Bachelor-Thesis

Efficient Execution of Synchronous Guarded Actions using CUDA

submitted by

Thorsten Ropertz

31. Mai 2010

University of Kaiserslautern
Department of Computer Science
Embedded Systems Group

supervisor: Prof. Dr. Klaus Schneider
tutor: Mike Gemünde

Zusammenfassung

Heutige Prozessoren erzielen ihren Leistungsgewinn durch die Einführung zusätzlicher Funktionseinheiten und das Auschöpfen von Parallelismus, anstatt der weiteren Erhöhung der Taktraten. Moderne Grafikkarten bestehen aus hunderten Prozessoren, die für verschiedene Berechnungen herangezogen werden können. CUDA stellt eine API zur Verfügung, die die Benutzung der stark parallelisierten Architektur ermöglicht. Diese Arbeit beschreibt die Synthese von synchronen, bedingten Aktionen zu CUDA Programmen, sodass die generierten Programme die Parallelität der Grafikkarten möglichst gut ausnutzen können. Dazu werden die allgemeine NVIDIA GPU Architektur und das Averest Intermediate Format, welches die synchronen, bedingten Aktionen bereitstellt, illustriert. Um die synchronen, bedingten Aktionen effizient auszuführen, wird ein einfacher Vektorisierungsalgorithmus eingeführt und die beschränkenden Faktoren der Grafikkarte aufgezeigt. Des Weiteren wird ein allgemeiner Ausblick auf die mögliche Leistung von optimierten CUDA Programmen, die synchrone, bedingte Aktionen ausführen, gegeben.

Abstract

Todays processors gain performance by introducing additional functional units and using the parallelism instead of just increasing the clock rates. Modern graphics devices come up with hundreds of cores for general purpose computations. CUDA provides an API for using this highly parallel architecture. This work describes the synthesis of synchronous guarded actions to CUDA programs such that the generated CUDA program can make use of the massive parallelism. Therefore, the general NVIDIA GPU architecture is illustrated as well as the description of the Averest Intermediate Format which contains the synchronous guarded actions. In order to execute the synchronous guarded actions efficiently, a simple vectorization algorithm is introduced and the limiting factors of the graphics device are emphasized. Additionally, a general perspective to the performance of optimized CUDA programs which execute synchronous guarded actions is given.

Eidesstattliche Erklärung

Hiermit versichere ich, die vorliegende Bachelorarbeit selbständig verfasst und keine anderen als die angegebenen Quellen und Hilfsmittel verwendet zu haben. Alle wörtlich oder sinngemäß übernommenen Zitate sind als solche kenntlich gemacht worden. Diese Arbeit hat in gleicher oder ähnlicher Form noch keiner Prüfungsbehörde vorgelegen.

Kaiserslautern, den 31. Mai 2010

Thorsten Ropertz

Contents

1 Introduction

1.1 Motivation

Synchronous Guarded Actions can be used to model synchronous systems. So far, it is possible to synthesize sequential C code out of this model. Synchronous Guarded Actions are able to model different forms of concurrency, but this ability is not used by the sequential code generation. Newly developed computer architectures use parallelism in order to speed up the computation instead of using higher clock-rates. The newer graphics device generations carry this architecture to extremes and introduce hundreds of processing units. The introduction of the *General Purpose Graphics Processing Unit* (GPGPU) enables programmers to use the massive parallelism for general purpose. GPGPUs have lower clock-rates than CPUs but due to the high amount of cores, the throughput is much higher than the throughput of CPUs. Therefore, the performance of programs generated from synchronous Guarded Actions can be increased by using graphics devices for computation. The CUDA-API is a well documented application programming interface for GPGPU programming invented by NVIDIA which does a lot of research in the graphics device computing topic. It is tailored to the NVIDIA GPU architecture which is designed for efficient general purpose computation. Thus, CUDA is used to deal with the efficient execution of Guarded Actions on graphics devices.

1.2 Related Work

Baudisch, Brandt and Schneider present in [2] techniques to extract independent threads for OpenMP in order to generate multithreaded code from synchronous programs. In particular, they introduce graph algorithms to extract concurrency from the dependency graphs and explain how to make the parallelism usable for thread generation. Baskaran, Ramanujam and Sadayappan describe in [1] an automatic code transformation system that generates parallel CUDA code from C code, for affine programs. They optimize their generated code for efficient data access and gain a quiet good performance which touches the performance of manually optimized CUDA programs. Stratton, Stone and Hwu describe in [7] the other way around. They explain how to execute CUDA programs efficiently on shared memory, multicore CPUs. Ishizaki and Komatsu show in [3] a loop vectorization algorithm, which

formalizes and unifies the detection of vector prefetch communication and vector pipeline communication for loop parallelization.

1.3 Organization

In Chapter 2 the basics of the CUDA programming model and the GPU architecture caused limitations are described. In particular, the different performance issues and the need of vectorizing the synchronous Guarded Actions are illustrated. Afterwards, synchronous Guarded Actions are described as well as the AIF structure which contain them.
Then, the transformations that have to be applied to the AIF system in order to reduce the synthesis effort are mentioned in Chapter 3. These transformations additionally assure that the concurrency can be extracted and the synchronous Guarded Actions can be vectorized.
In Chapter 4 the vectorization is described which leads to SIMT code. Therefore, the synchronous Guarded Actions have to be scheduled which implies the identification of dependencies and the breaking of dependency cycles. Afterwards, they can be vectorized by building sets of synchronous Guarded Actions that perform the same operations independently.
The different types of memories and their usage is described in Chapter 5. In order to execute synchronous Guarded Actions efficiently on graphics devices, the different memory latencies and their properties in terms of parallel accesses have to be considered due to the fact that the memory bandwidth is the bottleneck of modern graphics devices. Different techniques are used to hide this limitations which are described in this chapter.
Chapter 6 describes the code synthesis. Therefore, the kernel functions and the host function as well as their structures are illustrated. The kernel functions have the same structure. Thus, only one kernel is described in detail.
Finally, the generated CUDA program is compared to the generated C program and to a manually optimized CUDA program which uses synchronous Guarded Actions. In particular, the performance and the program size is compared. Additionally, the CUDA Visual Profiler is used to analyze the GPU occupancy and the usage of the GPU.

2 Preliminaries

2.1 CUDA

The Compute Unified Device Architecture (CUDA) is a scalable parallel programming model which has been invented by NVIDIA in order to enable programmers to use the massive parallelism and computational power of modern graphics processing units (GPUs) for non-graphics computing.

The following description is based on [4]. In principle, CUDA consists of a runtime environment and a small set of extensions to the C programming language. But there are also wrappers for some other programming languages like java or python.
CUDA is supported by NVIDIA GPUs beginning with the GeForce 8 series. Different GPU generations can have different compute capabilities. The compute capability is identified by a major revision number and a minor revision number. The major revision number specifies the core architecture and the minor revision number corresponds to incremental improvements to the core architecture. Higher compute capabilities may also introduce new functions, e.g. the warp vote function, which evaluates a given predicate for all threads of a warp and returns true if the predicate evaluates to true for all threads of a warp, is only supported by devices of compute capability 1.2 and higher.
A CUDA program consists of two parts: On the one hand, there is the device code, which is compiled with the CUDA Compiler Driver NVCC and is executed by the graphics card. On the other hand, there is the host code, which is compiled with a standard C/C++ compiler and executed by the CPU. Both parts can be combined in a *.cu file and the separation is done by the NVCC. The device code consists of special functions, which are called kernel.

2.1.1 GPU Core

The following describes the architecture of the NVIDIA GeForce 280 GTX GPU because this is the GPU this work is mainly based on.
CPUs use single instruction, multiple data (SIMD) units for vector processing. So the CPU is able to apply a single instruction to a bunch of data synchronously. In contrast, GPUs use an architecture called single instruction, multiple thread (SIMT) for scalar thread processing. This means that there are several program executions called threads that are running in parallel executing the same code on possibly different data. So the programmer does not have to organize the data into vectors and threads are free to branch separately. Each thread has its own ID which can be used to address different data. The ID can be accessed through the three-dimensional build-in variable threadIdx. Threads are grouped in three-dimensional blocks which also have their own ID, accessed through the three-dimensional build-in variable blockIdx. Blocks are grouped in a two dimensional grid. An overview is given in Figure 2.1.

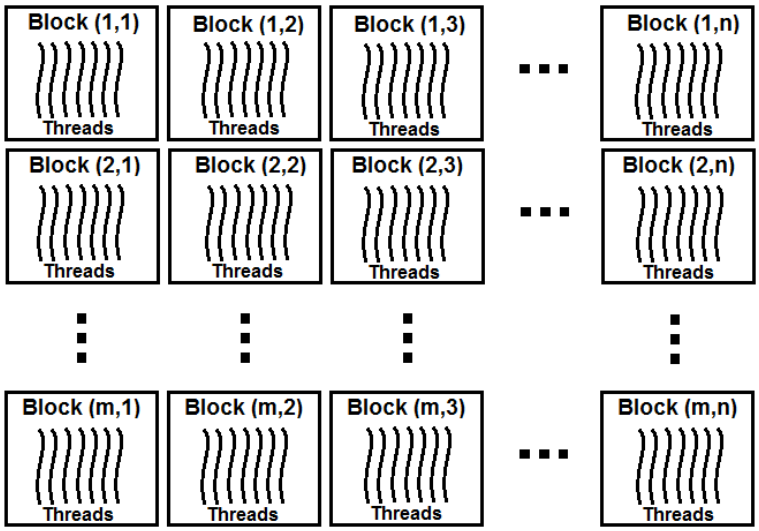

Figure 2.1: Organization of Threads in Blocks and Blocks in a Grid

The grid size and the number and arrangement of threads per block are specified by the kernel call, e.g.

$$myKernel <<< dim3(4,3,1), dim3(45,7,18) >>> (kernelParams);$$

defines a kernel with the name myKernel. dim3(4,3,1) specifies the two grid dimensions to 4 and 3, the third number has to be 1 because only two dimensional grids are supported. dim3(45,7,18) specifies the three dimensions of threads per block and kernelParams represents the parameters of the kernel function.

A kernel launch is asynchronous and the CPU can execute some other code and synchronize to the device by a synchronous data transfer, by another kernel launch or the *cudaThreadSynchronize* function. A kernel can consist of up to 2 000 000 instructions, but unfortunately, the whole execution time is limited to 5 sec.
Threads are executed on Scalar Processors (SPs), where 8 of them form a Streaming Multiprocessors (SMs). The NVIDIA GeForce 280 GTX GPU contains 30 SMs, which is shown in Figure 2.2. Each block is assigned to one SM, because the content of the shared memory has to be copied otherwise. For a detailed description see Section 2.1.2.

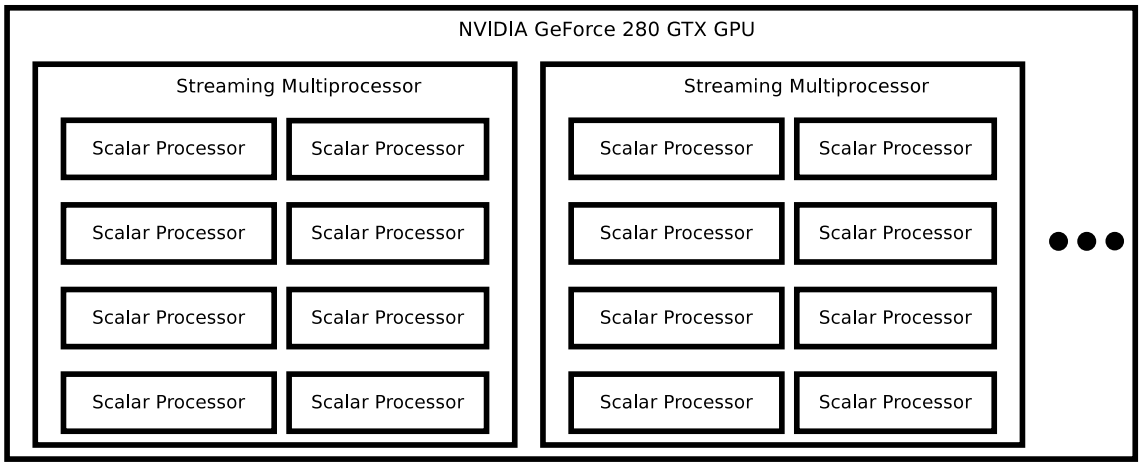

Figure 2.2: Organization of a NVIDIA GeForce 280 GTX GPU

In general, the maximum number of threads per block is 512. The maximum size of each dimension of the grid is 65535. Thus, it is possible to launch up to 65535^2x512 threads depending on the needed memory. Threads within a block can be synchronized by the *__syncthreads()* function while there is no option to synchronize different blocks. Each thread is executed independently from others with its own instruction address and register state, but the independence is restricted like described in the following. Threads are created, managed, scheduled and executed in groups of 32 successive parallel threads called warps. For memory operations, half warps are considered which are either the first or the second half of a warp. All threads within a warp start together at the same program address, but they are free to branch independently. Unfortunately, all these threads execute one common instruction at a time. This may lead to a problem if branch divergences occur, because this leads to a serial execution of each branch path taken. Therefore full efficiency is realized when all 32 threads of a warp agree on their execution path. If neither a branch conflict nor a memory conflict occur, a SM is able to process one common cycle of the 32 threads of a warp in four clock cycles.

2.1.2 Memory

In general, there are three types of memory available, which differ in locality: the device memory, the shared memory and local registers.

