Vrije Universiteit Brussel

Faculteit Wetenschappen
Departement Informatica
en Toegepaste Informatica

Anti-parallel Patterns in Fine-grain Data-parallel Programs

Proefschrift ingediend met het oog op het behalen
van de graad van Master in de Toegepaste Informatica

Jan Cornelis

Promotor: Prof. Dr. Jan Lemeire
Begeleider: Prof. Dr. Jan Lemeire

2009-2010

Vrije Universiteit Brussel

Faculty of Science
Department of Computer Science
and Applied Computer Science

Anti-parallel Patterns in Fine-grain Data-parallel Programs

Graduation thesis submitted in partial fulfillment of the
requirements for the degree of Master in Applied Informatics

Jan Cornelis

Promotor: Prof. Dr. Jan Lemeire
Advisor: Prof. Dr. Jan Lemeire

2009-2010

Abstract

Parallel systems and parallel programming are becoming increasingly more important. The developer in want of raw speed can no longer expect sequential processors to become faster and needs to turn to parallel platforms and parallel programs to satisfy his need for speed. But writing a parallel program is difficult and writing one with a decent performance even more so.

This thesis introduces the umbrella concept of "anti-parallel patterns" - parts of parallel programs that cause its performance to be less than expected. These patterns aim at helping developers understand, estimate and improve the performance of their parallel programs. To achieve these goals we model the effect of a given pattern on the performance and we supply solutions that can be applied in the face of an anti-parallel pattern. To help us model the behaviour of a pattern, we will define benchmark programs; programs that contain only one anti-parallel pattern. We will also discuss and test remedies that can be applied to decrease the performance loss the pattern causes. An additional advantage of the benchmark programs is that they can be used to compare the effect of a pattern on different parallel platforms.

This work defines four anti-parallel patterns that are commonly found in fine-grain data-parallel programs such as those running on NVIDIA GPUs. For each anti-parallel pattern we give a definition and a thorough discussion of its behaviour on these GPUs. We present a number of benchmark programs - written using NVIDIA's CUDA technology - in order to model the behaviour of the pattern. We also present remedies and examples of how to apply them. Finally, we demonstrate the usefulness of anti-parallel patterns by considering a prefix sum implementation in CUDA. We show how we can use the patterns to understand, estimate and improve the program's performance.

Samenvatting

Parallelle systemen en parallel programmeren worden steeds belangrijker. De ontwikkelaar op zoek naar grote snelheden kan niet langer verwachten dat sequentiele processoren steeds sneller worden en is verplicht parallelle programmas te schrijven en deze uit te voeren op parallelle platformen. Het schrijven van parallelle programmas is echter niet gemakkelijk; het schrijven van parallelle programmas met een goede performantie is zelfs nog moeilijker.

Deze thesis introduceert “anti-parallelle patronen” - delen van parallelle programmas die er voor zorgen dat hun performantie minder is dan men zou kunnen verwachten. Deze patronen streven ernaar ontwikkelaars te helpen bij het begrijpen, schatten en verbeteren van de performantie van hun parallelle programmas. Teneinde deze doelen te bereiken, modelleren we het effect van een patroon op de performantie en reiken we oplossingen aan die kunnen toegepast worden tegenover een anti-parallel patroon. Om ons te helpen bij het modelleren van het gedrag van een patroon, definiëren we benchmark programmas; programmas die één enkel anti-parallel patroon bevatten. We bespreken en testen ook oplossingen die kunnen toegepast worden om het performantieverlies ten gevolge van een patroon te beperken. Een bijkomend voordeel van deze benchmark programmas is de mogelijkheid om het effect van een anti-parallel patroon op verschillende parallelle platformen te vergelijken.

Dit werk definieert vier anti-parallelle patronen die vaak teruggevonden worden in fijnkorrelige data-parallelle programmas op NVIDIA GPUs. Voor ieder anti-parallel patroon geven we een definitie en een grondige bespreking van zijn gedrag op deze GPUs. We stellen een aantal benchmark programmas voor - geschreven met NVIDIA’s CUDA technologie - teneinde het gedrag van het patroon te modelleren. We stellen oplossingen voor, alsook voorbeelden van hoe ze kunnen toegepast worden. Tenslotte tonen we het nut van anti-parallelle patronen aan de hand van een CUDA implementatie van een prefix som. We tonen hoe we de patronen kunnen gebruiken om de performantie van een programma te begrijpen, te schatten en te verbeteren.

Acknowledgements

I would like to thank my promotor Prof. Jan Lemeire for giving me the opportunity to do this thesis. I thank him for his many suggestions, his help and for proofreading this thesis many times. Of course, any remaining errors are my own.

It has taken me a long time to get to this point, and I am glad I did not give up. I take this opportunity to thank my family and friends who have supported me throughout these years.

Contents

Abstract i

Samenvatting ii

Acknowledgements iii

1 Introduction 1
- 1.1 Problem . . . 2
- 1.2 Approach . . . 2
- 1.3 Goal . . . 3
- 1.4 Previous Work . . . 4
- 1.5 Scientific Questions . . . 5
- 1.6 Thesis Outline . . . 6

2 Background: Parallel Computing 7
- 2.1 Parallel Computing . . . 7
  - 2.1.1 Types of Parallelism . . . 7
  - 2.1.2 Parallel Platforms . . . 7
  - 2.1.3 Programming Models . . . 8
- 2.2 Parallel Performance . . . 9
  - 2.2.1 Speedup . . . 9
  - 2.2.2 Sources of Overhead . . . 9
- 2.3 GPUs and CUDA . . . 11
  - 2.3.1 GPU . . . 11
  - 2.3.2 CUDA . . . 11
  - 2.3.3 Architecture of an NVIDIA GPU . . . 15
  - 2.3.4 Sources of Overhead . . . 17
  - 2.3.5 GPUs in this Thesis . . . 18

3 Branching 19
- 3.1 Description . . . 19
- 3.2 Benchmark Algorithm . . . 20
- 3.3 Remedies . . . 23
  - 3.3.1 Table Lookup . . . 23

3.3.2 Static Thread Reordering . . . 23
3.3.3 Dynamic Thread Reordering . . . 24
3.4 Automation . . . 27

4 Parallel Data Access **29**
4.1 Description . . . 29
4.2 Benchmark Algorithm . . . 31
4.3 Remedies . . . 36
4.3.1 Array of Structures . . . 36
4.3.2 Matrix Transpose . . . 37
4.4 Automation . . . 41

5 Synchronization Points **42**
5.1 Description . . . 42
5.2 Benchmark Algorithm . . . 43
5.3 Remedies . . . 45
5.4 Automation . . . 46

6 Data Partitioning and Mapping **47**
6.1 Description . . . 47
6.2 Benchmark Algorithm . . . 48
6.3 Remedies . . . 49
6.4 Automation . . . 52

7 Case Study: Prefix Sum **53**
7.1 Exclusive Prefix Sum . . . 53
7.2 Implementation in CUDA . . . 56
7.2.1 A Naive Implementation . . . 57
7.2.2 Anti-parallel Pattern: (Global) Parallel Data Access . 59
7.2.3 Anti-parallel Pattern: Branching . . . 59
7.2.4 Anti-parallel Pattern: Data Partitioning and Mapping 60
7.2.5 Anti-parallel Pattern: (Shared) Parallel Data Access . 61

8 Conclusion and Future Work **63**
8.1 Goals Evaluation . . . 63
8.2 General Conclusion . . . 66

Bibliography **70**

Chapter 1

Introduction

Parallel systems and parallel programming are becoming increasingly more important. In the past, sequential processors continuously became faster, providing traditional sequential programs a "free" performance gain with every new processor generation. This progress is coming to an end due to physical limitations. As a consequence, programmers are forced to write parallel programs. CPUs are becoming multi-core, and many-core processors like cell processors or GPUs can be afforded and programmed by many. However, writing a program for a parallel system is not always evident; writing one with a decent performance is even less so. The reason lies in the many sources of parallel overhead that are incurred by the program's parallel execution.

The goal of this work is to help the parallel programmer by identifying patterns in parallel programs causing overhead. Given that these patterns act against "free" parallelism, we will call them "anti-parallel patterns". For each anti-parallel pattern we will define a number of benchmark algorithms. Each of these benchmarks isolates a single pattern. We will use these benchmarks not only to model the effect of a given pattern on the performance of a program running on a parallel platform, but also to compare different parallel architectures or different configurations of the same architecture. Finally, anti-parallel patterns can be used to understand and evaluate the performance of a given parallel system.

The focus of this work is on fine-grain data-parallel programs running on NVIDIA GPUs - graphical processing units that can be used for general computing using CUDA. We chose this platform because of the relative low barrier of entry and the increased interest in GPUs and manycore architectures. The idea of anti-parallel patterns, nevertheless, is general enough to be used for other parallel platforms and other types of parallelism.

In section 1.1 we discuss the problem we are trying to solve. We describe our approach in section 1.2. The goal of our work is set in section 1.3. We make a comparison with previous work in section 1.4. A number of scien-

tific questions that are raised during this work are discussed in section 1.5. Finally, we conclude this chapter with an outline of this work in section 1.6.

1.1 Problem

Given the two main reasons for writing parallel programs - performance and the need to solve bigger problems - it is safe to say that developers of parallel programs are concerned about performance. However, it is not always easy to understand the performance of a parallel program running on a parallel platform, let alone to improve it. We can ask ourselves the question how, given a parallel program and a parallel platform, we can understand, predict and improve the performance of the program.

1.2 Approach

We will analyze the performance of a parallel program using, what we will call, "anti-parallel patterns" and that we define as follows:

> An anti-parallel pattern is a common part of parallel programs that cause the performance of the program to be less than ideal, namely a speedup of P when run by P processing elements.

Example 1 (Branching on SIMD Platforms)

Consider SIMD platforms. These are parallel platforms on which each processor executes the same instruction, but on different parts of the data. Conditional execution on SIMD platforms is typically made possible by the use of a hardware flag that allows processors to be "disabled" for certain instructions. For example, consider the following code run by a separate thread on each processor:

```
1 if (B == 0)
2     C = A;
3 else
4     C = A/B;
```

It will be executed in two steps. First, all processors for which the condition of the if-statement is true will execute, while all other processors are idle. Next, all processors for which the condition is false execute. As can be seen in the execution profile of two processors - one for which the condition is true and another one

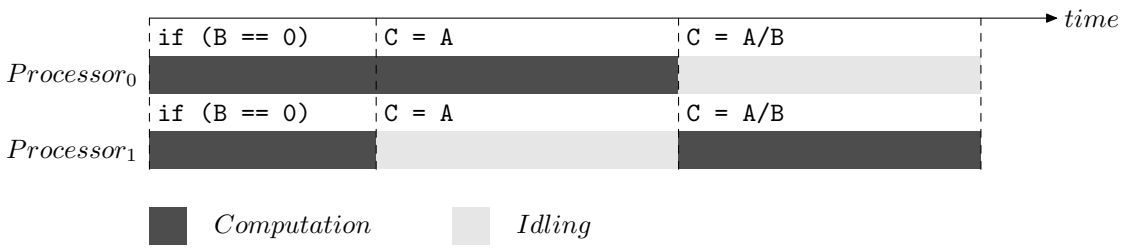


Figure 1.1: Execution Profile of 2 processors on an SIMD platform executing conditional code

for which the condition is false - in figure 1.1, this will cause processors to idle. Thus on SIMD platforms branching can be considered as an anti-parallel pattern that causes a performance degradation.

We will identify the most typical anti-parallel patterns of data-parallel programs and discuss the following elements for each of them:

- A description: we will give a *definition* of the pattern, and describe its effects on the performance of a parallel system. We will also enumerate the parameters that determine the behaviour of the pattern.
- Benchmark algorithm(s): we will define one or more benchmark algorithms that isolate the pattern and that allow us to measure its effect on the performance in function of its parameters.
- A model and heuristics: from the results of running the benchmark algorithm(s) on a parallel platform, we will try to model its behaviour for that parallel platform and formulate heuristics that can help evaluate the performance of a program running on the given platform in function of the anti-parallel patterns it contains.
- Remedies: a number of remedies that alleviate the problems caused by a pattern are discussed. This discussion includes the trade-offs and impacts of the remedies in question.
- The possibility to automate detection of the pattern, estimate performance degradation and propose remedies.

1.3 Goal

The goal of this work is to investigate a novel approach for improving the efficient usage of parallel processing power. More in particular, it tries to

find out to which extent anti-parallel patterns can help achieve the following goals:

1. Model the performance loss caused by an anti-parallel pattern on a given platform using the experimental results of our benchmark algorithms and our understanding of the parallel architecture.

2. Understand the performance of a program running on a parallel platform by identifying the anti-parallel patterns it contains and using the models we constructed. We strive to get a qualitative as well as a quantitative understanding: ideally the anti-parallel patterns will help us estimate the performance of a program running on a parallel platform.

3. Improve the performance of a program for a parallel platform. Each anti-parallel pattern a program contains points to a potential inefficiency and is therefore linked to solutions or remedies to overcome the parallel overhead.

4. Compare different parallel platforms: running the benchmark algorithms on different parallel platforms allows us to compare the effect of an anti-parallel pattern between them. These platforms can have a different architecture or can be platforms with the same architecture but a different configuration.

1.4 Previous Work

There exists already a large volume of work that shared our goal to assist the programmer to evaluate the performance of their parallel programs. Paradyn [1] automatically locates potential bottlenecks in parallel code and tries to explain why, where and when the problems occur. KappaPi [11] is a tool that relates the execution behaviour of an application with its structure. [24] tries to predict the performance of an application using algebraic mappings - convolutions - of application profiles to machine signatures of the machines on which the application will run. [10] fits overhead measurements to analytical forms in order to predict the performance of a given application.

Typically, these tools try to help the programmer find the main bottlenecks that cause their program to have a less than optimal performance. Our approach is complementary: given a parallel program, a developer or a tool will identify the anti-parallel patterns it contains. She will use these patterns to understand and estimate the performance of her program. Finally, she can use the remedies and solutions that we proposed to improve the performance of her program.

It should be pointed out that anti-parallel patterns are not in itself a novel idea. As will become clear later, a lot of work has already been done to identify these disadvantageous patterns and to propose solutions or remedies to alleviate their effect on the performance. But the results of these efforts are scattered. The novelty of our approach lies in the systematic categorization and modeling of the patterns. In doing so we provide a single point of information as well as a fairly simple tool to help programmers to understand and improve the performance of their parallel programs.

1.5 Scientific Questions

This thesis comprises a study of the usefulness of anti-parallel patterns. The anti-parallel patterns theme raises a number of scientific questions.

I. Does there exist an exhaustive list of patterns qualitatively explaining all performance degradations? If so, to what extent is the decomposition of a program into its anti-parallel patterns beneficial for performance analysis?

II. To what extent does the decomposition of a program into its anti-parallel patterns result into a decomposition of the performance? In this context we can ask the following subquestion:

    (a) Can the effect on the performance of a single pattern be described by a relatively simple model?

III. Can the patterns be identified automatically by a tool, (a) at compile or (b) at runtime?

IV. Can remedies be categorized according to the patterns?

V. Is it possible to automatically derive wether remedies apply for a given program?

VI. Are anti-parallel patterns useful in characterizing the performance of a given parallel platform. If so, is such a characterization complete?

VII. Can the performance drop caused by an anti-parallel pattern be measured at runtime by the hardware?

We will try to give partial answers to these questions in the conclusion of this work. Some questions will not be answered, but are given for the sake of completeness.

1.6 Thesis Outline

Chapter 1 is this introduction. It describes the problem we are trying to solve and the approach we want to take to solve it. It also compares our approach with previous work and raises a number of interesting questions.

Chapter 2 provides the background for this work. It gives an overview of parallel computing and provides a small introduction to parallel performance. It contains an extensive discussion of GPUs and CUDA.

