*, 68(10):1370–1380, 2008.

[5] Gerard de Haan. *Digital Video Post Processing*. 2006.

[6] G. Diamos, A. Kerr, and M. Kesavan. Translation GPU binaries to tiered SIMD architectures with Ocelot. Technical report, 2009.

[7] M. Fatica and W.K. Jeong. Accelerating Matlab with CUDA. MIT, 2007.

[8] Wilson W. L. Fung, Ivan Sham, George Yuan, and Tor M. Aamodt. Dynamic warp formation and scheduling for efficient GPU control flow. In *MICRO '07: Proceedings of the 40th Annual IEEE/ACM International Symposium on Microarchitecture*, pages 407–420, Washington, DC, USA, 2007. IEEE Computer Society.

[9] David Geer. Vendors upgrade their physics processing to improve gaming. *Computer*, 39(8):22–24, 2006.

[10] Z. Guz, E. Bolotin, I. Keidar, A. Kolodny, A. Mendelson, and U.C. Weiser. Many-core vs. many-thread machines: Stay away from the valley. *IEEE Computer Architecture Letter*, 2008.

[11] Mark Harris. Mapping computational concepts to GPUs. In *SIGGRAPH '05: ACM SIGGRAPH 2005 Courses*, page 50, New York, NY, USA, 2005. ACM.

[12] Mark Harris. Optimizing CUDA. In *SC07: High Performance Computing With CUDA*, 2007.

[13] Sunpyo Hong and Hyesoon Kim. An analytical model for a GPU architecture with memory-level and thread-level parallelism awareness. *SIGARCH Comput. Archit. News*, 37(3):152–163, 2009.

[14] Erik Lindholm, John Nickolls, Stuart Oberman, and John Montrym. NVIDIA Tesla: A unified graphics and computing architecture. *IEEE Micro*, 28(2):39–55, 2008.

[15] David Luebke, Mark Harris, Jens Krüger, Tim Purcell, Naga Govindaraju, Ian Buck, Cliff Woolley, and Aaron Lefohn. GPGPU: general purpose computation on graphics hardware. In *SIGGRAPH '04: ACM SIGGRAPH 2004 Course Notes*, page 33, New York, NY, USA, 2004. ACM.

[16] Aaftab Munshi. OpenCL: Parallel computing on the GPU and CPU. 2008.

[17] NVIDIA. CUDA zone. `http://www.nvidia.com/cuda`.

[18] NVIDIA. Speak visual with Adobe Photoshop. `http://www.nvidia.com/object/adobe_photoshop.html`.

[19] NVIDIA. *NVIDIA Compute PTX: Parallel Thread Excecution*, 1.1 edition, 2007.

[20] NVIDIA. *The CUDA Compiler Driver NVCC*, 2.1 edition, 2009.

[21] NVIDIA. *NVIDIA CUDA Programming Guide*, 2.1 edition, 2009.

[22] Larry Seiler, Doug Carmean, Eric Sprangle, Tom Forsyth, Michael Abrash, Pradeep Dubey, Stephen Junkins, Adam Lake, Jeremy Sugerman, Robert Cavin, Roger Espasa, Ed Grochowski, Toni Juan, and Pat Hanrahan. Larrabee: a many-core x86 architecture for visual computing. In *SIGGRAPH '08: ACM SIGGRAPH 2008 papers*, pages 1–15, New York, NY, USA, 2008. ACM.

[23] W.J. van der Laan. Decuda and cudasm: the cubin utilities package, 2007. `http://www.cs.rug.nl/ wladimir/decuda/`.