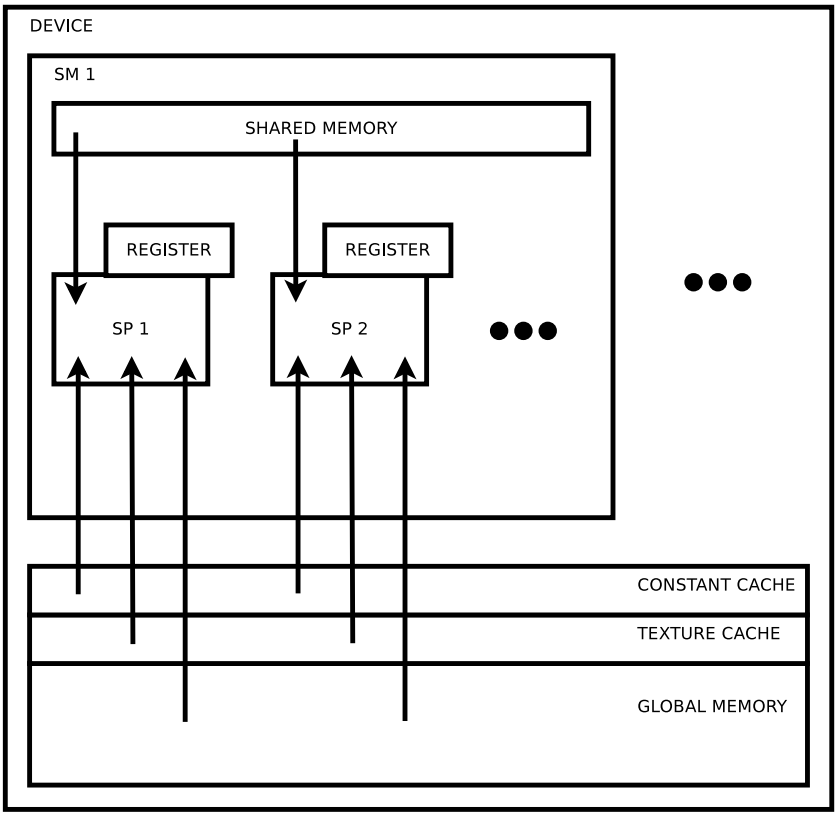

Figure 2.3: Memory Overview of a NVIDIA GeForce 280 GTX GPU

Device Memory

The device memory is accessible by all SMs respectively by all blocks as well as the host. It is divided into three parts: two read-only memory spaces, the constant memory and the texture memory, which are cached, and the uncached global memory.

Constant Memory
The constant memory is a cached read-only memory which is accessible by all SMs. The host indeed can write to this memory. The lifetime of variables in the constant memory corresponds to the application lifetime. The total amount of constant memory on GPUs with Compute Capability 1.3 is 16 KB and the cache working set for constant memory is 8 KB per SM. For all threads of a half-warp, accessing the constant memory is as fast as accessing a register if they read the same address.

Texture Memory
The texture memory is a cached read-only memory which is accessible by all SMs. The host indeed can write to this memory. The texture memory and the global memory share the same address space. Thus, the total amount of texture memory depends on the programmers choice, but it is limited to the size of the device memory. The cache working set for texture memory varies between 6 and 8 KB per SM.

Global Memory
The global memory is an uncached read/write memory which is accessible by all blocks and the host. It is used to share data between different blocks and between the host and the device. It is also used by the device to swap register contents. Accessing the global memory is very expensive, because it takes up to 600 clock cycles while accessing the cached area takes this time only once.
Fortunately, there are some techniques to speed up global memory accesses. For instance *coalescing* tries to exhaust the memory bandwidth by wrapping memory accesses of a half-warp into less transactions, but there are some restrictions which depend on the compute capability. For devices with compute capability 1.2 and higher, threads are allowed to access data in any order. In particular, several threads are allowed to access the same address at a time without causing any disadvantages. A single memory transaction is issued for each addressed segment. Additionally, the transaction size is tailored to the addressed memory size by the hardware in order to maximize the bandwidth. In contrast, threads running on devices with compute capability 1.1 have to access the memory in sequence, the accessed words have to lie in the same segment and all threads have to read words of the same size in order to use coalescing. Otherwise, a separate memory transaction is issued for each thread. An example for coalescing on devices with compute capability 1.3 is shown in Figure 2.4, where in the left case only one transaction of size 16 x (word-size) is needed, while in the right case two transactions of size 5 x (word-size) and 10 x (word-size) are necessary due to the data distribution to two segments. For further information see [4].

Local Memory
It is only used by the compiler for some automatic variables and register swaps.

Shared Memory
The shared memory is on-chip. This means that it is local to a SM and therefore much faster than global memory. Thus, a cache is not needed and not implemented. It is accessible only by the corresponding SM respectively by the blocks executed by this SM. The total amount of shared memory on GPUs with Compute Capability 1.3 is 16 KB per SM organized into 16 banks, which can be accessed simultaneously. Banks are organized such that successive 32-bit words are assigned to successive banks which have a bandwidth of 32 bits per two clock cycles cf.[4] Section 5.1.2.5 . For all threads of a half-warp, accessing shared memory is almost as fast as accessing local registers if there are no bank conflicts. Otherwise, the accesses are serialized to conflict-free groups. Additionally, the shared memory features a broadcast mechanism, which

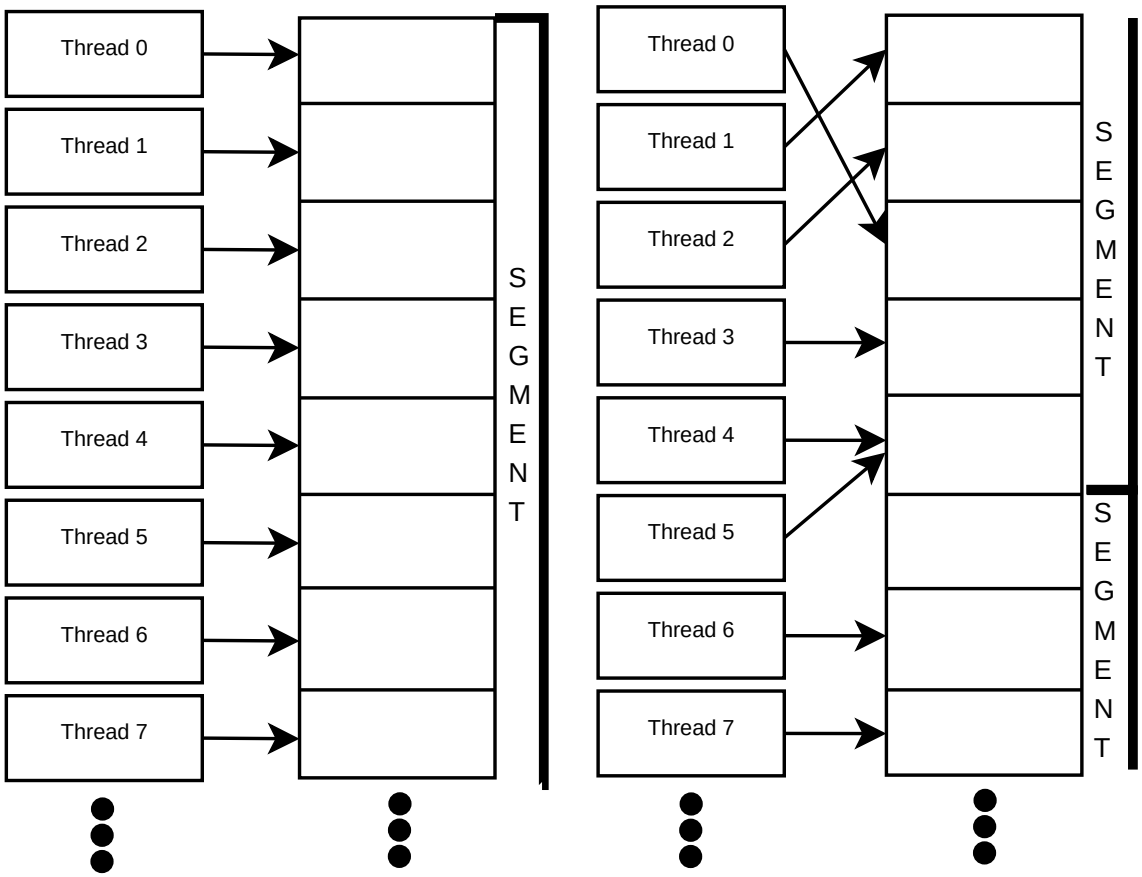

Figure 2.4: Global Memory Access Example

allows to access a 32-bit word by several threads at a time. An example is shown in Figure 2.5, where for the left case no bank conflicts occur. For the right case, there are also no bank conflicts if thread 2, 3 and 5 access the same address within the bank.

2.1.3 Instruction Throughput

In general, a SM takes the times specified in Table 2.1 to execute the instructions, but there are some special instructions like __mul24 and __umul24 which provide signed and unsigned 24-bit integer multiplication with a 32 bit result in 4 clock cycles while a standard 32-bit integer multiplication takes 16 clock cycles. Another example for these special functions is __fdividef(x,y) which provides single-precision floating-point division in only 20 clock cycles instead of 36. Unfortunately, Table 2.1 considers only the GPU time. In practice, the instruction throughput is heavily limited by the memory. E.g. an addition of two integers which are located in the global memory takes 8 clock cycles GPU time to issue the accesses for the whole warp plus two times

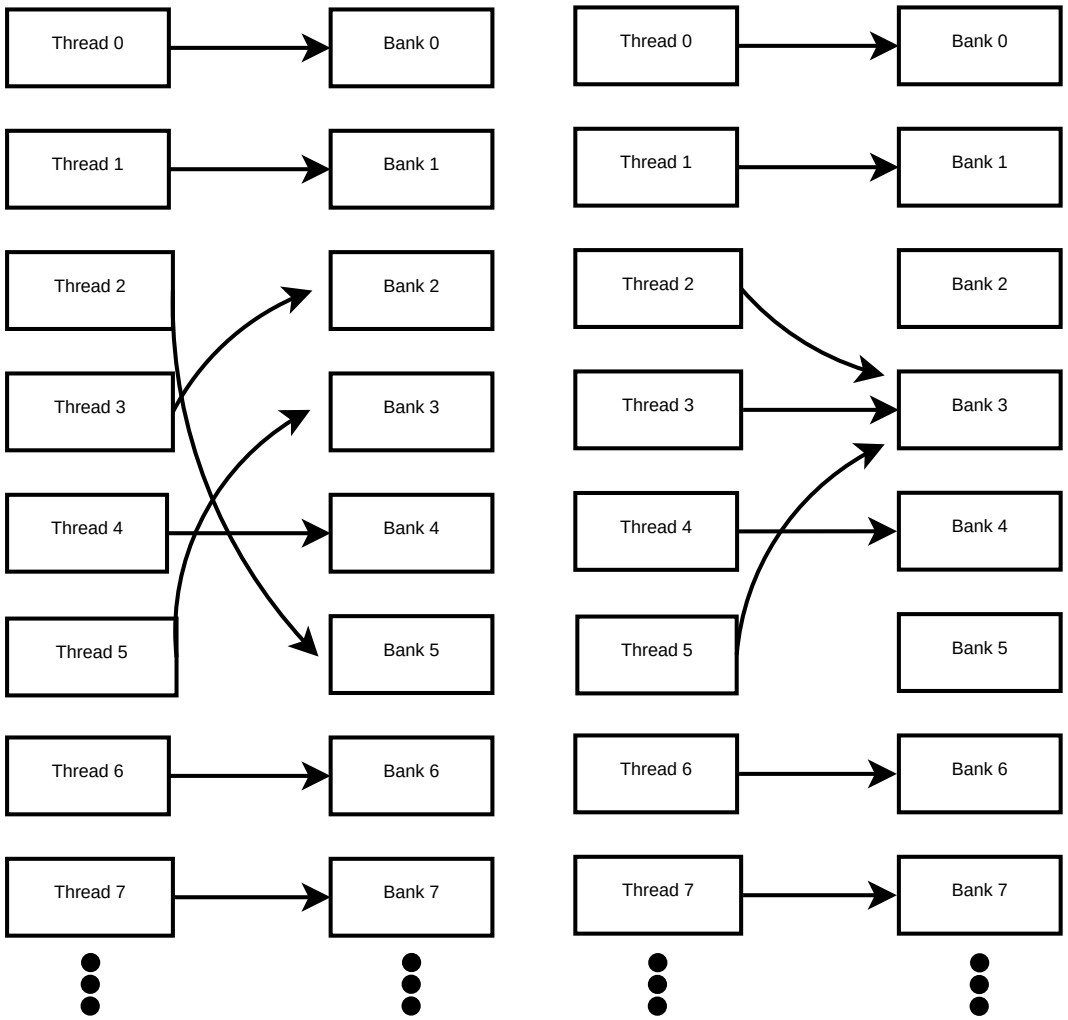

Figure 2.5: Shared Memory Access Example

about 600 cycles waiting for the memory. If there are not enough threads waiting for GPU time this lag cannot be hidden and the throughput decreases significantly.