Chapters 3, 4, 5 and 6 discuss four anti-parallel patterns: branching, data access, synchronization and data partitioning.

Chapter 7 presents a case study where we implement a parallel prefix sum and try the usefulness of our anti-parallel patterns.

Finally, chapter 8 provides a conclusion to this work and makes suggestions for future work.

Chapter 2

Background: Parallel Computing

This chapter provides the background for this work. In section 2.1, we recapitulate the most important terms and concepts of parallel computing and try to place our work in this domain. Then, in section 2.2 parallel performance and parallel overhead are discussed. Finally, in section 2.3 we provide a detailed discussion of GPUs and CUDA.

2.1 Parallel Computing

2.1.1 Types of Parallelism

We differentiate between two types of parallelism: data and task parallelism. They are defined as follows [16]:

- A data parallel computation is one in which parallelism is applied by performing the same operation on different items of data at the same time; the amount of parallelism grows with the size of the data.
- A task parallel computation is one in which parallelism is applied by performing distinct computations - or tasks - at the same time. Since the number of tasks is fixed, the parallelism is not scalable

2.1.2 Parallel Platforms

A popular categorization of computer platforms is the taxonomy of Flynn [12]. In this taxonomy computer platforms are categorized by the number of instruction and data streams they manage. The following categories are defined:

- SISD: Single Instruction Single Data.

- SIMD: Single Instruction Multiple Data.
- MISD: Multiple Instruction Single Data.
- MIMD: Multiple Instruction Multiple Data.

For parallel computing we only consider SIMD and MIMD computer platforms.

Another categorization of parallel platforms is by the way processing elements communicate. Some platforms will have a shared address space, and communication is achieved by reading from and writing to this shared address space. On other platforms processing elements mus communicate by sending messages.

2.1.3 Programming Models

A number of parallel programming models have been developed to facilitate the creation of source code for different parallel platforms. The most important ones are:

- Message Passing: different processing elements are able to execute operations on data that resides in their own exclusive workspace, however, they can only communicate with each other by sending messages. The best known implementation of message passing is probably MPI, the Message Passing Interface [2].
- Threads: different processing elements execute operations on data that reside in a shared address space. A POSIX standard was defined to specify the interface that an adhering programming interface must implement. Those implementations are known as Pthreads [3].
- Automatic parallelization: in some cases a compiler can generate parallel code automatically. For example OpenMP [4] , is an application program interface that may be used to explicitely direct multi-threaded shared memory parallelism. This is done by adding directives to the source code of a sequential program. The compiler will use these directives to generate parallel code.

Although message passing is most suitable for message passing platforms and threading for shared memory platforms, it is possible to implement message passing on shared memory platforms and threading on message passing platforms.

2.2 Parallel Performance

Parallel performance is the central theme of this work. We shall briefly review the most important definitions related to parallel performance and see that a program running on P processors does not necessarily run P times faster.

2.2.1 Speedup

We define the speedup of a parallel program as the execution time of a sequential program divided by the execution time of a parallel program that computes the same result. In particular [14]:

$$Speedup = \frac{T_s}{T_p} \tag{2.1}$$

We say that a parallel program exhibits a linear or ideal speedup when the speedup equals the number of processors on which the program is run:

$$T_p = \frac{T_s}{P} \tag{2.2}$$

Alternatively, we can say that the efficiency of the parallel program is equal to 1. The efficiency of a parallel program captures the fraction of time for which a processor is usefully employed by computations that also have to be performed by a sequential program:

$$E = \frac{T_s}{P \cdot T_p} \tag{2.3}$$

The speedup of a program with an efficieny equal to 1 is linear or ideal. Beginning programmers often expect linear speedup in parallel programs, however this is seldom the case. Programs with a higher efficiency are said to have a superlinear speedup, while those with a lower efficiency are said to have a sublinear speedup.

The superlinear speedup anomaly is explained by the fact that sometimes the parallel performance of the computation is at the advantage for example because the cache of each participating processor is large enough to hold the elements it needs to do its part of the computation.

Most programs suffer from sublinear speedup. For those programs we need a performance analysis. Sublinear speedup is typically caused by a combination of several sources of parallel overhead. We will discuss these sources of overhead in the next section.

2.2.2 Sources of Overhead

Many different categorizations of overhead in parallel programs exist. The categorization we use, is taken from [10].

Load Imbalance

When a computation needs to be executed in parallel, the tasks (for task parallelism) or the data (for data parallelism) need to be divided among the available processors. Sometimes, it can be difficult to provide the same amount of work to each processor. As a consequence, some processors will be idle while others are still doing work.

Insufficient Parallelism

Most computations have a part that is inherently sequential. This sequential part limits the performance that can be achieved by a parallel program. The maximal speedup that can be achieved is given by Amdahl's law [6]:

$$Speedup_{max} = \frac{P}{1 + (P - 1) \cdot S} \tag{2.4}$$

where P is the number of processors the program is running on and S is the fraction of the computation that is inherently sequential.

Synchronization Loss

When parts of a computation are dependent on each other, these parts need to synchronize. The time spent in synchronizing with other processors can be a major source of overhead. Some well known examples are lock acquisition and time spent waiting in a barrier.

Communication Time

Time spent in communication that is not overlapped with computation is another reason for parallel overhead. Communication refers to the exchange of data between processes. Typically, processes exchange data by sending each other messages or by reading and writing data in a shared memory. The waiting time will increase when the data needs to travel further i.e. when the locality of the data decreases.

Resource Contention

Because some resources are shared by more than one processor, they can cause waiting times when more than one processor tries to access them. Obviously, while a processor is waiting to access a shared resource, it cannot continue computing[1].

[1] Unless the access is asynchronous.

Figure 2.1: CPU versus GPU Chip area [21]

2.3 GPUs and CUDA

We concern ourselves with data parallel programs written in CUDA and run on NVIDIA GPUs. The execution of a CUDA program on an NVIDIA GPU corresponds to running a great number of threads that operate in SIMD mode on data that is stored in a common address space.

2.3.1 GPU

Today almost every new computer contains a graphics processing unit (GPU) to handle the computations needed to render complex graphics. This evolution has been driven by the gaming industry and its demand for interactive 3D graphics. Because of the specific demands of real-time computer graphics, GPUs have been optimized for high computation throughput, rather than for low latency [17].

In practice, where CPUs use a lot of chip area for cache and control logic, GPUs use most of the chip area for processors (figure 2.1). As a consequence, the computational power of GPUs is orders of growth larger than the computational power of even the most powerful CPUs [21].

Given this enormous computational power, a lot of research has taken place on general purpose computing on graphics hardware or GPGPU. By now, it has become possible for the average programmer to use the GPU to speed up his general purpose programs [17].

2.3.2 CUDA

CUDA stands for "Compute Unified Device Architecture". It was introduced by NVIDIA in November 2006 and allows the average programmer to use NVIDIA GPUs for general purpose computations [21]. CUDA presents

itself as a set of extensions to the C programming language and a runtime library.

A CUDA program will typically transfer data to the GPU, launch a kernel on the GPU to process it and finally retrieve the resulting data from the GPU. It is interesting to note that in this context the CPU is referred to as the "host" and the GPU as the "device".

A kernel is a procedure, identified by the `__global__` keyword, that is executed across a set of parallel threads. In order to identify the thread that is running a kernel, threads are organized in a 2-level hierarchy: threads belong to 3-dimensional thread blocks, that in turn belong to a 2-dimensional grid. The position of a thread in a block can be identified at runtime by accessing the `threadIdx` variable, while the position of a thread block in the grid can be identified by accessing the `blockIdx` variable. The dimensions of thread blocks and grids can be identified at runtime by accessing the `blockDim` and `gridDim` variables respectively (figure 2.2).

It is up to the programmer to specify the execution environment of a kernel i.e. the dimension of the grid and the thread blocks it contains. This is done by defining both dimensions and specifying them as follows when the kernel is launched:

```
kernel<<<dimGrid, dimBlock>>>(par1, par2);
```

Parallel execution and thread management is handled automatically in hardware and is not the programmer's concern. Threads in the same thread block can synchronize using barrier synchronization. This is done with the `__syncthreads()` function. Threads that belong to different thread blocks cannot communicate or synchronize with each other. This is a trade off that has been made for the sake of scalability: because of it, thread blocks execute independent of each other. This independence allows thread blocks to be executed in any order across any number of cores and makes the CUDA model scalable across an arbitrary number of cores, as well as across a variety of parallel architectures [19].

The CUDA programming model explicitely presents a hierarchy of memories to the programmer. Each thread has a private local memory. All threads in a thread block have access to a shared memory. All threads in the grid have access to the same global memory. Finally, there are two additional read-only memory spaces that are accessible to all threads: the constant and texture memory spaces.

The global memory is handled from the host: allocating global GPU memory is done by calling the function `cudaMalloc`, while it is freed by calling the funtion `cudaFree`. The function `cudaMemcpy`, finally, is used to transfer data from the host to the device, the device to the host or within the device.

We finalize our presentation of the CUDA programming model by enumerating a number of restrictions CUDA programs must adhere to:

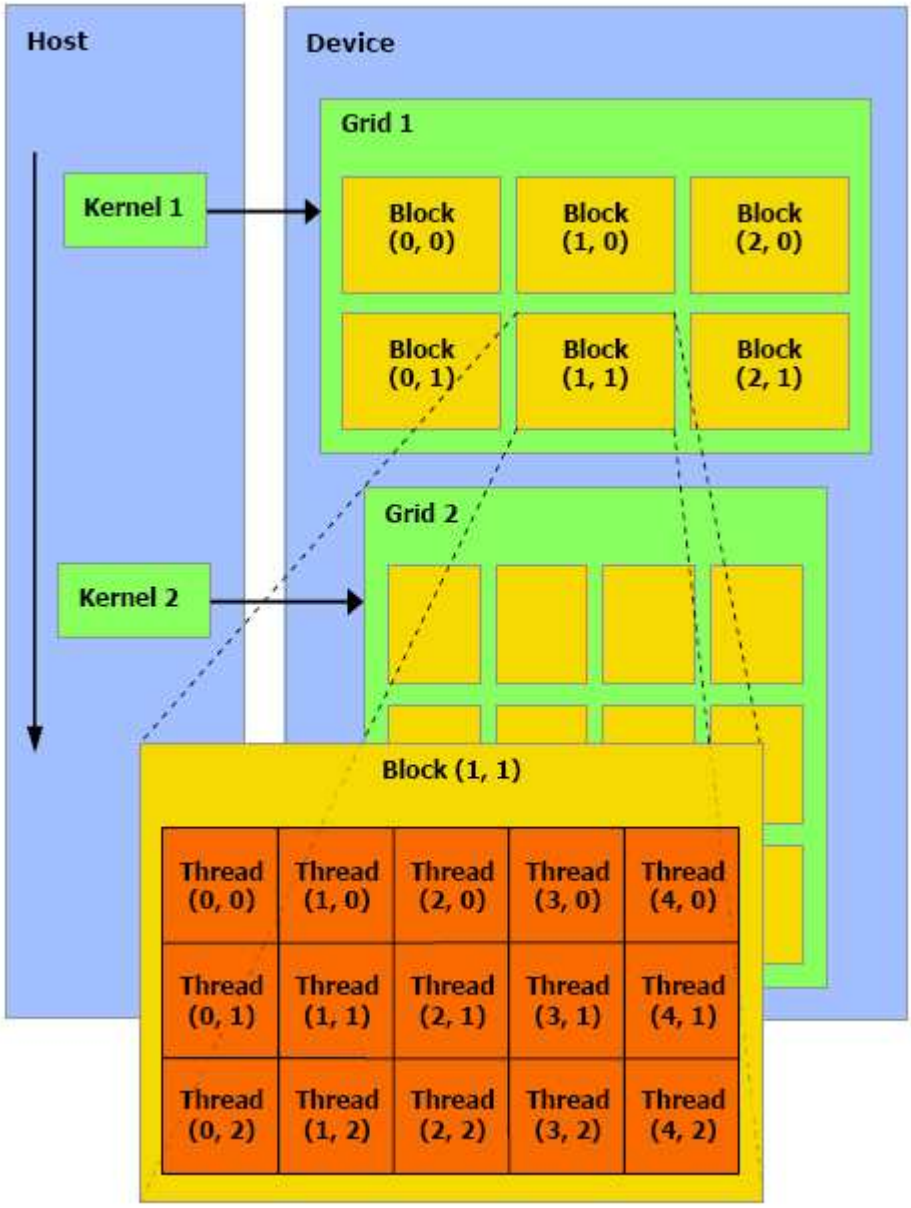


Figure 2.2: Kernel Execution on the GPU

Listing 2.1: A Simple Kernel

```
__global__ void saxpy(int n, int alpha, float *x, float *y)
{
    int i = blockIdx.x * blockDim.x + threadIdx.x;

    if (i < n) y[i] = alpha * x[i] + y[i];
}
```

- Threads can only be created by the invocation of a parallel kernel.
- Communication between thread blocks is sheer impossible.
- Recursive kernels are forbidden.
- Static variables in kernels are forbidden.
- Kernels with variable argument lists are forbidden.

Example 2 (A Simple Kernel [19])

Consider the following equation where α is a scalar value and X and Y are vectors (arrays).

$$Y = \alpha X + Y$$

A kernel that can be used to compute this in parallel is shown in listing 2.1. The `__global__` keyword identifies this function as a kernel. Next, we notice how the index of the element in the X and Y arrays is determined from the values of `blockIdx`, `blockDim` and `threadIdx`. Finally, it is necessary to test wether this index falls in the range of valid indices of the X and Y vectors.

Code that could launch this kernel is shown in listing 2.2. As shown it is first necessary to allocate the necessary GPU memory and transfer the data to this freshly allocated memory. Then, the kernel is launched. Here, the expressions that determine the grid and block dimensions are simple because both use only one dimension. We have a grid that contains enough thread blocks of 256 threads to process all data. When the data has been processed, the result is copied back to the host memory and the GPU memory that was allocated is freed.

Listing 2.2: A Kernel Wrapper

```
// xh and yh point to the X and Y vectors in host memory
// xd and yd point to the X and Y vectors in device memory

cudaMalloc((float **) &xd, n * sizeof(float));
cudaMalloc((float **) &yd, n * sizeof(float));
cudaMemcpy(xd, xh, n * sizeof(float), cudaMemcpyHostToDevice);
cudaMemcpy(yd, yh, n * sizeof(float), cudaMemcpyHostToDevice);

int nblocks = (n + 255) / 256;
saxpy<<<nblocks, 256>>>(n, 2.0, x, y);

cudaMemcpy(yh, yd, n * sizeof(float), cudaMemcpyDeviceToHost);
cudaFree(xd);
cudaFree(xh);
```

2.3.3 Architecture of an NVIDIA GPU

Although the above description of CUDA should be sufficient to write a working CUDA program, it can be beneficial to know the architecture of a typical NVIDIA GPU.

Each GPU contains an array of multi-threaded streaming multiprocessors (SM), a number of external DRAM partitions and a Compute Work Distribution Unit (CWDU) that is responsible for distributing the available work to the available SMs (figure 2.3). The CWDU enumerates the thread blocks of the grid and distributes them to the SMs with available capacity. When a thread block terminates, a new thread block is assigned to the SM on which it was running. It should be noted that a thread block is run on one SM only.

An SM consists of 8 scalar streaming processors (SP), 2 special function units (SFU) and a multi-threaded instruction unit (MTIU). An SM creates, manages and executes up to 768 concurrent threads or 8 concurrent thread blocks (which ever boundary is hit first). The `__syncthreads()` primitive is implemented with a single instruction in hardware. An SM also contains on-chip memory that is used to implement the CUDA shared memory discussed earlier. This shared memory is limited to 16 KB per SM and is organized in banks.

Threads are run in groups of 32, called "warps". Basically, these will all execute the same instruction. The 32 first threads of a thread block will form the first warp, the next 32 the second, and so on. An SM can handle a pool of 24 warps (corresponding to 768 threads running concurrently on one SM).

