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

Cedric Nugteren
Eindhoven University of Technology, The Netherlands
c.nugteren@tue.nl

Bart Mesman
Eindhoven University of Technology, The Netherlands
b.mesman@tue.nl

Henk Corporaal
Eindhoven University of Technology, The Netherlands
h.corporaal@tue.nl

ABSTRACT

With GPU architectures becoming increasingly important due to their large number of parallel processors, NVIDIA's CUDA environment is becoming widely used to support general purpose applications. To efficiently use the parallel processing power, programmers need to efficiently parallelize and map their algorithms. The difficulty of this task leads to the idea to investigate CUDA's compiler.

Part of the compiler in the CUDA tool-chain is entirely undocumented, as is its output. To draw conclusions on the behaviour of this compiler, the resulting object code is reverse engineered. A visualization tool is introduced, analyzing the previously unknown compiler behaviour and proving helpful to improve the mapping process for the programmer. These improvements focus on the area of register allocation and instruction reordering. This paper describes an extension to the CUDA tool-chain, providing programmers with a visualization of register life ranges. Also, the paper presents guidelines describing how to apply optimizations in order to obtain a lower register pressure.

In a case-study example, performance increases by 33% compared to already optimized CUDA code. This is achieved by optimizing the code with the help of the introduced visualization tool. Also, in 11 other case-study examples, register pressure is reduced by an average of 18%. The presented guidelines could be added to the compiler to enable a similar register pressure reduction to be achieved automatically at compile-time for new and existing CUDA programs.

Categories and Subject Descriptors

C.1.4 [**Processor Architectures**]: Parallel Architectures; D.3.4 [**Programming Languages**]: Processors—*Compilers, Optimization*

Keywords

Compiler, CUDA, GPU, Mapping, SIMD

1. INTRODUCTION

Graphical processing units are a hot topic in computer architecture design. Today's graphical processing units (GPUs) provide a higher raw computation potential than traditional general purpose CPUs, therefore, using GPUs for general purpose tasks becomes increasingly common [4]. In the past, GPUs have been used mainly for the acceleration of 3D-games. Now, however, GPUs come with increasing programmability and are used to provide hardware acceleration for high definition video decoding [1] and for physics calculations within 3D-games [8]. Adding to the popularity of GPUs for general purpose tasks is the in 2008 introduced hardware acceleration support for Adobe Photoshop [15] 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 [10]. In other words, the shift replaces one complex high frequency processor by many 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) [18].

Along with the trend towards parallel processors comes a shift within the programming model. Sequential programming languages are replaced by parallel programming languages that expose SIMD-style parallelism to the programmer. For GPUs, NVIDIA introduced CUDA[2] as a parallel programming environment for general purpose applications. With CUDA, the programmer can exploit the GPU's parallelism and accelerate general purpose applications. Although similar GPU programming environments and languages exist (such as AMD's Stream technology [2] and the Khronos group's open standard OpenCL [11]), the focus of this paper lies on CUDA.

1.1 Problem statement

Although CUDA is specifically designed for general purpose GPU programming, the mapping process of an algorithm onto a GPU remains non-trivial. Even though gen-

[1]Single Instruction, Multiple Data
[2]CUDA is an acronym for Compute Unified Device Architecture

eral purpose GPU programming has developed fast in the last three years, there is only a tiny fraction of programmers able to reach a near-optimal hardware usage for their algorithms. In order to achieve this, the programmer is required to have thorough knowledge of the algorithm, the programming language and the target hardware architecture. Achieving optimal hardware usage is non-trivial with distributed memories, caches and register files all mappable by the programmer. Additionally, the programmer is exposed to parallel computing problems such as data-dependencies, race-conditions, synchronization barriers and atomic operations. For example, programmers will have to ensure a low register pressure. A low register pressure can lead to a larger number of active threads, which can be paramount for the application's performance, as explained in this paper. An example is illustrated in table 1, in which a different instruction order yields a reduction of the register usage.

Table 1: Reducing register usage

Original	r1 r2	Optimized	r1 r2
load $r1		load $r1	
load $r2		use $r1	
use $r1		load $r2	
use $r2		use $r2	

Although it is clear that GPUs provide a strong platform for algorithm mapping, the development tools have not yet reached the level of maturity that programmers are accustomed to when programming CPUs. To maintain high quality solutions, register pressure has to be kept as low as possible. In order to do so, clear guidelines on register reduction techniques need to be presented, along with a visualization of the compiled binary's register life ranges. Only then will programmers have insight in the behaviour of their compiler and an opportunity to use their hardware efficiently in terms of register usage.