2.2 Averest

This work describes the execution of synchronous systems on graphic devices. The synchronous systems are developed within the Averest system (http://www.averest.org/). Averest is set of tools for the specification, verification and implementation of reactive systems. It also contains a compiler for synchronous programs, which translates system descriptions written in the imperative synchronous language *Quartz* to the *Averest Intermediate Format* (AIF) based on synchronous guarded actions.

clock cycles	type	instruction
4	single-precision floating-point	add, multiply, multiply-add
4	integer	add
4	other	compare, min, max, type conversion
16	other	reciprocal, reciprocal square root, __logf
32	single-precision floating-point	square root, __sinf, __cosf, __expf
36	single-precision floating-point	division

Table 2.1: Instruction Throughput

Statement	Description
nothing	empty statement
l:pause	time consuming statement that separates macro steps
emit x, emit next(x)	immediate and delayed assignment
x = τ, next(x) = τ	immediate and delayed variable assignment
if(σ) S1 else S2	conditional statement
S1 S2	sequential execution
S1 || S2	synchronous parallel execution
do S while(σ)	iteration
[*weak*][*immediate*] abort S when(σ)	process abortion
[*weak*][*immediate*] suspend S when(σ)	process suspension
α x	declaration of local variable x with type α

Table 2.2: Quartz Kernel Language

2.2.1 Quartz

Quartz is an imperative synchronous programming language. It is based on the concept of perfect synchrony, i.e. all reactions of the system are executed without any delay. The overall behavior of the system is given by a sequence of reactions. A reaction is also called macro-step and consumes one amount of logical time. Each macro step consists of finitely many computations called micro steps. In contrast to macro steps, micro steps do not consume time. Hence, micro steps within a macro step run in parallel. In each macro step, all inputs are read and new outputs are generated with respect to the current state and current inputs. In particular, all signals and variables are constant for a macro step. Quartz contains a lot of statements. Only a subset of them, called the kernel language, is described here, because all other statements are just syntactic sugar, they can be viewed as macros. The set of statements that forms the kernel language is ambiguous. A possible kernel is shown in Table 2.2. Other statements like **await** can be represented by a combination of kernel statements:

$$\textbf{await}(\sigma) := \textbf{abort}\ \ \textbf{do}\ \mathrm{l}\!:\!\textbf{pause};\ \textbf{while}(\textbf{true})\ \textbf{when}(\sigma).$$

An example of a Quartz module is shown in Listing 2.1.

```
module MatrixMul(int{32}[2][2] ?a, int {32}[2][2]  ?b,  int {32}[2][2]  !out)
{
    while(true) {
    out[0][0]  = (a[0][0]* b[0][0]) +(a[0][1]* b[1][0]) ;
    out[0][1]  = (a[0][0]* b[0][1]) +(a[0][1]* b[1][1]) ;
    out[1][0]  = (a[1][0]* b[0][0]) +(a[1][1]* b[1][0]) ;
    out[1][1]  = (a[1][0]* b[0][1]) +(a[1][1]* b[1][1]) ;
    pause;
    }
}
```

Listing 2.1: Quartz Module for Matrix Multiplication

This module gets two integer matrices as inputs in each step and computes the product of the inputs in each step in an infinite loop. At the beginning of the program all inputs are read, then the while loop is entered and all elements of the output array are computed immediately. The **pause**; statement finalize the actual macro step. Thus, the next macro step begins and the input arrays a and b are read again, the loop is reentered and the new outputs are generated immediately.

A given system description in Quartz can be translated to synchronous guarded actions by the *qrz2aif* compiler included in the Averest system. For further informations to Quartz see [6].

2.2.2 Synchronous Guarded Actions

A synchronous guarded action (GA) has the form $\gamma \Rightarrow \alpha$, where γ is a boolean expression called guard and α is an arbitrary expression called action. The guard triggers the corresponding action. In general, the actions can be immediate or delayed assignments with the following form: $x = \tau$ or $next(x) = \tau$. A synchronous system can be represented by a set of GAs. The system can be executed as follows: In each macro step all guards are evaluated synchronously and all enabled actions are executed immediately. If an enabled action is an immediate assignment then the environment is changed immediately else the assignment is deferred to the next macro step. For each variable, if neither an immediate assignment nor a delayed assignment of the preceding macro step is executed then the so-called *reaction to absence* is applied. It may also happen that two or more enabled GAs assign a different value to the same variable. These conflicting actions lead to semantic problems, but in the following only causally correct programs are considered which inhibit such a behavior. Listing 2.2 shows the GAs for the matrix multiplication example.

The translation from Quartz to Synchronous Guarded Actions is not obvious. The control flow and data flow have to be translated separately. Additionally, some spe-

cial cases like schizophrenia problems have to be considered. For further informations see [6].

2.2.3 Averest Intermediate Format

The Averest Intermediate Format, is a XML based system description. Synchronous guarded actions are used to model the system. They model control flow as well as data flow. Unfortunately, this way to describe a system can hide several control flow constructs like loops which are given in the Quartz program. Therefore, it is not possible to apply loop auto vectorization algorithms. Nevertheless, AIF is very powerful and can represent structural and behavioral aspects of a system as well as different forms of concurrency.
An AIF system contains the following informations:

- Needed variables are described by name, direction and type. The direction can be input, output and inout. Input means that this variable can only be read, while output means that this variable can only be written. Inout variables can be read and written. For the synthesis inout variables are regarded as outputs. For input variables, the type can be bool, int, nat, real, btv or arrays of them. Each type can be finite or infinite, but for synthesis only finite types can be used. Output and inout variables have an additional parameter, that defines if the variable is an event variable or a memorized one. The value of an event variable is reset every step while the value of a memorized variable is kept.
- Abbreviations are used to reduce the computational effort of evaluating the guards by reusing intermediate results.
- The initial part of the system represents the first step of the program. The description consist of the control flow, the data flow and assertions. Control flow and data flow are lists of guarded actions. For the synthesis, this initial part is moved to the main part by a given transformation of the Averest System.

```
true => next(__ell0) = true
true => out[0][0] = a[0][0]*b[0][0]+a[0][1]*b[1][0]
true => out[0][1] = a[0][0]*b[0][1]+a[0][1]*b[1][1]
true => out[1][0] = a[1][0]*b[0][0]+a[1][1]*b[1][0]
true => out[1][1] = a[1][0]*b[0][1]+a[1][1]*b[1][1]
__ell0 => next(__ell0) = true
__ell0 => out[0][0] = a[0][0]*b[0][0]+a[0][1]*b[1][0]
__ell0 => out[0][1] = a[0][0]*b[0][1]+a[0][1]*b[1][1]
__ell0 => out[1][0] = a[1][0]*b[0][0]+a[1][1]*b[1][0]
__ell0 => out[1][1] = a[1][0]*b[0][1]+a[1][1]*b[1][1]
```

Listing 2.2: Guarded Actions Extracted from the Matrix Multiplication example

- The main part represents all other steps of the program and also consists of the control flow, the data flow and assertions.
- Absence reactions are used to set a variable if no immediate assignment and no delayed assignment of the preceding step provide a value for this variable.
- An invariant and specifications are given, which are not used in this work.

An example is given in the following output of the *aif2txt* tool.

```
version: v2_0_alpha timestamp: Thu Apr 15 08:43:18 2010 UTC

system MatrixMul:
    interface :
        a: input int {32}[2][2]
        b: input int {32}[2][2]
        out: output memorized int{32}[2][2]
    locals :
        __ell0 : label bool
    abbreviations :
    init :
        control flow:
            true => next(__ell0) = true
        data flow:
            true => out[0][0] = a[0][0]*b[0][0]+a[0][1]*b[1][0]
            true => out[0][1] = a[0][0]*b[0][1]+a[0][1]*b[1][1]
            true => out[1][0] = a[1][0]*b[0][0]+a[1][1]*b[1][0]
            true => out[1][1] = a[1][0]*b[0][1]+a[1][1]*b[1][1]
        assertions:
            true => __rte0 : assert →
                a[0][0]*b[0][0]+a[0][1]*b[1][0]<32&−32<=a[0][0]*b[0][0]+a[0][1]*b[1][0]
            true => __rte1 : assert →
                a[0][0]*b[0][1]+a[0][1]*b[1][1]<32&−32<=a[0][0]*b[0][1]+a[0][1]*b[1][1]
            true => __rte2 : assert →
                a[1][0]*b[0][0]+a[1][1]*b[1][0]<32&−32<=a[1][0]*b[0][0]+a[1][1]*b[1][0]
            true => __rte3 : assert →
                a[1][0]*b[0][1]+a[1][1]*b[1][1]<32&−32<=a[1][0]*b[0][1]+a[1][1]*b[1][1]
    main:
        control flow:
            __ell0 => next(__ell0) = true
        data flow:
            __ell0 => out[0][0] = a[0][0]*b[0][0]+a[0][1]*b[1][0]
            __ell0 => out[0][1] = a[0][0]*b[0][1]+a[0][1]*b[1][1]
            __ell0 => out[1][0] = a[1][0]*b[0][0]+a[1][1]*b[1][0]
            __ell0 => out[1][1] = a[1][0]*b[0][1]+a[1][1]*b[1][1]
        assertions:
            __ell0 => __rte0@0 : assert →
                a[0][0]*b[0][0]+a[0][1]*b[1][0]<32&−32<=a[0][0]*b[0][0]+a[0][1]*b[1][0]
```

```
        __ell0 => __rte1@0 : assert →
            a[0][0]*b[0][1]+a[0][1]*b[1][1]<32&-32<=a[0][0]*b[0][1]+a[0][1]*b[1][1]
        __ell0 => __rte2@0 : assert →
            a[1][0]*b[0][0]+a[1][1]*b[1][0]<32&-32<=a[1][0]*b[0][0]+a[1][1]*b[1][0]
        __ell0 => __rte3@0 : assert →
            a[1][0]*b[0][1]+a[1][1]*b[1][1]<32&-32<=a[1][0]*b[0][1]+a[1][1]*b[1][1]
    absence reactions:
    invar : true
    specifications :
```

3 Transformations

Before beginning scheduling and system partitioning, some transformations provided by the Averest system have to be applied to the AIF system in order to reduce the effort for code generation and to adapt the system to the CUDA architecture. Afterwards, only GAs have to be considered. Reactions to absence and event variables do no longer exist in the resulting system. The init part only consists of independent constant assignments and it can be executed by the host together with other initializations. The main part only consists of data flow which is modeled by GAs. These GAs are executed on the graphics device, but before they have to be scheduled and distributed to possibly different blocks. Despite all these transformations, the system behavior and all parallelism is preserved. The following transformations are applied:

- The control flow has to be converted to data flow. Hence, the control flow has not to be considered separately and can be scheduled together with the data flow.
- In order to reduce the number of needed kernels, all initial actions are moved to the main part. In general, the beginning and the end of the program have to be distinguished, because otherwise the program would immediately restart after terminating. The distinction is done by separating the initial step and the following steps. By applying this transformation, the distinction is done by introducing a so-called boot variable of type event whose value is initialized to true and it is set to false in all other steps. This is done by reactions to absence which are eliminated by the next transformation.
- Event variables have to be converted to memorized. For the C synthesis, only events which are written by delayed GAs have to be eliminated because other event variables can be declared as local variables in the main loop. Therefore, they are reset automatically. For CUDA, it is not possible to declare them locally because this would destroy the vectorization. Unfortunately, a general event to memorized transformation is not implemented in the Averest system. Thus, the input system is assumed to be event free. But the event variable created by the init to main transformation can be eliminated by eliminating all event variables with delayed actions which is implemented in the Averest system.
- The reactions to absence has to be converted to GAs. Thus, they can be treated as ordinary data flow. Unfortunately, new initial GAs are produced which will be executed on the host.

- All abbreviations have to be substituted. Unfortunately, this leads to a lot of additional computations in the program execution. Otherwise, abbreviations would have to be converted to GAs, but this would lead to many additional dependencies which would inhibit the parallelism.
- Dynamic array accesses have to be eliminated by eliminating compound data types. Later on, pointer will be used to access variables. Given dynamic array accesses, it would be necessary to store a matrix of pointer for every access which would cause a too high memory consumption.

For instance, applying these transformations in the given order to the AIF system of the matrix multiplication example leads to the system shown in Listing 3.1 which is used to describe the general structure of the leading system. It yields a system which has the same interface and it has only immediate GAs in the init part. These GAs result from the *clear init* and the *remove all next assignments* transformation. In the init part of the example code, there are only guards that are true and all right-hand sides of the assignments are boolean constants which is valid for all systems after applying these transformations. Therefore, these assignments can be done by the host. In particular, there are no dependencies between these assignments. Hence, the init part has not to be scheduled. The system contains no more labels due to the first transformation. They are converted to event variables which are written by delayed actions. Thus, they are converted to memorized variables by the *remove all next assignments to (data) events* transformation. Unfortunately, they are no longer marked as labels because they could have directly been used for optimizations like storing them in the cached constant memory as described in Chapter 5. Neither the init part, nor the main part contain control flow as shown in the example Line 12 and 18. Thus, the control flow has not to be considered anymore. Furthermore, the system contains no abbreviations and no absence reactions. Next, the system has to be partitioned and scheduled in order to get SIMT code.

```
1  version: v2_0_alpha timestamp: Thu Apr 15 10:36:04 2010 UTC
2  system MatrixMul:
3      interface :
4          a: input int {32}[2][2]
5          b: input int {32}[2][2]
6          out: output memorized int{32}[2][2]
7      locals :
8          __init0 : local memorized bool
9          __ell0 : local memorized bool
10     abbreviations :
11     init :
12         control flow:
13         data flow:
14             true => __ell0 = false
15             true => __init0 = true
16         assertions :
17     main:
18         control flow:
19         data flow:
20             !( __init0 | __ell0) => next(__ell0) = false
21             true => next(__init0) = false
22             __init0 => next(__ell0) = true
23             __init0 => out[0][0] = a [0][0]* b[0][0]+a [0][1]* b [1][0]
24             __init0 => out[0][1] = a [0][0]* b[0][1]+a [0][1]* b [1][1]
25             __init0 => out[1][0] = a [1][0]* b[0][0]+a [1][1]* b [1][0]
26             __init0 => out[1][1] = a [1][0]* b[0][1]+a [1][1]* b [1][1]
27             __ell0 => next(__ell0) = true
28             __ell0 => out[0][0] = a [0][0]* b[0][0]+a [0][1]* b [1][0]
29             __ell0 => out[0][1] = a [0][0]* b[0][1]+a [0][1]* b [1][1]
30             __ell0 => out[1][0] = a [1][0]* b[0][0]+a [1][1]* b [1][0]
31             __ell0 => out[1][1] = a [1][0]* b[0][1]+a [1][1]* b [1][1]
32         assertions :
33             __init0 => __rte0 : assert →
                   a [0][0]* b[0][0]+a [0][1]* b[1][0]<32&−32<=a[0][0]*b[0][0]+a[0][1]*b[1][0]
34             __init0 => __rte1 : assert →
                   a [0][0]* b[0][1]+a [0][1]* b[1][1]<32&−32<=a[0][0]*b[0][1]+a[0][1]*b[1][1]
35             __init0 => __rte2 : assert →
                   a [1][0]* b[0][0]+a [1][1]* b[1][0]<32&−32<=a[1][0]*b[0][0]+a[1][1]*b[1][0]
36             __init0 => __rte3 : assert →
                   a [1][0]* b[0][1]+a [1][1]* b[1][1]<32&−32<=a[1][0]*b[0][1]+a[1][1]*b[1][1]
37             __ell0 => __rte0@0 : assert →
                   a [0][0]* b[0][0]+a [0][1]* b[1][0]<32&−32<=a[0][0]*b[0][0]+a[0][1]*b[1][0]
38             __ell0 => __rte1@0 : assert →
                   a [0][0]* b[0][1]+a [0][1]* b[1][1]<32&−32<=a[0][0]*b[0][1]+a[0][1]*b[1][1]
39             __ell0 => __rte2@0 : assert →
                   a [1][0]* b[0][0]+a [1][1]* b[1][0]<32&−32<=a[1][0]*b[0][0]+a[1][1]*b[1][0]
```

```
40              __ell0 => __rte3@0 : assert →
                    a[1][0]*b[0][1]+a[1][1]*b[1][1]<32&−32<=a[1][0]*b[0][1]+a[1][1]*b[1][1]
41      absence reactions:
42      invar : true
43      specifications :
```

