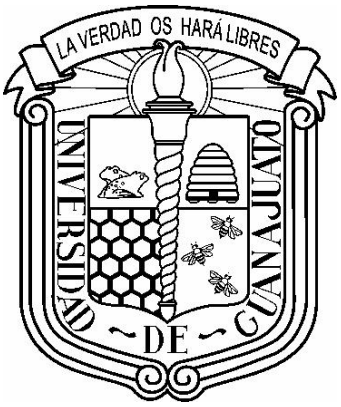

Universidad de Guanajuato

Campus Irapuato Salamanca
Division de Ingenierias

Study of Domain Wall Dynamics under Nonlocal Spin-Transfer Torque Using Heterogeneous Computing

Tesis Profesional

Para obtener el grado en Licenciatura en Ingeniera en Sistemas Computacionales

Autor:
Thomas Sanchez Lengeling

Director:
Dr. David Claudio González
Co-Director:
Dra. María Susana Ávila García

Sinodales:
Dr. Víctor Ayala Ramírez
Dr. Juan Gabriel Aviña Cervantes

Guanajuato, Mexico

April 2015

UNIVERSITY OF GUANAJUATO

Abstract

Campus Irapuato Salamanca
Division of Engineering
Department of Electronic Engineering

Bachelor of Computational Systems Engineering

Study of Domain Wall Dynamics under Nonlocal Spin-Transfer Torque Using Heterogeneous Computing

by Thomas Sanchez Lengeling

This work is an exploration of the role that Graphical Processing Units, also known as GPUs, can play in the acceleration of physical simulations. In particular, in the research of spintronic effects such as the dynamics of domain walls under nonlocal spin-transfer torque. Our study is relevant because it allows researchers to quantitatively test some of the effects of a phenomenon known as spin-diffusion on magnetic configurations at the nanoscale. Some of such configurations are known as domain walls. These magnetic configurations can be observed experimentally in NiFe soft nanostripes but they are really complicated to produce and image experimentally. Due to this, we use the massively parallel capabilities of a single GPU to numerically solve a mathematical equation, known as the Zhang-Li Model. As a consequence of our implementation, we have observed a 8.0x speed-up in the solution of the equation. This speed-up is obtained when we compare the time needed to obtain the result of a simulation in a GPU with that of a simulation with the same input parameters in a conventional processor e.g. Intel Xeon. The numerical method used for the solution is a the method known as Finite Differences in the Time Domain (FDTD) whose integration is done using a 4th order integrator.

Acknowledgements

To my advisor David Claudio for being supportive and flexible throughout the thesis endeavor; he allowed me to continue learning GPU though various experiences.

I want to thank my family, especially my parents, Manuel and Martha, and my two brothers Benjamin and Gabriel, as well as everyone that help me through the process. Karen encouraged me to finish fast and stop playing DOTA.

All my bachelor colleagues: we survived together through the courses, projects and work, and all the friends that I made along the way. Finally, to the University of Guanajuato for the support and assistance in the process. I had numerous experiences at the University that will allowed me to continue to improve myself.

Contents

List of Figures

Introduction

Commodity graphics processing units (GPUs) are becoming increasingly popular to accelerate scientific applications due to their low cost and potential for high performance when compared with central processing units (CPUs). A large number of contemporary problems and scientific research are being benefited from this new technology. There has been considerable progress in implementing the hardware and the supporting infrastructure for GPUs programming and streaming architectures. This thesis is a study of heterogenous computing using NVIDIA's GPU applied to computational physics.

The first chapter is a overview of heterogeneous architecture programming with NVIDIA's GPUs using NVIDIA's programming framework, Compute Unified Device Architecture (CUDA). The second chapter is the study of the phenomenon known as spin-diffusion on magnetic configurations at the nanoscale. Some of such configurations are known as domain walls. These magnetic configurations can be observed experimentally in NiFe soft nanostripes. Due to this, we use the massively parallel capabilities of a single NVIDIA GPU to numerically solve a mathematical equation, known as the Zhang-Li Model. The numerical implementation is done by using NVIDIA's CUDA platform, which is explained in chapter 3. In addition, the numerical solution is a the method known as Finite Differences in the Time Domain (FDTD) whose integration is done using a 4th order Runge-Kutta integration. The fourth chapter focuses on optimization techniques and practices, to gain the most performance out of the hardware capabilities. The fifth chapter are the results collected by applying optimization techniques to the initial CUDA code. Techniques applyed such as concurrent kernels, shared memory, branching and occupancy. The outcome is compared by launching the code on-to several GPUs nodes using the supercomputer Piritakua. Finally, the last chapter of the thesis is a conclusion of the work and future research.

Chapter 1

Heterogeneous Computing

Heterogeneous computing refers to a system that combines several processor types to gain more performance, typically using a single or multi-core computer processing units (CPUs) and a graphics processing units (GPUs). Frequently GPUs are known for 3D graphics rendering, video games and video editing, but GPUs are becoming increasingly popular for accelerating computing applications and scientific research due to their low price, high performance and relatively low energy consumption per FLOPS (floating point operations per second) when compared with the CPUs. This chapter provides an overview of GPUs within the High Performance Computing (HPC) context, their advantages and disadvantages and how they can be integrated in to a scientific software and research.