Finally, it is important to mention that each NVIDIA GPU has got a certain compute capability that describes the features of the hardware and reflects the set of instructions supported by the device as well as other specifications [20]. In practice, certain instructions are only possible on

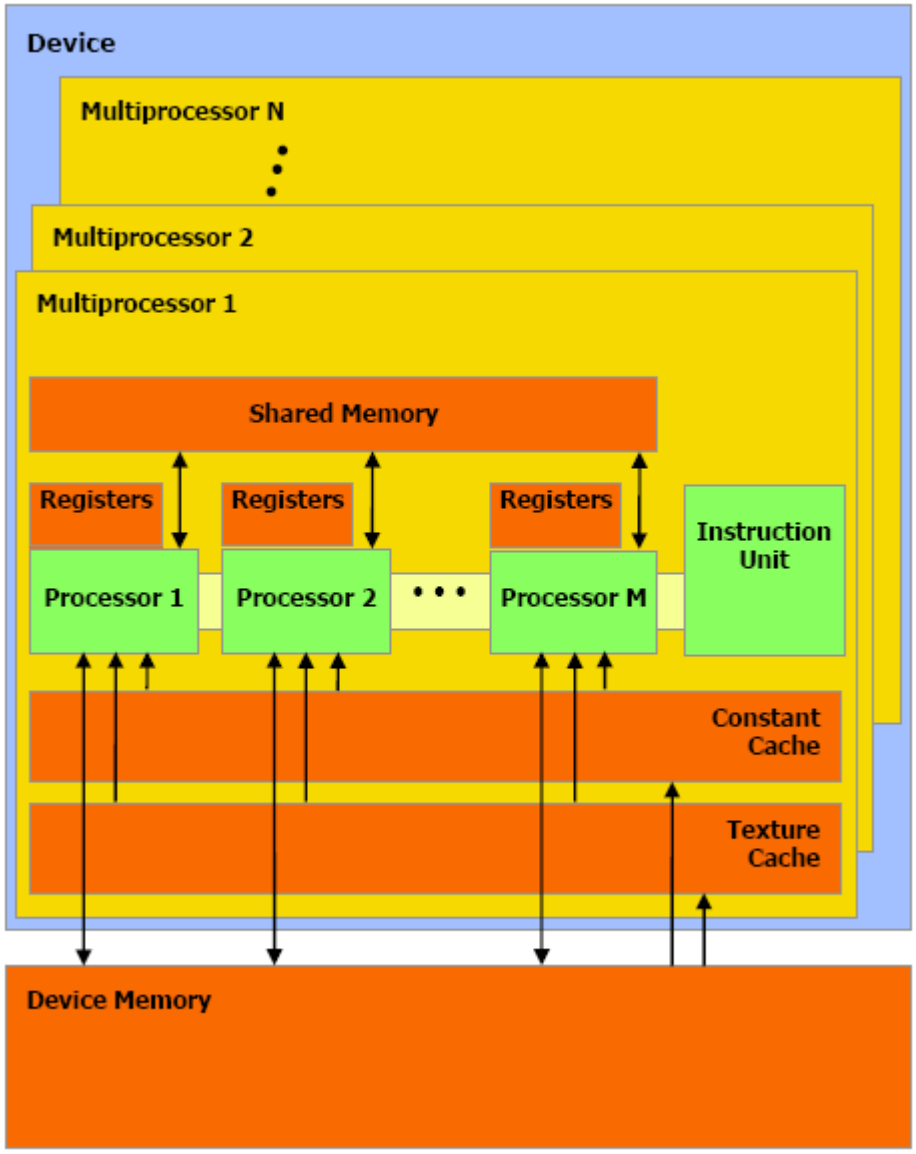


Figure 2.3: GPU Architecture [21]

GPUs with a higher compute capability and GPUs with different compute capabilities will have different performance characteristics.

2.3.4 Sources of Overhead

The following subsections discuss the sources of overhead that can occur when using an NVIDIA GPU for parallel computations [19][20][21].

Memory Latency

An SP that accesses global memory needs to wait hundreds of cycles for the data. This problem can be alleviated by using shared memory for data that is reused several times or by recalculating data rather than fetching it from global memory.

Single Instruction Multiple Threads

As explained earlier an SM executes instructions per group of 32 threads (warps). Basically, this means that all threads in a warp execute the same instruction. As a consequence, instruction divergence caused by branching can reduce the computation's performance.

Memory Coalescence

Access to global memory by 16 threads can be serviced in one transaction if the access is coalesced. In chapter 4 we will discuss coalescence in detail. There we will see that non coalesced access can cause serious performance penalties.

Bank Conflicts

A phenomenon similar to memory coalescence can be observed for shared memory. Shared memory is organized in equally sized memory modules, called banks. Because consecutive words are stored in consecutive banks and because different banks can be accessed simultaneously, it is important that access to shared memory is distributed across all banks. If two or more threads access the same memory bank, these accesses will be serialized.

Limited Resources

The resources of an SM are limited. The most important limitations are the size of the register file (8KB), the size of the shared memory (16KB) and the maximum number of 8 thread blocks or 768 threads that can execute at the same time on an SM. These limitations can seriously impact the performance of a computation. When an SM can execute less thread blocks

Characteristic	Geforce GTX280	Geforce 8500GT
CUDA Capability	1.3	1.1
Amount of Global Memory	1.054.539.776 Bytes	519.634.944 Bytes
Number of SMs	30	2
Number of SPs	240	16
Amount of Shared Memory per SM	16.384	16.384
Amount of Registers per SM	16.384	8.192
Warp Size	32	32
Maximum Threads per Block	512	512
Maximum Dimensions of a Block	512 x 512 x 64	512 x 512 x 64
Maximum Dimensions of a Grid	65.535 x 65.535 x 1	65.535 x 65.535 x 1
Clock Rate	1.30 GHz	0.92 GHz

Table 2.1: Characteristics of the GPUs we have used.

simultaneously because one of these limitations is met, it will have less opportunity to hide the latency of long memory access operations.

Synchronization

The means of synchronization in CUDA are limited: only threads within the same thread block can synchronize with each other using `__syncthreads()`. If all threads have to synchronize, it is necessary to stop the current kernel and start a new one. This adds the overhead of launching a kernel. Furthermore, the problem will probably be exacerbated by the introduction of extra data transfers between global and shared memory.

2.3.5 GPUs in this Thesis

The programs that are discussed throughout this thesis have been run on two NVIDIA GPUs: the small Geforce 8500GT and the larger Geforce GTX280, both from NVIDIA. Their most important characteristics are resumed in table 2.1.

Chapter 3

Branching

3.1 Description

Definition

> Branching refers to code that contains conditional statements such that, when run in parallel, it causes different processing elements to follow different execution paths.

Discussion

Depending on the parallel system running the code, branching can have a negative impact on the performance of the system. Branching statements do not constitute a problem on parallel systems with an MIMD architecture. In these systems each processing element can follow their own execution path. On systems with an SIMD architecture, however, branching can severely damage performance. In SIMD systems all processing elements must execute the same code. This results in lost cycles when conditional code is run. For example, consider the following code excerpt:

```
1 if (number < 0)
2     absNumber = -number;
3 else
4     absNumber = number;
```

While the processing elements for which the condition is true execute, all other processing elements remain idle and vice versa. Therefore, this particular piece of code will run two times slower than one might expect.

As mentioned in chapter 2 a GPU consists of a number of SMs. an SM contains 8 SPs and 1 MTIU. To maximize the processing power of an SM threads are scheduled in groups of 32 known as “warps”. Threads belonging to the same warp operate in an SIMD fashion i.e. they operate in lockstep.

When threads in a warp execute conditional statements the execution needs to be serialized. For example, if two branches are followed, first the threads that follow the first branch will be run, while the others remain active. Only thereafter will the threads that follow the second branch be run.

Threads that belong to the same warp but that follow different execution paths are called "divergent threads".

Parameters

The following parameters define the effect of the branching pattern:

- The number of branches followed.
- The work performed by each branch.
- The average number of branches followed per warp.

3.2 Benchmark Algorithm

A simple algorithm was defined to study the effect of branching. The algorithm executes an operation for each element of an input matrix and stores the result in an output matrix. However, the operation that is executed for an element depends on the element's value. This is accomplished with a big *switch* statement that executes the appropriate operation for a given element. For the sake of simplicity we chose to execute the same operation in each branch of the switch statement[1]. A kernel implementing this algorithm is shown in listing 3.1.

The branching is parameterized. By "abusing" the input matrix as a map of thread indexes and branch numbers, we can mimick any possible branching configuration.

For reasons that will become clear later, we have run two versions of the algorithm. One applies a convolution[2] - a data intensive operation - to each element of the matrix, while the other applies a compute intensive operation.

Results

Figure 3.1 shows the runtime in function of the average number of branches of the first version of our algorithm on the Geforce GTX280 for a square

[1] And also because the compiler does not optimize away this trick.

[2] Convolution filtering is a technique that is used in image processing for example for smoothing and edge detection. An image is convoluted with a filter by replacing the value of its pixel by its convoluted value. This value corresponds to the scalar product of the filter values with the values of the pixels that surround the pixel whose new value is being calculated [23].

Listing 3.1: Branching Kernel

```
__global__ void branchingKernel(int *in, int *out, int dataWidth)
{
    int row = blockIdx.y * BLOCK_WIDTH + threadIdx.y;
    int col = blockIdx.x * BLOCK_WIDTH + threadIdx.x;

    // Shared Memory
    __shared__ int shared[BLOCK_WIDTH][BLOCK_WIDTH];

    // Populate shared Memory - not shown

    int result;
    int element = shared[threadIdx.y][threadIdx.x];

    switch (element) {
        case 0:
            result = doOperation(element, shared, threadIdx.x
            break;
        case 1:
            result = doOperation(element, shared);
            break;
        case 2:
            result = doOperation(element, shared);
            break;
        case 3:
            result = doOperation(element, shared);
            break;
        case 4:
            result = doOperation(element, shared);
            break;
        case 5:
            result = doOperation(element, shared);
            break;
        case 6:
            result = doOperation(element, shared);
            break;
        case 7:
            result = doOperation(element, shared);
            break;
        case 8:
            result = doOperation(element, shared);
            break;
        case 9:
            result = doOperation(element, shared);
            break;
        default:
            break;
    }

    if (row < dataWidth && col < dataWidth) {
        out[row * dataWidth + col] = result;
    }

    return;
}
```

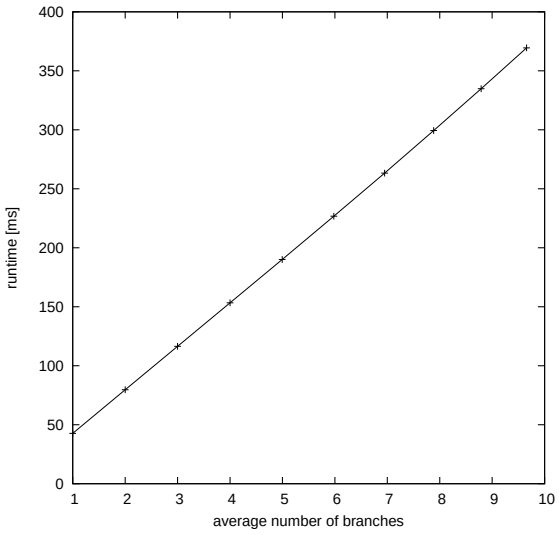


Figure 3.1: Runtime in Function of the Number of Branches Followed

matrix of width 2048. Running the algorithm on the Geforce 8500GT yielded similar results. Our observations correspond to our expectations: for each additional branch that is followed by a warp, the runtime increases with a constant amount. In our case this time roughly corresponds to the time that would be needed to run the algorithm when only this additional branch is followed. More precisely if $t_{outside}$ is the time spent outside any branch, t_{branch} the time spent in one branch and n the average number of branches followed per warp, the total runtime can be estimated by:

$$t_{total} = t_{outside} + n \times t_{branch} \tag{3.1}$$

Because different work is done in different branches, the time spent in each branch will be different. We modified our algorithm to take this into account: the second branch was changed to run twice as long as the first, the third branch to run thrice as long and the fourth branch to run four times as long. The results of this experiment are shown in figure 3.2. As can be expected, the total runtime corresponds roughly to the runtimes of the involved branches:

$$t_{total} = t_{outside} + \sum t_{branch} \tag{3.2}$$

Finally, the branches that are followed might not be evenly distributed. For example if there are two branches, the first might be followed in 90 % of the cases and the second only in 10 % of the cases. This can be taken into account by assigning a relative weight w to each branch and reformulating the estimation of the total runtime as follows:

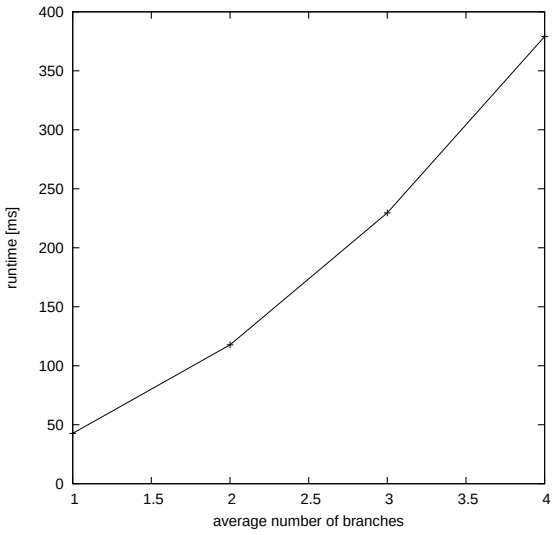


Figure 3.2: Runtime in Function of the Number of Branches Followed for Branches with different Runtime

$$t_{total} = t_{outside} + n \cdot \sum w_{branch} \times t_{branch} \tag{3.3}$$

3.3 Remedies

A number of remedies exist to alleviate the problems caused by branching. These remedies will be discussed in order of increasing complexity. Interestingly, this ordering corresponds to one of increasing genericity.

3.3.1 Table Lookup

In some cases it is possible to replace conditional code by simple data lookup. For example, if the operations performed in the different branches only differ by some constant factor, it is a better idea to store these constant factors in a table and to look them up at runtime. This approach was used in a CUDA implementation of the DES algorithm [9].

The disadvantages of this approach are the extra operations introduced to determine the appropriate index in the table, the additional data access and the reduced clarity of the resulting source code.

3.3.2 Static Thread Reordering

In many algorithms branching is determined by the position of the element that is being processed, rather than its value. If this is the case, it is better to

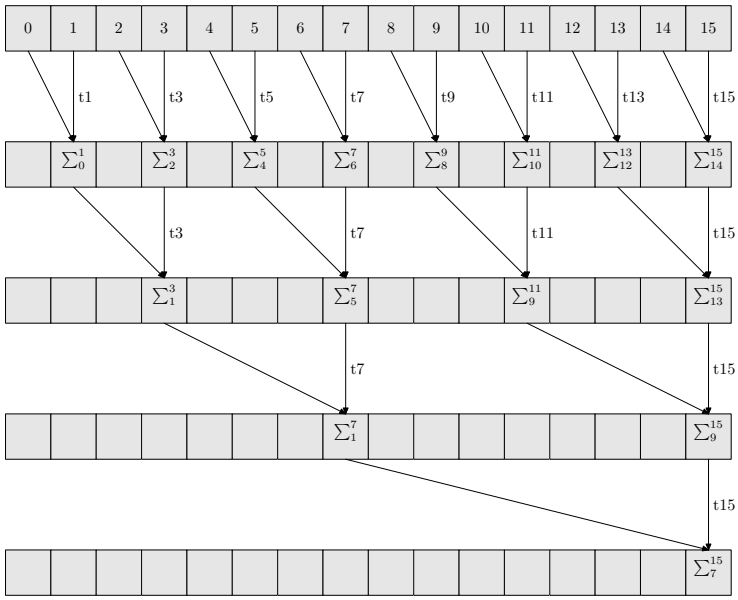


Figure 3.3: Computing the Sum of an Array of 16 Elements

reorder the threads statically such that threads that follow the same branch are grouped together.