1.2 Related work

An existing adjustment to the CUDA compilation flow is the addition of a source-to-source compiler to increase hardware utilization. In the work of S. Baghsorkhi et. al. such a compiler, named CUDA-lite [3], is presented. With CUDA-lite, performance can be increased by a factor up 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 using the GPU's virtual instruction set. 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.

1.3 Paper outline

The rest of this paper is organized as follows. First of all, section 2 provides background information on the CUDA environment and the GPU hardware. It first introduces the basic concepts, followed by an overview of the CUDA compilation flow. After the hardware is introduced, the coupling between the CUDA environment and the GPU hardware is evaluated, in order to obtain insight in the mapping process of an algorithm onto a GPU.

Through the case-study of a block matching algorithm [5] in section 3, the CUDA programming experience is evaluated. It quickly becomes clear that the current tool-chain needs to be extended with a decoding and visualization tool, presented in section 4. With this tool, previously hidden details of the compiled program are revealed, enabling for new optimizations and leading to an increased efficiency in hardware usage. This is presented in section 5, along with examples showing performance improvements of 33% over optimized code and more than 400% over a naive implementation.

Finally, in section 6, conclusions on the research are presented.

2. BACKGROUND

Graphical processing units (GPUs) are becoming more and more powerful, but also more suitable for general-purpose applications. The graphical pipeline found in the GPU has evolved to a homogeneous set of many programmable processing elements. In contrary to a traditional processor, the GPU has very simple processing elements, lacking complex control. While a CPU dedicates most of its chip area to control and cache, the GPU's chip area is mostly ALUs. This huge amount of computation power can be efficiently used for highly data-parallel applications, such as picture enhancement, image analysis and other media applications. To program GPUs for general purpose applications, the CUDA environment is used. This environment consists of among others a thread model and a compilation flow consisting of several tools.

2.1 The CUDA thread model

The CUDA thread model is introduced to support parallelism over different processing elements as well as to hide memory latency by switching to a different thread. An overview of the thread model is shown in figure 1 and described below:

- 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) [10]. In CUDA, the kernel is executed on the GPU (named *device* by NVIDIA), on which only one kernel can be active at any given time[3].
- A **thread** is an instance of a kernel. In CUDA, the number of threads one kernel instantiates can be tens to hundreds of thousands.
- Each thread belongs to a **threadblock**. Within one threadblock, threads can synchronize with each other

[3]Future hardware, such as the Fermi architecture [14], does support multiple concurrent kernels

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 [10].

- 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 synchronization is possible, except within individual threadblocks [10].

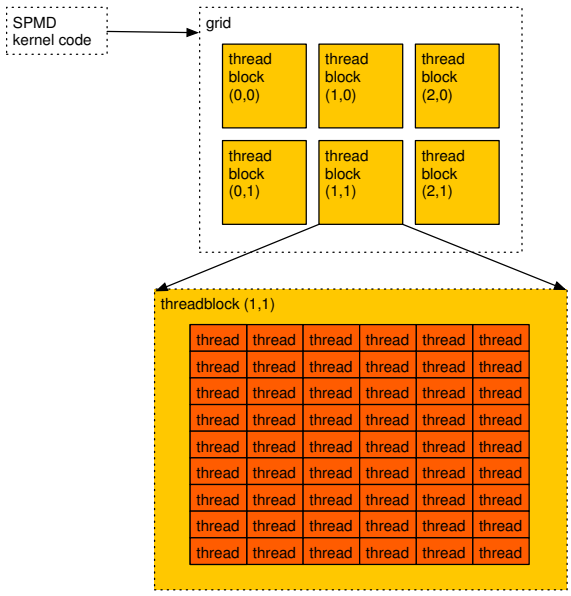

Figure 1: The thread model: an example layout

Threads can be characterized as programs that fetch their instructions from the same binary, but can take distinct control paths depending on thread-identifiers which are stored in special registers [10]. These identifiers may hold different values for each thread and correspond to both the coordinates (1D, 2D or 3D) of a thread in their threadblock and the coordinates of their threadblock in the grid.