Listing 3.1: *aif2aif -t cocf MatrixMul.aifs -> aif2aif -t clin MatrixMul.aifs -> aif2aif -t nxte MatrixMul.aifs -> aif2aif -t coar MatrixMul.aifs -> aif2aif -t suab MatrixMul.aifs -> aif2aif -t prcr MatrixMul.aifs*

4 SIMT Code Generation

Within a thread, all operations are performed sequentially. Hence, the Model of Computation has to be changed. Synchronous systems modeled by synchronous guarded actions (GAs) are based on the concept of perfect synchrony, which is just a model. But in CUDA programs, the execution time as well as the limited number of functional units has to be considered. Thus, it is not possible to check all guards synchronously and to perform actions immediately.
GAs could be executed on graphics devices by translating them to sequential C-code and embedding it in one kernel function. This is the easiest way to translate GAs to a CUDA program, but this approach does not use the parallelism of a GPU. It slows down the whole computation by using just one processing unit. Due to the GPU architecture described in Section 2.1.1, SIMT code has to be extracted in order to execute GAs efficiently on graphics devices. SIMT code consists of C instructions commonly used by all threads within a block/warp. GAs that consist of equal operations and work on possibly different data are used to form SIMT code.
In order to find such GAs, the dependencies as well as the problem of data sharing and thread/block synchronization have to be considered. Only the very slow global memory can be used to share data between blocks, while the fast shared memory can be used to share data within a block. Like mentioned in Section 2.1.1, synchronization is possible between threads within a block, but blocks are required to execute independently. Unfortunately, the thread synchronization requires at least 4 clock cycles. Hence, it is more efficient to execute depending GAs within the same thread. Therefore, a schedule has to be extracted that ensures the validity of read variables. The whole system is partitioned into three parts: the init part which consists of immediate GAs due to the transformations, the immediate GAs of the main part and the delayed GAs of the main part. As mentioned in the previous chapter, the init part is ignored in this chapter. The main part is divided into the immediate GAs and the delayed GAs due to the possible ordering mentioned in Section 4.1. These parts are scheduled and partitioned.
In order to derive a schedule, dependencies have to be extracted first. Afterwards, independent sets of GAs can be derived and scheduled independently. Finally, a comparison can be performed to identify sets of GAs that can be subsumed by the same SIMT code. By way of illustration, each step is applied to the system shown in Listing 4.1 respectively the GAs extracted from that system shown in Listing 4.2.

```
module MyModule(bool ?trigger)
{
    bool a,b;
    int{32} c,d,e,f;
    loop{
        if(a) c = 5;
        else c = 3;
        if(trigger) d = 2;
        else d = 4;
        a = true;
        pause;
        e = c;
        b = a & trigger;
        pause;
    }
}
```

Listing 4.1: Example Quartz Module for Illustration

```
1  !(__init0|__ell1) => next(__ell0) = false
2  !__ell0 => next(__ell1) = false
3  true => next(__init0) = false
4  __init0 => next(__ell0) = true
5  __init0&a => c = 5
6  __init0&!a => c = 3
7  trigger&__init0 => d = 2
8  __init0&!trigger => d = 4
9  __init0 => a = true
10 __ell1 => next(__ell0) = true
11 __ell0 => next(__ell1) = true
12 __ell1&a => c = 5
13 __ell1&!a => c = 3
14 trigger&__ell1 => d = 2
15 __ell1&!trigger => d = 4
16 __ell1 => a = true
17 __ell0 => e = c
18 __ell0 => b = trigger&a
```

Listing 4.2: Transformed GAs Extracted from Listing 4.1

4.1 Dependency Extraction

There are three kinds of dependencies which can occur:

- true dependency (read after write) (immediate action → immediate/delayed action)
- anti dependency (write after read) (immediate/delayed action → delayed action)
- output dependency (write after write) (immediate action → delayed action)

A true dependency can occur between an immediate action and an arbitrary action because if variable x is written by an immediate assignment and an immediate or delayed GA reads x, the writing GA has to be executed before the others. An anti dependency can occur between an arbitrary action and a delayed action because delayed actions assign values that are valid for the next step and not for the current one. Output dependencies can only occur between immediate and delayed actions because of the same reason as for anti dependencies.

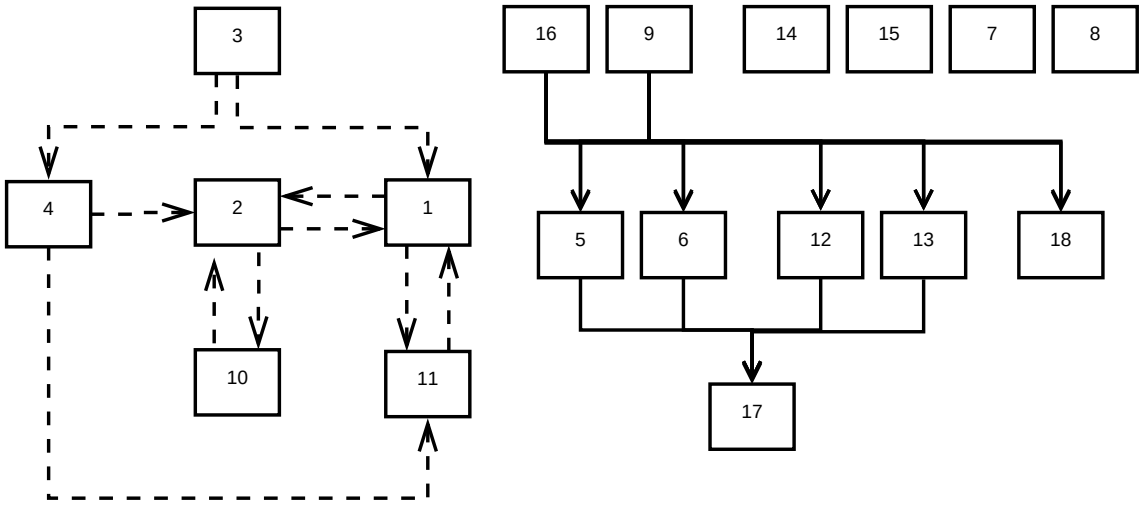

Figure 4.1: left: anti dependencies between delayed GAs, right: true dependencies between immediate GAs

Using these dependencies, a directed *action dependency graph* (ADG) can be build. An example is shown in Figure 4.1, where the numbers correspond to the line of a GA in Listing 4.2.

An ADG may contain cyclic dependencies. Cycles of true dependencies can be broken if the program is causally correct. This can be done by building the *extended finite state machine* (EFSM) or checking the conjunction of all guards in the cycle for satisfiability, but this is not considered in this work. In the next sections only cyclic free true dependencies are considered. Other cycles can be completely eliminated.

```
1 !( __init0 | __ell1' ) => next(__ell0) = false
2 ! __ell0 => __ell1' = __ell1
3 ! __ell0 => next(__ell1) = false
4 true => next(__init0) = false
5 __init0 => next(__ell0) = true
6 __ell1' => next(__ell0) = true
7 __ell0 => __ell1' = __ell1
8 __ell0 => next(__ell1) = true
```

Listing 4.3: Transformed Acyclic Delayed GAs Extracted from Listing 4.2

For sequential code generation, delayed GAs can be viewed as immediate GAs that are executed after processing real immediate GAs. Hence, immediate GAs and delayed GAs are scheduled separately in order to prevent mixed cycles and output dependencies.
Anti dependencies can be eliminated by introducing local variables, e.g. given the delayed GA $\gamma \rightarrow$ next(x) = τ of the cycle, a new variable x' has to be introduced, that replaces x in all further readings. Additionally, an immediate GA with the same guard has to be added that copies x to x': $\gamma \rightarrow x' = x$. This immediate guarded action introduces new true dependencies but it eliminates the cycle. An example is shown in Figure 4.2 and Listing 4.3. The additional copy operations are shown in Line 2 and Line 7. In Line 1 and 6, the occurrence of __ell1 are replaced by the copy __ell1'. In the other lines, __ell1 is only accessed for writing, thus it is not replaced.
So far, the set of GAs can be transformed such that it is possible to get an acyclic dependency graph, which is used to derive a schedule.

4.2 Scheduling

In order to derive independent threads, each dependency graph has to be partitioned into its components. Afterwards, each component has to be scheduled independently. This is done by deriving a topological order from the dependency graph which is possible due to the cycle elimination. An example is shown in Listing 4.4 and Listing 4.5.

In order to achieve a preferably huge amount of SIMT threads, it is very important to derive schedules deterministic, i.e. switching the order of GAs in the incoming system should not influence the final schedule. An example is shown in Figure 4.3, where an ADG (upper mid) is scheduled (lower left/lower right). Either the lower left or the lower right ordering can be chosen without changing the overall behavior of the system. Ordering them randomly may lead to a separation of independent threads into different blocks even if they could perform the same operations at the same position. Therefore, an arbitrary order has to be defined on expressions. Thus,

the order of GAs that can be swapped without changing the result equals in every thread. As a result, all threads that could perform the same operations at the same positions will be grouped. The ordering of GAs with the same expression types does not influence the comparison, because for the SIMT code, only the type and number of operations is important.

4.3 Vectorization

As mentioned in Section 2.1.1, every thread is mapped to a SP, where 8 of them form a SM. Thus, at least 8 independent threads should compose a block which is mapped to a certain SM. Unfortunately, all threads within a block execute the same instructions (SIMT). Hence, the scheduled sets of GAs have to be grouped by operation type and order. Therefore, the GA lists have to be compared element wise to each other.

GAs are compared recursively by first comparing the guards and afterwards comparing the actions. In general, only one recursive comparing function is needed, which compares two expressions. This function returns true if all expressions are equal apart from variables and array accesses because only operations are regarded and array elements are regarded as variables. The function returns false otherwise. In particular, constants have to be the same. Otherwise, each constant has to be stored in a separate variable in order to get SIMT code. This would lead to additional memory wastage and slow down the whole computation. Guards are boolean expressions and therefore, they are processed by the comparing function. For actions, only the right-hand sides of the assignments are compared because left hand sides of these can only be variables or static array accesses. This is also done by the comparing function. Please note that commutativity and associativity of operations are not regarded in this work because this would cause too much effort for the synthesis. Sets of GAs which are equal in the previously defined way can be grouped in order to form a block and they can be represented by the same SIMT code.
An example is shown in Listing 4.6 and Table 4.1. In Listing 4.6, there are five independent threads which have to be compared. The first, third and last thread consist of two GAs while the second and the fourth thread consist of only one GA. At the beginning, the first thread is compared to all others. Therefore, the first GA of the first thread is compared to the first GA of all other threads. The GA shown in Line 4 matches the GA shown in Line 1 because the constants and the operations are equal. Accordingly, the other GAs of these threads would have to be compared, but the first one consists of more GAs than the second one. Thus, they are not grouped in order to form SIMT code. It would be possible to group them by using a variable configuration such that the guard is not true for the second thread. But this is not considered in this work because in general, it could also be possible that the additional GA is placed in the middle like illustrated in Figure 4.4. The upper thread consists of three GAs while the lower one only contains two GAs. GA A and A' as well as C and C' are equal. Thus, both threads could be grouped by setting the guard of B for the

Block0	Block1	Block2	Block3
a&!b => e = c + 5 - d b => f = c + 5	g&!w => h = c + 5 - o	u => p = h + 5 - d (p==1) => z = c + 6 - o	g&!w => r = c - o + 5
i&!j => l = k + 5 - d g => m = l + 5			

Table 4.1: Compared GAs for the Comparison Example

lower thread to false. But there could also be more than one GA that is exclusively contained in one thread. Therefore, it is too complex to analyze all threads in order to find GAs that may match even if they are placed on different positions. The GA shown in Line 6 matches the GA shown in Line 1 because they execute the same operations at the same position and constants also match. But the first thread consists of two GAs. Thus, the second GA of the first thread shown in Line 2 has to be compared to the second GA of the third thread shown in Line 7. They do not match because the guards consists of different operations. Therefore, these two threads can also not be grouped. Next, the first GA of the first thread has to be compared to the GA of the fourth thread. They also do not match even if they perform the same operations and have the same constants because the commutativity is not considered in this comparing function. Afterwards, the first GA of the first thread is compared to the GA shown in Line 11. They do match because they perform the same operations at the same positions and the constants are equal. Thus, the GA shown in Line 2 and the one shown in Line 12 have to be compared. They also match. Therefore, the first thread and the last thread can be grouped. Next, the second thread and all following have to be compared which is done the same way. Finally, the threads can be grouped like shown in Table 4.1. All threads in a column can be represented by the same SIMT code.