Consider for example the problem of calculating a sum of a series of numbers[3]. Typically, this is done by first adding even/odd pairs of data values, yielding intermediate sums that are themselves added in pairs until only one pairwise sum is left [15] (figure 3.3).

If we implement this algorithm in a tradional manner where each thread handles the element whose position corresponds to the position of the thread, the performance will be horrible. During the first iteration only half of the threads belonging to a warp are active, during the second iteration only one fourth, and so on... A better way is to reorganize the source code such that the threads that perform the actual additions are always the first threads of a thread block (figure 3.4).

In chapter 7 we will show in practice how code can be reorganized to achieve a static thread reordering.

3.3.3 Dynamic Thread Reordering

A static reordering of threads can only be applied if we know on forehand which branches will be followed by which threads. Then we can modify the source code and impose such a reordering. Often the branch followed by a given thread is only known at runtime. In such cases it can sometimes be

[3]This is a specific case of a parallel reduction.

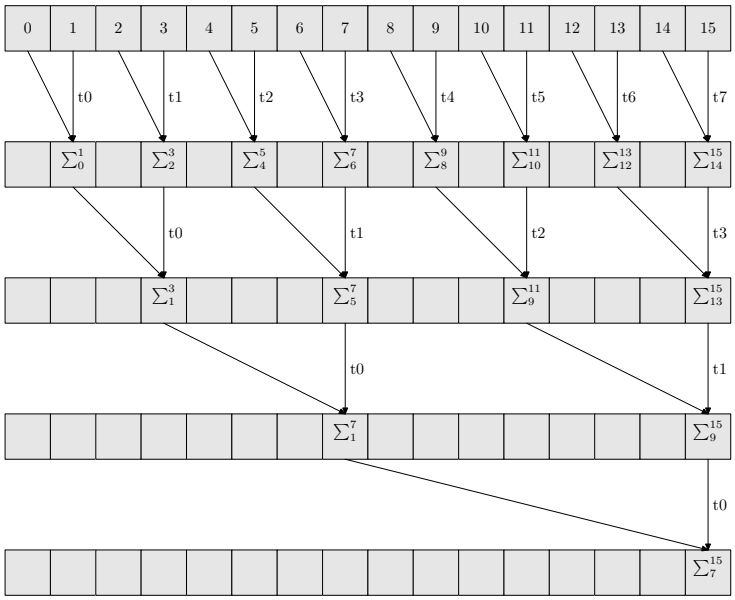


Figure 3.4: Statically Reordered Threads

useful to perform a dynamic reordering of the threads i.e. a reordering at runtime. We will first give an informal description of how this can be done and then turn to a practical implementation.

Two additional arrays are needed to implement dynamic thread reordering: one that is indexed by thread number and that we will call T and another one that is indexed by branch number and that we will call B. These arrays can be stored in shared memory. First, every thread will determine which branch it would follow if it was to process the element with the same index. It will then increment the corresponding value in B. Once all threads have done this, B will contain for every branch the number of times it is followed. Then, an exclusive prefix sum is executed on B. Now B contains the first index into T at which an element that follows a branch can be stored. Finally, every thread stores the number of the element it would process in table T. To do so, the thread executes an atomic addition on the appropriate value in B. This finalizes the dynamic thread reordering. Now, rather than processing the element that corresponds to it, a thread will lookup the element it has to process in T. Because we have ordered the elements in this table in such a way that elements that follow the same branches can be found in contiguous parts of T, we expect the average number of branches per warp to decrease and the performance to increase.

Listing 3.2 shows a practical implementation of dynamic thread reordering that can be used for kernels that work with one-dimensional data. It is interesting to note a few things concerning this implementation. First, the

Listing 3.2: Dynamic Thread Reordering

```
1
2  __shared__ int threadIndices[BLOCK_SIZE];
3  __shared__ int branchIndices[NUMBER_OF_BRANCHES];
4
5  int element = shared[threadIdx.x];
6
7  if (threadIdx.x < NUMBER_OF_BRANCHES) {
8      branchIndices[threadIdx.x] = 0;
9  }
10 __syncthreads();
11
12 atomicAdd(&branchIndices[element], 1);
13
14 __syncthreads();
15
16 exclusivePrefixSum(branchIndices);
17
18 __syncthreads();
19
20 int index = atomicAdd(&branchIndices[element], 1);
21 threadIndices[index] = threadIdx.x;
22
23 __syncthreads();
```

two extra arrays are stored in shared memory; dynamic thread reordering is done for a thread block and shared memory provides fast and convenient access. Second, we are using the *atomicAdd* operation that provides atomic access to variables in shared memory. Executing an *atomicAdd* on a variable will return its old value and increment the variable's value with the amount given. As the name suggests, this operation is executed in an indivisible manner. Finally, the function *exclusivePrefixSum* performs an exclusive prefix sum on the array *threadIndices*. We will discuss an efficient implementation of an exclusive prefix sum in CUDA in chapter 7.

A number of trade-offs should be made when choosing for dynamic reordering of threads. First of all, performing the reordering incurs some extra work that might cause the new code to run slower than the original code. Second, the introduction of a number of new variables and the usage of extra shared memory might cause the SMs of the GPU to run less thread blocks in a concurrent manner. Finally, dynamic reordering of threads might cause less efficient data access for data intensive algorithms.

We implemented dynamic thread reordering for our branching algorithm. Figure 3.5(a) compares the runtimes of running the algorithm using convolution with and without table lookup. Figure 3.5(b) does the same for the algorithm using a compute intensive operation. It appears that for these algorithms using dynamic thread reordering becomes interesting from a certain number of branches followed per warp. Also, this number appears to be lower for the algorithm using the compute intensive operation. Probably this is caused by the introduction of bank conflicts for the algorithm that

uses convolution.

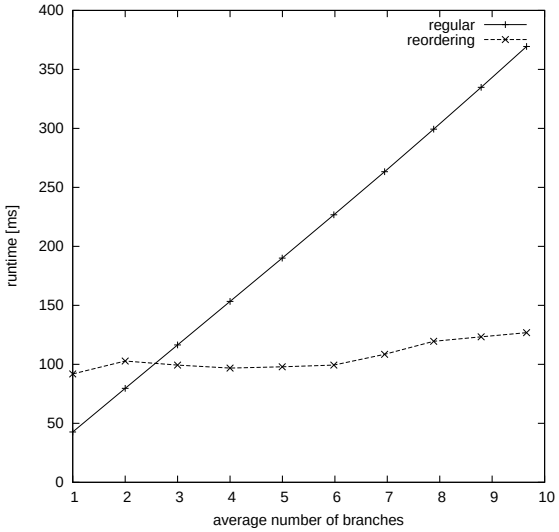
A thorough elaboration of dynamic thread reordering and more advanced implementations can be found in [26].

A similar idea but on hardware level is presented in [13] The authors of this paper propose a hardware implementation that dynamically regroups threads into new warps on the fly following the occurrence of diverging branch outcomes. According to them this could improve performance by an average of 20.7% for an estimated chip area increase of 4.7%.

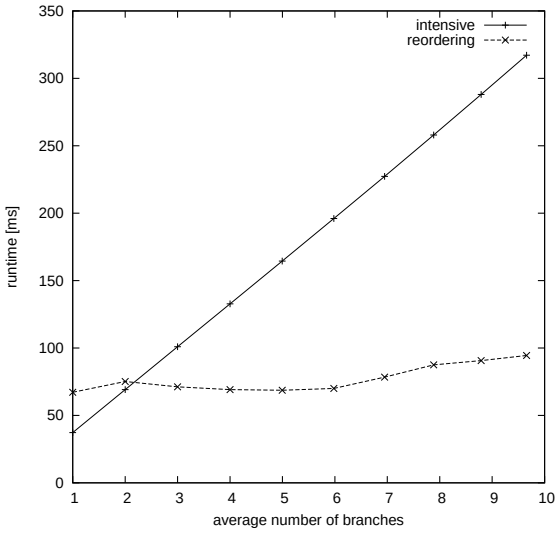
3.4 Automation

Ofcourse it is possible to automatically detect conditional statements - *if*-, *switch*-, *while*- or *for*- statements - in the source code. It is also possible to discern wether the belonging condition is a function of the thread index or not. In any case, an appropriate remedy - dynamic or static thread reordering - can be proposed to the programmer. However, any automatic tool should avoid "false alarms". For example, *if*- statements for bound checking are very common and necessary in CUDA kernels.

We asked the question wether it would be possible to measure the performance drop of a given pattern in hardware. Given that conditional execution on the GPU is handled by the hardware (by associating a flag with each core), we think this could be done in a straightforward manner.



(a) Data Intensive Branches



(b) Compute Intensive Branches

Figure 3.5: Influence of Dynamic Thread Reordering on the Runtime

Chapter 4

Parallel Data Access

4.1 Description

Definition

> Parallel data access refers to the concurrent access of different processing elements to shared memory.

Discussion

The performance of parallel systems is often limited because of high I/O latency. Most parallel platforms typically have a lot of processing power, but their access to external memory is not at par. When processes access memory, they become idle waiting for their request to be satisfied. CPUs hide latency by using large amounts of cache that is present on the processor chip. Using cache on parallel platforms is more difficult. It requires the usage of cache coherency mechanisms and furthermore there is not enough chip area on many-core platforms like GPUs.

The strategy to hide latency on the GPU is massive multi-threading. If enough threads are created, the GPU will be able to keep all its SMs busy by running ready threads while other threads are waiting for their data access requests to be serviced. Because on the GPU threads are completely handled in hardware, this is virtually a free operation.

In this chapter we will consider the extra latency that is caused by the concurrent access of memory. Parallel platforms support parallel reading and writing by different processing elements in order to decrease memory access latency and increase the memory throughput. Roughly speaking, to keep P processing elements busy the I/O capacity must be P times greater. GPUs support parallel memory access to both shared and global memory. However, the degree of parallelism achieved will depend on the memory

access pattern. We will try to explain what we mean by this both for global and shared memory in the following paragraphs.

Global Memory On NVIDIA GPUs data access is processed per group of 16 threads - or half-warps - and global memory is accessed via transactions of 32, 64 or 128 bytes that are naturally aligned. This means that the start address of the segment must be a multiple of its size. Under certain conditions data access for a half-warp can be performed in only one or a few transactions. These conditions depend on the compute capability of the GPU and we will briefly describe what they are for the GPUs we worked with and for words of 4 bytes. For a complete discussion we refer to [21].

For devices of compute capability 1.0 and 1.1 all words read or written by the threads in a half-warp must lie in the same 64 bytes segment. Furthermore, threads must access the words in sequence i.e. the kth thread of the half-warp must access the kth word of the segment.

The conditions for devices of compute capability 1.2 and 1.3 are not so stringent. All words read or written by the threads in a half-warp must lie in the same 128 byte segment. If only the lower or the upper half of this segment is used, a transaction of 64 bytes suffices to service all threads. The access order within the half-warp no longer matters.

Another phenomenon that needs to be taken into account is "partition camping". Global memory is made up of a number of partitions of width 256 bytes i.e. the first 256 bytes will reside in the first partition, the following 256 bytes in the second partition and so on. Partition camping takes place when more than one concurrent half-warp tries to access data in the same partition. These requests need to be queued, thus resulting in an increase of the data latency [22]. Partition camping concerns global memory accesses amongst active half-warps.

Shared Memory Shared memory is organized in 16 banks such that successive 4 byte words are assigned to successive banks (figure 4.1). Each bank can service 4 bytes per two clock cycles [21]. Given that data is accessed per half-warp, access to shared memory will be optimal if each thread of a half-warp accesses another bank. If more than one thread accesses the same bank, these accesses need to be serialized, causing threads to be idle for a longer time[1].

Parameters

The following parameters define the condition of the data access pattern:

- The pattern of the data access i.e. how do threads of the same half-warp access the memory.

[1] The only exception being many threads that read the same element.

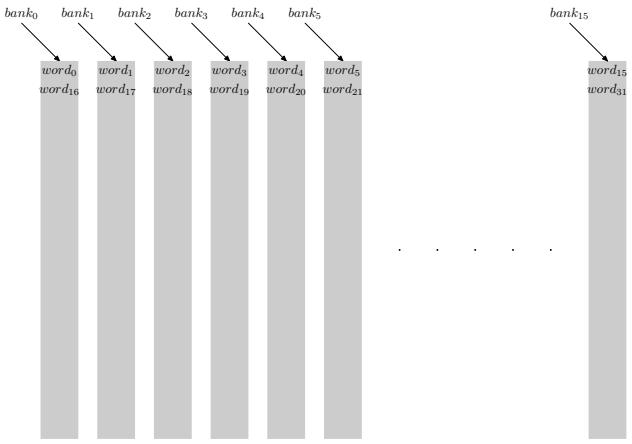


Figure 4.1: Organization of shared memory banks.

- The data access type: global or shared memory.

We do not include the amount of computation performed by the processing elements. As such, we will study the effect of different access patterns in the absence of latency hiding. Modeling the data access pattern in the presence of latency hiding is more intricate. However, it should be clear that the greater the potential for latency hiding, the smaller the impact of the data access pattern on the performance will be.

4.2 Benchmark Algorithm

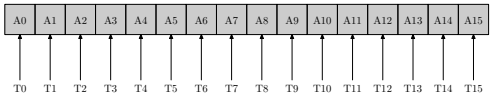
Note Ideally, we would like to test each possible access pattern and indeed this was tried. We created a kernel whose access pattern was stored in a table in constant memory. However, the time needed to access this table dominated the runtime of the kernel and made it difficult to get a good benchmark. Therefore, we limited ourselves to a number of simple kernels that will be described next. Given that the best-case and the worst-case access are comprised in these kernels, they are sufficient to get an idea about the performance loss when the access patterns are less than ideal.

Global Memory The reference kernel is shown in listing 4.1. It simply copies an input matrix to an output matrix[2]. Given the dimension of a thread block - 16×16 - the kernel can be adapted to exhibit non-coalesced

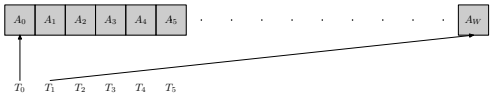
[2]We chose a two-dimensional structure because this allowed us to introduce non-coalesced data access in a "fair" manner by simply exchanging *threadIdx.x* and *threadIdx.y*. For one-dimensional data it would be necessary to introduce very expensive modulo calculation.

Listing 4.1: Global Data Reference kernel

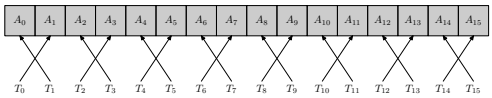
```
__global__ void globalCopy(float *in, float *out, int width)
{
    int inRow = BLOCK_WIDTH * blockIdx.y + threadIdx.y;
    int inCol = BLOCK_WIDTH * blockIdx.x + threadIdx.x;
    int outRow = BLOCK_WIDTH * blockIdx.y + threadIdx.y;
    int outCol = BLOCK_WIDTH * blockIdx.x + threadIdx.x;

    if (inRow < width && inCol < width && outRow < width && outCol < width) {
        in[inRow * width + inCol] = out[outRow * width + outCol];
    }

    return;
}
```



(a) Coalesced Access



(b) Uncoalesced Access (W is the width of the matrix)



(c) Crossed Access

Figure 4.2: Access to Global Memory

data access by simply exchanging $threadIdx.x$ and $threadIdx.y$ when calculating the row and column of the target element.