2.2 G80's hardware architecture

In this work, NVIDIA's G80 architecture is taken as an example. This GPU has a clustered SIMD layout, dividing its processing elements in clusters of 8 over different multiprocessors[4]. With a total of at most 16 clusters and a clock speed of more than 1GHz, its computational performance can be in the order of hundreds of giga floating-point operations per second (GFLOPS). The G80 GPU uses a large off-chip memory and a small high-speed memory shared between the processing elements within a cluster. Because no read/write level 1 or level 2 cache is implemented in the GPU's design, accesses to the off-chip memory will have a latency of several hundred clock cycles. To compensate for the omitted caches, each multiprocessor has a large register file to store numerous threads. With this large register file, the GPU can switch to other threads instantly, performing useful computations while waiting for data to be read from the off-chip memory. The G80 architecture does include a *texture cache*, only supporting memory reads from a GPU perspective. Future NVIDIA GPUs (the Fermi architecture [14]) do add a traditional level 1 and 2 cache hierarchy, accelerating memory access operations even further.

[4]Strictly spoken, what NVIDIA refers to as a Streaming Multiprocessor (SM), is not a multiprocessor as it contains a single instruction cache. Nevertheless, the term multiprocessor is used to concur with NVIDIA and related work

2.3 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 [18].

On execution of a kernel, a warp is started onto a 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[5] of each thread within the warp [9]. Then, the first instruction of the second warp is executed, taking another 4 clock cycles. When all warps finish their first instruction, the first warp starts executing the second instruction. This process continues until an instruction is encountered that needs to wait for data from a previously issued load instruction. 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 stalled.

Besides scheduling in groups of warps, the thread model also introduces constraints on the scheduling possibilities. As mentioned before, threadblocks can synchronize and use a shared memory. Therefore, one entire threadblock must be scheduled onto one multiprocessor. The number of threadblocks that fit onto one multiprocessor depends among others on the register usage per thread and the shared memory needed per threadblock.

From all the above mentioned concepts and models, it is important to realize the impact of the 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. Additionally, for the pipeline latency (24 cycles in G80 architecture [10]) to be hidden, the number of warps should be at least equal to the pipeline latency multiplied by the number of processing elements per multiprocessor (8 in G80 architecture [10]). But lastly - the most important - the total number of threads needs to be chosen correctly to hide memory latency. 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

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

completely or partially hidden - if enough warps are available to switch to. The memory operation intensity together with the computation intensity can determine the number of warps required to ensure completely hidden memory request times, which can be analytically computed [9]. However, increasing the number of threads - and thus warps - will either show the same performance or help to hide this latency, independent of the structure of the program.

One way to increase the number of threads - and thus the potential performance - is to optimize and reorganize the kernel. By reducing the number of registers used, the register pressure per thread decreases, allowing for more threads to simultaneously fit onto the register file of one multiprocessor.

2.4 The CUDA compilation flow

To get more insight on the characteristics of threads, the CUDA compilation flow is introduced, consisting of two proprietary compilers, bundled together in *nvcc* [17]. 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 a modified *Open64* [12], compiling high level code into an intermediate instruction set - the PTX[6] format. The resulting PTX code is ran through the second compiler, *ptxas*, which produces a GPU binary [17]. Since the host part is executed on the CPU, a standard C compiler is used.

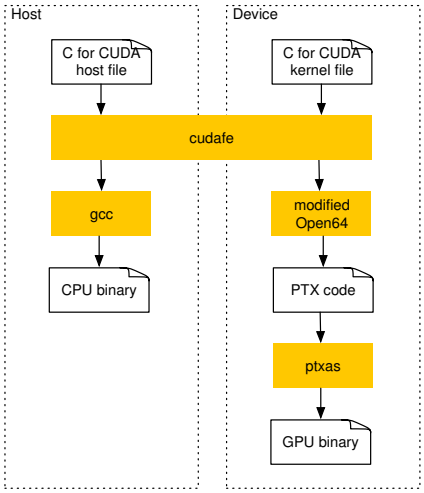

Figure 2: A simplified compilation flow

Please note that the compilation flow described and seen in figure 2 is an abstraction of the actual (more complicated) compilation flow. For the research purposes of this work, this abstraction provides sufficient information. Research is focused on the trajectory from CUDA code through ptxas and the modified Open64, resulting in a GPU binary.

The first compiler, based on Open64 (open-source), 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 is produced is chosen to be hardware independent, still being able to run on any CUDA enabled GPU [17]. Because of the introduced hardware independence, several compiler tasks are omitted. This includes register assignment and instruction re-ordering. Thus, PTX code can be seen as a virtual instruction set, targeting current and future hardware architectures. PTX is completely specified by NVIDIA [16].

The second compiler performs hardware specific compilation. As of 2010, four different hardware specifications exist [16]. The compiler can compile a GPU binary for any of these target specifications. In the case that a different hardware architecture is used, the compiler 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 [10]. Ptxas needs to perform register allocation, because PTX code assumes an infinite number of registers available, while different hardware specifications vary the number of available registers. In order to perform efficient register allocation, ptxas can also perform instruction re-ordering.