So far, the system is scheduled and partitioned to threads which are grouped to blocks. In order to get an efficient CUDA program, SIMT code has to be extracted, i.e. the same code has to be created for all threads within a block. In particular, variable accesses have to be considered. Note that constants are not considered in the following because they can be left due to the scheduling.
There are two options for variable accesses: either all threads within a block access the same variable at the same position, so nothing is left to do, or threads access different variables at the same position, then all variables, or cells in terms of arrays, have to be arranged such that all accesses can be mapped to array accesses which depend on the thread ID and block ID. The first option have to be considered at the scheduling. Unfortunately, this may lead to very small blocks which cannot exploit the power of SMs. Therefore, all variables are arranged in an array such that it is possible to derive functions for each access which compute the right array index for each

ThreadID/AccessNo.	0	1	2	3	4	5
0	__init0	out[0][0]	a[0][0]	b[0][0]	a[0][1]	b[1][0]
1	__init0	out[0][1]	a[0][0]	b[0][1]	a[0][1]	b[1][1]
2	__init0	out[1][0]	a[1][0]	b[0][0]	a[1][1]	b[1][0]
3	__init0	out[1][1]	a[1][0]	b[0][1]	a[1][1]	b[1][1]
4	__ell0	out[0][0]	a[0][0]	b[0][0]	a[0][1]	b[1][0]
5	__ell0	out[0][1]	a[0][0]	b[0][1]	a[0][1]	b[1][1]
6	__ell0	out[1][0]	a[1][0]	b[0][0]	a[1][1]	b[1][0]
7	__ell0	out[1][1]	a[1][0]	b[0][1]	a[1][1]	b[1][1]

Table 4.2: Cells Extracted from Listing 4.7

thread depending on the thread ID and block ID. An example is shown in Listing 4.9 which provides the SIMT code for the block shown in Listing 4.7. In the SIMT code all variables/cells are stored in the array *var* which is accessed by the functions *v1 ... v6*.

Unfortunately, it would be too complex, sometimes even impossible, to derive a pair of ordering and corresponding functions. A simplification would be to arrange all variables, which are accessed at the same position in the code of one block, in an array such that a variable can be accessed by the thread ID. An example is shown in Listing 4.10, where the first number of var_b_ specifies the the position and the second one the block ID. To reduce the total amount of arrays, all arrays for one block can be organized in a two-dimensional one.

Unfortunately, many variables are used on different places. Thus, there are many copies which have to be maintained. Therefore, almost every array has to be updated after a write due to the true dependencies. Hence, all variable accesses that aim for the same variable should read/write from/to the same address. Obviously, pointer are a good choice to implement. Thus, a two-dimensional array of pointer is generated for every block. One dimension distinguishes threads and the other is used to distinguish different access positions. An example is shown in Listing 4.11. All variables/cells can be arranged in an one-dimensional array.

Unfortunately, the memory consumption grows significantly. Therefore, only a multiplication of up to two 9x9 matrices can be realized using 32 threads per block. Decreasing the number of threads per block would allow to multiply larger matrices, but this would slow the computation because slow memory accesses could not be hidden by executing another warp meanwhile. So far, the SIMT code is extracted from the scheduled and partitioned GAs. The next steps are allocating the memory, extracting a mapping for the previously described pointer array and initializing all arrays.

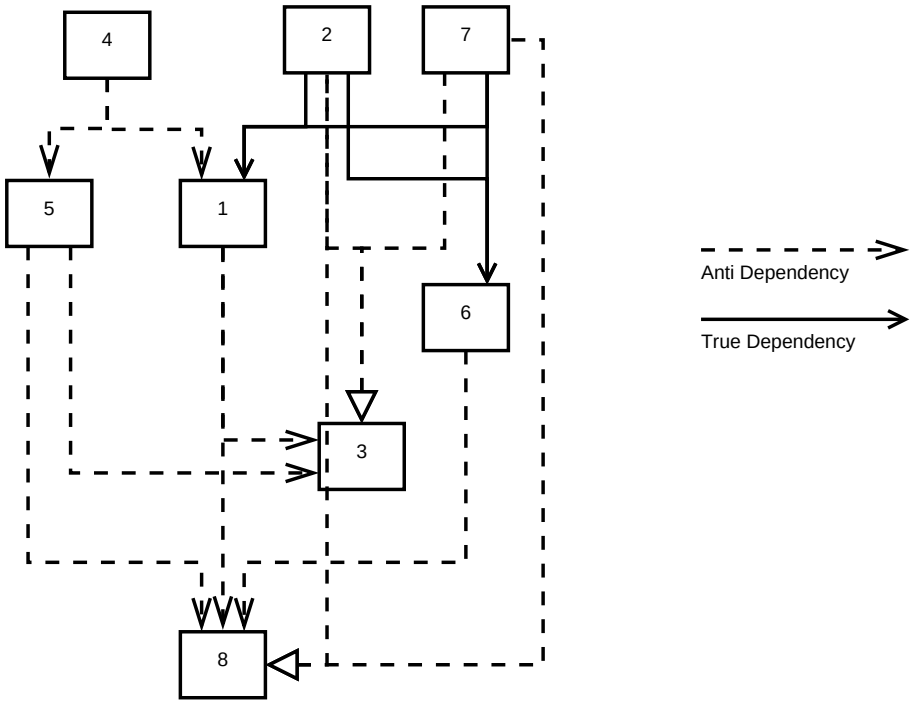

Figure 4.2: Acyclic Dependency Graph Extracted from the System Shown in Listing 4.3

```
1 __init0 => a = true
2 __ell1 => a = true
3 __init0&a => c = 5
4 __init0&!a => c = 3
5 __ell1&a => c = 5
6 __ell1&!a => c = 3
7 __ell0 => e = c
8 __ell0 => b = trigger&a
```

Listing 4.4: Scheduled Immediate GAs Extracted from Listing 4.2

```
true => next(__init0) = false
__init0 => next(__ell0) = true
!__ell0 => __ell1′ = __ell1
__ell0 => __ell1′ = __ell1
!( __init0 | __ell1′) => next(__ell0) = false
__ell1′ => next(__ell0) = true
!__ell0 => next(__ell1) = false
__ell0 => next(__ell1) = true
```

Listing 4.5: Scheduled Delayed GAs Extracted from Listing 4.3

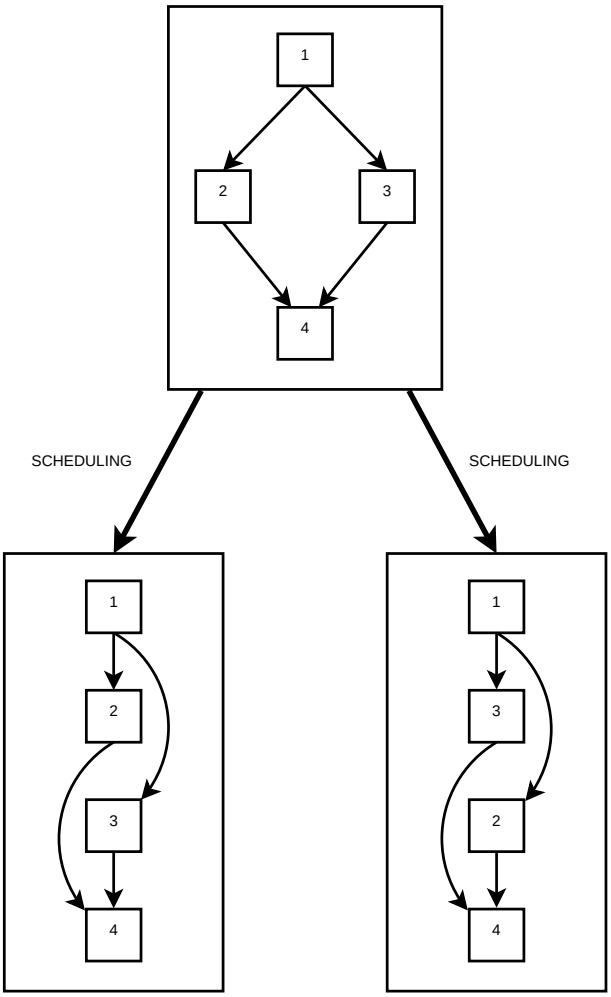

Figure 4.3: Example ADG and Corresponding Schedules

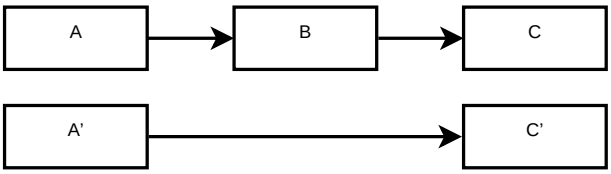

Figure 4.4: Two Independent Threads that Are Not Grouped

```
1  {a&!b => e = c  + 5  − d
2   b    => f  = e  + 5}
3
4  {g&!w => h = c + 5 − o}
5
6  {u    => p = h + 5  − d
7   (p==1) => z = c + 6  − o}
8
9  {g&!w => r = c − o + 5}
10
11 {i&!j  => l  = k  + 5  − d
12  g    => m = l + 5}
```

Listing 4.6: GAs for the Comparison Example

```
__init0  => out[0][0]  = a [0][0]∗ b[0][0]+a [0][1]∗ b [1][0]
__init0  => out[0][1]  = a [0][0]∗ b[0][1]+a [0][1]∗ b [1][1]
__init0  => out[1][0]  = a [1][0]∗ b[0][0]+a [1][1]∗ b [1][0]
__init0  => out[1][1]  = a [1][0]∗ b[0][1]+a [1][1]∗ b [1][1]
__ell0  => out[0][0]  = a [0][0]∗ b[0][0]+a [0][1]∗ b [1][0]
__ell0  => out[0][1]  = a [0][0]∗ b[0][1]+a [0][1]∗ b [1][1]
__ell0  => out[1][0]  = a [1][0]∗ b[0][0]+a [1][1]∗ b [1][0]
__ell0  => out[1][1]  = a [1][0]∗ b[0][1]+a [1][1]∗ b [1][1]
```

Listing 4.7: One Block Extracted from the Matrix Multiplication Example

```
var[v0(threadID, blockID)] => var[v1(threadID, blockID)] = var[v2(threadID, blockID)] ∗ →
    var[v3(threadID, blockID)] + var[v4(threadID, blockID)] ∗ var[v5(threadID, blockID)]
```

Listing 4.8: Vectorized Guarded Action Extracted from Listing 4.7

```
if (var[v0(threadID, blockID)]) var[v1(threadID, blockID)] = var[v2(threadID, blockID)] ∗→
    var[v3(threadID, blockID)] + var[v4(threadID, blockID)] ∗ var[v5(threadID, blockID)]
```

Listing 4.9: SIMT Code Extracted from Listing 4.7

```
var0b0[threadID] => var1b0[threadID] = var2b0[threadID] ∗ var3b0[threadID] + →
    var4b0[threadID] ∗ var5b0[threadID]
```

Listing 4.10: Vectorized Guarded Action Extracted from Listing 4.7 using One Array per Position

```
if (∗var0[threadID][0]) (∗var0[threadID][1]) = (∗var0[threadID][2]) ∗ →
    (∗var0[threadID][3]) + (∗var0[threadID][4]) ∗ (∗var0[threadID][5])
```

Listing 4.11: SIMT Code Extracted from Listing 4.7

5 Memory Management

The memory presents the bottleneck of the CUDA architecture due to the huge latency. Hence, the usage of the different kinds of memory as well as their limitations have to be considered carefully.