We also defined a kernel that can be used to determine the compute capability to which the memory of a given GPU device responds. This kernel copies an input array to an output array. However, rather than reading and writing the element whose index corresponds to the thread's index, threads with even indices read and write elements whose index inmediately follows the thread's index, while threads with odd indices handle elements whose index inmediately preceeds the thread's index. For GPUs of compute capability 1.2. and 1.3. we expect the performance of this kernel to equal the performance of an optimal copy kernel. For GPUs of lower compute capability the performance will be much worse. The access pattern of the three kernels are shown in figure 4.2.

Listing 4.2: Shared Data Reference kernel

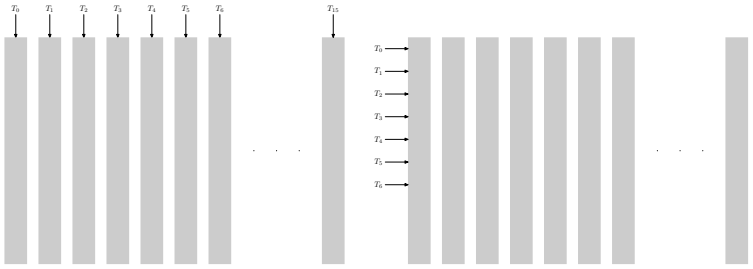
```
__global__ void sharedCopy(float *in, float *out, int width)
{
    __shared__ float sharedIn[BLOCK_WIDTH][BLOCK_WIDTH];
    __shared__ float sharedOut[BLOCK_WIDTH][BLOCK_WIDTH];

    int row = BLOCK_WIDTH * blockIdx.y + threadIdx.y;
    int col = BLOCK_WIDTH * blockIdx.x + threadIdx.x;

    if (row < width && col < width) {
        sharedIn[threadIdx.y][threadIdx.x] = in[row * width + col];
    }
    else {
        sharedIn[threadIdx.y][threadIdx.x] = 0.0;
    }
    __syncthreads();

    for (int i = 0; i < 100; ++i) {
        sharedOut[threadIdx.y][threadIdx.x] = sharedIn[threadIdx.y * BLOCK_WIDTH +
            threadIdx.x];
    }
    __syncthreads();

    if (row < width && col < width) {
        out[row * width + col] = sharedOut[threadIdx.y][threadIdx.x];
    }

    return;
}
```

Shared Memory The reference kernel is shown in listing 4.2 This kernel reads data from an input matrix in global memory to a matrix in shared memory. The data is then written 100 times to another matrix in shared memory. Finally, the data is read from the second matrix in shared memory and written to an output matrix in global memory. Exchanging $threadIdx.x$ and $threadIdx.y$ in the statement that performs the write from shared memory to shared memory will maximize bank conflicts. The access pattern of a half-warp to shared memory is shown in figure 4.3 for both the best and worst case.

Results

Global Memory The runtimes of the global data kernels as a function of the matrix width are shown in figure 4.4(a) for the Geforce 8500GT and in figure 4.4(b) for the Geforce GTX280. The graphs shown correspond to the four possible combinations for reading from and writing data to global memory.

The first thing we notice is the peaked behaviour for the kernels that write to memory in a non-coalesced fashion. These peaks occur at regular intervals. For the Geforce 8500GT they occur every 128 elements, while for



(a) No bank conflicts (b) Maximal bank conflicts

Figure 4.3: Access to Shared Memory

Read	Write	Geforce 8500GT	Geforce GTX280
C	C	1.0	1.0
N	C	2.5 ± 0.25	4.0 ± 0.25
C	N	3.5 ± 0.25	5.5 ± 0.5
N	N	5.0 ± 0.5	9.0 ± 1.0

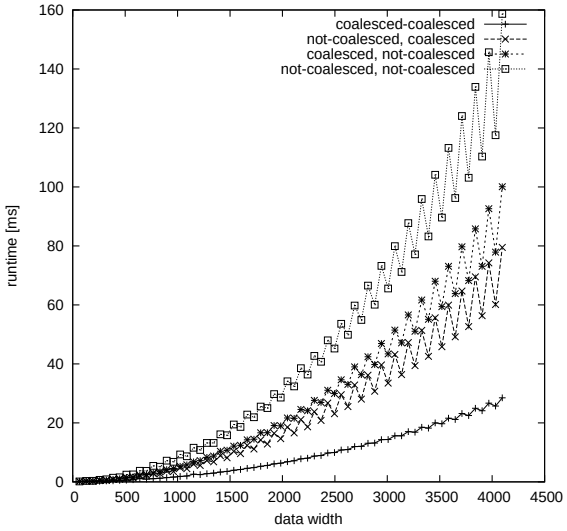
Table 4.1: Approximate slowdown due to non coalesced data access.

the Geforce GTX280 they occur every 256 elements and are less pronounced. These peaks are caused by partition camping. We will discuss partition camping in section 4.3.2 where its effects are more pronounced.

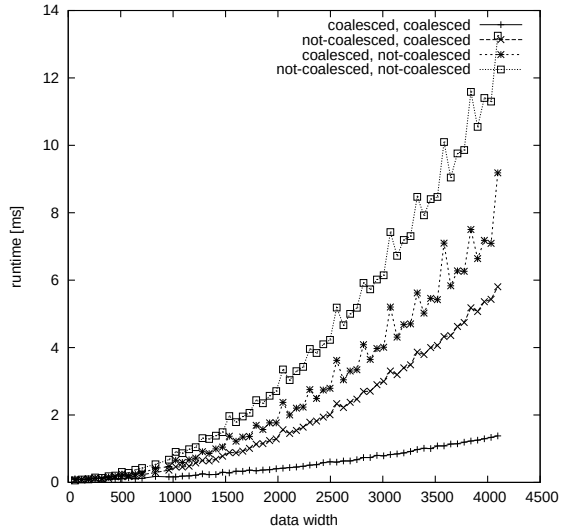
It should be clear that introducing non-coalesced reading and writing drastically increases the runtime of a kernel. Table 4.2 shows approximate slowdown factors for the different kernels. In this table C stands for "coalesced" and N for "not coalesced".

Finally, running the crossed data kernel on both the Geforce 8500GT and the Geforce GTX280 confirmed that their memory model corresponds to the announced compute capability of the GPU i.e. the access was not coalesced for the former, while it was for the latter. However, when we ran this kernel on the Geforce 9400M - a GPU with announced compute capability 1.1 - the access appeared coalesced. This suggests that the memory model of this GPU follows the one for GPUs with compute capability 1.2 or 1.3.

Shared Memory The approximate slowdown factors for the different kernels are shown in table 4.2. In this table U stands for "unconflicted" i.e. with no bank conflicts, while C stands for "conflicted" i.e. with maximal bank conflicts. It is interesting to see that these factors appear to be half as big as one would expect (When bankconflicts are maximal, 16 threads access the same bank).



(a) Geforce 8500GT



(b) Geforce GTX280

Figure 4.4: Runtimes of the Global Data Kernel

Read	Write	Geforce 8500GT	Geforce GTX280
U	U	1	1
C	U	4.15 ± 0.05	4.25 ± 0.05
U	C	4.00 ± 0.05	4.25 ± 0.05
C	C	7.60 ± 0.05	7.55 ± 0.05

Table 4.2: Approximate slowdown due to bank conflicts

4.3 Remedies

The impact of the data access pattern can be minimized by taking into account some simple rules of thumb:

1. Access to global memory should be coalesced if possible.
2. Access to global memory should be minimized through the use of shared memory.
3. Access to shared memory should be organized such that bank conflicts are minimized.

We will give two examples of how these rules of thumb can be put in practice. First we consider memory problems experienced by code that processes an array of structures. Second, we show how our rules of thumb improve the performance of a matrix transposition algorithm.

4.3.1 Array of Structures

Sometimes it is advantageous to restructure the data that is to be processed in order to enforce coalesced data access. Consider for example a program that needs to process an array of structures that are defined as follows:

```
1 typedef struct {
2     int a;
3     int b;
4     int c;
5 } AOS_T;
```

Given the size of one AOS_T structure - 12 bytes - it will never be possible to read an array of these structures in a coalesced fashion. Therefore reordering the data to be a structure of arrays rather than an array of structures will dramatically improve the performance. For example[3]

```
1 typedef struct {
2     int *ap;
3     int *bp;
4     int *cp;
5 } SOA_T;
```

Table 4.3 compares the runtimes of a kernel using an array of 8.388.608 structures and another one using a structure of 3 arrays each containing 8.388.608 elements. As can be seen in the table the difference is greater for the GPU of the lower compute capability because of the more stringent conditions for coalesced data access.

[3]Note that in the example we are using pointers rather than arrays for greater generality.

GPU	Array of Structures	Structure of Arrays
Geforce 8500GT	182 ms	31 ms
Geforce GTX280	5 ms	1.8 ms

Table 4.3: Array of Structure versus Structure of Arrays

Listing 4.3: First Transpose Kernel

```
1  __global__ void transpose(float *in, float *out, int width)
2  {
3      int inRow = BLOCK_WIDTH * blockIdx.y + threadIdx.y;
4      int inCol = BLOCK_WIDTH * blockIdx.x + threadIdx.x;
5
6      if (inRow < width && inCol < width) {
7          out[inCol * width + inRow] = in[inRow * width + inCol];
8      }
9
10     return;
11 }
```

4.3.2 Matrix Transpose

The transpose of a $m \times n$ matrix A, is a $n \times m$ matrix A^T obtained by exchanging the rows and colums of A. For example:

$$\begin{bmatrix} 1 & 2 \\ 3 & 4 \end{bmatrix}^T = \begin{bmatrix} 1 & 3 \\ 2 & 4 \end{bmatrix}$$

The construction of a transpose kernel is an instructive example of how the rules of thumb can be applied to improve the parallel performance.

The first kernel is shown in listing 4.3. Each thread reads the element at position (row, col) of the input matrix and writes it to position (col, row) of the output matrix. row and col are determined by the position of the thread in it's threadblock and of the threadblock in the grid. The memory access of this kernel is very bad. The addresses in global memory written by adjacent threads in a half-warp lie $4 \times width$ bytes apart[4]. Global data is not written in a coalesced manner.

The kernel shown in listing 4.4 ensures all global memory access is coalesced through the use of shared memory. First, data is read from global memory and written to shared memory. When this is done, the shared memory corresponding to threadblock (i, j) contains the data of the input's submatrix (i, j). The data is then read from shared memory and written to the output. To ensure a proper transposition thread (y, x)[5] of threadblock (i, j) will write the element at position (x, y) of shared memory to position (y, x) of the output's submatrix (j, i).

[4]The matrix is stored in row major-order in the GPU's global memory.

[5]y corresponds to the row and x to the column of the thread in in the threadblock.

Listing 4.4: Second Transpose Kernel

```
__global__ void transpose(float *in, float *out, int width)
{
    __shared__ float shared[BLOCK_WIDTH][BLOCK_WIDTH];

    int inRow = BLOCK_WIDTH * blockIdx.y + threadIdx.y;
    int inCol = BLOCK_WIDTH * blockIdx.x + threadIdx.x;

    if (inRow < width && inCol < width) {
        shared[threadIdx.y][threadIdx.x] = in[inRow * width + inCol];
    }
    __syncthreads();

    int outRow = BLOCK_WIDTH * blockIdx.x + threadIdx.y;
    int outCol = BLOCK_WIDTH * blockIdx.y + threadIdx.x;

    if (outRow < width && outCol < width) {
        out[outRow * width + outCol] = shared[threadIdx.x][threadIdx.y];
    }

    return;
}
```

The previous kernel is still not optimal. During the second phase of the kernel, data is read from shared memory in a way that maximizes bank conflicts: the threads of a half-warp access elements that belong to the same column of the submatrix residing in shared memory. Given the dimensions of these submatrices, these elements lie in the same memory bank (figure 4.5). In this case, it is easy to fix this by defining the submatrix in shared memory to have dimensions 16×17. Now all threads of a half-warp access a different bank as (figure 4.6). It suffices to modify one line in the kernel to achieve this:

```
    __shared__ float shared[BLOCK_WIDTH][BLOCK_WIDTH + 1];
```

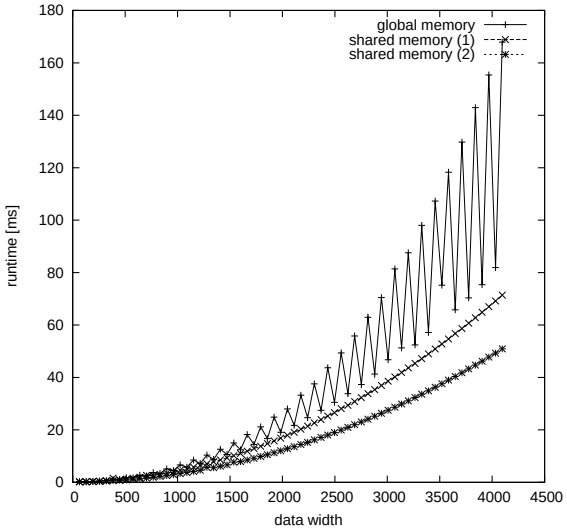
The runtimes of the 3 kernels as a function of the matrix width are shown in figure 4.7(a) for the Geforce 8500GT and in figure 4.7(b) for the Geforce GTX280. The first kernel's runtime exhibits a strongly peaked behaviour for both GPUs. This behaviour is explained by partition camping: the thread blocks that run concurrently on the same SM, write data to blocks in the output matrix that reside in the same block row. For certain matrix widths these blocks reside in the same global data partition, hence causing partition camping. The distance between two peaks can be used to infer the global memory layout. Finally, it should be noted that our last kernel did not provide performance improvements for the Geforce GTX280 GPU. The reason herefore is still unclear.

B0	B1	B2	B3	B4	B5	B6	B7	B8	B9	B10	B11	B12	B13	B14	B15
0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1
2	2	2	2	2	2	2	2	2	2	2	2	2	2	2	2
3	3	3	3	3	3	3	3	3	3	3	3	3	3	3	3
4	4	4	4	4	4	4	4	4	4	4	4	4	4	4	4
5	5	5	5	5	5	5	5	5	5	5	5	5	5	5	5
6	6	6	6	6	6	6	6	6	6	6	6	6	6	6	6
7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7
8	8	8	8	8	8	8	8	8	8	8	8	8	8	8	8
9	9	9	9	9	9	9	9	9	9	9	9	9	9	9	9
10	10	10	10	10	10	10	10	10	10	10	10	10	10	10	10
11	11	11	11	11	11	11	11	11	11	11	11	11	11	11	11
12	12	12	12	12	12	12	12	12	12	12	12	12	12	12	12
13	13	13	13	13	13	13	13	13	13	13	13	13	13	13	13
14	14	14	14	14	14	14	14	14	14	14	14	14	14	14	14
15	15	15	15	15	15	15	15	15	15	15	15	15	15	15	15

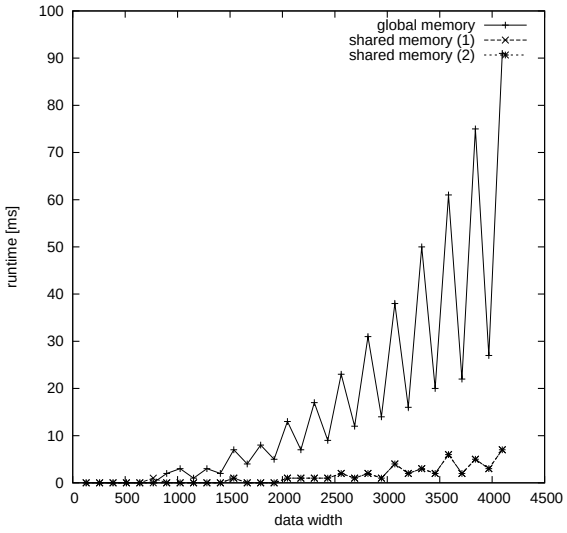
Figure 4.5: Elements accessed by the first half warp for a 16×16 matrix (elements are indicated by their rownumber)

B0	B1	B2	B3	B4	B5	B6	B7	B8	B9	B10	B11	B12	B13	B14	B15
0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
0	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1
1	1	2	2	2	2	2	2	2	2	2	2	2	2	2	2
2	2	2	3	3	3	3	3	3	3	3	3	3	3	3	3
3	3	3	3	4	4	4	4	4	4	4	4	4	4	4	4
4	4	4	4	4	5	5	5	5	5	5	5	5	5	5	5
5	5	5	5	5	5	6	6	6	6	6	6	6	6	6	6
6	6	6	6	6	6	6	7	7	7	7	7	7	7	7	7
7	7	7	7	7	7	7	7	8	8	8	8	8	8	8	8
8	8	8	8	8	8	8	8	8	9	9	9	9	9	9	9
9	9	9	9	9	9	9	9	9	9	10	10	10	10	10	10
10	10	10	10	10	10	10	10	10	10	10	11	11	11	11	11
11	11	11	11	11	11	11	11	11	11	11	11	12	12	12	12
12	12	12	12	12	12	12	12	12	12	12	12	12	13	13	13
13	13	13	13	13	13	13	13	13	13	13	13	13	13	14	14
14	14	14	14	14	14	14	14	14	14	14	14	14	14	14	15
15	15	15	15	15	15	15	15	15	15	15	15	15	15	15	15

Figure 4.6: Elements accessed by the first half warp for a 16×17 matrix (elements are indicated by their rownumber)



(a) Geforce 8500GT



(b) Geforce GTX280

Figure 4.7: Matrix Transpose Runtimes

4.4 Automation

The presence of access to global or shared memory can easily be detected at compile time. Determining the access-patterns that half-warps will follow might be possible in a limited number of cases, but might prove a lot more difficult.

