Mikko Hulkkonen

Graphics Processing Unit Utilization in Circuit Simulation

School of Electrical Engineering

Thesis submitted for examination for the degree of Master of Science in Technology.

Espoo 17.8.2011

Thesis supervisor:

Prof. Martti Valtonen

Thesis instructor:

Lic.Sc. (Tech.) Jarmo Virtanen

Aalto University
School of Electrical
Engineering

AALTO UNIVERSITY
SCHOOL OF ELECTRICAL ENGINEERING

ABSTRACT OF THE
MASTER'S THESIS

Author: Mikko Hulkkonen

Title: Graphics Processing Unit Utilization in Circuit Simulation

Date: 17.8.2011 Language: English Number of pages:8+45

Department of Radio Science and Technology

Professorship: Circuit theory Code: S-55

Supervisor: Prof. Martti Valtonen

Instructor: Lic.Sc. (Tech.) Jarmo Virtanen

Graphics processing units (GPU) of today include hundreds of multi-threaded, multicore processors and a complex, high-bandwidth memory architecture, making them a good alternative to speed up general-purpose parallel computation where large data quantities are processed with same functions. Some successful applications of GPU computation have also been introduced in the field of circuit simulation. The objective of this thesis is to examine the GPU's computing potential in the APLAC circuit simulation software. The realization of a diode model on a GPU device is also presented.

The nonlinear diode model was implemented on NVIDIA's Compute Unified Device Architecture (CUDA), that is a single-instruction, multiple-thread (SIMT) architecture. A CUDA device was programmed using the CUDA C application programming interface, which is an extension of the standard C language.

The test results revealed that due to the diode's simple nonlinearity, its evaluation is computationally too light to gain any speed benefit from the GPU's computation power. The required modifications to the circuit analysis structure and data handling resulted in a marginally longer total simulation time than initially. However, when the diode model is made more complex by multiplying its evaluation, the CUDA implementation is faster than the original model. This gives a rough estimate of how complex a model benefits from the GPU computation.

Although, the diode model evaluation was not faster on the GPU, this implementation is a good foundation for future CUDA applications in APLAC. The next of these applications will be the computationally more complex BSIM3 transistor model, which will most likely benefit from the computing power of GPU devices.

Keywords: CUDA, circuit simulation, diode model, parallel computing

AALTO-YLIOPISTO
SÄHKÖTEKNIIKAN KORKEAKOULU

DIPLOMITYÖN
TIIVISTELMÄ

Tekijä: Mikko Hulkkonen

Työn nimi: Grafiikkaprosessorin Hyödyntäminen Piirisimuloinnissa

Päivämäärä: 17.8.2011 Kieli: Englanti Sivumäärä:8+45

Radiotieteen ja -tekniikan laitos

Professuuri: Piiriteoria Koodi: S-55

Valvoja: Prof. Martti Valtonen

Ohjaaja: TkL Jarmo Virtanen

Nykypäivän grafiikkaprosessorit (GPU) koostuvat sadoista monisäikeisistä, moniytimisistä prosessoreista ja monimutkaisesta korkean kaistanleveyden muistiarkkitehtuurista. Tämän vuoksi niistä on tullut hyvä vaihtoehto nopeuttamaan rinnakkaistettua yleislaskentaa, jossa suuria datamääriä käsitellään samoilla funktioilla. Myös piirisimuloinnin alalla on esitelty menestyksellisiä GPU-laskennan sovellutuksia. Tämän opinnäytteen tavoitteena on tutkia GPU-laskennan mahdollisuuksia APLAC-piirisimulointiohjelmassa. Työssä esitellään myös diodimallin laskennan toteutus GPU:lla.

Epälineaarinen diodimalli toteutettiin NVIDIAn CUDA-arkkitehtuurilla, joka on niin sanottu SIMT-arkkitehtuuri (single-instruction, multiple-thread) eli yksi käsky suoritetaan kerrallaan usealle säikeelle. CUDA-laite ohjelmoitiin CUDA C -ohjelmointirajapinnalla, joka on standardin C-kielen laajennus.

Testitulokset paljastivat että diodin yksinkertaisesta epälineaarisuudesta johtuen sen laskenta on liian kevyt, jotta GPU:n tehokkuudesta olisi mitään nopeusetua. Vaadittavat muutokset piirianalyysin rakenteeseen sekä datan hallintaan johtivat marginaalisesti alkuperäistä pidempään kokonaissimulointiaikaan. Kun diodimallia monimutkaistetaan moninkertaistamalla sen laskenta, CUDA-toteutus on nopeampi kuin alkuperäinen malli. Tämä antaa karkean arvion siitä kuinka monimutkainen malli hyötyy GPU-laskennasta.

Vaikka diodimalli ei ollutkaan nopeampi GPU:lla, tämä toteutus on hyvä perusta tuleville CUDA-sovelluksille APLACissa. Näistä seuraavana on huomattavasti monimutkaisempi BSIM3-transistorimallin laskenta, joka mitä todennäköisimmin hyötyy GPU:n laskentatehosta.

Avainsanat: CUDA, diodimalli, piirisimulointi, rinnakkaislaskenta

Preface

I want to thank my instructor Jarmo Virtanen and supervisor Martti Valtonen. Also, my thanks go to Luis Costa for proofreading and the entire Circuit Theory group for all the support and ideas I've received throughout this work. Special thanks go to my beloved wife Mari-Anne.

Otaniemi, 17.8.2011

Mikko Hulkkonen

Contents

Symbols and abbreviations

Symbols

a	relative device area of a diode
α	energy gap temperature dependency factor
β	energy gap temperature dependency factor
C_d	diode model shunt capacitance
E_dc	the DC voltage of a voltage source
E_g	energy gap of a diode
η	emission coefficient
η_BV	high reverse breakdown ideality factor
η_BVL	low reverse breakdown ideality factor
η_R	recombination current emission coefficient
$g(u_0)$	conductance of the diode iteration model at operating voltage u_0
I_d	the diode model current
I_BV	high reverse breakdown current
I_BVL	low reverse breakdown current
I_KF	high-injection knee current
I_S	saturation current
I_SR	recombination constant current
I_t	thermal current of the diode model
$i(u_0)$	current through the diode at the operating voltage u_0
$J(u_0)$	current source of the diode iteration model at the operating voltage u_0
k	the Boltzmann constant $\approx 1.380662 \times 10^{-23}$ J K^{-1}
k_g	diode model's recombination current dependency on the depletion layer width
N	the number of parallel processors available in the Amdahl's law
P	the fraction of the total execution time that can be parallelized in the Amdahl's law
q	the elementary charge $\approx 1.6021892 \times 10^{-19}$ C
p_t	saturation current temperature exponent
R	the internal serial resistance of a voltage source
R_s	diode model serial resistance
S	the maximum achievable speedup according to the Amdahl's law
T	diode temperature
T_nom	nominal temperature of a diode
T_RS1	linear temperature coefficient of R_S
T_RS2	quadratic temperature coefficient of R_S
t_BV1	linear temperature coefficient of V_BV
t_BV2	quadratic temperature coefficient of V_BV
t_KF	temperature coefficient of current I_KF
u_0	operating voltage of the diode iteration model
u_tJ	thermal potential of the diode model's thermal node $\mathrm{n_J}$
V_BV	reverse breakdown voltage
V_j	junction potential

Operators

$\frac{\partial}{\partial u}$	partial derivative relative to variable u

Abbreviations

ALU	arithmetic logic unit
API	application programming interface
APLAC	A circuit simulation software originally developed by professor Martti Valtonen in Helsinki University of Technology [1]
CUBLAS	CUDA Basic Linear Algebra Subprograms
CUDA	Compute Unified Device Architecture
CUDPP	CUDA Data Parallel Primitives
DC	direct current
DRAM	dynamic random access memory
FLOPS	floating-point operations per second
GPGPU	general-purpose graphics processing unit
GPU	graphics processing unit
ISA	instruction set architecture
JIT	just-in-time
NVCC	The CUDA compiler driver
PTX	parallel thread execution
SIMD	single-instruction, multiple-data
SIMT	single-instruction, multiple-thread
SM	streaming multiprocessor
SpMV	sparse matrix-vector multiplication

1 Introduction

The need for faster, high-resolution computer graphics processing has driven the development of graphics processing units (GPU) to a point where they can be utilized as powerful general-purpose computational devices. Compared to common central processing units (CPU), the GPU's advantage is its capability to process large amounts of data in parallel. This has introduced a new possibility in many fields requiring heavy data processing: blocks or algorithms of software that process large data quantities with same functions can now be accelerated by implementing them on a GPU.

Several successful applications of general-purpose GPU computation have already been introduced in the fields of computational finance and biology, physics, electromagnetic and electronic circuit simulation. The objective of this thesis is to examine the potential of GPU computation in the APLAC circuit simulation software.

Modern GPUs are based on a single-instruction, multiple-thread (SIMT) architecture. The two leading SIMT device platforms are NVIDIA's Compute Unified Device Architecture (CUDA) and AMD/ATI's Stream. Several research papers present CUDA as the more efficient one of the two [2]. Consequently, also this thesis focuses on CUDA architecture.

CUDA is based on streaming multiprocessors (SM), each of which consists of multiple cores or processors. Each SM has its own control logic and internal memory spaces. A CUDA device also has other memory spaces common to all multiprocessors. Overall, the CUDA memory structure and handling is highly sophisticated, allowing efficient processing of large data quantities when utilized properly.

Computation on the device is carried out by executing CUDA-specific functions called kernels. A CUDA kernel, when launched, is executed on a number of threads in parallel by the multiprocessors and cores of a device. The CUDA device code can be programmed on various application programming interfaces (API). Two of the most commonly used are CUDA C and CUDA driver API. CUDA C follows the standard C language with a set of extensions used for kernel configurations and memory handling. Also, this thesis focuses on CUDA C. The CUDA driver API gives more freedom to the programmer but at the same time, requires a lot more expertise.

This thesis investigates some previous GPU applications in the field of circuit simulation. After evaluating these applications also from the viewpoint of APLAC, a suitable object for APLAC's GPU development is found. The chosen program block is the diode model evaluation.

A diode is a nonlinear component that has exponential behaviour. Its model in APLAC is also capable of modeling the component's characteristics under dynamically changing temperature. The diode model is evaluated by the function `DiodeId`, which is implemented as a CUDA kernel. When called, the kernel is executed for every diode in a circuit. Each thread evaluating a diode is computed by one core.

Implementing the kernel with CUDA C was quite straightforward. The original `DiodeId` function written in the C language required only minor modifications. On

the other hand, analysis and data structures were quite unsuitable for the CUDA kernel. Hence, some modifications to the analysis structure and additional data handling functions were required.

The CUDA implementation of the diode evaluation did not perform as well as was hoped. The diode's nonlinearity turned out to be too simple for the GPU computation to bring any speed benefit. However, this implementation is a good foundation for future CUDA applications in APLAC. One of these will be the BSIM3 transistor model evaluation, which is substantially more complex and will most likely benefit from a CUDA device's computational power.

2 Background

2.1 From CPU to GPU computation

Throughout the long-term development of computer technology, an obvious division of computation tasks has dominated between the central processing unit (CPU) and the graphics processing unit (GPU). The CPU has been designed to handle all general computation, whereas the GPU handles all the graphic-related computation, thus relieving the CPU of these operations. Although the growing need for more efficient graphics rendering has guided GPU development, the high capacity to process large amounts of data in parallel has made the GPU an excellent alternative for general-purpose parallel computation in various fields, such as computational finance and biology, physics, electromagnetic and electronic circuit simulation [3].

Due to their different use, CPU and GPU architectures are almost complete opposites of each other. The CPU concentrates on a more complex flow control and data caching using only a low level of parallelization, as a typical modern CPU comprises of 1–16 cores. As for the GPU, in order to achieve the fast, high-definition graphics processing of modern computer software, the GPU is required to process hundreds of pixels in parallel. This has lead to a highly parallel GPU architecture, which is based on a great number of parallel, multithreaded and multicore processors with a complex, high bandwidth memory system. Figure 1 shows the differences of the CPU and the GPU architectures. CPU has a few complex processing cores or arithmetic logic units (ALU), one complicated control unit, and a large cache (or several different level caches). Whereas the GPU comprises several small and simple ALUs and one simple control unit and cache per multicore processor (one row in the figure).