5.1 Memory Selection

CUDA programs can use four memory spaces for their variables like mentioned in Section 2.1.2. Thus, needed variables have to be distributed over these memories such that there is enough space and the memory latency does not slow down the whole computation.
There are no more event variables due to the transformations. Therefore, all variables are memorized and have to be saved after each step at the global or constant memory. The shared memory is not usable because all the data allocated in the shared memory has the lifetime of a block and therefore, it is transient. The easiest option is to allocate all variables in the global memory. But, in order to speedup the computation, faster memories like the constant memory should be used.
Unfortunately, the CUDA compiler NVCC can't resolve the right type of memory when using an array of pointer to mixed memory targets which is accessed dynamically. Thus, it automatically assumes the global memory as target. Therefore, all used variables have to be placed in the global memory. Fortunately, graphic devices with compute capability 2.0 and higher have a cached global memory. Thus, the memory latency is significantly reduced. But for the NVIDIA GeForce 280 GTX, it is a big handicap. Almost all memory accesses will be serialized because it is impossible to order all variables such that coalescing can be applied. Thus, real concurrency can only be realized on SM/block level.
Nevertheless, the shared memory can be used to cache the pointer array. Like mentioned in [5] Section 5.3, coalescing can be used to maximize the memory throughput when filling the cache. In order to prevent bank conflicts, the two dimensions are ordered such that the first dimension is used to distinguish access positions, while the second dimension is used to distinguish threads.
In order to gain some concurrency, the scheduling can be extended. Blocks are partitioned such that all threads of a block access inputs at the same position. Due to the fact that inputs are only changed after a step, inputs can also be cached in the shared memory. To avoid the pointer problem, the inputs can be copied directly into

```
if (*var[blockID][threadID][0]) (*var[blockID][threadID][1]) = →
    ((int)var[blockID][threadID][2]) * ((int)var[blockID][threadID][3]) + →
    ((int)var[blockID][threadID][4]) * ((int)var[blockID][threadID][5])
```

Listing 5.1: Vectorized Guarded Action Extracted from Listing 4.7

the pointer array instead of their address. Since all threads within a block access an input at the same position the redirect operator can be left out in the SIMT code. By applying this optimization to the matrix multiplication SIMT code shown in Listing 4.11, the SIMT code is changed as shown in Listing 5.1. The gain for this example is significant, because by using the shared memory almost all computations can be performed concurrently, while all computations are serialized without this optimization because of the serialized memory accesses. The same optimization could be done for labels in the immediate part.
Inputs and outputs are written/read every step by the host. Therefore, they should be arranged in separate arrays in order to maximize the memory throughput.
Inputs are constant for one step. Therefore, they can be stored in the constant memory which is cached and much faster for that reason.
Additionally, one could mention streams to parallelize data transfers and kernel execution by using streams. CUDA programs manage concurrency through streams which are sequences of operations. Within each stream, all operations are serialized while different streams may execute their operations out of order or concurrently with respect to another, i.e. executing data transfer 1 and kernel 1 in stream 1 and data transfer 2 and kernel 2 in stream 2 may lead to an overlapping of transfer 2 and kernel 1. Unfortunately, the other way around is also possible: transfer 1 overlaps kernel 2. But it is essential to preserve the ordering of the immediate and the delayed kernel as well as the data transfers. This can be done by using the *cudaThreadSynchronize* function which blocks the host until all streams are finished. In order to transfer inputs concurrently to a kernel execution, the input array has to be duplicated. One copy for the immediate kernel and one copy for the delayed kernel. Thus, new inputs can be transfered to the immediate inputs while the delayed kernel operates on its own copy of the old inputs. Hence, only the first initialization of the immediate inputs has to be serial. The outputs can also be transfered while executing the delayed kernel, because it does not write to the outputs due to the transformations. The update of the delayed inputs can be done while executing the immediate kernel. Thus, 2 streams are needed. The immediate kernel is assigned to stream 0 and the input copy to stream 1. Then *cudaThreadSynchronize* is called to make sure these two streams are finished. Afterwards, these streams are destroyed and two new streams are created. The output transfer and the immediate input transfer are assigned to stream 1 and the delayed kernel is assigned to stream 0. Then, *cudaThreadSynchronize* is called again. The streams are destroyed and two new streams are created. Then, the whole procedure starts again.

0	1	2	3	4	5	6	7
out[1][1]	out[1][0]	out[0][1]	out[0][0]				

Table 5.1: Output Array for the Matrix Multiplication Example

0	1	2	3	a
__tmp1	__tmp2	__tmp3	__init0	__ell0

Table 5.2: Variable Array for the Matrix Multiplication Example

Not considering the pointer problem, there would be many other optimizations which are not applied in this work. Labels are only written by delayed GAs. For that reason they are constant for the immediate kernel and can be stored at the constant memory. For the delayed GAs another copy has to be saved in the global memory. The shared memory could be used to cache the global memory. Unfortunately, the shared memory can only be accessed by threads within the same block and it is very small. Consequently, the set of needed variables should be partitioned into one set for every block. There would be still variables that are commonly used by blocks, so they should be stored in a separate array. Therefore, every block would have to cache its pointer array, variable array and the array of common variables.

5.2 Mapping

In order to derive the targets for the pointer array, a map from pointer array index to the variable array index respectively input/output array index has to be extracted. First, all needed arrays are allocated in the global memory. Then, an arbitrary order is chosen for the variables like shown in Table 5.1, 5.2 and 5.3. In general, there are two input arrays that look exactly the same. But in this example only the immediate kernel accesses inputs. Thus, only one instance of the inputs is needed. Afterwards, all GAs are processed and the names of variables which are accessed are extracted in chronological order. Finally, the list of names is processed and each name is compared to the order to get the right position in the variable/input arrays.

In order to reuse this mapping each step, the pointer array should be stored in the global memory. By using coalescing, the pointer array can be cached efficiently in the shared memory. The initialization can be done by a separate kernel which is executed in the beginning of the program. The initialization kernel for the matrix multiplica-

0	1	2	3	4	5	6	7
a[1][0]	b[0][1]	a[1][1]	b[1][1]	b[0][0]	b[1][0]	a[0][0]	a[0][1]

Table 5.3: Input Array for the Matrix Multiplication Example

tion example is shown in Listing 5.2. The initialization kernel needs the input array, variable array, output array and all pointer arrays. The kernel is started with the number of blocks of the immediate kernel plus the number of blocks of the delayed kernel. Each block contains only one thread that performs the copying ,because these operations are always serialized due to the memory access.
As described in Section 5.1, the inputs are handled separately. Inputs are stored directly into the pointer array. Therefore, each block first caches the inputs in the shared memory. Afterwards, one thread per block copies the values into the pointer array. An example is shown in Listing 5.3. Obviously, only the first block uses inputs and loads them into the pointer array. An explicit type conversion from type *int* to type *int** has to be performed in order to prevent type conflicts. This type conversion has to be performed vice versa in the SIMT code of the GAs.

```
__global__ void initializePointer(int *outputs, int *vars, int **b0p
    , int **b1p)
{
    switch (blockIdx.x +(blockIdx.y * blockDim.x))
    {
        case 0:
        {
            b0p[0] =  &(vars[3]);
            b0p[8] =  &(outputs[3]);
            b0p[1] =  &(vars[3]);
            b0p[9] =  &(outputs[2]);
            b0p[2] =  &(vars[3]);
            b0p[10] =  &(outputs[1]);
            b0p[3] =  &(vars[3]);
            b0p[11] =  &(outputs[0]);
            b0p[4] =  &(vars[4]);
            b0p[12] =  &(outputs[3]);
            b0p[5] =  &(vars[4]);
            b0p[13] =  &(outputs[2]);
            b0p[6] =  &(vars[4]);
            b0p[14] =  &(outputs[1]);
            b0p[7] =  &(vars[4]);
            b0p[15] =  &(outputs[0]);
        }
        case 1:
        {
            b1p[0] =  &(vars[0]);
            b1p[1] =  &(vars[4]);
            b1p[2] =  &(vars[4]);
            b1p[3] =  &(vars[1]);
            b1p[4] =  &(vars[3]);
            b1p[5] =  &(vars[0]);
            b1p[6] =  &(vars[3]);
            b1p[7] =  &(vars[2]);
            b1p[8] =  &(vars[3]);
            b1p[9] =  &(vars[0]);
            b1p[10] =  &(vars[1]);
            b1p[11] =  &(vars[3]);
            b1p[12] =  &(vars[2]);
            b1p[13] =  &(vars[3]);
        }
    }
}
```

Listing 5.2: Initialization Kernel for the Matrix Multiplication Example

```
b0[2][0] =  (int*)(inputs[2]);
b0[3][0] =  (int*)(inputs[5]);
b0[4][0] =  (int*)(inputs[3]);
b0[5][0] =  (int*)(inputs[7]);
b0[2][1] =  (int*)(inputs[2]);
b0[3][1] =  (int*)(inputs[4]);
b0[4][1] =  (int*)(inputs[3]);
b0[5][1] =  (int*)(inputs[6]);
b0[2][2] =  (int*)(inputs[0]);
b0[3][2] =  (int*)(inputs[5]);
b0[4][2] =  (int*)(inputs[1]);
b0[5][2] =  (int*)(inputs[7]);
b0[2][3] =  (int*)(inputs[0]);
b0[3][3] =  (int*)(inputs[4]);
b0[4][3] =  (int*)(inputs[1]);
b0[5][3] =  (int*)(inputs[6]);
b0[2][4] =  (int*)(inputs[2]);
b0[3][4] =  (int*)(inputs[5]);
b0[4][4] =  (int*)(inputs[3]);
b0[5][4] =  (int*)(inputs[7]);
b0[2][5] =  (int*)(inputs[2]);
b0[3][5] =  (int*)(inputs[4]);
b0[4][5] =  (int*)(inputs[3]);
b0[5][5] =  (int*)(inputs[6]);
b0[2][6] =  (int*)(inputs[0]);
b0[3][6] =  (int*)(inputs[5]);
b0[4][6] =  (int*)(inputs[1]);
b0[5][6] =  (int*)(inputs[7]);
b0[2][7] =  (int*)(inputs[0]);
b0[3][7] =  (int*)(inputs[4]);
b0[4][7] =  (int*)(inputs[1]);
b0[5][7] =  (int*)(inputs[6]);
```

Listing 5.3: Input Handling for the Matrix Multiplication Example

6 Synthesis

So far, the SIMT code is generated and a variable mapping is extracted. Now, the SIMT code has to be integrated in the corresponding kernels and the host code has to be generated.

6.1 Kernel Synthesis

The SIMT code is partitioned into an immediate and a delayed set which anon are partitioned into blocks. Thus, two kernels, which are structurally equal, are created for the SIMT code. The general structure is explained with help of the immediate kernel of the matrix multiplication example shown in Listing 6.1.

```
1  __global__ void MatrixMul_now_d(int *inputs_d,int **b_0_d)
2  {
3      switch (blockIdx.x +(blockIdx.y * blockDim.x))
4      {
5          case 0:
6          {
7              __shared__ int* b0[6][8];
8
9              if(threadIdx.x<8)
10             {
11                 b0[1][threadIdx.x] = b_0_d[1*8+threadIdx.x];
12                 b0[0][threadIdx.x] = b_0_d[0*8+threadIdx.x];
13
14             }
15
16             if(threadIdx.x<1)
17             {
18                 b0[2][0] =  (int*)(iInputs[2]);
19                 b0[3][0] =  (int*)(iInputs[5]);
20                 b0[4][0] =  (int*)(iInputs[3]);
21                 b0[5][0] =  (int*)(iInputs[7]);
22                 b0[2][1] =  (int*)(iInputs[2]);
23                 b0[3][1] =  (int*)(iInputs[4]);
24                 b0[4][1] =  (int*)(iInputs[3]);
25                 b0[5][1] =  (int*)(iInputs[6]);
```

```
26                  b0[2][2] =  (int*)(iInputs[0]);
27                  b0[3][2] =  (int*)(iInputs[5]);
28                  b0[4][2] =  (int*)(iInputs[1]);
29                  b0[5][2] =  (int*)(iInputs[7]);
30                  b0[2][3] =  (int*)(iInputs[0]);
31                  b0[3][3] =  (int*)(iInputs[4]);
32                  b0[4][3] =  (int*)(iInputs[1]);
33                  b0[5][3] =  (int*)(iInputs[6]);
34                  b0[2][4] =  (int*)(iInputs[2]);
35                  b0[3][4] =  (int*)(iInputs[5]);
36                  b0[4][4] =  (int*)(iInputs[3]);
37                  b0[5][4] =  (int*)(iInputs[7]);
38                  b0[2][5] =  (int*)(iInputs[2]);
39                  b0[3][5] =  (int*)(iInputs[4]);
40                  b0[4][5] =  (int*)(iInputs[3]);
41                  b0[5][5] =  (int*)(iInputs[6]);
42                  b0[2][6] =  (int*)(iInputs[0]);
43                  b0[3][6] =  (int*)(iInputs[5]);
44                  b0[4][6] =  (int*)(iInputs[1]);
45                  b0[5][6] =  (int*)(iInputs[7]);
46                  b0[2][7] =  (int*)(iInputs[0]);
47                  b0[3][7] =  (int*)(iInputs[4]);
48                  b0[4][7] =  (int*)(iInputs[1]);
49                  b0[5][7] =  (int*)(iInputs[6]);
50              }
51              __syncthreads();
52
53              if(threadIdx.x<8)
54              {
55                  if((*b0[0][threadIdx.x])) {
56                      (*b0[1][threadIdx.x]) = (((int)(b0[2][threadIdx.
                           x]) * (int)(b0[3][threadIdx.x])) + ((int)(b0
                           [4][threadIdx.x]) * (int)(b0[5][threadIdx.x])
                           ));
57                  }
58              }
59          }
60      }
61  }
```

Listing 6.1: Immediate Kernel of the Matrix Multiplication Example

Each kernel needs the corresponding input array and the pointer arrays as parameters like shown in Line 1. The SIMT code was extracted because all threads within a kernel have to use the same code. It is possible to make every thread execute another part of the code by introducing branches depending on the thread ID, but this leads to branch divergences which anon lead to a sequential execution of the threads

like described in Section 2.1.1. In contrast, threads in different blocks can execute different code by introducing branches depending on the block ID without any disadvantages. Thus, each block get its own case branch like shown in Line 5. First, the shared memory for the inputs and the pointer array is allocated within each case branch, which is illustrated in Line 7 and 8. Then, the array pointer is loaded from the global memory to the shared memory by using coalescing (Line 10 ff). Due to the fact that all blocks are executed by the same amount of threads and the blocks are not equal in the amount of needed threads, the block size has to be set to the maximum of needed threads per block. Thus, it has to be guaranteed that only the needed amount of threads execute the operations. Otherwise, array accesses may be out of bounds. Therefore, each code block within a case branch is surrounded by an if statement that checks the thread ID. This is illustrated in the example Line 10,16,20,57. Afterwards, the inputs are copied to the pointer array like described in Section 5.2. Only one thread can be used to copy the inputs because the sources and targets are disorganized as illustrated in Line 20. After all the memory transfers are done the threads have to be synchronized in order to ensure that the needed data is present (Line 55). Finally, the SIMT code is executed by the corresponding amount of threads. The same is done for all other immediate blocks. By now, the kernels for the immediate GAs, delayed GAs and the initialization of pointer arrays are generated. Only the function that organizes the data transfer and kernel launches is missing.