On the other hand, it can be straightforward to instrument the source code such that every data access is written to a trace file. This trace file can be analyzed automatically to detect the presence of less than ideal access patterns. For example, the authors of [8] have done this to analyze the access to shared memory of CUDA applications.

We asked the question wether it would be possible to measure the performance drop of a given pattern in the hardware. Given that scheduling threads is handled by the MTIU, it should be possible for this element to indicate the impossibility to hide the latency of data parallel access.

Chapter 5

Synchronization Points

5.1 Description

Definition

> Synchronization points are points in the program at which processing elements join up to ensure that a part of the work has been done.

Discussion

Most parallel programs contain parts whose sole task is to synchronize the processes or threads in order to coordinate the overall work. Synchronization will typically cause processes to become idle because they have to wait for other processes to finish their work before they can continue. On NVIDIA GPUs there exist 3 manners to achieve synchronization.

Thread block synchronization The *__syncthreads()* operation provides barrier synchronization for the threads of a thread block. No thread can pass the barrier before all threads of the block have reached it.

Global synchronization All threads that run on a GPU are implicitly synchronized when the kernel terminates. That is, to synchronize all threads it is necessary to end the kernel and launch a new one.

Atomic operations CUDA provides a number of *atomic* operations. They allow threads to modify data in global or shared[1] memory in a thread-safe way. Although these operations do not really constitute synchronization points, they are related to synchronization and mentioned for completeness.

[1] Atomic operations on data in shared memory are only possible on GPUs having a compute capability of at least 1.2.

Each of these mechanisms has a negative impact on the performance of a CUDA program. *__syncthreads()* statements will decrease the potential for latency hiding within a thread block. The scheduler cannot run threads from the same block that have already reached the barrier to hide data access latency of threads from the same block that did not yet reach the barrier. Similarly, launching different kernels decreases the potential for latency hiding within and across all thread blocks. Furthermore, because shared memory is not persistent across kernel calls, launching different kernels induces extra data access. Finally, if two or more threads attempt to execute an atomic operation on the same data, they will have to be serialized.

Parameters

We will consider only thread block synchronization via *__syncthreads()* and global synchronization across kernel calls in our benchmark programs. We will study the effect of the following parameters:

- The number of statements between two synchronization points.
- The presence of shared memory for global synchronization.

5.2 Benchmark Algorithm

We define two kernels. The first one investigates the effect of intra-threadblock synchronization, while the second investigates the effect of inter-threadblock synchronization.

The first kernel reads data from global memory, carries out n operations on this data and finally writes the result to global memory. However, every m operations a *__syncthreads()* statements is executed. Both m and n are passed as arguments to the kernel. This kernel measures the impact of thread block synchronization.

The second kernel also reads data from global memory, but only executes m operations on it and writes the result to global memory. There are no *__syncthreads()* statements in this kernel. The kernel is wrapped in a sequence of kernel calls, such that in total n operations are executed. This kernel measures the impact of global synchronization in the absence of shared memory.

Finally, a variant of the second kernel in which data is written to shared memory at the start and back to global memory at the end has also been tested.

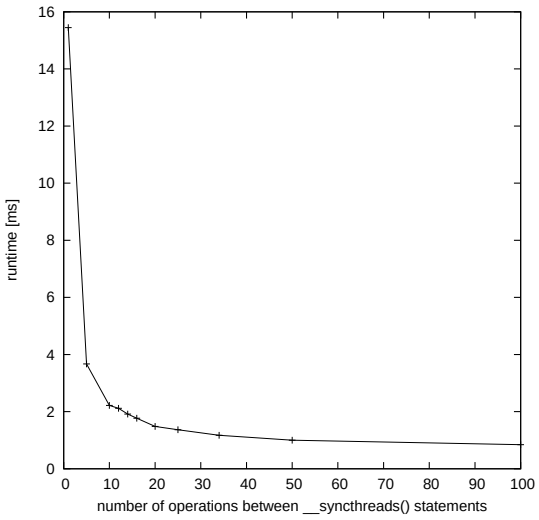


Figure 5.1: Intra Thread Block Synchronization Kernel Runtime

Results

We ran our benchmark programs such that each program executed the same number of operations: $n = 100$ and for a datasize of 8.388.608 elements. By varying m - the number of operations between synchronization points - we naturally vary the number of synchronization points. The number of synchronization points is given by $\lfloor \frac{n}{m} \rfloor$.

The results of running the first kernel on the Geforce GTX280 are shown in figure 5.1. We see that as the number of operations between synchronizations decreases, the total runtime of the kernel increases. This corresponds to our expectations. There is a factor 20 difference between running the kernel with no synchronizations at all, and running the kernel with a synchronization between every two operations.

The results of running the second and the third kernel on the Geforce GTX280 are shown in figure 5.2. The first thing we notice is that the presence of shared memory does not make a very big difference. Second, global synchronization can have a dramatic impact on the performance. Running 100 kernels with one statement is almost a 100 times slower than running 1 kernel with 100 statements. This suggests that both kernels take the same amount of time to complete when run only once. This is quite normal, given that every time a new kernel is launched, data needs to be accessed in global memory.

Running the kernels on the Geforce 8500GT yielded similar results.

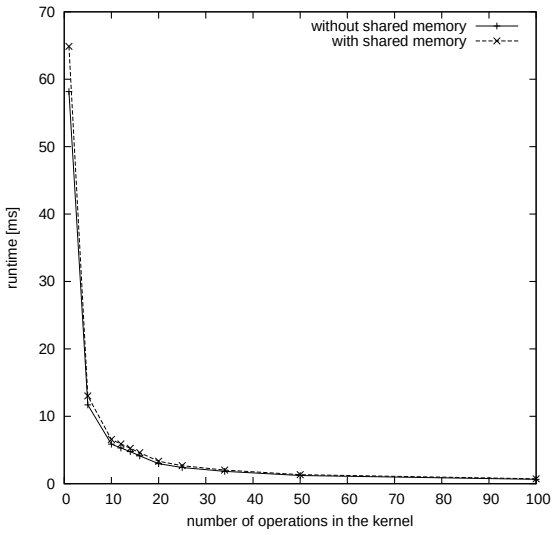


Figure 5.2: Extra Thread Block Synchronization Kernels Runtime

5.3 Remedies

Most of the time synchronization is necessary and cannot be removed from the program. However, sometimes it is possible to reduce the amount of synchronization needed by the program.

- The threads that are part of the same warp execute in lockstep and, are therefore inherently synchronized. It is not necessary to insert explicit *__syncthreads()* statements to synchronize them. An example of this technique can be found in the optimized prefix sum kernel in chapter 7.

- Sometimes it can be more advantageous to calculate the same results several times. Doing so can remove the necessity of synchronization to exchange data.

  For example, in many applications data is typically divided in a number of blocks that themselves are made up of cells[2]. The computation of the value of a cell will often depend on the values of the neighbouring cells. If we structure such an application in a traditional manner, values of outer cells will have to be exchanged between neighbouring threadblocks, thus necessitating global synchronization. In this case it can be interesting to let each thread block compute more data to avoid

[2]This is reminiscent of the way CUDA organizes threads in thread blocks, that themselves are part of a grid. As a consequence this type of applications is ideal to implement in CUDA.

global synchronizations. This technique is illustrated in a CUDA implementation of HotSpot, a program that computes the temperature on processor chips [9].

- There has been some research in how to implement global synchronization in CUDA. The authors of [25] have investigated three approaches to inter-block GPU synchronization. For information about these approaches we refer the interested reader to the paper.

5.4 Automation

Detecting synchronization points in CUDA programs at compile time is straightforward. It is also possible to detect *__syncthread()* statements that are not necessary, for example because they serve to synchronize threads of the same half-warp.

Automatically providing advanced remedies might prove impossible given that they require a restructuring of the application.

We asked the question wether it would be possible to measure the performance drop of a given pattern in hardware. Given that the MTUI that schedules threads must have a way to exclude threads that reached a synchronization barrier from scheduling, we are inclined to believe that it should be possible to measure the performance loss at runtime. Similarly, it should be possible for the hardware to indicate that certain SMs are no longer active in the case of global synchronization.

Chapter 6

Data Partitioning and Mapping

6.1 Description

Definition

> Data partitioning and mapping refers to the way data is mapped to the processors that execute the computation. This pattern is very specific to MIMD architectures and GPUs

Discussion

In data-parallel programs parallelism is applied by performing the same operation on different items of data at the same time. This means that the data is mapped against the processing elements.

In the case of NVIDIA GPUs and CUDA we must assign the data to threads that themselves are organized in thread blocks. There exist different possibilities to perform this mapping. The particular organization that is chosen, will influence the performance of the kernel, hence the reason to include data partitioning and mapping in our collection of anti-parallel patterns.

In CUDA programs data is typically partitioned in blocks that are commonly called tiles. Most of the time a tile is assigned to a thread block and very often one thread will process one element of the tile. However this is not necessarily the case and furthermore different dimensions can be chosen for the thread blocks. In what follows, we have a look at the influence on the performance of the block dimensions and the number of elements processed per thread.

Parameters

The following parameters determine the behaviour of the "data partitioning and mapping" pattern:

- The dimensions of the data being processed. In particular, we will consider 1-dimensional and 2-dimensional data.
- The size of the thread blocks.
- The mapping of data elements to threads i.e. the number of data elements that are processed by each thread.

6.2 Benchmark Algorithm

We developed 3 kernels to measure the impact of the above parameters on the performance of our kernels.

The first kernel operates on 1-dimensional data. The threadblock size and the tile size are equal. The threadblock size - and thus the tile size as well - can be varied. The kernel reads data from global memory, executes a number of operations and writes the result back to global memory. The number of operations is configurable.

The second kernel serves the same purpose, but for 2-dimensional data.

Finally, the third kernel tests the impact of the data to thread mapping. It operates in the same fashion as the previous ones, but one thread might process more than one element. This is done by varying the tile size.

Results

Figure 6.1 shows the influence of the thread block size[1] on the runtime of the 1-dimensional kernel for a total data size of 6.291.456 elements and running on the Geforce GTX280. The peaked behaviour of the runtime can easily be explained. Ideally, the thread block size should be a multiple of 32. In this case all warps will be fully occupied. In the case the thread block size is a multiple of 16, all half-warps will be fully occupied (except the last one that will not be occupied). Finally, if the block size is not a multiple of 16, the occupation will be less than ideal both for thread execution and data access, explaining the peaks in the runtime.

Figure 6.2 shows the influence of the thread block size on the runtime of the 2-dimensional kernel for a total data size of 2048×2048 elements. This curve also exhibits a peaked behaviour that can be explained readily. If we take the thread block width to be 4, the thread block size will only be 16 or a warp (group of 32 threads) will not fit into a thread block. The ideal thread block widths appear to be 8 and 16. However, a thread block size

[1] All the thread block sizes tested were multiples of 4.

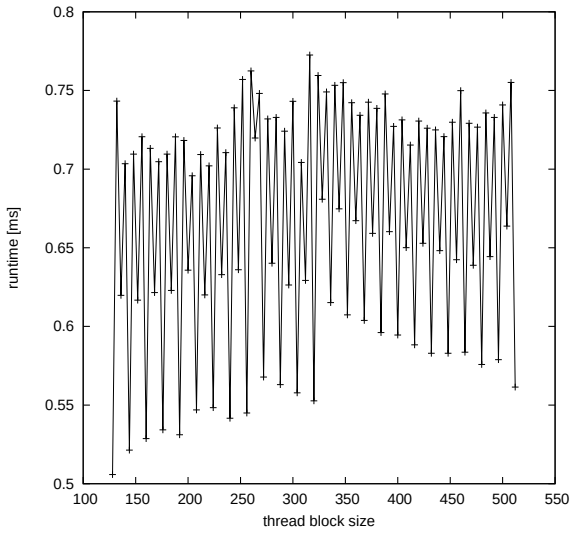


Figure 6.1: Influence of the Thread Block Size on a 1-dimensional Kernel

of 16 is preferable given the greater data structures that can be processed with such a thread block size and the better memory alignment of memory accessed by half-warps.

Finally, figure 6.3 shows the influence of the tile size on the runtime of the third kernel. That is, we have run the kernel for a fixed thread block size 256 but varied the tile size such that one thread may process less or more than one data element. Letting one thread process more than one element seems to be more advantageous for compute intensive programs. This can be explained by considering the difference between a kernel that directs a thread to process more than one element and a traditional kernel. The first kernel can be run with less threadblocks than the second. However, the first kernel exhibits more data access than the second one. When there is enough available computation, the first kernel will be at the advantage.

6.3 Remedies

Determining the ideal thread block size or the number of elements that should be processed by one thread is often a case of trial and error. However, there are a few thumbs of rule and guidelines that can be followed when deciding on the values of these parameters.

- The block size should not be taken too small. Although small blocks allow the GPU to simultaneously run more thread blocks per SM, using too small blocks will make it difficult to process large data structures.
- The block size should not be taken too large. Given that both the

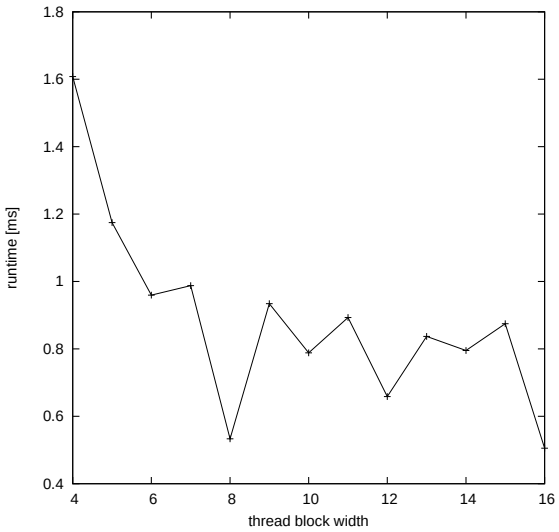


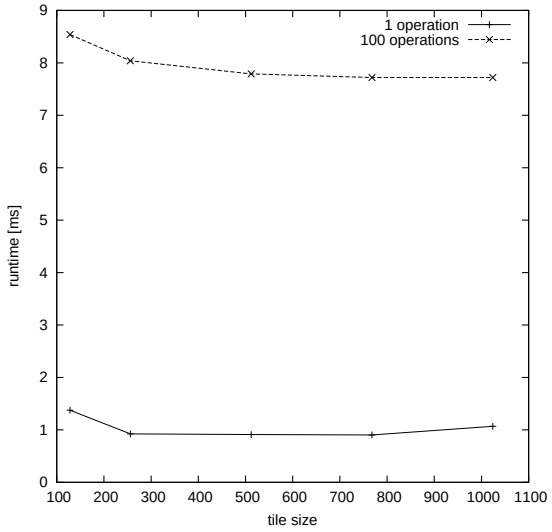
Figure 6.2: Influence of the Thread Block Size on a 2-dimensional Kernel

number of threads and the number of thread blocks that can run simultaneously on an SM are limited, taking a too large block size might limit the potential for concurrency of our kernel. For example, if we take the thread block size to be 512, only one thread block can be run at the same time on any given SM. Typically, the thread block size is taken to be 256. This allows to run 3 thread blocks simultaneously on the same SM for a total of 768 threads (the maximum number of threads that can run simultaneously on one SM.).