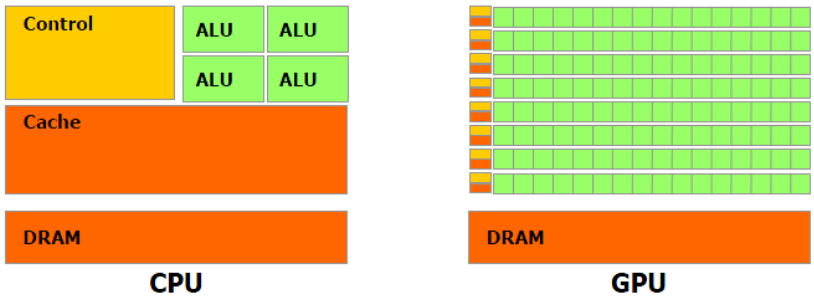

Figure 1: Comparison of CPU and GPU architectures [3].

Throughput improvements of GPUs and CPUs are compared in figure 2 as floating-point operations per second (FLOPS) [3]. Although comparing performance solely based on FLOPS is a rather naive perspective, it still gives a good estimate of the theoretical maximum performances in the case of architecture optimized code. On this basis, NVIDIA's GeForce GTX 280 with its peak performance of over 900 GFLOPS seems quite superior to Intel's Harpertown and its performance of 120 GFLOPS.

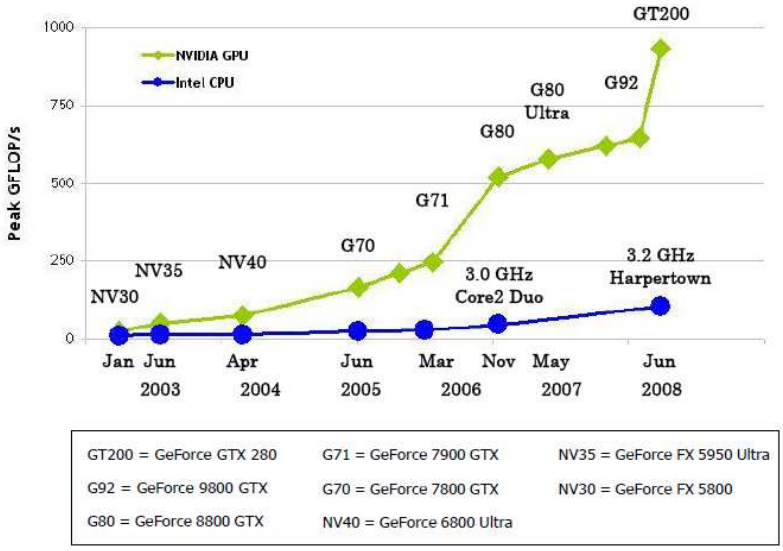

Figure 2: Throughputs for CPUs and GPUs as FLOPS [3].

Despite the superiority of the theoretical GPU throughput, reality is a lot more complex. GPU computation is efficient with highly parallelized tasks. If a task is not suitable for parallel processing, it will not benefit from GPU. In fact, such a task would be slower on a GPU. It should be kept in mind that the CPU is always faster on complex and serial tasks or when throughput comparison is normalized to the number of cores. Because of this, the GPU cannot replace the CPU as the main processing unit. However, it can accelerate parallel tasks significantly, as will be shown later in this thesis.

2.2 NVIDIA CUDA versus AMD/ATI Stream

As mentioned before, this thesis focuses solely on the CUDA architecture and development environment. This is mainly due to the fact that the hands-on software development was done on CUDA. Nevertheless, there is an alternative to NVIDIA and CUDA. AMD/ATI has released its own technology to general purpose GPU (GPGPU) computing, carrying the name Stream [4].

Several test results imply that the same algorithms realized on CUDA and ATI Stream platforms perform significantly better on CUDA. In [2], results show a GPU accelerated algorithm performing 9.2× faster on ATI Stream and 22.9× faster on CUDA compared to the CPU version of the algorithm. A significant reason for CUDA's notably larger speed-up is the difference in the memory architectures [2].

3 CUDA architecture

Section 2 gave a rough idea of the differences between CPU and GPU computation. In order to understand the principals of GPU computing and its capabilities, one needs to have at least some preliminary knowledge of the architecture. Supplying this knowledge is the purpose of this section.

The relevant portions of CUDA architecture are explained from the programmer's point of view. This includes memory structures, instruction set architecture, and processor architecture related to the execution. The entire thesis relies on the concepts and notations presented in this section making this an essential part of the thesis.

3.1 Device architecture

The NVIDIA CUDA device architecture defines a set of multicore and multithreaded processors called streaming multiprocessors (SM). Figure 3 shows the hardware architecture of a CUDA device. One device comprises numerous streaming multiprocessors. These SMs each involve a number of processors or cores (the expression varies). Every multiprocessor has one common instruction unit for all of its cores. Figure 3 also presents some of the memory spaces which are discussed in section 3.3.

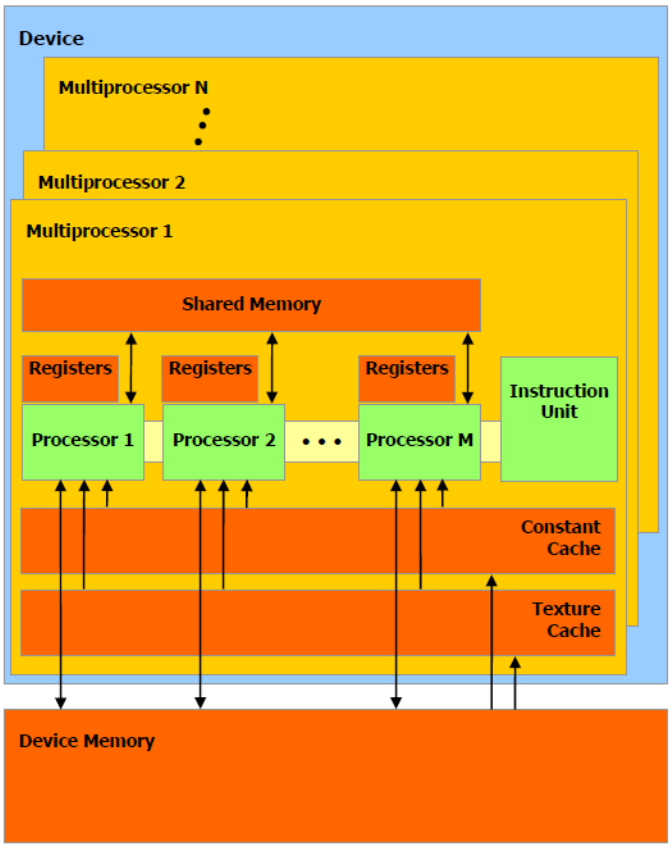

Figure 3: CUDA device architecture.

The multiprocessors are based on a so-called single-instruction, multiple-thread (SIMT) architecture. The name SIMT refers to the way that a SM arranges data processing. When a CUDA function, known as kernel, is called from the main program, it is launched with hundreds of concurrent threads. (This is discussed in section 4.) These threads are divided into blocks that are distributed to the multiprocessors. Each multiprocessor divides the threads of a block into groups of 32 threads called warps. Every multiprocessor manages the execution of its threads and thread blocks independently. The instruction unit issues one instruction to all of the cores which then process the threads of a warp, one thread per core.

This entire execution scheme is designed to be highly parallel and flexible. Every multiprocessor can process several thread blocks as well as different warps of a block concurrently. Every multiprocessor is bound by SIMT processing, but different multiprocessors of a CUDA device may have differing execution paths in the same kernel.

SIMT architecture is partly similar to the single-instruction, multiple-data (SIMD) vector machine architecture. However, these two should never be confused. Both of these share the principle of processing multiple data elements concurrently with the same instructions. A SIMT device is capable of processing thread-level parallel code where independent threads may branch, whereas a SIMD device requires identical execution paths.

3.2 PTX and instruction set architecture

Every processor architecture, CPU or GPU, has its own instruction-set architecture (ISA). In general, ISA defines the available native machine instructions, data types, registers, memory architecture, addressing modes, and interrupt and exception handling. When a program is compiled, it has to be targeted to a specific ISA in order to work. Regarding CPU code, the number of commonly used ISAs is rather limited, therefore targeting at compilation time is convenient and common practice. But when it comes to GPU architectures this is not as practical anymore. Even NVIDIA's CUDA-supporting product family involves several architectures. This makes targeting a CUDA program to every available architecture version rather difficult. Although, it would likely be possible, the new features of the next generation architectures would be left useless. This is where NVIDIA's parallel thread execution (PTX) steps in [5].

PTX defines a low-level virtual machine and its own ISA. This way, it provides a stable, static programming model and instruction set that remains unchanged regardless of the evolving GPU architecture generations. When a CUDA program is compiled it is targeted to the PTX instructions. It is not until at run time when the PTX instructions are translated and optimized to the native target hardware instructions. Figure 4 shows an illustration of this so called just-in-time (JIT) compilation and device targeting. The programming languages shown in the figure are introduced in section 4.

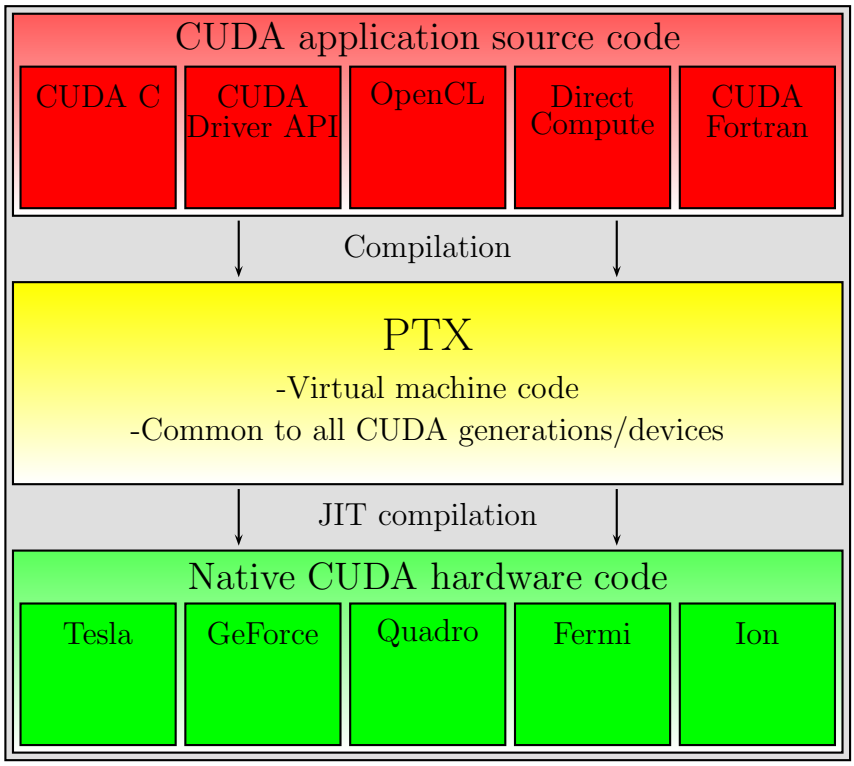

Figure 4: JIT compilation and device targeting of a CUDA application.

This kind of compilation and targeting strategy has many advantages. The stable ISA and programming model can be used by the several GPU generations and sizes. Figure 5 shows an example of this. Distribution of thread blocks is not determined until at run time, so the programmer does not have to know how many GPU cores will be available. Another benefit is that, because the PTX ISA is machine-independent, it is a convenient target for the different compilers and a common interface for PTX-to-GPU translator optimizations. Development of more efficient compilers and PTX-to-GPU translators can be conducted individually as long as both are compatible with PTX. It should be mentioned that traditional compilation directly to binary device code is also possible for CUDA applications. However, due to the benefits of JIT compilation, its use is common practice in CUDA programming.

3.3 Memory architecture

In this section, the CUDA memory architecture is described. There are six different memory spaces available in a CUDA device. The programming details are examined later in section 4.3. Figure 6 presents the different memory spaces available: global, local, shared, texture, constant memories, and registers.