3. MOTIVATION

To illustrate the current state of the ptxas compiler, a case-study has been performed in which the block matching algorithm[7] is mapped onto the GPU. The algorithm consists of the calculation of a sum of absolute difference (SAD) between two blocks of pixels. This is repeated numerous times within a so-called search window, after which the smallest SAD value is selected. This is then repeated multiple times for different blocks of pixels, until a complete frame is processed. More details can be found in [5] and [13].

A mapping of the block matching algorithm results in the creation of different kernels. One kernel shows a pressure of 17 register entries at the kernel's bottleneck. During this bottleneck, several register values are stored for later use. To reduce the register usage, these values can be recalculated. In this case, introducing 4 additional instructions reduces the kernel's register usage by one, which can lead to a significant performance increase due to an increased active thread count. In the case-study example, the size of a threadblock is set by the programmer to 256, resulting in the data shown in table 2.

Table 2: Reducing the kernel's register pressure

	Number of instructions	Registers per thread	Registers per block
Original	20	17	4352
Optimized	24	16	4096

In G80's hardware architecture, the register file has a size of 8192 entries. The original kernel's register usage implies the mapping of one threadblock on a multiprocessor (since $2 \cdot 4352 > 8192$), while the kernel with a reduced register usage enables two threadblocks per multiprocessor. As stated,

[6]PTX is an acronym for Parallel Thread eXecution

[7]The block matching algorithm is part of motion estimation, which is used among others in MPEG decoding

the addition of more threadblocks - and thus threads - enables for better memory latency hiding. If the original kernel is memory latency bound, a best-case speed-up[8] of 1.7 is achieved ($\frac{20}{24} \cdot 2$), due to the scheduling of 2 threadblocks instead of 1.

The reader should note that although a small part of the kernel is modified, the performance of the complete kernel is altered due to the nature of the thread-based architecture. Even so, performance increase can only be expected in the modified kernel - not in any other kernels or in the host code. As always, for the complete algorithm's performance, Amdahls law has to be kept in mind.

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. To automate the mapping and optimization process, the CUDA compilation flow can be improved by either adding tools or modifying the compilers.

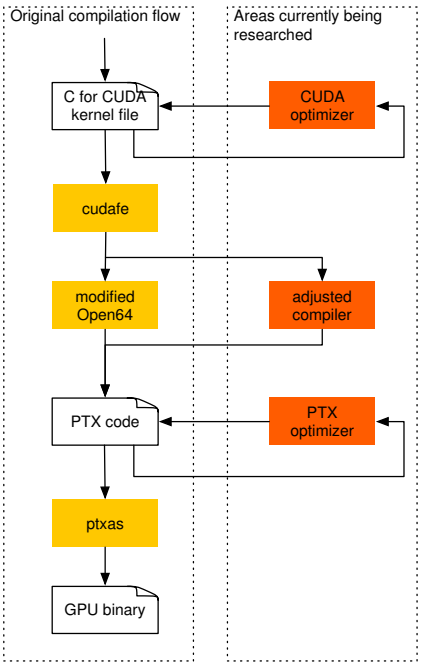

Figure 3: Automating the mapping and optimization process

Currently, a small number of CUDA source-to-source optimizers exist [3]. These optimizers are positioned at the beginning of the CUDA compilation flow, as can be seen in figure 3. Shown in the same figure are adjustments to the Open64 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 little 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 an unknown area. It is however an important area to investigate, since register pressure can determine the thread count and thus influence the level of parallelism. Register allocation is performed in ptxas and is thus not present at PTX level. CUDA provides the programmer with a setting for a maximum register count, but this involves spilling register data onto the off-chip memory.

[8]This is only true under several assumptions, the simplified formula estimates the actual value

4. VISUALIZING GPU BINARIES

To perform research in the area of ptxas, W.J. van der Laan created a decoder for GPU binaries in 2007, named *decuda* [19], decoding binaries to an assembly language close to PTX. However, his decoder, designed through reverse engineering, has several drawbacks:

- Since the designer of decuda started from an empty set of known instructions, the source code of the tool is unorganized. For every known rule, an additional if-then-else statement is introduced, expanding the source code while gathering more information and ending up with complex source-code.
- Since decuda is solely based on reverse engineering, it cannot be proven 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.
- With the use of decuda, information can only be extracted with understanding of the PTX virtual instruction set. Then, in order to draw any conclusions, the obtained decoded instructions need to be analyzed and compared with the PTX or CUDA source code.