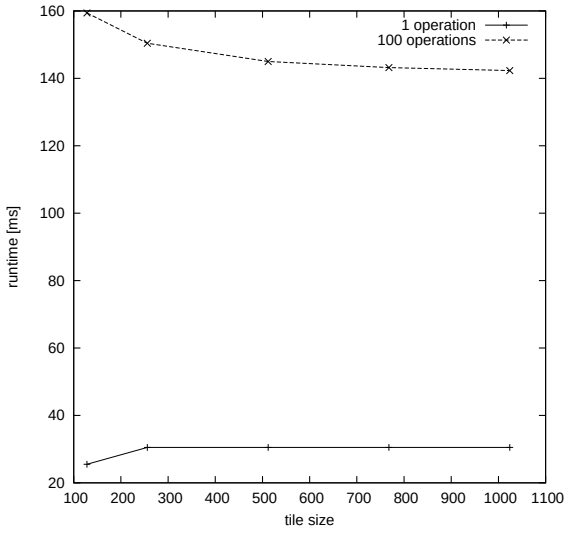
- Choosing the number of elements per thread presents a trade-off. Letting one thread process more than one element often amortizes the cost of calculating indices for 2-dimensional data. However, processing more data per thread block might exhaust an SM's resources - shared memory in particular - thus limiting the amount of concurrency that can be reached. If this is not the case it is sometimes straightforward to calculate the speedup achieved by processing more elements per thread rather than one.

  Consider for example the case where two elements are processed per thread rather than one. If we let t_1 be the time needed by one SM to finish its work in the case a thread is processing one data element and t_2 the time needed by one SM to finish its work in the case a thread is processing two data elements, then the speedup achieved is given by:

$$speedup = \frac{2t_1}{t_2}$$



(a) Geforce GTX280



(b) Geforce 8500GT

Figure 6.3: Influence of the Tile Size on the Runtime

The factor 2 represents the fact that the first kernel needs twice as much thread blocks to handle the same amount of data. From the above equation it is clear that the speedup is greater than one if t_2 is smaller than $2t_1$.

To assist the developer in determining the best threadblock dimensions, NVIDIA provides a tool in the form of an Excel spreadsheet: the occupancy calculator [5].

6.4 Automation

It should be possible to automatically determine the occupation of an SM given a kernel's source code and the execution environment in which it will run. As mentioned earlier NVIDIA already provides an Excel sheet that has a similar function.

Automatically determining the ideal number of elements to be handled by each thread is more intricate. First of all, the source of the kernels will probably be different. Second, it is difficult to estimate upfront how fast a kernel will run. It would be interesting, however, to instrument the source code such that it traces the runtime of every threadblock. This knowledge, together with the knowledge of how many threadblocks need to be run for a given kernel can then be used to determine the ideal configuration.

We asked the question wether it would be possible to measure the performance drop of a given pattern in hardware. For data partitioning and mapping it is not clear how this could be done.

Chapter 7

Case Study: Prefix Sum

This chapter is devoted to a case study: the implementation of a prefix sum algorithm in CUDA. We will use this case to analyze the benefits of the anti-parallel patterns approach.

Section 7.1 explains what a prefix sum is and how it can be computed in parallel using Blelloch's algorithm [7]. In section 7.2, we present a naive implementation of the algorithm. We then show how this kernel can be improved by identifying the anti-parallel patterns it contains and applying the appropriate remedies. At the same time we analyze the performance of the resulting kernels in the context of anti-parallel patterns and compare this analysis with the experimental results obtained by running the kernels on our GPUs.

Much of the inspiration for the CUDA implementation of the prefix sum was taken from chapter 39 of [18].

7.1 Exclusive Prefix Sum

The prefix sum used in this chapter is the exclusive prefix sum. It is defined as follows:

The exclusive prefix sum takes a binary associative operator $\oplus$ with identity I, a set of n elements

$$[a_0, a_1, ..., a_{n-1}]$$

and returns the ordered set

$$[I, a_0, (a_0 \oplus a_1), ..., (a_0 \oplus a_1 ... \oplus a_{n-1})]$$

In what follows, we will limit ourselves to exclusive prefix sums where the set of elements is implemented as an array and where $\oplus$ and I correspond to $+$ and 0. We will use the term "prefix sum" rather than the longer and explicit "exclusive prefix sum".

Example 3 (Sample Prefix Sum)

The prefix sum of the array

$$[0, 1, 2, 3, 4, 5, 6, 7]$$

is the array

$$[0, 0, 1, 3, 6, 10, 15, 21]$$

Implementing a prefix sum for a sequential processor is fairly easy. The following C code is a straightforward implementation:

```
1  void prefixSum(int *in, int *out, int size)
2  {
3      out[0] = 0;
4
5      for (int i = 1; i < size; ++i) {
6          out[i] = out[i-1] + in[i-1];
7      }
8
9      return;
10 }
```

Finding an efficient parallel algorithm is more difficult. We will use Blelloch's algorithm that is especially suited for PRAM architectures [7]. It comprises two phases called the up-sweep and the down-sweep. We shall briefly explain the work done in each phase.

Up-sweep The algorithm presents the elements of the array as the nodes and leaves of a binary balanced tree[1]. During the first step of the up-sweep the elements of the array represent the leaves of the tree. The up-sweep reduces the array by repeatedly performing pairwise sums until the last element of the array - that now represents the root - contains the sum of all elements. Figure 7.1 illustrates the up-sweep phase for the array of our example.

[1]This requires the number of elements to be a power of 2. If this is not the case, the array needs to be padded with zeros.

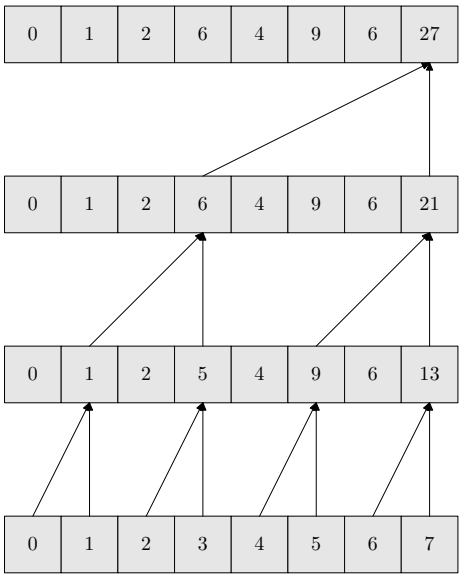


Figure 7.1: Prefix Sum Up-sweep Phase

Down-sweep After the up-sweep the array contains many partial sums over regions of the original array. These partial sums are used to generate the final result. This is done by moving down the tree. First, the identity element, 0 is inserted at the root of the tree. Then, at each step, the involved nodes pass their values to their left child and pass the sum of the old value of their left child and their own value to their right child which is stored at the same position. When the process has completed for all levels of the tree, the array contains the prefix sum of the original array. Figure 7.2 illustrates the down-sweep phase for the array of our example.

A Note on Performance

The prefix sum algorithm is a data intensive algorithm and therefore we express its performance in terms of its data throughput i.e. in GBytes per second. We shall only consider the throughput that is visible to the outside world. In our case, for an array containing n elements of size b bytes[2] and a runtime t seconds, the throughput is given by:

$$throughput = \frac{2 \times b \times n}{t}$$

[2]In our case the elements are integers, thus b is equal to 4.

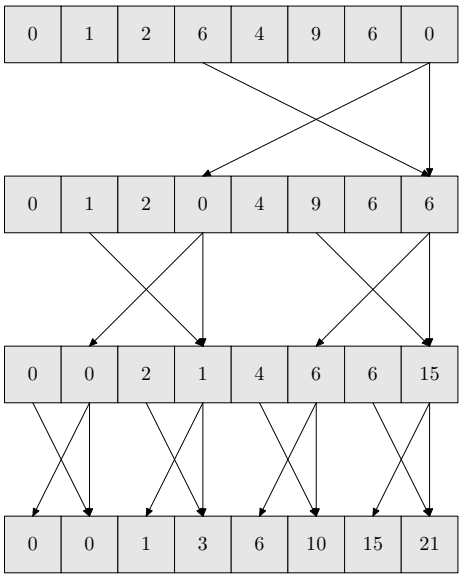


Figure 7.2: Prefix Sum Down-sweep Phase

7.2 Implementation in CUDA

This section desribes the different implementations of the algorithm in CUDA. For each improvement we will explain the anti-parallel pattern it deals with and the remedy that was applied. For each kernel we will try to analyze and estimate its performance in the context of the anti-parallel patterns it contains. We will then compare this with the results obtained by running the kernel on a GPU. Before discussing the actual kernels we will start with a note on the implementation.

A Note on the Implementation

The kernels we present in the following sections perform a prefix sum over a part of the array. More specifically, the threads of the kth thread block only compute the prefix sum of the kth subarray of the input array. That is, when the kernel has run the output array will not yet contain the complete prefix sum. There are a number of good reasons for doing so. First, cooperating threads can easily be synchronized using *__syncthreads()*. Second, shared memory can be used to perform the up-sweep and the down-sweep phase. Finally, the size of the subarrays are equal to the size of the thread blocks[3] which is a power of 2.

The complete prefix sum is calculated as follows: each thread block stores

[3] As it appears, they might also be twice the size of a thread block.

Listing 7.1: Naive Prefix Sum

```
__global__ void prefix(int *deviceIn, int *deviceOut, int *maxVals, int dataSize)
{
    int index = BLOCK_SIZE * blockIdx.x + threadIdx.x;

    // up-sweep
    deviceOut[index] = deviceIn[index];
    for (int d = 2; d <= BLOCK_SIZE; d *= 2) {
        __syncthreads();
        if (threadIdx.x % d == d - 1) {
            deviceOut[index] = deviceOut[index] + deviceOut[index - d/2];
        }
    }

    if (threadIdx.x == BLOCK_SIZE - 1) {
        maxVals[blockIdx.x] = deviceOut[BLOCK_SIZE * blockIdx.x + BLOCK_SIZE - 1];
        deviceOut[BLOCK_SIZE * blockIdx.x + BLOCK_SIZE - 1] = 0;
    }

    // down-sweep
    for (int d = BLOCK_SIZE; d >= 2; d /= 2) {
        __syncthreads();
        if (threadIdx.x % d == d - 1) {
            int temp = deviceOut[index - d/2]; // value left child
            deviceOut[index - d/2] = deviceOut[index];
            deviceOut[index] = deviceOut[index] + temp;
        }
    }

    return;
}
```

the value that is stored at the root of the tree at the corresponding position of an auxiliary array whose size corresponds to the number of subarrays and that resides in global memory. Then, when the prefix sum kernel has completed, a complete prefix sum is applied to the auxiliary array whereafter the resulting values are added to the corresponding subarrays.

7.2.1 A Naive Implementation

We start with a kernel that could have been written by a CUDA neophyte. The thread block size (*BLOCK_SIZE*) is equal to the subarray size (*TILE_SIZE*), thus the number of threads is equal to the number of elements. Every thread computes the sum that has to end up at the position with the same index. All operations are executed in global memory.

This naive kernel is shown in listing 7.1

Performance Analysis

The runtime of the kernel is dominated by data access. For a subarray of size 256, there are 17 steps during which data is read from and written to

GPU	data size	runtime	performance	speedup
Geforce 8500GT	8,388,608	508 ms	0.132 GBps	1.0
Geforce GTX280	16,515,072	46 ms	2.87 GBps	1.0

Table 7.1: Results - Naive Kernel

global memory: 1 at the start, 8 during the up-sweep and 8 during the down-sweep. Because these steps are separated by *__syncthreads()* calls, we can analyze the performance of every step independent from the other steps. We shall do so for the down-sweep and for the Geforce 8500GT.

During an iteration of the down-sweep, each active thread reads and writes the element with the same index. This read and write is coalesced. Furthermore, until the fourth iteration all half-warps contain active threads. Therefore, we expect the contribution of these reads and writes to be at least the time needed to copy an array of the same size in global memory. Every iteration also contains a non-coalesced read/write pair[4]. This complicates our analysis. Our benchmarks have only measured the time needed to perform a complete copy in global memory across the various dimensions of coalescence, not to do partial copies. However, we expect their contribution to be at least as big as the one due to coalesced reading and probably bigger. Therefore, if we estimate the last 4 iterations of the down-sweep to take half as much time as the first 4, we can say the down-sweep will take at least:

$$t_{down} > 4 \times 2 \times t_{RW} + \frac{1}{2} \times 4 \times 2 \times t_{RW}$$

Where t_{RW} stands for the time needed to copy the complete array in global memory. If we take the up-sweep to take as long as the down-sweep, the lower-bound on the runtime is given by:

$$t_{runtime} > 25 \times t_{RW}$$

The experimental results show that we underestimated the runtime by a factor 2. On the one hand, we did not include the cost of the modulo operations in the conditions of the if-statements. On the other hand, our benchmark algorithms for data-access did not help us to determine the time needed for partial uncoalesced access to global memory.

Table 7.1 shows experimental results for both GPUs on which we ran the prefix kernel. The column *speedup* gives the speedup compared to the previous version of the kernel.

[4]At least for GPU devices of compute capability 1.1 such as the Geforce 8500 GT.

GPU	data size	runtime	performance	speedup
Geforce 8500GT	8,388,608	481 ms	0.140 GBps	1.06
Geforce GTX280	16,515,072	46 ms	2.87 GBps	1.00

Table 7.2: Results - Using Shared Memory

7.2.2 Anti-parallel Pattern: (Global) Parallel Data Access

The first anti-parallel pattern we discern is "data access". All operations take place in global memory. Although, most of the data access is coalesced, we will perform both the up-sweep phase and the down-sweep phase in shared memory. To do so we declare an array in shared memory and populate it at the very start of the kernel. We also change all operations to take place in shared memory and we insert a write from shared memory to global memory at the end of the kernel.

Using shared memory has an additional advantage. Contrary to the first kernel, this kernel computes a correct prefix sum for arrays of any size. Indeed, the first kernel only produces a correct result for array sizes that are an integer multiple of the subarray size. Handling other cases would complicate the first kernel too much. In general, handling this is easier when working with shared memory.

Performance Analysis

It is difficult to estimate what we have gained by applying shared memory. We have not taken benchmarks to compare the runtime of kernels that operate in global memory with other kernels that operate in shared memory. However, we expect to gain more on GPU devices of compute capability 1.1 given the more stringent conditions for coalesced data access on these devices.

The results in table 7.2 show a very small performance gain for the Geforce 8500GT. For the Geforce GTX280 we see no performance gain at all. This is also explained by the modulo operations whose presence is used to achieve latency hiding.

7.2.3 Anti-parallel Pattern: Branching

Because each thread processes the elements with the same index and because of the way the computation is organized, the active threads are distributed uniformly across all warps. As a consequence, our kernel suffers a lot from "branching". To remedy this problem we will apply "static thread reordering". We reorganize the code such that the threads that do the work, are the threads that reside in the beginning of each thread block.