Data transfers between host and device are handled through global, constant, and texture memories, which are located in the dynamic random access memory (DRAM) of the device. Constant and texture memories are the only cached memory spaces in devices with compute capability of under 2.0 (see section 3.4 for the

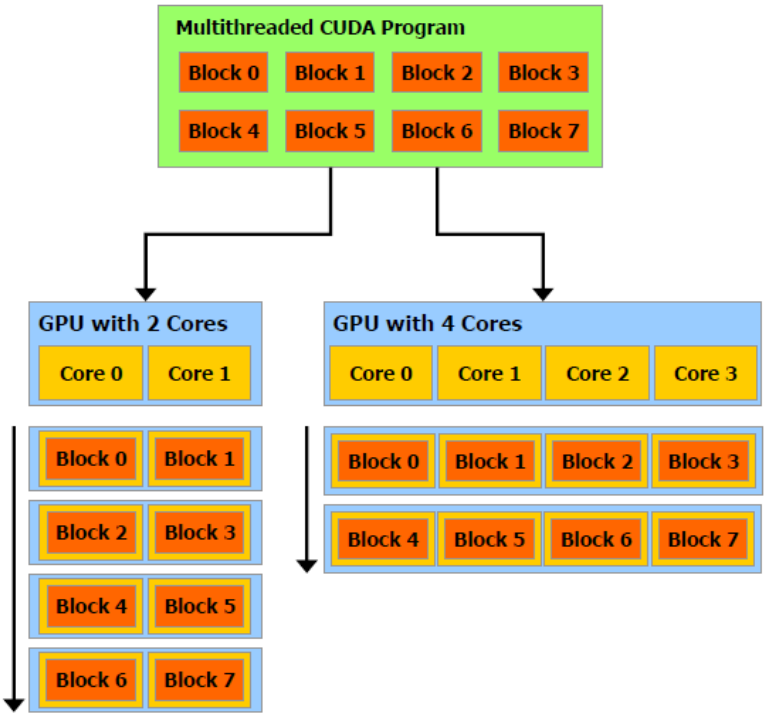

Figure 5: Example of thread block distribution on different sized GPUs [3].

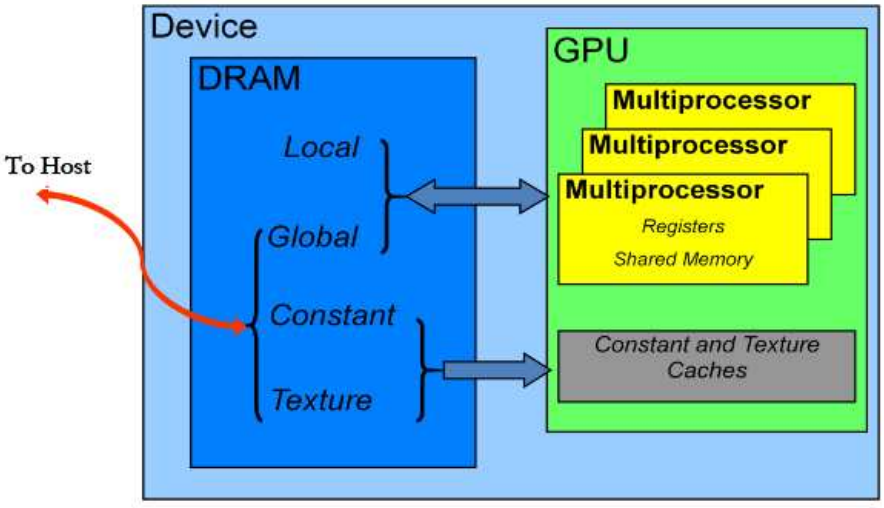

Figure 6: CUDA device memory spaces and their physical location on the device.

explanation of compute capability). They are also the only read-only memory spaces on the device. In addition, there is a local memory space in the device's DRAM space. In the multiprocessors, there are two on-chip memory spaces, namely registers and shared memory. All of the memories have their own scopes and data lifetimes. Figure 7 clarifies these for the three commonly used memory spaces.

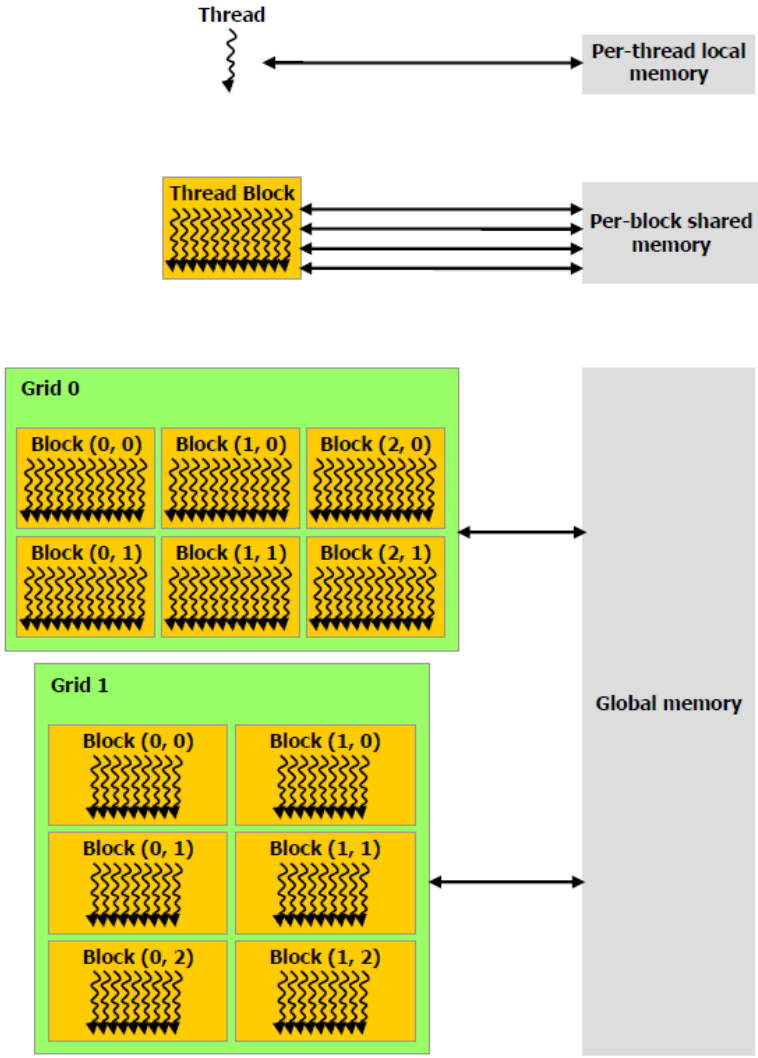

Figure 7: CUDA memory hierarchy presented in relation to the kernel threads.

Each thread has its own local memory which is accessible only from that particular thread and has a lifetime of that thread. Similarly, each thread uses private registers which also store data for the maximum of that thread's lifetime. In the same manner, every thread block has its own per-block shared memory accessible from that block and with a lifetime of the thread block. As the name implies, global memory space is accessible from every thread. The lifetime of global memory data is controlled by the programmer, meaning that it is not dependent of kernel launches and terminations. The same access rights and lifetime apply to the constant and texture memory spaces as does for the global memory.

3.4 Compute capability

CUDA architecture has developed in leaps and bounds from its first version. Just as with CPUs, every new CUDA version or generation has a more powerful architecture

and more advanced qualities. These properties are manifested by a version number called compute capability.

Current CUDA devices can be divided into two main categories: compute capability 1.x and 2.x. These are the major revisions. Devices with the same major revision number share the same core architecture and core number in a multiprocessor. The minor revision number (x) specifies the new features and incremental enhancements to the core architecture. As an example, double-precision floating-point numbers are supported from devices of compute capability 1.3 onwards. A full list of features and technical specifications of different compute capabilities can be found in Appendix G of [3].

4 Programming CUDA

Before GPU computation was used in general purpose applications, programming of GPUs could be done with specialized application programming interfaces (APIs), such as DirectX [6] and OpenGL [7]. The drawback of these APIs is that they have been designed specifically for graphics programming. Hence, they are quite impractical for general purpose GPU programming and require a high level of expertise from the programmer.

After the introduction of NVIDIA's CUDA architecture, several programming interfaces have been developed to meet better the requirements of general purpose GPU programming better. Next, some of these interfaces are introduced, following a more profound discussion of CUDA C, which is the API utilized in this thesis. In addition, a general CUDA programming model and strategies to enhance performance are introduced in this section. Compilation, debugging, and multi-GPU programming are also briefly discussed.

4.1 CUDA programming interfaces

At present, there are numerous programming interfaces applicable to CUDA devices. Every interface has its benefits and drawbacks. Therefore, the interface should be chosen with the complexity level of the programming task and the skill of the programmer in mind. Here are the five most commonly used APIs for CUDA programming.

- CUDA C
- CUDA driver API
- OpenCL
- DirectCompute
- CUDA Fortran

The first two interfaces in the list, CUDA C and CUDA driver API, are the ones directly supported and developed by NVIDIA. CUDA C is a minimal set of extensions to the standard C language whereas the CUDA driver API is a lower-level C API. Both interfaces can be used side by side, but it is typical to use only one. The CUDA driver API gives better control to the code and is language-independent but it requires more code than CUDA C and it is also more difficult to program and debug. CUDA C will be discussed further in section 4.3.

OpenCL (Open Computing Language) is an open API standard developed by Khronos Group [8]. The OpenCL language is based on the modern standard C language (C99). Its advantage is that it is supported by both NVIDIA and AMD/ATI GPUs. However, this leads to an inevitable drawback that a GPU kernel written in OpenCL usually executes slower than with CUDA C. This is obvious because

OpenCL cannot be fully optimized specifically to the CUDA architecture, unlike CUDA C.

The DirectCompute API is Microsoft's solution to general purpose GPU programming. It is a part of the DirectX API collection [9]. The CUDA Fortran API enables programming CUDA devices with the Fortran language [10]. It is a set of extensions to Fortran language just as CUDA C is to the C language. In the field of circuit simulation, GPU implementations using DirectCompute or CUDA Fortran are quite hard to find. Thus, they will not be discussed further in this thesis.

4.2 Programming model

Programming a GPU enhanced application may sound complicated, but the general guidelines are actually very simple. The traditional work flow for programming CUDA can be simplified into the following five stages [11]:

1. Allocating global device memory
2. Copying data from the host memory to the global memory
3. Executing the CUDA kernel and storing results to the global memory
4. Copying the results back to the host memory
5. Freeing global device memory

The first step is run in the host code. Section 3.3 covered the CUDA device memory architecture and stated that the common data path between host and kernel is global memory. From there, optional memory transfers between different device memory spaces can be made to enhance performance.

The second step is quite obvious. The device cannot access host memory, therefore all data required by the kernel has to be transferred to the device memory. The third stage is self-explanatory. The desired computations are performed on the device and the results are stored in the device memory. As the kernel cannot read host memory, it certainly cannot write onto it. The fourth step is, of course, similar to the second step, except that data is transferred to the opposite direction.

The fifth and last stage is identical to any host code programming. Any memory that is allocated should also be deallocated when it is no longer needed. Section 4.6 discusses this topic further in terms of performance, but in brief, if there is any chance of further use of the allocated data, it should not be freed too early. Memory transfers between host and device are extremely expensive and should be avoided whenever possible.

This work flow can be quite well generalized to various CUDA programming APIs. As mentioned earlier, GPU programming in this thesis was done on CUDA C, which will be discussed next in detail.

4.3 CUDA C

CUDA C consists of a set of extensions to the standard C and a runtime library. In practice, this enables programming CUDA devices with the C language. The extensions to standard C are mainly for defining and launching kernels and managing memory. Complete descriptions of the CUDA C extensions are found in Appendix B of [3].

Next, the kernel and memory extensions are discussed through a simple C program, that consists of a main function and a vector addition function. Here are the conventional C language program executed solely on a CPU(host) and the CUDA C program, that consists of the host part (main function) and the kernel executed on a CUDA device. The five stages of the general CUDA programming model can easily be seen from the CUDA C example.