In order to overcome the mentioned drawbacks of decuda, a new decoder is introduced, named *CUDAvis*[9]. CUDAvis is used to both decode and visualize GPU binaries.

4.1 Design of CUDAvis

Using the scripting language Ruby[10] and the graphical library Tcl/Tk[11], a cross platform tool is created. The design of the tool can be divided into two stages - the decoding stage and the visualization stage. Both stages are depicted in figure 4. Among others, branches and labels are visualized, followed by a register life range checking algorithm. This results in a full view of every register, indicating if it is occupied, written, read or unused. Additionally, bottlenecks in terms of register pressure are identified, pointing the programmer to the critical part of the code.

CUDAvis is positioned within the CUDA compilation flow as seen in figure 5. The usage of CUDAvis is twofold:

- Firstly, CUDAvis is used by programmers to evaluate compiled code. Programmers can identify structures, register life ranges and bottlenecks in the code, which

[9]CUDAvis is short for *CUDA visualization tool*
[10]See *www.ruby-lang.org*
[11]See *www.tcl.tk*

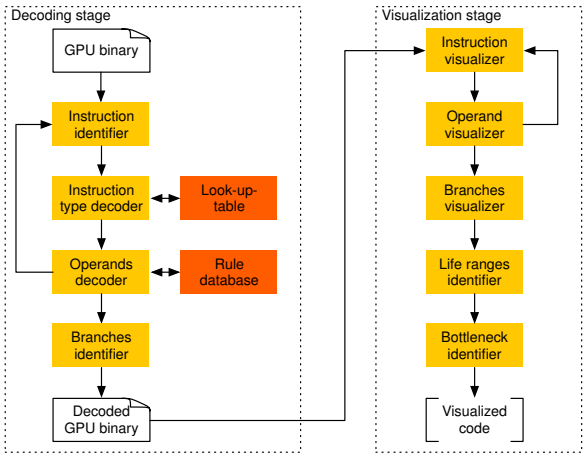

Figure 4: The design of CUDAvis

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. A number of improvements to the ptxas compiler are proposed, resulting in automated optimizations within the CUDA environment.

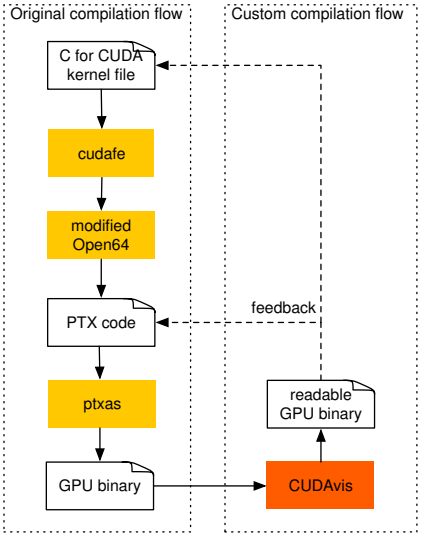

Figure 5: CUDA compilation flow extended with CUDAvis

Since a GPU includes an instruction decoder, hardware exists that has the same functionality as CUDAvis' decoding stage. 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 data classes. The result is readable, compact and intuitive code, leaving room for adjustments and expansions.

4.2 Usage of CUDAvis

In figure 5, a feedback-loop is drawn, which involves adjusting program code manually. This is done according to the following steps:

1. The original high-level code is compiled with an unmodified compiler.

2. The resulting binary (.cubin) is taken as an input by CUDAvis. The execution of CUDAvis decodes the binary and displays a visualization; an example is shown in figure 6 and elaborated in section 4.3.

3. The programmer identifies high register pressure sections in the code and finds better register allocation possibilities by examining the visualized code.

4. The original high-level code is adjusted to reduce register pressure. Depending on the solutions needed, several types of adjustments can be made to direct the compiler to reduce register usage. This includes the declaration of volatile or constant variables, the explicit declaration of expressions which are part of other statements and the rearrangement of lines of code. These are examples of adjustments that typically do not influence register allocation, but do for CUDA's compilation flow.

4.3 CUDAvis' visualization

In figure 6, a snapshot is shown of the visualization obtained using CUDAvis. In this case, lines 38 up to 68 of the Black-Scholes algorithm are shown, which is taken as a case-study in section 5.2.