GPU	data size	runtime	performance	speedup
Geforce 8500GT	8,388,608	223 ms	0.301 GBps	2.16
Geforce GTX280	16,515,072	19 ms	6.95 GBps	2.42

Table 7.3: Results - Static Thread Reordering

This kernel has an extra advantage: it no longer contains expensive modulo operations.

Performance Analysis

By applying static thread reordering we have decrease the average branching factor per warp from 2 to 1. Furthermore, we got rid of the expensive modulo operations. This leads us to believe that the runtime will decrease by at least a factor of 2. Our belief is confirmed by the measured speedup shown in table 7.3.

7.2.4 Anti-parallel Pattern: Data Partitioning and Mapping

Until now, half of the threads of a thread block only populate shared memory at the beginning and write to global memory at the end; they do not participate in neither the up-sweep nor the down-sweep. As we have discussed in chapter 6, it can sometimes be advantageous to let each thread handle two data elements. This is such a case.

The change is implemented by letting each thread load 2 elements in shared memory and write 2 elements to global memory. The execution environment in which the kernel is launched is modified accordingly.

Performance Analysis

Letting each thread handle two elements changes a few things. First, the loading and unloading of shared memory by a thread block will take twice as long. Second, both the up-sweep and the down-sweep perform an extra iteration. Third, we need half the number of thread blocks to process the same amount of data.

The first and the second change cause a thread block to run longer. The third change causes the GPU to run only half as much thread blocks. If we take the time to load and unload shared memory to be t_1, the time to run the up-sweep and the down-sweep to be t_2 and the extra time to run the extra iteration for both the up-sweep and the down-sweep to be t_δ, the expected speedup is given by:

$$speedup = \frac{t1 + t2}{\frac{1}{2}(2t_1 + t_2 + t_\delta)}$$

GPU	data size	runtime	performance	speedup
Geforce 8500GT	8,388,608	164 ms	0.409GBps	1.36
Geforce GTX280	16,515,072	14 ms	9.44GBps	1.36

Table 7.4: Results - Data Mapping

It is easy to see that the speedup will be larger than 1 if $t_\delta < t_2$. Here, we expect this to be the case. Our measurements shown in table 7.4 confirm our expectations.

7.2.5 Anti-parallel Pattern: (Shared) Parallel Data Access

Finally, the access to shared memory exhibits a great deal of bank conflicts. As we have seen in chapter 4, it is sometimes possible to remove bank conflicts by padding the data structure that resides in shared memory. The solution proposed in [18] is to introduce en empty position between every 16 elements of the array in shared memory. In practice, the size of the array in shared memory is increased by 31 (the *TILE_SIZE* is 512, thus there are 32 times 16 elements in an array in shared memory) and the real index of an element is determined by a macro that performs the following computation:

$$index_{real} = index + \frac{index}{16}$$

The kernel resulting from all our optimizations is shown in listing 7.2. Note that in this kernel multiplications and divisions by 2 are replaced by a shift left and a shift right respectively. Measurements have pointed out that this is not done by the compiler.

Performance Analysis

The performance gain won by removing bank conflicts will be largest for the iterations that act upon the upper part of the tree. These are also the iterations in which least work is done. We will not try to estimate what was gained, but inmediately look at the experimental results.

Table 7.5 shows the measurements for the kernel where we applied the removal of bank conflicts and the replacement of multiplication operations by shift operations. Therefore, the speedup corresponds to the speedup in comparison with the previous kernel using shift operations.

Listing 7.2: Optimal Prefix Sum

```
#define INDEX(i) ((i) + ((i) >> 4))

__global__ void prefix(int *deviceIn, int *deviceOut, int *maxVals, int dataSize)
{
    int index = TILE_SIZE * blockIdx.x + threadIdx.x;

    __shared__ int shared[TILE_SIZE + 31];
    shared[INDEX(threadIdx.x)] = index < dataSize ? deviceIn[index] : 0;
    shared[INDEX(threadIdx.x + BLOCK_SIZE)] = index + BLOCK_SIZE < dataSize ?
         deviceIn[index + BLOCK_SIZE] : 0;

    int delta = 1;
    int increment = threadIdx.x + 1;
    int idx = threadIdx.x;

    // up-sweep
    for ( ; delta < TILE_SIZE; delta <<= 1, increment <<= 1 ) {
        __syncthreads();
        idx += increment;
        if (idx < TILE_SIZE) {
            shared[INDEX(idx)] += shared[INDEX(idx - delta)];
        }
    }

    if (threadIdx.x == 0) {
        maxVals[blockIdx.x] = shared[INDEX(TILE_SIZE - 1)];
        shared[INDEX(TILE_SIZE - 1)] = 0;
    }

    // down-sweep
    for (delta >>= 1, increment >>= 1; delta > 0; delta >>= 1, increment >>= 1) {
        __syncthreads();
        if (idx < TILE_SIZE) {
            int tmp = shared[INDEX(idx - delta)];
            shared[INDEX(idx - delta)] = shared[INDEX(idx)];
            shared[INDEX(idx)] += tmp;
        }
        idx -= increment;
    }
    __syncthreads();

    if (index < dataSize) deviceOut[index] = shared[INDEX(threadIdx.x)];
    if (index + BLOCK_SIZE < dataSize) deviceOut[index + BLOCK_SIZE] = shared[INDEX(
         threadIdx.x + BLOCK_SIZE)];

    return;
}
```

GPU	data size	runtime	performance	speedup
Geforce 8500GT	8,388,608	128 ms	0.524 GBps	1.16
Geforce GTX280	16,515,072	11.5 ms	11.5 GBps	1.09

Table 7.5: Results - Removing Bank Conflicts

Chapter 8

Conclusion and Future Work

This thesis presented a novel approach for improving the efficient usage of parallel processing power in the context of fine-grain data-parallel programs. Herefore, it introduced the concept of "anti-parallel patterns" - patterns that represent parts of parallel programs that cause their performance to be less than expected. We elaborated this concept for a number of patterns typically found in data-parallel programs.

Most performance analysis approaches focus on the detection of bottlenecks. We proposed a complementary approach in which we focused on the patterns of a parallel program causing a performance degradation. In this work we investigated wether a structured, systematic performance analysis based on anti-parallel patterns can help the developer. Furthermore, the idea of anti-parallel patterns provides an umbrella concept that enables us to group previous work which is now scattered.

For each pattern we gave a definition and discussed its effect on the parallel platform that was used for this thesis: NVIDIA GPUs. We created a number of benchmark programs for each anti-parallel pattern and used its results to model the behaviour of the anti-parallel pattern and provide heuristics to the programmer to determine the impact of a pattern on her program's performance. Finally, we provided remedies to minimize the performance loss caused by the patterns.

This work tried to make an analysis of the most important aspects of anti-parallel patterns without going into detail at each point. In the text and in the rest of this chapter, we indicated the places where there exists opportunity for further work.

8.1 Goals Evaluation

We set four different goals for anti-parallel patterns. The following paragraphs discuss to which extent these goals were met. Throughout our discussion, we will give (partial) answers to the scientific questions that were

raised in the introduction. We will refer to the scientific questions by their Roman numerals. However, for the reader's sake we repeat them here.

I. Does there exist an exhaustive list of patterns qualitatively explaining all performance degradations? If so, to what extent is the decomposition of a program into its anti-parallel patterns beneficial for performance analysis?

II. To what extent does the decomposition of a program into its anti-parallel patterns result into a decomposition of the performance? In this context we can ask the following subquestion:

   (a) Can the effect on the performance of a single pattern be described by a relatively simple model?

III. Can the patterns be identified automatically by a tool, (a) at compile or (b) at runtime?

IV. Can remedies be categorized according to the patterns?

V. Is it possible to automatically derive wether remedies apply for a given program?

VI. Are anti-parallel patterns useful in characterizing the performance of a given parallel platform. If so, is such a characterization complete?

VII. Can the performance drop caused by an anti-parallel pattern be measured at runtime by the hardware?

Before we discuss to which extent each goal of the anti-parallel patterns was met, we start with a summary overview of the answers to the scientific questions for each pattern in table 8.1. Note that in this table we only give an answer to questions that can be asked for a single pattern. These answers should be understood as follows:

answer	meaning
++	very positive
+	positive but uncomplete
-	negative (only positive in a limited number of cases)
+-	in some cases positive, in others negative
?	unknown

Although these answers are by no means definitive, they reveal a number

Patterns	II	II (a)	III (a)	III (b)	V	VII
Branching	+	++	+	++	+-	++
Parallel Data Access	+-	+	+-	++	-	++
Synchronization Points	+-	+	++	++	-	+
Data Partitioning and Mapping	?	?	++	?	+	?

Table 8.1: Summary Analysis of the 4 Anti-parallel Patterns.

of patterns. They show a clear difference between what is possible and where the challenges lie. The table shows that patterns can clearly be recognized (columns III (a) and III (b)). Further, it should be possible for most patterns to let their performance impact be measured by the hardware (column VII). Finally, column V - the possibility to automatically propose appropriate remedies - suggests that there is still work remaining for the programmer.

Model Performance Loss

We modeled the performance loss caused by an anti-parallel pattern using the results from our benchmark programs. This exercise served as a partial answer to scientific question II (a). The behaviour of some patterns - for example branching - is almost deterministic, allowing us to create a simple model. For other patterns - for example parallel data access - the effect on the performance of a CUDA program depends on many factors like the amount of computation that is available in the kernel. Consequently it is harder to capture the behaviour of such a pattern in a simple model.

Evaluate and Estimate Parallel Performance

We tried to understand the performance of a parallel program running on a GPU using anti-parallel patterns. Using a prefix sum as a case study, we proved that anti-parallel patterns are helpful in analyzing the performance of a CUDA program, given that one can identify and locate the anti-parallel patterns it contains. The study showed decomposing a program into its anti-parallel patterns is beneficial for performance analysis (scientific question I).We felt however that some patterns were still missing, like the cost of expensive operations. Also, we believed some models proved too simple for accurate performance estimation. Consequently, we think it is difficult to answer scientific question II. However, if we expand our collection of anti-parallel patterns and provide better models, it might be possible to obtain good estimations for a great class of programs.

Other architectures than GPUs might have a less deterministic behaviour. Therefore, it might prove difficult to use anti-parallel patterns for accurate

performance estimation.

In this work we also reasoned about the possibility of detecting the pattern at compile time or at runtime and about the possibility of letting the hardware measure the performance drop at runtime. A summary answer to this question for each pattern is shown in table 8.1.

Improve Parallel Performance

We improved the performance of the parallel prefix sum implementation by identifying the anti-parallel patterns it contains and applying the appropriate remedies. Doing so, we made the program run four times faster.

This was a relatively easy task because the remedies were categorized according to the anti-parallel patterns whose effect they try to alleviate (scientific question IV). It sufficed to recognize the pattern and apply the appropriate remedy.

Throughout this work we hinted at the possibility of finding remedies automatically once the pattern is detected and the performance drop understood. An idea about the possibility to do this for each pattern is given in table 8.1.

Compare Architectures and Platforms

We used anti-parallel patterns and the benchmark programs written for them to compare two different NVIDIA GPUs: the Geforce 8500GT and the Geforce GTX280. Doing so gave us an idea of the performance of each GPU and of their behaviour in the face of an anti-parallel pattern. The benchmarks written for the parallel data access pattern clearly exhibited the different memory architecture of both GPUs. Furthermore, the benchmarks revealed information that could not be extracted from the vendor's manuals[1]. This allows us to partially answer scientific question VI. Anti-parallel patterns are useful in characterizing the performance of a given parallel platform, however it is not yet clear wether this characterization is complete.

8.2 General Conclusion

We conclude that the anti-parallel patterns concept is a promising method that complements other methods that assist the developer in analyzing, understanding and improving the performance of her parallel programs.

Ofcourse, the anti-parallel patterns concepts is no silver bullet. Rather, it provides a structured way to look at a parallel program's performance

[1] In particular we showed that the memory of the NVIDIA Geforce 9400M, a GPU with compute capability 1.1, follows the memory model of GPUs with compute capability 1.2 or 1.3.

and that is complementary to other ways to analyze, estimate and improve performance.

Bibliography

[1] http://www.paradyn.org/.

[2] https://computing.llnl.gov/tutorials/mpi/.

[3] https://computing.llnl.gov/tutorials/pthreads/.

[4] https://computing.llnl.gov/tutorials/openMP/.

[5] http://developer.download.nvidia.com/compute/cuda/3_1/sdk/docs/CUDA_Occupancy_calculator.xls.

[6] Gene M. Amdahl. Validity of the single processor approach to achieving large scale computing capabilities. In *AFIPS '67 (Spring): Proceedings of the April 18-20, 1967, spring joint computer conference*, pages 483–485, New York, NY, USA, 1967. ACM.

[7] Guy E. Blelloch. Prefix sums and their applications. Technical report, Synthesis of Parallel Algorithms, 1990.

[8] M. Boyer, K. Skadron, and W. Weimer. Automated Dynamic Analysis of CUDA Programs. In *Third Workshop on Software Tools for MultiCore Systems*, 2008.

[9] 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.

[10] Mark E. Crovella and Thomas J. LeBlanc. Parallel performance prediction using lost cycles analysis. In *Proceedings of Supercomputing '94*, pages 600–609, 1994.

[11] A. Espinosa, F. Parcerisa, Tomàs Margalef, and Emilio Luque. Relating the execution behaviour with the structure of the application. In *Proceedings of the 6th European PVM/MPI Users' Group Meeting on Recent Advances in Parallel Virtual Machine and Message Passing Interface*, pages 91–98, London, UK, 1999. Springer-Verlag.

[12] Michael J. Flynn. Some computer organizations and their effectiveness. *IEEE Trans. Comput.*, pages 948–960, 1972.

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

[14] Ananth Grama, George Karypis, Vipin Kumar, and Anshul Gupta. *Introduction to Parallel Computing (2nd Edition)*. Addison Wesley, 2 edition, 2003.

[15] W. Daniel Hillis and Guy L. Steele, Jr. Data parallel algorithms. *Commun. ACM*, 29(12):1170–1183, 1986.

[16] Calvin Lin and Larry Snyder. *Principles of Parallel Computing*. Addison Wesley, 1 edition, 2008.

[17] David Luebke and Greg Humphreys. How gpus work. *Computer*, 40(2):96–100, 2007.

[18] Hubert Nguyen. *GPU gems 3*. Addison-Wesley Professional, 2007.

[19] John Nickolls, Ian Buck, Michael Garland, and Kevin Skadron. Scalable parallel programming with cuda. *Queue*, 6(2):40–53, 2008.

[20] NVIDIA. *NVIDIA CUDA C Programming Best Practices Guide*. 2009.

[21] NVIDIA. *NVIDIA CUDA Programming Guide 2.3.1*. 2009.

[22] Greg Ruetch and Paulius Micikevicius. *Optimizing Matrix Transpose in CUDA*. 2009.

[23] S.W. Smith. The scientist and engineer's guide to digital signal processing. In *California Technical Publishing*, 1997.

[24] Allan Snavely, Nicole Wolter, and Laura Carrington. Modeling application performance by convolving machine signatures with application profiles, 2001.

[25] Shucai Xiao and Wu-chun Feng. Inter-Block GPU Communication via Fast Barrier Synchronization. In *Proceedings of the 24th IEEE International Parallel and Distributed Processing Symposium (IPDPS)*, Atlanta, Georgia, USA, April 2010.

[26] Eddy Z. Zhang, Yunlian Jiang, Ziyu Guo, and Xipeng Shen. Streamlining gpu applications on the fly: thread divergence elimination through runtime thread-data remapping. In *ICS '10: Proceedings of the 24th*

ACM International Conference on Supercomputing, pages 115–126, New York, NY, USA, 2010. ACM.