Standard C example:

```
//Function
void vectorSum(double *A, double *B, double *C)
{
    C[0] = A[0] + B[0];
}

//Main program
int main()
{

    //Vectors in the host memory
    double *A, *B, *C;

    /* Vector allocations (size N) in host memory */
    /* Vector fills */

    //Loop for function call
    for (int i=0; i<N; i++) {
        vectorSum(A+i, B+i, C+i);
    }

    /* Vector deallocations in host memory */

}
```

CUDA C example:

```
//Kernel
__global__ void vectorSum(double *A, double *B, double *C)
{
    int thr_id = threadIdx.x;
    C[idx] = A[idx] + B[idx];
}
```

```
//Main program
int main()
{

    //Vectors in the host memory
    double *A, *B, *C;

    //Vectors in device memory
    double *A_d, *B_d, *C_d;

    /* Vector allocations (size N) in host memory */
    /* Vector fills */

    //Device memory allocations
    cudaMalloc((void**) &A_d, sizeof(double)*N);
    cudaMalloc((void**) &B_d, sizeof(double)*N);
    cudaMalloc((void**) &C_d, sizeof(double)*N);

    //Data transfer from host to device
    cudaMemcpy(A_d, A, sizeof(double)*N, cudaMemcpyHostToDevice);
    cudaMemcpy(B_d, B, sizeof(double)*N, cudaMemcpyHostToDevice);

    //Kernel launch for vectors with N elements
    vectorSum<<< 1, N >>>(A_d, B_d, C_d);

    //Data transfer from device to host
    cudaMemcpy(C, C_d, sizeof(double)*N, cudaMemcpyDeviceToHost);

    //Device memory deallocations
    cudaFree(A_d);
    cudaFree(B_d);
    cudaFree(C_d);

    /* Vector deallocations in host memory */

}
```

Kernels

As mentioned earlier, the heart of the CUDA execution is the kernel. It is fundamentally a C language function with some differences related to the definition, calling and thread organization. Kernel definition is similar to a C function definition except for the `__global__` declaration specifier, which defines the device kernel to be called from the host code. Functionality inside the kernel is similar to the C function. The only difference inside the kernel is the variable

```
int thr_id = threadIdx.x;
```

`threadIdx` is a built-in variable that tells the thread's index in a block. The thread index is used as an index for the vectors. This way every thread knows which elements to use.

A kernel call includes an execution configuration syntax `<<< 1, N >>>`. It is used to specify the number of executed threads and how they are arranged in blocks

(see section 3.1). The first value in the configuration is the number of thread blocks and the second is the number of threads in one block. In the example, only one block is used and the number of threads is the size of the vectors. Both the numbers of blocks and threads can be specified in three dimensions. (Hence the `.x` in `threadIdx` syntax). Further discussion of the 3D thread arrangement can be found in [3].

It should be pointed out that this is a naive example and it does not utilize a GPU device's total capacity. One block is always sent to one multiprocessor, and to avoid idle SMs the number of blocks should be at least the number of SMs. Threads in this example are executed only on one multiprocessor. If multiple blocks were used in this example, the thread index definition would be

```
int thr_id = blockIdx.x * blockDim.x + threadIdx.x;
```

As said, `threadIdx` is the thread index inside the block. So, to get the thread's index in the entire kernel the offset to the current block has to be determined. This is taken from the built-in variables `blockIdx.x` and `blockDim.x`. They are this block's index and the number of threads in a block.

Memory handling

Basic memory management in CUDA programming is quite straightforward. Memory is allocated in the device's global memory space with the function

```
cudaMalloc((void**) &A_d, sizeof(double)*N);
```

where the first parameter is a pointer to the allocated memory and the second is the size of allocation in bytes. Allocated memory is freed with the function

```
cudaFree(A_d);
```

Memory transfers are done by the function

```
cudaMemcpy(A_d, A, sizeof(double)*N, cudaMemcpyHostToDevice);
```

The function parameters are destination and source memory addresses, transfer size in bytes, and direction of transfer.

These are the basic memory functions that are adequate to some extent. Besides these there are dozens of more sophisticated CUDA memory handling functions, which are required to manage the CUDA device's various memory spaces (see section 3.3) and more complicated memory transfers such as asynchronous transfers. Applications presented in this thesis rely on the basic functions discussed here. More on the additional functions and their use can be found in [3], [12], and [13].

4.4 Compiling a CUDA program

The compilation of a CUDA program is handled by the CUDA compiler driver nvcc [14]. The compilation process includes several splitting, compilation, preprocessing, and merging steps. The developer does not have to pay any attention to these

detailed phases since all of this is managed by nvcc. Nvcc has been designed to mimic the GNU compiler gcc [15]. It takes a number of compiler options, macro definitions, file and library paths. A CUDA-compatible program includes source code for both the host (CPU) and the device (GPU). During preprocessing, nvcc separates the host code and forwards it to a supported host C/C++ compiler. On Linux platforms the supported compiler is the GNU compiler (gcc) and on Windows platforms it is the Microsoft Visual Studio compiler cl. All compiler options concerning the host code compilation are passed on to the host compiler [14].

4.5 Debugging

Debugging a CUDA program is rather similar to debugging a normal multi-threaded host program. The host code part of a program can be debugged as usual. For instance, all the debugging properties of Visual Studio are fully functional for the host code.

Nevertheless, the device code cannot be debugged as easily. Neither Visual Studio nor GNU debugger GDB in Linux is able to retrieve any debugging information from the CUDA code. NVIDIA has released solutions for both Linux and Windows to counter this problem. The NVIDIA CUDA debugger, CUDA-GDB, is an extension to the GNU project debugger for Linux [16]. It provides the user with a debugging environment enabling all the traditional tools, such as breakpoints, variable watches, and memory checking, for both CPU and GPU code. Correspondingly, NVIDIA Parallel NSight is a Windows-based debugging environment which integrates fully into Visual Studio. It enables all the corresponding tools as CUDA-GDB does in Linux. Besides this, NSight includes an analyzer tool for capturing events and profiling kernel performance. Still NSight has one very inconvenient drawback: all the real-time debugging properties are available only when at least two GPUs are used.

4.6 Enhancing performance

After a simple version of a CUDA enhanced application has been successfully built, the speed performance is most likely to be of the same order as the original host program. At this point, there are certain performance optimization guidelines to make the performance of the CUDA application superior to the original one. These guidelines can be summarized as three basic strategies, which are examined further in this section [12]:

1. Maximizing the level of parallel execution
2. Optimizing memory usage for maximum memory bandwidth
3. Optimizing instruction usage for maximum instruction throughput

The first strategy is rather obvious: the application or algorithm in hand should be exposed to as much data parallelism as possible. The kernels should be launched with a configuration which makes the mapping to hardware as efficient as possible.

Besides this, a higher level of parallelism is encouraged. Also, attention should be paid to the concurrency of the host and the device execution.

The second of the strategies could be, in many cases, described as the most important [2]. Data transfers between host and device have a significantly lower bandwidth compared to transfers within the device or the host. Hence, these low bandwidth data transfers should be absolutely minimized. It is possible that moving some serial code to the device kernels may bring a speed advantage if this reduces the low bandwidth data transfers. Actions inside the device include avoiding kernel access to the global memory. Instead, the faster shared memory should be used whenever possible. In some cases, speed-up may even be obtained by recomputing data instead of transferring it.

Concerning the third strategy, a general method of trading precision for speed whenever possible is suggested in [12]. In practice, this could mean using single instead of double precision and intrinsics instead of regular functions. NVIDIA has also supplied a fast math library which includes faster but less accurate versions of common mathematic functions. These functions are encouraged to be used whenever possible.

One other matter of interest is control flow instructions. As described in section 3, a CUDA device uses a SIMT architecture. This leads to serialization of execution whenever a control instruction with differing branches between threads of a warp are processed. Naturally, control instructions are unavoidable, but they should be avoided if possible for maximum throughput.

4.7 Multi-GPU programming

With common CPU computation, there is the possibility to enhance parallel computing by forming clusters of computers. In a similar way, GPU calculations can be enhanced by using multiple GPU units. This naturally requires highly parallelized, efficient CUDA code to bring any extra advantage to the single GPU situation. In this thesis, the subject of multi-GPU programming is overviewed only at a general level. The subject is presented in more details in [12].

Basically, programming a CUDA application utilizing multiple GPUs is similar to programming an application using multiple CPU threads or cores. Thus, multi-GPU support can be added to a pre-existing multi-threaded host application as easily as single-GPU support to a single-thread application. The only new concept to basic single-GPU programming is selecting the correct GPU. A CPU thread assigns work to a GPU using a context that needs to be established between the host thread and the GPU. This procedure is valid even with a single-thread and a single GPU, it is handled by the compiler without the need for the programmer to intervene. In any case, only one context can be active on both the GPU and the CPU thread. Because of this, the only possible solution when using CUDA C is that each CPU thread controls one GPU. When the application spawns as many host threads as there are GPUs available, this approach functions properly in most cases. Even the GPU indices can be acquired directly from the CPU thread numbers. When the CUDA driver API is utilized, controlling multiple GPUs by a single host thread is

possible by pushing and popping contexts.

Multi-GPU CUDA code is compiled in the same manner as normal single-GPU code. That is, the nvcc compiler driver controls the whole compilation invoking gcc [15] on Linux or Microsoft Visual C/C++ compiler on Windows. The compilation process was already discussed in sections 3.2 and 4.4.

5 Utilization of GPU in circuit simulation

In this section, programming the GPU is examined from the viewpoint of circuit simulation. Section 5.1 examines when and where GPU programming should and can be used. Then, some working examples from the field of circuit simulation are presented and evaluated.

5.1 Where to start

Preceding sections established the ultimate purpose of exploiting GPU computation in circuit simulation, or in any other field — performance. The CPU architecture is more sophisticated which makes it the best alternative for almost any application except graphics processing. Because of this, GPU programming should be applied only in situations where it can bring noticeable performance gain.

When the objective is enhancing performance, the first task is obviously locating the data processing bottlenecks. Which are the slowest blocks of a program, and on which tasks does the program spend most of its processing time. This investigation reveals where GPU programming could be used to attain the best possible gain. The next step is to examine which of these program blocks are suitable for GPU processing and can benefit from GPU characteristics. As stated in section 3, the efficiency of GPU processing is based solely on heavily parallel computation. This fact rules out all parts of a program involving only a handful of data and mostly sequential code. No matter how time consuming these kinds of blocks are for the program, translating them to GPU code would only slow the processing. To conclude, that part of a program which is both heavily utilized and handles large amounts of data in a fashion that can be realized in parallel needs to be found.

5.2 Previous work

The bottlenecks of circuit simulation and their potential as GPU implementations has been researched widely in recent years. Some of these circuit simulation tasks that have been successfully implemented on CUDA are discussed next.

Matrix-vector multiplication

Solving the current or voltage unknowns of a circuit involves (sparse) matrix-vector multiplication (SpMV) [17]. This is done several times during a circuit analysis, thus making it a potential candidate for CUDA enhancement. SpMV is a highly complicated task in any architecture due to the large number of indirect and irregular memory accesses.

Because of the great need for SpMV, NVIDIA too has developed its own CUDA versions of the algorithm: CUDA libraries CUDPP [18] and CUBLAS [19] include different kernels for SpMV. Other implementations have also been developed. The SpMV kernel presented in [20] outperformed the CUDPP version by $2-8\times$ and was at least equal to the CUBLAS kernel. Another example of CUDA SpMV implementation was even $20-40\times$ faster than normal CPU implementations [21].

Indeed, matrix-vector multiplication is a crucial part of circuit simulation and its acceleration would decrease the total simulation time. Nevertheless, section 6 examines the integration of CUDA in the APLAC simulator and discovers that in this case a CUDA matrix-vector multiplication is not beneficial.

Transistor models

BSIM3 and BSIM4 transistor models have a dominant role in modern real-life circuits. In one study 66% of transient-analysis time was spent evaluating the BSIM4 model [22]. This was based on 27 test circuits. Another study reported that on average 75% of the total simulation time was spent in the BSIM3 model evaluations [23]. Based on these results, it is evident that even reasonable speed enhancements in the model evaluation can bring about serious reduction in the total simulation time.

Indeed, in both cases, migrating the model evaluation code to CUDA resulted in significant improvement. In the first case, the speed-up was $3 - 6\times$ for circuits with more than 900 transistors and in the second case it was approximately $4\times$.

General bottlenecks

A slightly different approach to GPU speed enhancement is proposed in [24]. The idea is to concentrate on the common bottlenecks and overheads, presuming that the parallelization has already been done. Some of these issues regarding performance enhancement have been discussed in section 4.6. Some solutions and general guidelines for overcoming these bottlenecks are presented in that section. In section 6, these challenges are discussed in detail from the viewpoint of CUDA parallelization in APLAC. The following bottlenecks can be generalized for parallelization on GPUs [24]:

- Communication bottlenecks: The most classical of bottlenecks is that the data needed in the GPU needs to be transferred from the host memory to the device memory. This delay should be hidden beneath the data processing as well as possible.

- Conditional control flow: SIMT architecture restricts the execution to one instruction for all threads of a warp. This leads to forced serialization when deviant execution paths of a conditional statements are confronted.

- Kernel invocation overheads: Implementing threads with less parameters simplifies the generation of threads and is consistent with the nature of the ALU. However, every kernel invocation raises new overheads. Therefore, the number and size of kernels as well as the number of inputs and outputs per kernel need to be balanced by testing.