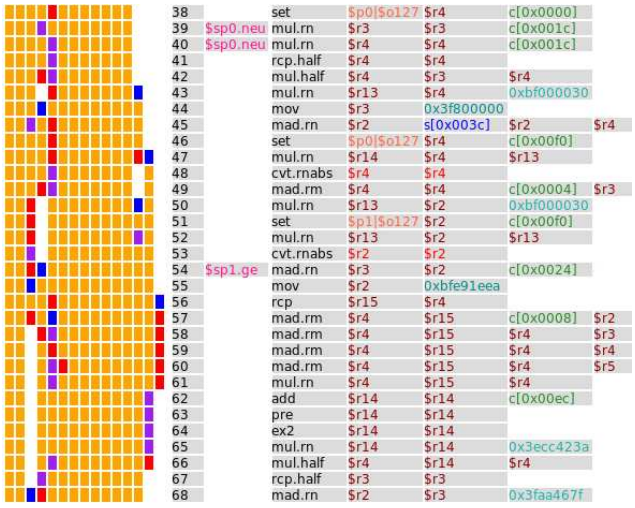

Figure 6: Snapshot of the decoded Black-Scholes binary

The left hand side of figure 6 shows the register life ranges, ranging from register 0 up to 15 from left to right. For each instruction and each register, the status is indicated. Red indicates a read, blue indicates a write and purple indicates both a read and a write. If the register is occupied it is given an orange colour, else it is left blank.

Then, on the right hand side of figure 6, the decoded instructions are shown, complete with register names, constant memory accesses, constants and conditional statements. In CUDAvis, each type of instruction operand is given a distinct colour.

5. RESULTS

To illustrate the use of CUDAvis, two case-study examples are taken. First, the block matching algorithm is considered. Secondly, the Black-Scholes algorithm is taken as an example, which is part of the CUDA SDK. Other examples from the SDK are included as well. Concluding this section, a number of guidelines is presented to improve CUDA's compilation flow, in particular the compiler ptxas.

5.1 Block matching as a case-study example

Part of the block matching algorithm is taken as an example. A total of 7 instructions are considered, reading from and writing to the register file as seen in table 3. In this table, the register life ranges are shown. The 7 instructions are executed sequentially and show a peak register usage of 5.

Table 3: Original register life ranges

	r1	r2	r3	r4	r5	r6	Registers	Instruction
1							2	load to ($r1,$r2)
2							4	load to ($r3,$r4)
3							**5**	$r5 = $r2 * 64
4							**5**	$r6 = $r4 * 64
5							**5**	$r2 = $r1 + $r5
6							4	$r4 = $r3 + $r6
7							3	$r1 = $r2 + $r4

With some basic instruction reordering steps, the structure can change as seen in table 4. It is now a trivial task to alter the instructions to reduce the register pressure to 4 which enables the scheduling of more threads, resulting in a better hardware utilization. With CUDAvis' register life range checker, programmers can identify such opportunities.

Table 4: Modified register life ranges

	r1	r2	r3	r4	r5	r6	Registers	Instruction
1							2	load to ($r1,$r2)
2							3	$r5 = $r2 * 64
3							3	$r2 = $r1 + $r5
4							3	load to ($r3,$r4)
5							**4**	$r6 = $r4 * 64
6							**4**	$r4 = $r3 + $r6
7							3	$r1 = $r2 + $r4

Reducing the number of registers per thread can greatly improve performance, as explained in section 2.3. For the block matching algorithm, a naive and a highly optimized GPU implementation where created [13]. Even then, with the use of CUDAvis, more optimizations can be applied, reducing the register pressure and resulting in a speed-up of 33% over the optimized code, as shown in table 5.

Table 5: Case-study: the block matching algorithm

Architecture	Optimization level	Speed-up
CPU	Naive	1x
GPU	Naive	17x
GPU	Optimized	56x
GPU	Optimized using CUDAvis	75x

Optimizations applied before CUDAvis range from memory coalescing to shared memory usage, while optimizations performed after the use of CUDAvis only result in a reduced register usage. This includes:

- The **explicit declaration** of variables, to point the compiler to parts of an expression that can be calculated prior to the rest of the expression. This technique can be applied in cases where an expression must be calculated every iteration of a loop, while a part of it can be pre-calculated outside the loop. Omitting or using this technique can alter the register pressure in specific parts of the code.
- Specifying a **volatile** variable forces the compiler to recalculate the variable if it is used later on in the code. This can alter the register pressure, but can introduce additional instructions.
- The use of a **const** variable indicates a non-changing variable, to be kept in the register file as long as needed. Using or omitting constant variables can influence register pressure.
- The **rearrangement** of the instruction order (while preserving functionality) can influence the compiler's behaviour, possibly resulting in a different register mapping.

5.2 Black-Scholes as a case-study example

A second case-study example is taken from the CUDA SDK. The Black-Scholes algorithm represents a mathematical description of financial markets. As the algorithm is provided, it shows a register pressure of 16. Part of the kernel from the CUDA SDK is given in listing 1.