6.2 System Function

The system function is used to allocate the memory, it initiates the data transfers and launches the kernels. It also performs the initial step which consists of constant variable initializations due to the transformations. The general structure is described using Listing 6.2 for illustration.

```
1  void MatrixMul ()
2  {
3      int *outputs_d, *outputs_h;
4      int *ninputs, *dinputs, *inputs_h;
5      int **initPtr;
6      int *vars_h;
7      int *vars_d;
8      int **b0p_d,**b1p_d;
9
10     cudaHostAlloc((void**)&inputs_h, 8*sizeof(int),
           cudaHostAllocPortable);
11     cudaHostAlloc((void**)&outputs_h, 4*sizeof(int),
           cudaHostAllocPortable);
12
13     cudaStream_t stream[2];
14
```

```
15      vars_h = new int[5];
16      initPtr = new int* [2];
17
18      cudaMalloc((void**) &b0p_d, 6*8*sizeof(int*));
19      cudaMalloc((void**) &b1p_d, 14*1*sizeof(int*));
20      cudaMalloc((void**) &vars_d, 5*sizeof(int));
21      cudaMalloc((void**) &outputs_d, 4*sizeof(int));
22      cudaMalloc((void**) &ninputs, 8*sizeof(int));
23      cudaMalloc((void**) &dinputs, 8*sizeof(int));
24
25      for(int i=0; i<5; i++) vars_h[i] = 0;
26      READ_inputs(inputs_h)
27      initPtr[0] = &vars_h[3];
28      initPtr[1] = &vars_h[4];
29
30      {
31          (*initPtr[0]) = 0;
32          (*initPtr[1]) = 1;
33      }
34
35      cudaMemcpy(vars_d, vars_h, 5*sizeof(int), cudaMemcpyHostToDevice
            );
36
37      cudaMemcpyToSymbol("iInputs", inputs_h, 8*sizeof(int), 0,
            cudaMemcpyHostToDevice);
38      cudaMemcpy(outputs_d, outputs_h, 4*sizeof(int),
            cudaMemcpyHostToDevice);
39
40      initializePointer<<<2,1>>>(outputs_d, vars, b1_d, b1p_d);
41
42      while(true)
43      {
44          cudaStreamCreate(&stream[0]);
45          cudaStreamCreate(&stream[1]);
46
47          cudaMemcpyToSymbolAsync("dInputs", inputs_h, 8*sizeof(int),
                0, cudaMemcpyHostToDevice, stream[1]);
48          MatrixMul_now_d<<<1,8,0,stream[0]>>>(ninputs,b0p_d);
49
50          cudaThreadSynchronize();
51          cudaStreamDestroy(stream[0]);
52          cudaStreamDestroy(stream[1]);
53          cudaStreamCreate(&stream[0]);
54          cudaStreamCreate(&stream[1]);
55
```

```
56          cudaMemcpyAsync(outputs_h, outputs_d, 4*sizeof(int),
                cudaMemcpyDeviceToHost, stream[1]);
57
58          {
59              READ_inputs(inputs_h)
60              cudaMemcpyToSymbolAsync("iInputs", inputs_h, 8*sizeof(
                    int), 0, cudaMemcpyHostToDevice, stream[1]);
61              MatrixMul_nxt_d<<<1,1,0,stream[0]>>>(dinputs,b1p_d);
62          }
63          cudaThreadSynchronize();
64          cudaStreamDestroy(stream[0]);
65          cudaStreamDestroy(stream[1]);
66          WRITE_outputs(outputs_h)
67      }
68  }
```

Listing 6.2: System Function Generated for the Matrix Multiplication Example

First, the needed pointers are defined in Line 3 to 8. One for the outputs on the device, two for the inputs on the device, one array of pointer for the initial step, one pointer to the variables on the host and one to the variables on the device and finally the pointer arrays for every block. A pointer array is needed for the initial step. Therefore, it is possible to reuse the extracted pointer mapping described in Section 5.2. The variables are also allocated on the host in order to initialize them.
Afterwards, the arrays for inputs and outputs on the host are defined like illustrated in Line 10 and 11. These are used for the data transfer between the host and the device and serve as the new system interface. They are allocated in the page-locked host memory using the *cudaHostAlloc* function in order to transfer them asynchronously.
In Line 13 the stream array used for data transfer and kernel execution parallelization is defined.
Then, memory for the variables and the init pointer array are dynamically allocated.
Next, Line 18 to 25 reveal the memory allocation on the device. The memory allocation function gets a pointer to a void pointer where the address of the first byte of allocated memory is saved to and the amount of memory in bytes which should be allocated in the global memory space. Afterwards, all variables are initialized by 0.
Then, the initial step is performed like shown in Line 29 to 36. First, all inputs have to be read (Line 29). Then, the init pointer array is initialized using the mapping (Line 30,31). Afterwards, the initial step is performed (Line 34,35).
Next, the device memory is initialized by transferring the initialized data from the host to the device using *cudaMemcpy* like illustrated in Line 38 to 41. *cudaMemcpyToSymbol* has to be used to transfer data from the host to the constant device memory. Only the inputs for the delayed kernel are not initialized at this point because this is always done while executing the immediate kernel like described in Section 5.1. *cudaMemcpy* needs a pointer to the target, a pointer to the source, the number of bytes

that should be copied and the direction which can be *cudaMemcpyHostToDevice* for data transfer from host memory to device memory, *cudaMemcpyDeviceToDevice* for data transfer from device memory to device memory or *cudaMemcpyDeviceToHost* for data transfer from device memory to host memory. Note that the device memory is always global memory. *cudaMemcpyToSymbol* has one more parameter than *cudaMemcpy*, which is the offset used to specify the begin of data to be transfered.
Then, the pointer initialization kernel is launched to initialize the pointer arrays. Altogether, there are 2 blocks in the immediate and delayed kernel. Therefore, the pointer initialization kernel uses 2 blocks. The initialization within a block can't be executed concurrently because it would be very difficult to create corresponding SIMT code. Thus, every block contains only one thread as shown in Line 43. Fortunately, the initialization is done only once per system execution. As mentioned in Section 5.2, the parameters are the output array in device memory, the variable array in device memory and the needed pointer arrays.
So far, the initialization of the system is generated. Next, the functionality is implemented. As shown in Line 45, the kernel launches are surrounded by an infinite loop due to the fact that GAs are used to model reactive systems which are nonterminating. In general, two parts can be identified. On the one hand, there is the immediate kernel launch and the data transfer from immediate inputs to delayed inputs and on the other hand, there is the delayed kernel and the output and input data transfers.
First, the two needed streams are created by the *cudaStreamCreate* function in Line 47 and 48. Afterwards, the data transfer from immediate inputs to delayed inputs is assigned to stream[0] (Line 50). The bandwidth between device memory and device memory is higher than the bandwidth between the host memory and the device memory. Thus, instead of copying the data from the host to the device, the data could be copied from device to device. But a *cudaMemcopyDeviceToDevice* data transfer can not run concurrently to a kernel execution. Therefore, it is better to use *cudaMemcpyHostToDevice* data transfer which is slower but it can run in parallel to the kernel execution and is hidden by this. The launch of the immediate kernel is assigned to stream[1] (Line 51). Thus, the data transfer and the kernel execution can run in parallel.
Then, *cudaThreadSynchronize* is called in order to make sure that the immediate kernel finished. Next, both streams created for the first part are destroyed and two new streams are created for the second part as shown in Line 54 to 57.
Afterwards, the output data transfer is assigned to stream[1] (Line 59). The inputs for the next step are read (Line 60) and they are transfered to the device by stream[1] (Line 61) and the delayed kernel is executed by stream[0] (Line 62). Afterwards, *cudaThreadSynchronize* is called again (Line 64) and both streams are destroyed. Finally, the output macro is called as shown in Line 67.
By now, the synthesis is finished, but there are many improvements which could be applied in order to speed up the computation. They are just mentioned in Chapter 7.

7 Further Improvements

As mentioned in Chapter 6, there are many further improvements to speed up the execution of GAs, which have not been applied in this work. They aim the extraction of concurrency and a better use of the different memory types. The more concurrency can be extracted, the more threads respectively blocks can be created which is necessary to use the whole capacity of a GPU. Usage of the global memory should be minimized due to the high latency. Instead the shared memory ,which has a very low latency or one of the other two cached memories, should be used. Some improvements are described in the following :

- By using the extended finite state machine (EFSM) it is possible to allocate GAs to different states. Therefore, dependencies that were extracted not considering the state may be broken. Thus, more concurrency can be extracted which leads to more threads and a better use of the parallelism of GPUs.
- By identifying variables that are constant for different states, these variables could be stored in the cached constant memory to reduce the usage of the global memory.
- By using heuristics to identify variable orderings and access functions as described in Chapter 5, the usage of pointer could be prevented and the variables itself could be cached in the shared memory. Thus, less shared memory is used per block and it is possible to handle greater systems and more threads per block. Additionally, a variable access leads to only one access to the shared memory instead of an access to the shared memory and one to the global memory. Identifying such orderings and formulas is very difficult because the independent threads that perform the same operations at the same positions have to be partitioned once more to be able to compute such a formula anyway. Additionally, bank conflicts have to be considered defining a variable ordering.
- The dependencies between delayed GAs can be ignored when storing all read values in the pointer array instead of their address. Thus, each access works on its own copy of the current data and the values in the global memory can be changed without any effect to other GAs. Consequently, all GAs can form independent threads. The vectorization can be tried and many blocks can be generated which leads to a more efficient execution of the delayed GAs. Additionally, the scheduling can be omitted which prevents the introduction of additional GAs for the cycle elimination. This improvement would not change the results of the benchmark shown in 8 significantly due to the small number of delayed GAs in this benchmark.

The first improvement is not only used to gain more concurrency. The use of the EFSM make the scheduling of some programs possible which cannot be scheduled without differentiating states. The second improvement is a very simple one, which is used to reach less memory latency by using a cached memory. Unfortunately, there are only 16 KB of constant memory. Thus, there is few space left because the inputs are also stored in this memory type. The third improvement is very complex one. General vectorization is quite complex, but for CUDA the graphics device specific properties like memory latencies and memory access patterns (e.g. coalescing) have to be considered. As mentioned in Chapter 8, this improvement is the most effective. By using a better vectorization, larger systems can be handled and the performance increases significantly.

8 Comparison

8.1 Comparison to C

CUDA programs are extended C programs which are partially executed on a graphics device. CUDA uses a special form of parallelism in order to speedup the computation. Unfortunately, the generated code is not as efficient as hand-written code because it is very difficult to vectorize code automatically. Additionally, there are many properties of CUDA, like the fact that pointers may be assumed to point to the global memory instead of faster memories (see Section 5.1), that impede the automated code generation.
In the following, a CUDA program and a C program are compared due to the similarity. The general comparison is illustrated by the matrix multiplication example introduced in Section 2.2.1. First, the size of the programs is regarded. Afterwards, the performance is compared. Finally, several other problems which appear in the automated code generation are discussed.

8.1.1 Program Size

The source code of the CUDA program is much larger than the pure C code due to the memory initializations and memory transfers which have to be done in the CUDA program. For instance, the matrix multiplication example contains about 252 lines of CUDA code and it has an object-file size of 126.6 KB, while the C implementation needs only 79 lines of code and it has an object-file size of 8.1 KB. This huge difference is caused by the overhead in the CUDA program which is needed by the memory management and the vectorization.

8.1.2 Performance

CUDA programs use the massive parallelism to gain performance while standard C programs can use the higher clock-rates as well as a complex cache hierarchy to gain performance.
Unfortunately, CUDA programs need vectorized code. It is very difficult to extract enough independent GAs that can run in parallel to use all SPs. The previously described approach uses pointer, which are stored in the global memory and cached in

the shared memory, to vectorize GAs or lists of GAs which perform the same operations at the same position. Therefore, it is possible to gain relatively huge vectorized sets of GAs. For a 9x9 matrix multiplication, 14 blocks with 16 threads each can be extracted for the immediate kernel. Thus, about the half amount of SPs can be used on a NVIDIA GeForce 280 GTX for the immediate kernel not considering the memory limitations. The delayed kernel consists of only one thread. Therefore, only one SP is used. The initializing kernel consists of 15 blocks with one thread each. Due to the organization of threads into warps only one SP per SM can be used. According to [4] Chapter 5, there should be at least 64 threads per block in order to hide just the register latencies. Hence, it is not possible to utilize the capacity of a GPU due to the small amount of threads.
Unfortunately, the NVCC compiler cannot distinguish the target memory type if a pointer to different memory types is dynamically chosen. Therefore, all targets (variables) have to be stored in the global memory. The global memory supports parallelism only if the needed data is arranged within the same segment, which is not possible for the execution of GAs. Thus, many memory accesses are serialized.
In order to speedup the computation, the inputs are stored in the constant memory which is cached. In the beginning of the computation the pointer array is transfered from the global memory to the shared memory using coalescing. Pointer to inputs are not copied, instead the input values are copied from the constant memory. Thus, the operations working on inputs can run in parallel. Therefore, the matrix multiplication is well suited.
The comparison between the 9x9 matrix multiplication executed on the CPU and on the GPU shows that the GPU is significantly slower than the CPU. While the CPU takes only 234 milliseconds for 10 000 steps, the GPU takes 3 573 milliseconds. Unfortunately, this difference does not shrink according to the number of performed steps. The time needed for 10 000 additional steps amounts about 3 600 ms on the GPU while it amounts about 225 ms on the CPU as shown in Table 8.1.