- Data structures: A typical code organization of a common CPU program consists of linked lists, pointers, and dynamic memory allocation. This raises severe difficulties with SIMT computing which requires data in the form of

vectors. The processed data has to be rearranged or collected in vectors for a kernel. This increases the simulation time.

- Data size: The strength of GPU computation lies in processing huge data sets. If the data size is too small, all the speed-up is lost to the communication overheads.

- Compatibility with SIMT program flow: A host program might be implemented in a way that is rather unsuitable for SIMT architecture. This might be the case if the data processing that could be parallelized is separated by a complicated control flow for each data block.

The results in [24] were excellent. By working on the above-mentioned bottlenecks, the realized speed-ups were in the range of $10 - 50\times$. Still, it has to be noted that dealing with these bottlenecks is mostly fine-tuning. For this to be useful, the code must already be working properly. Besides, some of these matters cannot be even manipulated.

6 CUDA programming in APLAC

Section 5 discussed the topic of identifying the bottlenecks and possible programming tasks for CUDA enhancement on a general level. Also, some previous work in the field of circuit simulation was examined. In this section, the possible targets for CUDA enhancement in the APLAC simulation software are examined.

6.1 Matrix-vector multiplication

Matrix-vector multiplication is a serious time consumer in circuit simulation. Therefore, it would seem to be a good program block to obtain speed-up with CUDA. Nevertheless, matrix-vector multiplication in APLAC has been implemented in a way that falls partly under the general bottleneck: compatibility with the SIMT program flow described in section 5.2.

The multiplication has not been implemented with few straightforward functions. It consists of several inner function calls, control statements, and data fetches from different data structures. Even this kind of implementation could be modified into CUDA kernels. The only problem is that this CPU-optimized structure would require serious rewriting which would probably lead to a slower, clumsy CUDA implementation without any performance gain.

Because of this, matrix-vector multiplication was discarded as a choice for APLAC's CUDA enhancement. Still, it should be mentioned that, due to the massive role of SpMV in circuit simulation, the possibilities to implement it with CUDA also in APLAC should be re-examined at some point of development. One solution could be to optimize APLAC's SpMV with pre-existing kernels from NVIDIA's libraries, such as CUDPP [18], CUBLAS [19], and CUSPARSE [25].

6.2 BSIM3 and BSIM4 transistor models

A typical real-life circuit contains thousands of transistors, which are modeled with BSIM3 or BSIM4 models. Evaluation of these models takes as much as 75% of the total simulation time [23]. Therefore, the BSIM3 model is an excellent candidate for CUDA development also in APLAC. Despite the theoretical speed-up that CUDA enhancement of BSIM3 model could bring, it is left for a later stage of the APLAC development and will not be covered in this thesis. This is due to the complicated structure of the model evaluation function.

6.3 Diode model

The diode model is not as commonly used as BSIM3. Nevertheless, it has a similar evaluation function as BSIM3, only a lot more simplified. This is why the diode model is a perfect starting point for CUDA development in APLAC. The aim is to program a functioning CUDA version of the diode model. After the original analysis code has been modified suitably for the diode, the development of a CUDA version of BSIM3 is much easier.

7 CUDA diode implementation in APLAC

This section covers the implementation of the CUDA diode model in detail. First, the model structure and characteristics are introduced. Then, the actual CUDA implementation is discussed. Finally, the performance optimization and expectations of the CUDA model are examined.

7.1 Diode model

The diode model follows the general structure for APLAC models described in [26]. It has its own model parameters, of which a name and nodes for the anode and cathode are the only obligatory parameters. The remaining of the roughly 50 parameters are optional for the user. If some of these parameters are not defined, they are either ignored or default values are used, depending on the parameter. A complete parameter list can be found in [27].

The model's internal structure is presented in figure 8. It consists of the current source I_d, linear resistance R_s, and a shunt capacitance C_d. The capacitance is left out of this discussion because it has not been implemented in the CUDA version yet. The diode's behaviour has been programmed as several functions of which the one required here is the `DiodeId` function. Its functionality is described next. Equations (1)-(4) determine the current I_d going through the ideal pn-junction.

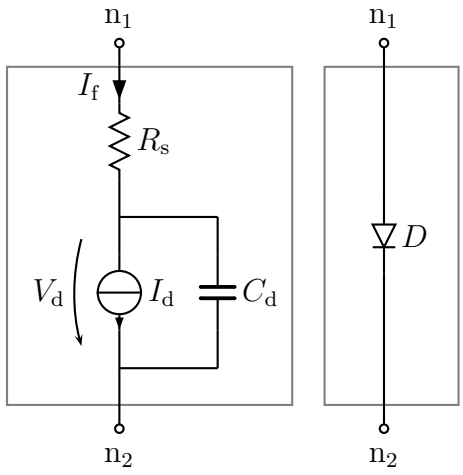

Figure 8: Diode model structure.

$$I_\mathrm{d} = I_\mathrm{f} - I_\mathrm{r}, \tag{1}$$

$$I_\mathrm{f} = a\left\{\frac{I_\mathrm{n}}{1+\sqrt{I_\mathrm{n}/I_\mathrm{KF}}} + k_\mathrm{g} I_\mathrm{SR}\exp\left(\frac{qV_\mathrm{d}}{\eta_\mathrm{R}kT}\right)\right\}, \tag{2}$$

$$\begin{aligned} I_\mathrm{r} = {} & aI_\mathrm{BV}\left(\exp\left(\frac{-qV_\mathrm{d}}{\eta_\mathrm{BV}kT}\right)-1\right)\exp\left(\frac{-qV_\mathrm{BV}}{\eta_\mathrm{BV}kT}\right) \\ & + aI_\mathrm{BVL}\left(\exp\left(\frac{-qV_\mathrm{d}}{\eta_\mathrm{BVL}kT}\right)-1\right)\exp\left(\frac{-qV_\mathrm{BV}}{\eta_\mathrm{BVL}kT}\right), \end{aligned} \tag{3}$$

$$I_\mathrm{n} = I_\mathrm{S}\left(\exp\left(\frac{qV_\mathrm{d}}{\eta kT}\right)-1\right). \tag{4}$$

In the equations, $k = 1.380662\times10^{-23}$ is the Boltzmann constant, $q = 1.6021892\times 10^{-19}$ is the elementary charge, and T is the element's temperature in Kelvins. In equation (2), k_g is the recombination current dependency on the depletion layer width and is given as

$$k_\mathrm{g} = \left[\left(1-\frac{V_\mathrm{d}}{V_\mathrm{j}}\right)^2 + 0.005\right]^{m/2}. \tag{5}$$

Table 1 presents the parameters used in the equations along with their default values. All of these are optional for the user. Also, the parameter names that are applicable in the APLAC input files (`.i`) are shown in the table.

Table 1: Diode model parameters.

Parameter	`.i`-file syntax	Description	Default
a	AREA	Relative device area	1
η	N	Emission coefficient	1
η_BV	NBV	High reverse breakdown ideality factor	1
η_BVL	NBVL	Low reverse breakdown ideality factor	1
η_R	NR	Recombination current emission coefficient	2
I_BV	IBV	High reverse breakdown current	100pA
I_BVL	IBVL	Low reverse breakdown current	0
I_KF	IKF	High-injection knee current	∞
I_S	IS	Saturation current	10fA
I_SR	ISR	Recombination constant current	∞
R_s	RS	Series resistance. Divided by a.	0
V_BV	BV	Reverse breakdown voltage	∞
V_j	VJ	Junction potential	1V

Electrothermal model

Besides modeling the electrical behaviour of a diode, the model can also simulate a diode's operation under dynamically changing temperature. Generally this kind of APLAC model is referred to as electrothermal model (concurrent electric and thermal simulation) [28]. The general structure of an electrothermal model is presented in figure 9.

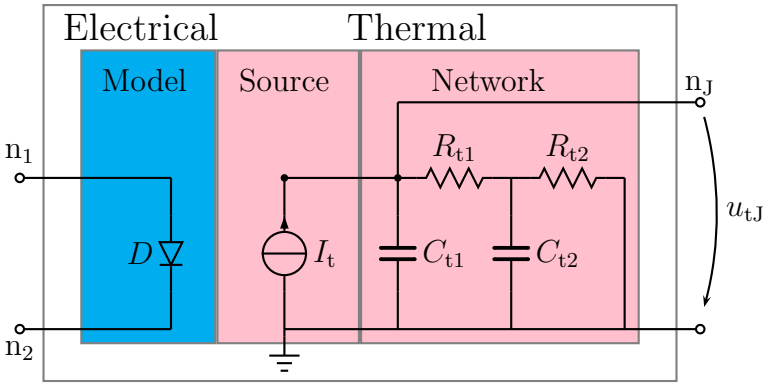

Figure 9: Electrothermal diode model.

A diode's temperature rise is modeled by a thermal current source I_t and an RC-ladder thermal network. Power dissipating from the component is fed to the thermal network by the source. The resistance and capacitance values for the thermal network have been defined so that the potential u_tJ in thermal node $\mathrm{n_J}$ is equivalent to the temperature rise above the ambient temperature. The relation is 1 V = 1 K by default.

The thermal potential u_tJ is passed on to the `DiodeId` function as a controlling voltage. `DiodeId` then takes into account this dynamic temperature while evaluating the diode's electrical behaviour. Equations (6)-(11) show the temperature dependencies of I_S, I_SR, I_KF, V_j, V_BV and R_S. Values I_S0, I_SR0, I_KF0, V_j0, V_BV0 and R_S0 are the respective parameters at nominal temperature T_nom.

$$I_\mathrm{S} = I_\mathrm{S0}\left(\frac{T}{T_\mathrm{nom}}\right)^{p_\mathrm{t}/\eta}\exp\left[\frac{qE_\mathrm{g0}}{\eta k T_\mathrm{nom}} - \frac{qE_\mathrm{g0}}{\eta k T}\right] \tag{6}$$

$$I_\mathrm{SR} = I_\mathrm{SR0}\left(\frac{T}{T_\mathrm{nom}}\right)^{p_\mathrm{t}/\eta_\mathrm{R}}\exp\left[\frac{qE_\mathrm{g0}}{\eta_\mathrm{R} k T_\mathrm{nom}} - \frac{qE_\mathrm{g0}}{\eta_\mathrm{R} k T}\right] \tag{7}$$

$$I_\mathrm{KF} = I_\mathrm{KF0}(1 + t_\mathrm{KF}(T - T_\mathrm{nom})), \tag{8}$$

$$V_\mathrm{j} = \frac{T}{T_\mathrm{nom}}V_\mathrm{j0} - 3\frac{kT}{q}\ln\left(\frac{T}{T_\mathrm{nom}}\right) - \left(\frac{T}{T_\mathrm{nom}}E_\mathrm{g0} - E_\mathrm{g}\right) \tag{9}$$

$$V_\mathrm{BV} = V_\mathrm{BV0}(1 + t_\mathrm{BV1}(T - T_\mathrm{nom}) + t_\mathrm{BV2}(T - T_\mathrm{nom})^2) \tag{10}$$

$$R_\mathrm{S} = R_\mathrm{s0}(1 + T_\mathrm{RS1}(T - T_\mathrm{nom}) + t_\mathrm{RS2}(T - T_\mathrm{nom})^2) \tag{11}$$

E_g is the energy gap with a temperature dependence

$$E_g = E_{g0} - \frac{\alpha T^2}{\beta + T} + \frac{\alpha T_{nom}^2}{\beta + T_{nom}}, \tag{12}$$

where E_{g0} is E_g at T_{nom}. The rest of the parameters required in equations (6)-(12) are listed in Table 2.

Table 2: Diode model electrothermal parameters.

Parameter	`.i`-file syntax	Description	Default
α	ALPHA	Energy gap temperature dependency factor [V/K]	Si: 702×10^{-6}
β	BETA	Energy gap temperature dependency factor [K]	Si: 1108
p_t	XTI	Saturation current temperature exponent	3
t_{BV1}	TBV1	Linear temperature coefficient of V_{BV} [K^{-1}]	0
t_{BV2}	TBV2	Quadratic temperature coefficient of V_{BV} [K^{-2}]	0
t_{KF}	TIKF	Temperature coefficient of current I_{KF}	0
T_{RS1}	TRS1	Linear temperature coefficient of R_S [K^{-1}]	0
T_{RS2}	TRS2	Quadratic temperature coefficient of R_S [K^{-2}]	0

The `DiodeId` function

The heart of the diode model evaluation is the function `DiodeId`. It computes the current and conductance for the nonlinear diode model at a given operating point based on the equations (1)-(5). It also evaluates the diode's temperature dependency (equations (6)-(12)) when the electrothermal model is enabled. The `DiodeId` function returns the current and conductance. Additionally, it updates some of the diode model variables.