Now, using CUDAvis, the kernel can be visualized. When evaluating the lines in which the register bottleneck can be found (lines 56 and 57 in figure 6), it appears that a number of registers are kept alive (orange), but not used at all during the actual computation. One of the values occupying such a register can be traced back to high-level code as the variable *thread_N*.

To reduce register pressure for the Black-Scholes algorithm, the type of the variable *thread_N* can be changed from *int* to *volatile int*. This will force the hardware to recalculate *thread_N* every iteration, resulting in a slightly increased execution time. Since *thread_N* can be calculated in one

Listing 1: Simplified Black-Scholes algorithm

```
void BlackScholesGPU(float* data, int optN)
{
 int opt;
 int tid;
 int thread_N;

 tid = blockDim.x*blockIdx.x+threadIdx.x;
 thread_N = blockDim.x*gridDim.x;

 for(opt=tid; opt<optN; opt+=thread_N)
 {
   BlackScholesBodyGPU(data);
 }
}

void BlackScholesBodyGPU(float *data)
{
 // Calculate something
}
```

instruction using only input values from the register, execution time will increase minimally. Register pressure will now reduce from 16 to 15. A reduction of register pressure by one does not lead to any performance increase. This can be explained from the fact that the Black-Scholes algorithm has a block-size of 480 threads. With 16 registers, this results in a register pressure of 7680 registers per threadblock. The hardware has 8192 registers available, resulting in an occupation of one threadblock per multiprocessor ($\lfloor \frac{8192}{16 \cdot 480} \rfloor = 1$). Changing the register pressure to 15 will not result in a higher occupation ($\lfloor \frac{8192}{15 \cdot 480} \rfloor = 1$), thus not leading to an increase in performance.

However, when the size of a threadblock is changed to 270, a register pressure of 15 will result in a higher occupation compared to a register pressure of 16. In this case, the resulting performance increase is limited (2%), due to other bottlenecks than just the thread-count. The results of register pressure reduction for the Black-Scholes algorithm (both with 480 and 270 threads per threadblock) can be found in table 6.

5.3 Register pressure reduction in examples

Apart from the two case-study algorithms, CUDAvis was used to reduce register pressure on a set of algorithms taken from the CUDA SDK. Without any algorithm knowledge, register pressure was reduced by 1 to 4 registers, as shown in table 6.

Although table 5 shows a performance increase of 33% due to the use of CUDAvis for the block-matching algorithm, it may not be the case for other algorithms[12]. From table 6 it is clear that programmers are able to reduce register pressure by directing the compiler in the right direction. Even though performance may not increase, reducing the register pressure might enable a significant (up to a factor of 2) relaxation of memory latency constraints. This implies that in that case a slower and cheaper memory - in terms of cost and power - will not have a performance impact.

[12]It should be noted however, that the kernels given in the CUDA SDK are already highly optimized for performance

Table 6: Register reduction for CUDA SDK examples

Algorithm	Register pressure (original)	Register pressure (modified)	Performance gain
Black-Scholes_480	16	15	0%
Black-Scholes_270	16	15	+2%
BoxFilter_rgba	16	14	0%
ConvFFT2D_mod	11	10	-36%
Histogram_64	14	12	-225%
Denoising_NLM	24	21	-6%
MarchingCubes	31	27	0%
MersenneTwister	16	12	-10%
QuasiRandom_CND	10	7	-3%
ScalarProd	15	13	+3%
ScanNaive	8	7	-5%
ThreadFenceRed	11	10	0%

As seen in table 6, applying the register pressure reduction techniques to the Histogram_64 algorithm shows a significant performance loss. When comparing the original with the resulting binary, the cause is found: the compiler completely restructered the code. For the other algorithm showing a significant performance loss, ConvFFT2D_mod, the reason can be found in the relatively small kernel. Originally, the inner loop contained 13 instructions, but now, after register pressure reduction, it contains 16 instructions. The signficiant difference is a cause for the performance loss.

5.4 Guidelines to reduce register pressure

As stated, CUDAvis helps to evaluate the behaviour of ptxas. Because the development of CUDA's tool-chain lacked manpower [12], an opportunity for possible improvements to ptxas exists [13]. From section 2.3, it is clear that a higher number of schedulable threads can lead to an increase of performance. To increase the number of threads schedulable on one multiprocessor, a reduction of the register usage per thread could be required. The relation between the register pressure and the active thread count is elaborated in this section, along with three techniques to increase performance.