In spite of using the constant memory and the shared memory, the memory latency slows down the whole computation. As shown in Table 8.2, the input and output transfer between the host and the device takes about 23.15% GPU time. Thus, about 827 ms of 3 573 ms are used for input and output data transfer in the matrix multiplication with 10 000 steps. Within these 827 ms the CPU could perform more than 30 000 steps while the GPU has not done any computation. The time wasted waiting for the data transfer between the GPU and the global memory is even not considered.

Due to the chosen structure and the equality of GAs that form a block, a small number of branch divergences is achieved. Branch divergences can only occur if the a GA is not executed in a step due to the control flow while other GAs within the same block are executed or whole threads are not used by a block. Therefore, only about 3.2 % of the branches diverge as shown in Table 8.3.

Consequently, the low performance is caused by the huge memory latency as well as the small number of threads. The immediate kernel has only one active block per SM

Steps	GPU	increase	CPU	increase
10 000	3 573		234	
20 000	7 144	3 571	468	234
30 000	10 749	3 605	687	219
40 000	14 446	3 697	920	233
50 000	18 034	3 588	1 139	219
60 000	21 762	3 728	1 373	234
70 000	25 319	3 557	1 591	218
80 000	29 063	3 744	1 826	235
90 000	32 760	3 697	2 059	233
100 000	36 286	3 526	2 293	234

Table 8.1: Measurements of the 9x9 Matrix Multiplication on a NVIDIA GeForce GTX 280 (GPU) and on a Intel Atom 330 @ 2x1,6 GHz (CPU) in Milliseconds Depending on the Performed Amount of Steps.

Method	#Calls	%GPU time	instruction throughput
matmult_now	10 000	74.01	0.101235
matmult_next	9 999	2.82	0.00536741
initializePointer	1	0	0.245984
memcpyHostToDevice	4	0	
memcpyHostToDeviceAsync	19 999	21.28	
memcpyDeviceToHostAsync	10 000	1.87	

Table 8.2: Instruction Throughput and Percentage of GPU Time used for the 9x9 Matrix Multiplication Measured by the CUDA Visual Profiler on a NVIDIA GeForce GTX 280

Method	branches	branch divergences
matmult_now	478676	15824
matmult_nxt	3070	0
initializePointer	15	0

Table 8.3: Branches of the 9x9 Matrix Multiplication Analyzed by the CUDA Visual Profiler on a NVIDIA GeForce GTX 280

out of 8 and 16 active threads per block out of 1 024 like illustrated in the occupancy analyses 8.1 Line 17 and 18. Unfortunately, it is not possible to launch much more threads per SM or to have more than one active block per SM due to the high shared memory occupancy. In the following, only the immediate kernel is regarded because it takes about 75% GPU time. The immediate kernel uses 84% of the shared memory using only 16 threads per block as illustrated in Line 16. Only 0.25 % of the registers are used. The maximum overall occupancy is just 3.125% which is caused by the block-size as shown in Line 10, 22 and 34. But the block-size can't be significantly increased due to the shared memory occupancy. Additionally, increasing the block-size would reduce the amount of blocks which also inhibits the occupancy.

```
1  Occupancy analysis for kernel ' initializePointer '  for context 'Session1 : Device_0 : →
       Context_0' :
2  Kernel details : Grid size: 15 x 1, Block size: 1 x 1 x 1
3  Register Ratio       = 1   ( 16384 / 16384 ) [60 registers per thread]
4  Shared Memory Ratio = 0.125 ( 2048 / 16384 ) [88 bytes per Block]
5  Active Blocks per SM    = 4 : 8
6  Active threads per SM   = 4 : 1024
7  Occupancy       = 0.125   ( 4 / 32 )
8  Max achieved occupancy  = 0.03125 (on 15 SMs)
9  Min achieved occupancy = 0  (on 15 SMs)
10 Occupancy limiting factor    = Block−Size
11 Warning: Grid Size (15) is less than number of available SMs (30).
12
13 Occupancy analysis for kernel 'matmult_now' for context 'Session1 : Device_0 : →
       Context_0' :
14 Kernel details : Grid size: 14 x 1, Block size: 16 x 1 x 1
15 Register Ratio       = 0.25   ( 4096 / 16384 ) [60 registers per thread]
16 Shared Memory Ratio = 0.84375 ( 13824 / 16384 ) [13376 bytes per Block]
17 Active Blocks per SM    = 1 : 8
18 Active threads per SM   = 16 : 1024
19 Occupancy       = 0.03125   ( 1 / 32 )
20 Max achieved occupancy  = 0.03125 (on 14 SMs)
21 Min achieved occupancy = 0  (on 16 SMs)
22 Occupancy limiting factor    = Block−Size
23 Warning: Grid Size (14) is less than number of available SMs (30).
24
25 Occupancy analysis for kernel 'matmult_nxt' for context 'Session1 : Device_0 : →
       Context_0' :
26 Kernel details : Grid size: 1 x 1, Block size: 1 x 1 x 1
27 Register Ratio       = 0.03125   ( 512 / 16384 ) [8 registers per thread]
28 Shared Memory Ratio = 0.03125 ( 512 / 16384 ) [92 bytes per Block]
29 Active Blocks per SM    = 1 : 8
30 Active threads per SM   = 1 : 1024
31 Occupancy       = 0.03125   ( 1 / 32 )
32 Max achieved occupancy  = 0.03125 (on 1 SMs)
```

```
33 Min achieved occupancy = 0 (on 29 SMs)
34 Occupancy limiting factor = Block-Size
```

Listing 8.1: Occupancy Analysis of the 9x9 Matrix Multiplication by the CUDA Visual Profiler on a NVIDIA GeForce GTX 280

8.2 Comparison to Manually Optimized CUDA Code

In order to understand the possible performance of executing GAs using CUDA, the generated CUDA program is compared to a manually optimized CUDA program which uses just a smarter vectorization strategy. Therefore, the program size and the performance is compared again.

8.2.1 Program Size

The source code of the optimized CUDA program is much shorter than the source code of the generated CUDA program due to a missing pointer initialization kernel, other missing data transfers from the global memory to the shared memory and missing redirection operations. Therefore, the source code of the optimized program contains only 142 lines of code with an object file size of about 69 KB. Thus, the size is shrunken to about the half of the size of the generated program.

8.2.2 Performance

As shown in Table 8.4, the computation is about 30% faster using the manually optimized code instead of the generated one.

There is a big difference considering the GPU usage. While the immediate kernel of the generated code consumes about 74% the immediate kernel of the manually optimized program consumes only about 18% as shown in Table 8.5. In the optimized program, the problem of the low bandwidth between the host and device becomes a big problem because the data transfer of the inputs take about 64% GPU time. I.e. computing 10 000 steps of the matrix multiplication wastes about 1 571 ms for input transfer. Thus, the pure computation takes only about 893 ms which is quiet good considering that only one block and only 81 threads are used. The instruction throughput of the optimized program is less than the instruction throughput of the generated program due the smaller number of instructions in the optimized program. There are less instructions due to the missing redirection and the missing data transfers from the global memory to the shared memory, which is coalesced in the generated program.

steps	generated code	manual code
10 000	3 573	2 465
20 000	7 144	4 930
30 000	10 749	7 612
40 000	14 446	10 016
50 000	18 034	12 620
60 000	21 762	15 397
70 000	25 319	17 847
80 000	29 063	20 499
90 000	32 760	23 041
100 000	36 286	25 646

Table 8.4: Comparison of Elapsed Time in ms of the 9x9 Matrix Multiplication Generated by the Described Algorithm and a Manually Optimized Version Depending on the Performed Steps.

Method	#Calls	%GPU time	instruction throughput
matmult_now	10 000	18.73	0.00639983
matmult_next	9 999	11.87	0.000451878
memcpyHostToDevice	4	0	
memcpyHostToDeviceAsync	19 999	63.74	
memcpyDeviceToHostAsync	10 000	5.63	

Table 8.5: Instruction Throughput and Percentage of GPU Time Used for the 9x9 Matrix Multiplication Measured by the CUDA Visual Profiler on a NVIDIA GeForce GTX 280

Method	branches	branch divergences
matmult_now	31 845	1 844
matmult_nxt	1 449	0

Table 8.6: Branches of the 9x9 Matrix Multiplication Analyzed by the CUDA Visual Profiler on a NVIDIA GeForce GTX 280

Due to the smaller number of blocks, kernels and instructions, there are less branches in the optimized program as shown in Table 8.6. The optimized program uses about 6% of the branches used in the generated program while the amount of branch divergences decrease to about 11%.

Like illustrated in the occupancy analysis 8.2, only 9% of the shared memory is used by the optimized program while the generated program uses about 84%. Therefore, much larger systems can be handled by using a better vectorization. Thus, the GPU can be used more efficiently. The maximal achieved occupancy of the optimized program is about 9% which is about 3 times better than the occupancy of the generated program. Considering the small system size, the performance of the optimized program could reach the performance of the C program for large systems with small interfaces.

```
1  Occupancy analysis for kernel 'matmult_now_d' for context 'Session1 : Device_0 : →
       Context_0' :
2  Kernel details : Grid size: 3 x 1, Block size: 9 x 9 x 1
3  Register Ratio         = 0.375   ( 6144 / 16384 ) [15 registers per thread]
4  Shared Memory Ratio = 0.09375 ( 1536 / 16384 ) [28 bytes per Block]
5  Active Blocks per SM    = 3 : 8
6  Active threads per SM   = 243 : 1024
7  Occupancy         = 0.28125   ( 9 / 32 )
8  Max achieved occupancy  = 0.09375 (on 3 SMs)
9  Min achieved occupancy = 0  (on 27 SMs)
10 Occupancy limiting factor    = Block−Size
11 Warning: Grid Size (3) is less than number of available SMs (30).
12
13 Occupancy analysis for kernel 'matmult_nxt_d' for context 'Session1 : Device_0 : →
       Context_0' :
14 Kernel details : Grid size: 1 x 1, Block size: 1 x 1 x 1
15 Register Ratio         = 0.03125   ( 512 / 16384 ) [2 registers per thread]
16 Shared Memory Ratio = 0.03125 ( 512 / 16384 ) [24 bytes per Block]
17 Active Blocks per SM    = 1 : 8
18 Active threads per SM   = 1 : 1024
19 Occupancy         = 0.03125   ( 1 / 32 )
20 Max achieved occupancy  = 0.03125 (on 1 SMs)
21 Min achieved occupancy = 0  (on 29 SMs)
22 Occupancy limiting factor    = Block−Size
```

23 Warning: Grid Size (1) is less than number of available SMs (30).

Listing 8.2: Occupancy Analysis of the Optimized 9x9 Matrix Multiplication by the CUDA Visual Profiler on a NVIDIA GeForce GTX 280

All in all, this CUDA synthesis is not able to handle such big systems that the performance of the CUDA program can touch the performance of the corresponding C program. Perhaps, the new GPU generation implementing compute capability 2.0 is able to increase the performance significantly by hiding the global memory latency using a cached global memory. But using better vectorization algorithms would lead to a quiet performant CUDA programs.

9 Conclusion

Synchronous guarded actions are not well suited to synthesize CUDA code. Even if GAs can model different forms of concurrency, it is very difficult to extract them explicitly. In particular, CUDA programs need a special form of concurrency to act efficiently. SIMT code has to be generated. Furthermore, the SIMT code generation has to consider architecture specific properties like the latency of different memory types and different divisions of independent GAs into threads and blocks. The vectorization is the most difficult part of the synthesis. There are some methods for automated loop vectorization, e.g. the one described by Kazuaki Ishizaki and Hideaki Komatsu in [3]. But synchronous systems modeled by GAs do not contain loop constructs. The described vectorization method is just a simple one. Applying better strategies would lead to CUDA programs which could reach the performance of C programs or even better for large systems with small interfaces. For small systems it makes no sense to use the graphics device because of the huge overhead.
All in all, GAs can be executed efficiently on graphics devices using CUDA by applying intelligent vectorization algorithms and considering the architecture specific limitations. Additionally, the ability to use the graphics device and the CPU concurrently due to the asynchronicity, make the use of CUDA at least to a good extension to the standard C programs.

Bibliography

[1] Muthu Manikandan Baskaran, J. Ramanujam, and P. Sadayappan. Automatic c-to-cuda code generation for affine programs. In *CC 2010*, pages 244 – 263. Springer-Verlag, 2010.

[2] Daniel Boudisch, Jens Brandt, and Klaus Schneider. Multithreaded code from synchronous programs: Extracting independent threads for openmp.

[3] Kazuaki Ishizaki and Hideaki Komatsu. A loop parallelization algorithm for hpf compilers.

[4] NVIDIA. *NVIDIA CUDA Programming GUIDE 2.3.1*, 2009.

[5] NVIDIA. *NVIDIA CUDA Programming GUIDE 3.0*, 2010.

[6] Klaus Schneider. The synchronous programming language quartz, 2008.

[7] John A. Stratton, Sam S. Stone, and Wen mei W. Hwu. Mcuda: An efficient implementation of cuda kernels for multi-core cpus. In *LCPC 2008*, pages 16 – 30. Springer-Verlag, 2008.