Because the diode is a nonlinear component, the circuit has to be solved by iterating. Figure 10 shows the basic iteration model that represents the diode's current source. At the operating voltage u_0, the diode's current is $i(u_0)$ and conductance

$$g(u_0) = \left.\frac{\partial i(u)}{\partial u}\right|_{u=u_0}. \tag{13}$$

The current of the iteration model source is

$$J(u_0) = i(u_0) - g(u_0)u_0. \tag{14}$$

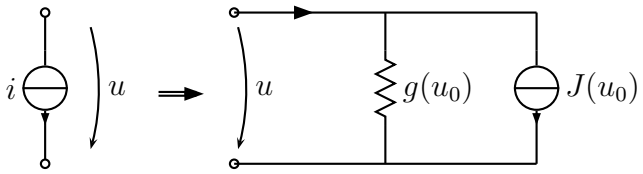

Figure 10: Iteration model for a nonlinear component [29].

7.2 The `DiodeId` kernel

In this case, implementing the diode model evaluation in CUDA means transferring the `DiodeId` function to a CUDA kernel, which can then be used to evaluate every diode in the circuit simultaneously. The original model, as well as APLAC in general, have been written in C. Therefore CUDA C (see section 4.3) is a consistent addition to be used in the CUDA blocks of the source code.

The `DiodeId` function is quite straightforward to implement in CUDA C. Roughly 90% of the function code can be left unmodified. The biggest modifications that had to be done concerned data fetches, parameters, and kernel definition. These are discussed more specifically in the following sections.

`DiodeId` calls some minor auxiliary functions during its execution, which had to be replaced in the kernel. Mostly these auxiliary functions are modifications of some mathematical functions containing additional error handling. In the `DiodeId` kernel, these functions were replaced with the default mathematical functions.

Naturally, the kernel requires the variable `threadIdx` for the thread index, which determines the correct elements in the parameter vectors (see section 4.3). `DiodeId` also contained some debugging data-printing that had to be left out from the kernel.

7.3 Data handling

In the original `DiodeId` function, all the required data was fetched from the model data structures during function execution. Obviously, this is not possible from a CUDA kernel because the structures are in the host memory. Instead, the kernel fetches the data from vectors that reside in the device memory. The function returns directly the calculated values whereas CUDA kernels are always of type void. Thus, the calculated results are stored in temporary vectors residing in the device memory. Pointers to all of these vectors are passed as kernel parameters.

Additional functions were required to fetch and store all data needed during model evaluation. To avoid excessive memory transfers, all the data that stays unmodified is fetched at the beginning of the analysis and is left in the device memory. Next, data handling functions are presented.

The `GetParams` function

The parameter fetch function `GetParams` handles all the model parameters which stay unmodified during the whole analysis. `GetParams` goes through the diodes of a circuit and fetches all the model parameters from the data structures and stores

them in vectors which can then be transferred to the device memory. This includes all obligatory and optional parameters that have been defined.

The `GetParams` function also checks which parameters have been specified via an integer-type vector `ParamsInt` containing information on the defined parameters. This is normally done inside the `DiodeId` function, which is not possible in the kernel.

Both of the vectors used by `GetParams` include data for all of the diodes. This way, the parameters can be used intuitively by the `DiodeId` kernel. The vector `Params` of type double stores the actual model parameters. It has space for 100 data elements per diode. The vector `ParamsInt` has space for 50 values per diode and primarily holds the information on defined parameters. Table 3 shows how the data in these vectors are arranged.

Table 3: Parameter vector arrangement.

Vector		Diode 1	Diode 2	...	Diode n
`Params`	Index:	[0] - [99]	[100] - [199]	...	[(n-1)*100] - [(n-1)*100+99]
`ParamsInt`	Index:	[0] - [49]	[50] - [99]	...	[(n-1)*50] - [(n-1)*50+49]

Vectors `Params` and `Paramsint` are allocated at the beginning of an analysis and `GetParams` is invoked for every diode successively. After this, memory space for these vectors in the device memory is allocated and the vectors are transferred. Because these vectors include only constant parameters, they can be left in the device memory for the entire analysis and with no further modifications.

The `GetTemp` function

Another function that had to be implemented for parameter fetching is `GetTemp`. When the ET model is used, the component's temperature and related values constantly change during the analysis. Therefore, this data needs to be retrieved between kernel calls. The `GetTemp` function gets the required ET-related data of every diode just before the next kernel call. This data is stored in the vector etParams, which is organized in the same manner as Params and ParamsInt. The vector has space for 15 double-type elements per diode as shown in Table 4.

Table 4: Arrangement of vector `etParams`.

	Diode 1	Diode 2	...	Diode n
Index:	[0] - [14]	[15] - [29]	...	[(n-1)*15] - [(n-1)*15+14]

Controlling voltages

In a manner similar to all the model parameters, the controlling voltages of the diode model are also taken from a data structure inside the `DiodeId` function. For

the kernel, the controlling voltages are taken from the structure before the kernel execution and stored in a double-type vector `uos`. Space for two controlling voltages per diode is allocated for this vector, one for the voltage across the component and the other for the ET model voltage. Even though, the latter is used only when the ET model is enabled, it is more straightforward to always allocate space for it. The arrangement of the `uos` vector is again similar to the previously presented parameter vectors and is shown in table 5.

Table 5: Arrangement of the `uos` vector.

	Diode 1	Diode 2	...	Diode n
Index:	[0] - [1]	[2] - [3]	...	[(n-1)*2] - [(n-1)*2+1]

Controlling voltages change between iterations, that is, between kernel calls. Because of this, they are updated in the `uos` vector and transferred to the device just before every kernel call.

Data saving

The current J and conductance g returned by the `DiodeId` kernel are stored in the `funcs` vector. It is allocated right before kernel execution and has space for five double-type values per diode. Besides this, `DiodeId` updates some other model parameters in the diode structure. In the kernel, these values are temporarily stored in the double-type vector `cuda_x`. This vector has space for 30 values per diode.

After the kernel execution, the function `storeX` is called successively for every diode. This function updates the data from `funcs` and `cuda_x` to diodes' structures. Next, the values of J and g are updated in the structures of the current sources modeling the diodes (see section 7.1).

Kernel data structures

The preceding sections introduced the different vectors required in the kernel execution. Table 6 summarizes the properties of these vectors. The table also shows when the vectors are updated and transferred between host and device. These vectors are actually the ones in host memory, but everyone of these has an identical copy in the device memory, denoted with `_d` in its name. Naturally, the one in device memory is used inside the kernel.

Table 6: Details of the vectors required in kernel execution.

Name	Type	Purpose	Elements /Diode	Updated	Transferred	Transfer direction
`Params`	double	Model parameters	100	Analysis initialization	Initialization	Host To Device
`ParamsInt`	integer	Specified parameters	50	Analysis initialization	Initialization	Host To Device
`etParams`	double	ET-related parameters	15	Between iterations	Before `DiodeId`	Host To Device
`uos`	double	controlling voltages	2	Between iterations	Before `DiodeId`	Host To Device
`funcs`	double	J and g from `DiodeId`	5	In `DiodeId`	After `DiodeId`	Device To Host
`cuda_x`	double	Other `DiodeId` return values	30	In `DiodeId`	After `DiodeId`	Device To Host

7.4 Analysis structure

In the previous sections, the modifications regarding data fetches and saves were discussed. The following pseudo-codes show a simplified comparison between the original and the modified analysis structures. The first code shows the idea of the original analysis: Data is retrieved and updated during the analysis and inside the `DiodeId` function. All the operations during one iteration are done successively in all diodes.

Original analysis structure:

```
Analysis initialization {}

Iteration Loop {
    Other parts of analysis

    Updating nonlinear elements: includes diodes {
        Loop(All diodes) {
            DiodeId()
            UpdateJandGms()
        }
    }
    Other parts of analysis
}
```

The following pseudo-code shows the rearranged analysis structure along with the new data transfers and calls to the functions previously introduced. Here, it is no longer possible to handle one diode at a time. In the original analysis, only one loop was required for the diodes, while now one loop is executed before the actual evaluation and one after. Inevitably, this increases the number of host instructions in one iteration loop.

Modified analysis structure:

```
Analysis initialization {
    Host memory allocation for:
        -Params
        -Paramsint
        -uos
        -funcs
        -cuda_x
        -etParams

    Loop(All diodes) {
        GetParams()
    }

    Data transfer from host to device:
        -Params
        -Paramsint
}


Iteration Loop {
    Other parts of analysis

    Updating nonlinear elements: includes diodes {

        Loop(All diodes) {
            Fetch controlling voltages -> store to uos-vector
            getTemp()                  -> ET-parameters to etParams-vector
            storeUo()                  -> controlling voltage is
                                          stored in diode struct
        }

        Data transfer from host to device:
            -uos
            -etParams

        DiodeId_kernel<<<>>>()
            -J and g to funcs vector
            -other return values to cuda_x vector

        Data transfer from device to host:
            -funcs
            -cuda_x

        Loop(All diodes) {
            storeX()
            UpdateJandGms()
        }
    }

    Other parts of analysis
}
```

7.5 Performance optimization

One major bottleneck in integrating CUDA code into APLAC is the code and data structure. It is obvious that a software originating from decades before the advent of general-purpose CUDA programming has not been designed with GPU integration in mind. Although this does not eliminate an ideal integration, such an integration is highly unlikely. The situation, where APLAC is concerned, is most unfortunate. The inner architecture of the data structures and the entire code has been designed for optimum performance on the host, leading to a source code that is contrary to the performance guidelines and restrictions of CUDA programming.

However, APLAC's performance can be accelerated with CUDA. This means that achieving speed-up requires thorough performance optimization of all aspects of the execution. Some of the strategies to achieve speed-up were introduced in section 4.6. Next, the application possibilities of these strategies to the diode implementation are examined in detail. Despite this research, most of the advanced optimization techniques have not been implemented yet. Therefore, their effect on the performance could not be tested and evaluated in the scope of this thesis.

Maximizing parallel execution

The first strategy, maximizing the level of parallel execution advises to parallelize as much of the algorithm and program in hand. Currently, this is fixed in the `DiodeId` kernel and code related to it. Further parallelization of APLAC is beyond the scope of this thesis. More interesting issues in this strategy are the kernel launch configuration and concurrent execution of the kernel and host code.

Regarding performance, the essential part of the kernel launch configuration is the distribution of threads into blocks. This has a major effect on the kernel efficiency through the level of core utilization, or occupancy. The optimum block size cannot be determined explicitly without testing, and the total thread count dependency on the simulated circuit complicate optimization even further. However, there are some guidelines to determine the best block size [12]. The number of threads per block should be the following:

- A multiple of 32 (warp size) to avoid incomplete warps
- 64 at minimum when multiprocessor executes multiple concurrent blocks
- 128-256 is a good initial number to start optimizing
- 3-4 smaller blocks are better than one large one to hide latency

Concurrent execution of device and host code is quite difficult in this situation. Kernel results are immediately utilized in the following host code. Therefore, it is unacceptable to continue free host execution before the kernel has finished. There is still one possibility to enhance this parallelization. If the number of diodes, that is, the number of launched threads is exceedingly more than the number of GPU cores, most of the threads are left in the queue. When a group of threads is finished, the

next threads in the queue begin execution. At this point, results from completed threads could be transferred to host memory and host code execution could continue on its part. Unfortunately, this contradicts the principal of large memory transfers. Because every separate memory transfer has some overhead, this may lead to an even longer process time than executing all threads and then transferring all the results at the same time. The benefits and drawbacks of these different procedures are difficult to estimate in theory. Therefore, the only way to discover the best of the approaches, is by trial and error.

There is also one more potential method to parallelize host and device execution. Every circuit involves components other than diodes that are evaluated in the host code. With a reasonable amount of analysis structure modification, these components could be evaluated in the host concurrently with diode evaluation in the device. This should not result in any data validity issues because every component is evaluated individually inside one iteration loop.