Traditional architectures have fixed register boundaries. The GPU however, has multiple virtual register boundaries. This is due to the fact that threads within threadblocks are scheduled as a whole; a complete threadblock can either fit or not fit onto the register file. Depending on the size of a threadblock, multiple register boundaries are defined. The number of threadblocks per multiprocessor can be calculated[13] as seen in equation 1. The CUDA compiler assumes a fixed number of registers available, whereas it should consider these virtual register boundaries imposed by the thread model.

[13]The number of threadblocks per multiprocessor could be limited by other factors, such as shared memory usage

$$blocks = \lfloor \frac{total\_registers}{registers\_per\_thread \cdot threads\_per\_block} \rfloor \quad (1)$$

From equation 1, it can be seen that if the denominator approaches half the number of total registers, the register file occupation will approach 50%. Reducing the register or thread count slightly can result in a full or almost full occupation of the register file, increasing the number of active threads. Using equation 1, programmers can tune their algorithms in the following ways:

- **Rematerialization** can lead to a lower register pressure. This is the case if variables are kept alive during the register bottleneck in the code. These variables can be found with CUDAvis and altered by adding or removing *volatile* or *const* in high-level code. This introduces a number of additional computational instructions, depending on the complexity of the expression to be recalculated, and should be taken into account in the decision whether or not to perform rematerialization.
- **Instruction reordering** has the potential to reduce register count. This can be done both in high-level code or by the explicit declaration of variables, steering the compiler to a different register assignment.
- The **thread count** can be adjusted to achieve a higher active threadblock count. This can be done using equation 1.

From equation 1, it can be seen that if the denominator is greater than the nominator, the resulting number of active threadblocks will be zero. In order to still execute the code, register spilling is performed into the off-chip memory. Since this reduces performance significantly, kernels that use too many registers are typically adjusted by the programmer 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, resulting in an access time reduction equal to the memory latency, typically a factor 400 to 600 [10].

Currently, the register pressure reduction techniques are to be performed by hand. It would be even better if NVIDIA would incorporate equation 1 and the guidelines in their compiler ptxas. This would require algorithm descriptions to automate decisions. These algorithm descriptions are presented in pseudo-code and more detail in [13].

6. CONCLUSION

Since GPUs provide orders of magnitude more raw computation power than traditional processors, they are a popular target for computationally intense applications. However, the development tools leave the programmer with the non-trivial task to exploit this computational power. One of these tasks is to reduce the register usage, potentially leading to a significant performance increase. Previously, details on the register usage where hidden from the programmer. Now, a new tool, named CUDAvis, is introduced to decode and visualize GPU binaries. With the help of the tool and the presented guidelines, register pressure can be reduced, leading to an increased active thread count.

To illustrate the use of CUDAvis and the presented guidelines, an example algorithm's speed-up is increased by 33% compared to already optimized GPU code. Also, for 11 CUDA SDK examples, register usage is reduced by an average of 18%. Additionally, this work presents the guidelines as potential improvements to the compiler, enabling register pressure reduction to be performed automatically. The guidelines include a virtual register boundary aware allocation, improved instruction reordering and rematerialization of register values. Both instruction reordering and rematerialization have the potential to reduce register usage. Together with a virtual register boundary aware allocation, this can lead to more active threads. In the case of a memory access latency bound kernel, twice as much active threads can reduce execution time by half.

7. REFERENCES

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

[2] AMD. ATI stream technology. `http://www.amd.com/stream`.

[3] S. Baghsorkhi, M. Lathara, and W. mei Hwu. CUDA-lite: Reducing GPU programming complexity. *Languages and Compilers for Parallel Computing*, 5335:1–15, 2008.

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

[5] G. 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. Jeong. Accelerating Matlab with CUDA. MIT, 2007.

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

[9] S. Hong and H. 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.

[10] E. Lindholm, J. Nickolls, S. Oberman, and J. Montrym. NVIDIA Tesla: A unified graphics and computing architecture. *IEEE Micro*, 28(2):39–55, 2008.

[11] A. Munshi. OpenCL: Parallel computing on the GPU and CPU. 2008.

[12] M. Murphy. NVIDIA's experience with Open64.

[13] C. Nugteren. Improving CUDA's compiler through the visualization of decoded GPU binaries. Master's thesis, 2009. `http://www.es.ele.tue.nl/~gpuattue`.

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

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

[16] NVIDIA. *NVIDIA Compute PTX: Parallel Thread*

Excecution, 1.1 edition, 2007.

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

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

[19] W. van der Laan. Decuda and cudasm: the cubin utilities package, 2007. `http://wiki.github.com/laanwj/decuda`.