Memory usage optimization

This is the second strategy. Memory usage should be optimized for maximum memory bandwidth, as mentioned above. This means that memory transfers between the host and device should be in large blocks and excess memory transfers should be avoided. Memory-wise implementation of the `DiodeId` kernel is quite suitable. Diode parameters are stored in vectors and transferred to the device. After this, they remain unchanged in the device memory. Only updated controlling voltages and ET parameters have to be transferred to the device for every new iteration. These data vectors do not have to be transferred back to host and can be destroyed after analysis. The only data that needs to be transferred from device to host is the result vector which contains only five double values per diode and the `cuda_x` vector. These transfers are done after each kernel execution. From this point of view, the `DiodeId` kernel follows quite well the principle of efficient parallelization on SIMT architecture: a relatively small quantity of data needs to be transferred between host and device, and the kernel includes heavy calculation.

In the diode implementation, only global memory is used in the device. This is straightforward to implement, but global memory is also the slowest. The efficiency of memory usage should increase if the other specialized memory spaces are used (see section 3.3). The constant parameter vectors could be transferred to the constant or texture memory space. Their access patterns have already been optimized for data that stays unmodified throughout the entire execution. Other data that is modified between kernel calls could be transferred to per-block shared memory spaces that are faster but accessible only from the current thread block. However, this might not bring any speed benefit due to the excessive transfers in the device and the extra instructions required to distribute the data to the blocks.

Instruction level optimization and circuit characteristics

The third strategy suggests a tradeoff between precision and speed whenever possible. One simple way to do this, is to use single precision instead of double. However,

results of nonlinear circuit analysis can be quite sensitive to the lost accuracy. That is why in the `DiodeId` kernel, double precision is used by default unless the required precision is guaranteed using single precision.

One other thing that affects the achieved speed-up is the number and characteristics of the diodes in analyzed circuit. In order to utilize all the cores of a device, the number of threads, that is the number of diodes, has to be at least the number of GPU cores. When all the overheads from invoking a kernel and transferring data to the device and back to the host are taken into account, it is very unlikely that any speed-up is achieved with few threads. Thus, roughly put, the more diodes there are the more speed-up can be achieved. Then there is the matter of diode characteristics. SIMT architecture restricts the execution to one instruction for every thread of a warp. This leads to partly serialized execution in every deviating branch (see section 4.6). Thus, if the number of diodes with similar characteristics is too small, some cores may remain idle on some execution path and a smaller speed-up is obtained. It is important to notice that here similar characteristics do not imply that the diodes are identical. The `DiodeId` kernel branches mostly according to specified parameters and not as much by the parameter values. Hence, different diodes most likely have the same execution path if they have the same parameters defined, regardless of differing parameter values.

The number of control branches could be reduced leading to some theoretical speed enhancement. However, this part of the development is still unfinished. On other parts, these circuit and element related matters are independent of the code. Hence, the achieved speed-up is largely case-specific.

7.6 Performance expectations

As discussed in the previous section, the achieved speed-up is rather difficult to predict. One main reason for this is the simulation time dependency on the circuit. Besides this, details in memory utilization make a great difference in the enhanced simulation time. However, the maximum achievable speed-up can be approximated by Amdahl's law [30] according to which the maximum speed-up S is

$$S = \frac{1}{(1-P) + \frac{P}{N}}, \tag{15}$$

where P is the fraction of the total serial simulation time that can be parallelized. In the `DiodeId` case, P is that fraction of the total simulation time that is used by the `DiodeId` function. N is the number of processors on which the parallelized CUDA code is executed. For the diode development and test setup $N = 192$.

When N is of this magnitude, the term $\frac{P}{N}$ is small enough to be left out from the equation and S can be safely approximated by the simplified Amdahl's law

$$S \approx \frac{1}{1-P}. \tag{16}$$

Figure 11 shows the speed-up S when the parallelized simulation time fraction is $0 \leq P < 1$ calculated from equation 16. It is easy to see from the figure, that

substantial speed-ups are not reached until the parallelized fraction of the program is rather large. However, it is worth mentioning that in many cases even a speed-up of $2\times$ is significant. This can be reached when $P = 0.5$.

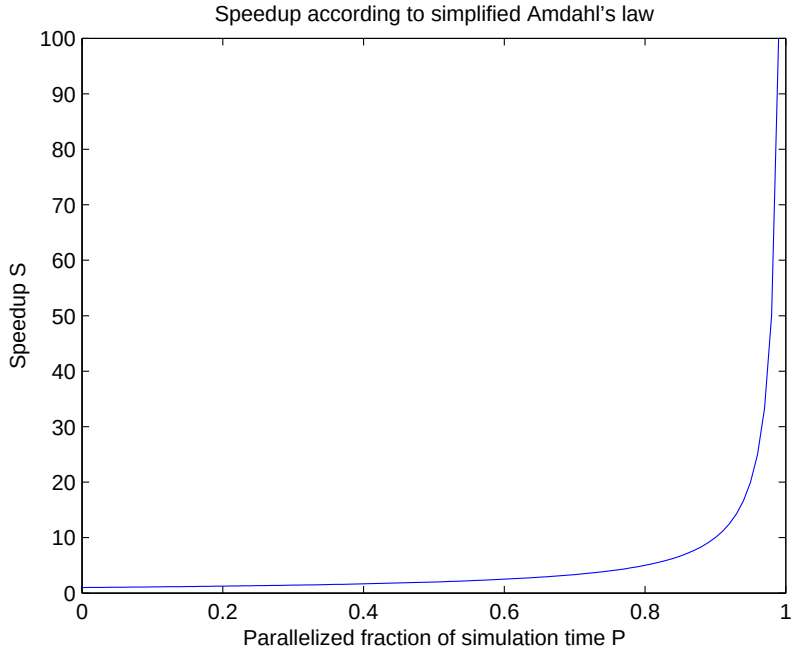

Figure 11: Speed-up calculated from simplified Amdahl's law of equation 16.

It is important to bear in mind that this is a rough estimate for the ideal, theoretical speed-up. In practice, introducing GPU enhancement to a program will also bring new overheads and other delays caused by memory transfers, kernel launches, and other added instructions. These will inevitably reduce the achieved speed-up.

In section 5, some earlier CUDA implementations of matrix-vector multiplication and transistor model evaluation were presented. The SpMV was even $40\times$ faster than the CPU implementation. Based on figure 11, this means that at least 97.5% of the original simulation time should have been spent on the SpMV-algorithm. One example of a transistor model evaluation showed a speed-up of $4\times$ when 75% of the total simulation time was spent in the evaluation. This seems to be quite consistent with the speed-up predicted by the simplified Amdahl's law.

Memory-wise performance

Previous sections repeatedly emphasized the slowness of memory transfers between the host and device. Although excessive memory transfers should be avoided, the execution time spent on memory transfers with the diode implementation turns out to be rather marginal. Theoretical calculations will follow to approximate the transfer times.

This estimate is presented for the parameters that are fetched during diode model initialization and transferred to device only once. For one diode's parameters, a total of 100 double-precision floating-point and 50 integer-type values are allocated. One double-precision value requires 8 bytes of memory and one integer-type requires 4 bytes. This results in a memory allocation of

$$100 \times 8 \text{ bytes} + 50 \times 4 \text{ bytes} = 1000 \text{ bytes} = 1 \text{ kilobyte} \tag{17}$$

Even for a circuit with 1000 diodes, the total memory consumption for these parameters is 1 MByte. NVIDIA GeForce GTX 260 (see section 8.1) has a theoretical external memory bandwidth of 111.9 GB/sec. From these figures, one can easily see that even the overheads of executing memory transfers are likely more significant than the theoretical transfer time of less than 9 microseconds.

Based on these approximations, an assumption can be made that memory transfers should have only a minor effect on the simulation time. Transfers are made between each kernel launch. Controlling voltages and changed values of model parameters have to be transferred several times during simulation. Nevertheless, these transfers involve even less data so their time consumption is quite fixed.

8 Testing and results

8.1 Test setup

All the CUDA development and testing was performed on the same hardware. Although the CUDA code was targeted for the device in hand, the CUDA enhanced program should function on any NVIDIA CUDA device with at least the same compute capability (1.3) (see section 3.4). Further testing and debugging will be conducted when the hardware becomes available. In this section, the test and development setup is described for both hardware and software.

Hardware

The GPU used is the NVIDIA GeForce GTX 260. It has a compute capability of 1.3. This is a minimum requirement because it is the lowest compute capability supporting double-precision floating point numbers. The IEEE 754-2008 standard for binary floating point arithmetic is followed with some deviations listed in Appendix G.2 of [3]. The number of multiprocessors in this device is 24 and it has 192 CUDA cores. The total amount of memory is 2 GB that works at a clock rate of 999 MHz with an external memory bandwidth of 111.9 GB/sec.

The used workstation has an Intel Core i7-860 CPU running at 2.80 GHz. The CPU has a 64-bit instruction set, 8 MB of cache, and 4 cores running a total of 8 threads. The system memory is 8 GB of DDR3.

Software

All the development and testing was done under the 64-bit Windows 7 Home Premium operating system. Nevertheless, the 32-bit development environment was utilized throughout. This was due to the guaranteed compatibility with the 32-bit APLAC software. The used development environment is Microsoft Visual Studio 2008 Express Edition with the Microsoft .NET Framework 3.5 SP1. The CUDA compiler driver NVCC rel. 3.1 was integrated into the Visual Studio environment. The CUDA toolkit version is 3.1. The NVIDIA Compute Visual Profiler version 3.1.1 was used for examining the operation of the CUDA code.

Circuits

The circuits used for debugging and testing were not actual industrially used circuits. This is merely because the diode is not a commonly or widely used component in modern high-tech circuits. Three circuits were used in all. Circuit 1 is the simplest one consisting of three diodes and three voltage sources. The diodes and sources are connected pair-wise in parallel. Every voltage source is defined with a DC voltage E_{dc} and an internal serial resistance R. The defined parameters of the diodes are the saturation current I_{S} and the serial resistance R_{S}. Default values are used for all the other model parameters. Test circuits 2 and 3 share the same topology presented

in figure 12, only the number of nodes differs. Circuit 2 contains 1000 and circuit 3 10 000 nodes.

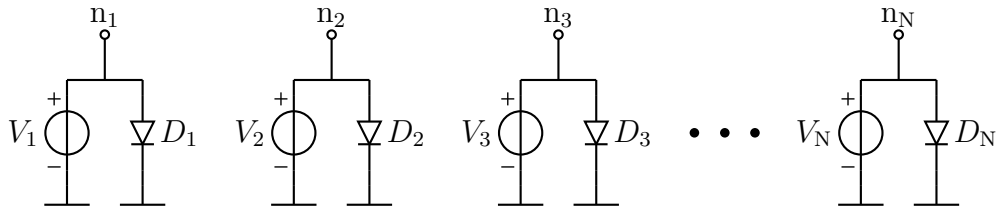

Figure 12: Test circuit topology.

Circuit 1 was used for debugging and verifying the proper functioning of the program. For this purpose, the diodes were defined having electrothermal modeling both enabled and disabled. Also, the `DiodeId` kernel's accuracy was tested with this circuit.

Circuit 2 was for verifying the kernel's correct operation with large circuits consisting of more diodes than the number of GPU cores utilized. Also, some speed testing was performed with this circuit. Circuit 3 was used for kernel speed testing. With this large a number of diodes, the kernel's true performance should come up clearly.

8.2 Accuracy

Overall, no evidence of any serious inaccuracy was detected during the testing. Table 7 shows the node voltages and relative errors for test circuit 1. The table shows that only the sixth and last decimal in one of the nodes is different in the original and CUDA versions.

Table 7: Test circuit 1 node voltages and relative errors.

Node	Original [mV]	CUDA [mV]	Relative error [%]
1	956.366	956.365	1.04×10^{-4}
2	838.961	838.961	0.00
3	978.510	978.510	0.00

One possible reason for this may be the difference in the math functions. In the original version, APLAC's own exponential and logarithm functions include some additional threshold handling for values too small or large, which have not been implemented in the CUDA version. Such small inaccuracies might just as well be caused by differences in the CPU and GPU architectures and differing rounding errors related to this.

8.3 Speed

Theoretical speed enhancements were speculated in section 7.6. As it was shown, the realized speed of a CUDA implementation is extremely hard to predict due to the various factors affecting it. The model evaluation should be faster in parallel but the additional memory transfers along with their overheads and complicated analysis structure reduce the obtained acceleration. Moreover, the acceleration of the evaluation may be useless if it is initially too simple and quick. Next, the realized speed of the diode model CUDA implementation is examined.

Realized speed

Speed testing was done on the three test circuits presented in section 8.1. It should be noted that when evaluating circuit 1, only three of the 192 cores of GeForce GTX 260 are used. This means that 98.4 % of its computing capacity remains idle. Hence, it cannot be expected to be any faster. Circuits 2 and 3, on the other hand, have enough diodes for the GPU. Thus, these circuits make use of the GPU's whole capacity and can potentially perform faster.

In practice, the test simulation times were not what was hoped. The CUDA implementation was slower for all of the circuits. Table 8 presents the total simulation times for both implementations as well as the relative speed difference.

Table 8: Total simulation times for the test circuits.

Circuit	Diodes	Original (s)	CUDA (s)	CUDA/Original	Speedup
1	3	0.578	0.671	1.16	0.86
2	1000	1.01	1.09	1.08	0.93
3	10 000	28.73	30.65	1.07	0.94

On second thought, these results are actually not very surprising. Analysis structure modifications and additional memory transfers increase the simulation time, as was expected. But why did the expected faster model evaluation not cancel this increase? The answer lies in the diode model structure. Equations (1)-(4) in section 7.1 show the nature of the diode model's nonlinearity: the entire model evaluation is based on computing the exponential function. The evaluation turns out to be a rather lightweight computation. Based on the numbers reported by the NVIDIA Compute Visual Profiler, the kernel execution spends less than one second of the GPU time in test circuit 3. This is only 3% of the total simulation time. It is obvious that no matter how much this kernel execution time is decreased, the remaining 97% of the total time spent elsewhere would still remain the same. The simplified Amdahl's law (equation (16)) presented in section 7.6 predicts the same: with a 3% parallelization, the best possible speed-up is only 3%.

8.4 Performance in relation to computational intensity

The poor performance of the `DiodeId` kernel compared to the original function can be proven to result from its lightweight computation. To this end, both the kernel and the original function were intentionally made more complex. The following pseudo-code shows how this is done.

Multiplied computation in `DiodeId`:

```
DiodeId kernel/function
{

    ...

    int k = ...;      //Multiplication factor of Id calculation
    IS = IS/k;

    ...

    Loop for k times
    {
        /* Equations for Id calculation */
        ...
        Id +=...;           //One loop calculates Id/k
    }

    ...

}
```

At the beginning of `DiodeId`, the saturation current I_S is divided by k. As a result, the diode's current I_d calculated from equations (1)-(4) is now I_d/k instead of the correct current. When the calculation of I_d is looped k times and the results are summed, the correct value of I_d is obtained. This way, the computation in `DiodeId` can be multiplied by approximately k times, which gives a rough estimate of how the GPU would perform compared to the CPU if the model was k times more complicated.

Test simulations were carried out on a circuit consisting of 10 000 diodes without electrothermal modeling. The circuit topology is the one shown in figure 12. Table 9 presents the total simulation times for both the CUDA kernel and the original function with different values of k from 1 to 10 000.

Simulations with $k = 1$ are equivalent to the model evaluation without the loop and the results are similar to the earlier tests: too simple a computation is faster on the CPU. Even when $k = 10$ and $k = 100$ the CPU function is slightly faster. However, when $k = 500$ the efficiency of CUDA computation starts to show. The CPU simulation time increases by 2 seconds, whereas the CUDA simulation time still remains approximately the same. When k is further increased the speed difference grows. The last simulations ($k = 10\,000$) bring out the true performance capability of the CUDA device. The CUDA simulation time is now only 25% of the CPU simulation time.

Table 9: Total simulation times for the complex `DiodeId`.

k	Original (s)	CUDA (s)	CUDA/Original	Speedup
1	14.20	15.69	1.10	0.91
10	14.46	15.66	1.08	0.92
100	15.16	15.59	1.03	0.97
500	17.22	15.49	0.90	1.11
1000	19.92	15.88	0.80	1.25
10 000	72.00	17.79	0.25	4.05

These test simulations show, that if the diode model was more complex, it would be faster to compute in the CUDA device. These tests also verify that it is possible to accelerate the evaluation of other component models in APLAC as long as the chosen model is computationally heavy enough. It should be noted that these tests cannot be used to determine what is a sufficiently heavy computation. Model structures of different components are not similar and required data quantities vary. Thus obtaining a CUDA implementation of a model that outperforms the original CPU model is case specific.

8.5 Conclusions after testing

The overall results of this CUDA implementation were actually quite positive. No performance gain was achieved, on the contrary, this implementation was marginally slower than the original. But as was proven in the previous section, the reason for this is the light computation in the `DiodeId` kernel. Because of the poor performance, this implementation does not have any actual use in the simulation software. Although several performance optimization strategies were examined, there is no reason, at this point, to apply them to this implementation in the hope of better performance. The benefits would be marginal.

However, with respect to research and future development of the simulator, this study and the diode implementation have all the more importance. Despite the starting point that was quite strongly against CUDA programming principles, GPU computation was eventually implemented in APLAC successfully. This shows that CUDA implementations of other models are also possible in APLAC. The test simulations in the previous section show that a CUDA version of a model can actually outperform the original model.

The most important outcome of the implementation concerns data handling and analysis structure. The modifications that were required for the diode model to work are applicable for almost any other model as well. It will be significantly easier to implement, for instance, a transistor model in CUDA now that these modifications have already been done.

9 Summary

In this thesis, the overall process of implementing GPU computation in a program has been covered at a general level. Also, a practical example for applying CUDA to a circuit simulation software has been presented. In the beginning, some background information on the development of general purpose GPU computation and its differences to CPU computation was given. GPU and CPU architectures turned out to be quite the opposites of each other. Then, the most efficient GPU architecture, NVIDIA's CUDA, was discussed in detail. The main point here is CUDA's high capacity to process massive data quantities in parallel. This part of the thesis created a basis for the reader to understand CUDA programming techniques better.

The next section concentrated on programming CUDA. First, some programming interfaces and a common CUDA programming model were introduced. Programming a CUDA device was shown to be rather simple for someone with basic knowledge of programming. Then, GPU programming with the CUDA C programming interface was discussed with the aid of a simple example code. This section also covered CUDA program compiling and debugging, as well as some performance optimization strategies. Lastly, there was a brief discussion on multi-GPU programming, which is quite similar to multithreaded CPU programming.

Next, the focus was moved to circuit simulation. The first topic of discussion was finding a suitable program block for CUDA development. Important aspects of this are the processing time taken by a task and how suitable a task is for parallel processing. Next, some earlier CUDA implementations in the field of circuit simulation were discussed. These included matrix-vector multiplication and transistor model evaluation. For both of these, some successful CUDA implementations were presented. After this, the possibilities of a CUDA implementation in APLAC were examined. Here, the SpMV and transistor model were also considered but discarded. However, a suitable target for development was found. The diode model was chosen because it has some similarities to the transistor model but it is significantly simpler.

The rest of the thesis focused on implementing APLAC's diode model with CUDA. The structure and characteristics of the model were first introduced. The diode's nonlinear characteristics are based primarily on the exponential behaviour. The model is also capable of simulating the diode's behaviour under dynamically changing temperature conditions.

Next, the required modifications to the APLAC source code and the actual diode evaluation kernel were explained. The kernel was rather straightforward to implement, but the analysis structure and data handling required somewhat more modification. After this, the application of the previously introduced optimization strategies to the diode model were examined. However, these strategies have not been applied to the model yet. Also, the expected performance of the CUDA model was speculated. This turned out be quite difficult due to several factors affecting the performance. The maximum theoretical speed-up was examined with Amdahl's law.

Finally, the test setup of the implementation was described, and the test results were presented and evaluated. The accuracy of the implementation was almost

perfect, only a marginal error was detected. No speed-up was actually achieved, and the implementation ran slightly slower than the original version. This was shown to be caused by the light computation in the diode model. However, this implementation was successful. It proved that implementing CUDA in APLAC is possible. It also establishes a good basis for future CUDA implementations in APLAC.

Future work

Although, the CUDA diode model implementation is now working, it is not complete. Several of the optimizations that were discussed in this thesis could still be applied to the model, although the implementation is practically impossible to accelerate. This model is, however, simple enough to study and test the impact of the various optimization strategies.

After further studies of this implementation are finished, focus will be turned to the BSIM3 transistor model. In section 5.2, some earlier CUDA applications of transistor models were introduced. The earlier studies show that the transistor model evaluation is extremely heavy and typically consumes most of the total simulation time. Because of this, a successful CUDA transistor model implementation should achieve some actual speed-up, unlike the lightweight diode evaluation.

So, the next objective will be to implement a CUDA BSIM3 transistor model. Because this is a widely used component and model in real-life industrial circuits, a successful implementation could have some actual use.

References

[1] AWR Corporation. http://web.awrcorp.com/

[2] Wang, B., Wu, T., Yan, F., Li, R., Xu, N., Wang, Y. RankBoost Acceleration on both NVIDIA CUDA and ATI Stream platforms. *2009 15th International Conference on Parallel and Distributed Systems*. 2009.

[3] NVIDIA CUDA Programming Guide Version 3.1. NVIDIA Corporation, 2010.

[4] ATI Stream Technology. http://www.amd.com/stream

[5] PTX: Parallel Thread Execution ISA Version 2.1. NVIDIA Corporation, 2010.

[6] DirectX Developer Center. http://msdn.microsoft.com/fi-fi/directx

[7] OpenGL. http://www.opengl.org/

[8] Khronos Group. http://www.khronos.org/

[9] NVIDIA Developer Zone. DirectCompute. http://developer.nvidia.com/directcompute

[10] The Portland Group. PGI CUDA Fortran Compiler. http://www.pgroup.com/resources/cudafortran.htm

[11] Breitbart, J. CuPP - A framework for easy CUDA integration. *Parallel and Distributed Processing Symposium (IPDPS)*. 2009.

[12] NVIDIA CUDA C Best Practices Guide Version 3.1. NVIDIA Corporation, 2010.

[13] NVIDIA CUDA Reference Manual Version 3.0. NVIDIA Corporation, 2010.

[14] The CUDA Compiler Driver NVCC Version 3.1. NVIDIA Corporation, 2010.

[15] GCC, the GNU Compiler Collection. http://gcc.gnu.org

[16] CUDA-GDB (NVIDIA CUDA Debugger) User Manual Version 3.2. NVIDIA Corporation, 2010.

[17] Dongarra, J. J., Duff, I. S., Sorensen, D. C., van der Vorst, H. A. Numerical Linear Algebra for High-Performance Computers. Philadelphia, Society for Industrial and Applied Mathematics, 1998.

[18] CUDA Data Parallel Primitives Library. http://code.google.com/p/cudpp/

[19] NVIDIA CUDA CUBLAS Library Version 3.1. NVIDIA Corporation, 2010.

[20] Baskaran, M. M., Bordawekar, R. Optimizing Sparse Matrix-Vector Multiplication on GPUs. IBM Research Report, 2009.

[21] Cao, W., Yao, L., Li, Z., Wang, Y., Wang, Z. Implementing Sparse Matrix-Vector Multiplication using CUDA based on a Hybrid Sparse Matrix Format. *2010 IEEE International Conference on Computer Application and System Modeling (ICCASM 2010)*. 2010.

[22] Poore, R. E. GPU-Accelerated Time-Domain Circuit Simulation. *IEEE 2009 Custom Integrated Circuits Conference (CICC)*. 2009.

[23] Gulati, K., Croix, J. F., Khatri, S. P., and Shastry, R. Fast Circuit Simulation on Graphics Processing Units. 2009.

[24] Bayoumi, A. M., Hanafy, Y. Y. Massive Parallelization of SPICE Device Model Evaluation on GPU-Based SIMD Architectures. *IFMT '08*. 2008.

[25] CUDA CUSPARSE Library Version 0.1. NVIDIA Corporation, 2011.

[26] Virtanen, J. APLACin mallituskieli (mallien implementointi C-kielellä). CT-group internal document, 2005. (in Finnish)

[27] APLAC RF Design Tool Version 8.50 Reference Vol. II. AWR-APLAC Corporation, 2009.

[28] APLAC RF Design Tool Version 8.50 Reference Vol. I. AWR-APLAC Corporation, 2009.

[29] Virtanen, J. Nonlinear DC Analysis. Lecture notes, Numerical Circuit Design Methods, 2010.

[30] Amdahl, G. Validity of the Single Processor Approach to Achieving Large-Scale Computing Capabilities. *AFIPS Conference Proceedings (30): 483-485*. 1967.