1.1 Motivation

The GPU has been an essential part of personal computer since its early use. Over the course of 30 years the graphics architecture has evolved from drawing a simple 3D scene to be being able to program each part of the GPU graphics pipeline. Their role became more important in the 90s with the first-person shooting video game DOOM by id Software. The demanding video game industry has brought year by year more realistic 3D graphics. Consequently new innovated hardware capabilities have been developed to increase the graphics pipeline and the render output. This leads to a more sophisticated programming environment with a massive parallel capabilities.

The fixed graphics pipeline (fixed functions on the GPU) was introduced in the early 90s and this allowed various customization of the rendering process. However, it allowed some modifications of the GPU output. Specific adjustment were extremely complicated

and did not allow custom algorithms. In 2001 NVIDIA and ATI (AMD) introduced the first programmability to the graphics pipeline, which could control millions pixels and vertex output in a single frame, Moreover, it out-performed the CPU for real-time rending. In addition, graphics shifted from the CPU to the GPU. This was the beginning of GPU parallel capabilities [20].

At first the GPUs were only used for general-purpose computing (GPGPU) like computer graphics, but in resent years the GPU has been used to accelerate scientific research, analytics, engineering, robotics and consumer applications.

GPUs are attractive for certain type of scientific computation as they offer potential seedup of multi-processors devices with the added advantages of being low cost, low maintenance, energy efficient, and relative simple to program. Many algorithms in applied physics are using GPUs to improve their performance over the CPU. Some area of scientific research that obtain the benefit of heterogenous computing are: Molecular Dynamics, Quantum Chemistry, Computational Structural Mechanics and Computational Physics [21].

In any case, for a given simulation a compromise between speed and accuracy is always made. The current tendency of the CPU relies on increasing the clock speed, decreasing the size of transistor and finally adding more cores per unit and be able to work and a parallel manner. Therefore there are limitations to this paradigm [25].

Power Wall

The CPUs single core has not gone beyond the 4GHz barrier, a paradigm shift from a single core to a multi-core CPUs. Also the power use of CPUs is very high per Watt. The figure 1.1 shows the comparison of performance between the GPU and CPU.

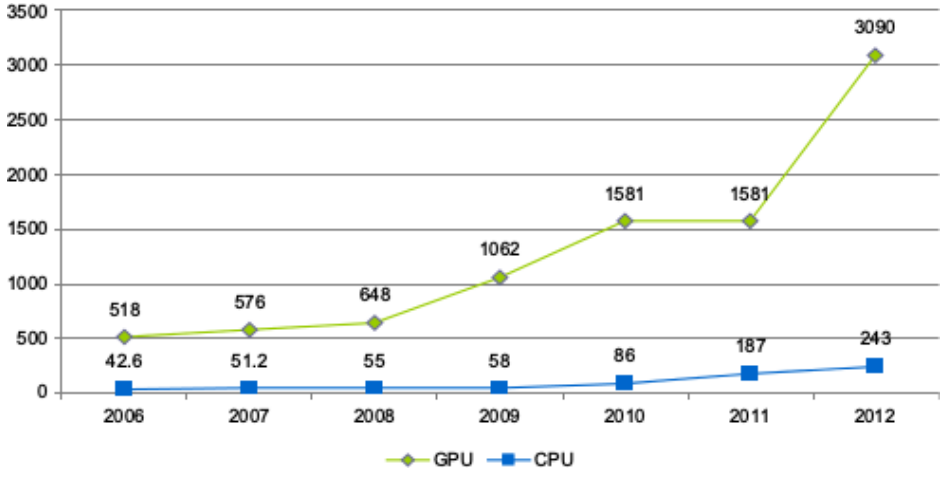

FIGURE 1.1: GPU and CPU peak performance in gigaflops

Memory Wall

This refers to the growing disparity of speed between CPU and the memory outside the CPU chip. Some applications have become memory bound, that is to say computing time is bounded by the transfer memory between the CPU and all the hardware devices connected to the CPU, commonly to the Peripheral Component Interconnect (PCI) chip. In conclusion, the computing time is bounded by the memory and not by the time calculations performed on the CPU.

Parallelism Wall

This indicates a law that indicates the number of parallel processes. The number N parallel processes is never ideal and always depends on the problem. The speedup is described by Amdahl's Law in terms of the fraction of parallelized work f [25].

$$Speedup \leq \frac{N}{f + N(1 - f)}$$

The current paradigm of using CPUs for computing is growth unsustainable. In 2012, Japan was among the countries with elite supercomputers and Japan built the machine "K Computer", with 705,024 multi-core CPUs, achieving up-to 11.3 petaflops (10^5 flops). Furthermore, the computer is one of the most power efficient supercomputer in the world with a total of 12.66 megawatts (MW), in other words 830 Mflops/watt. This is enough to power a small town of 10,000 homes. If the current thread of power use continues, the next supercomputer would require 200 MW of power; this would require a nuclear power reactor to run it [28]. However, in 2013 Oak Ridge Nacional Laboratory (U.S) built a supercomputer that combines CPUs and GPUs, the Titan. It obtains an astonishing 24 petaflops theoretical peak performance and with a power consumption of 8.2 MW. They demonstrated that it is possible to built a supercomputers that combines CPUs and GPUs, which enables a higher performance and lower power consumption compared to a CPU based supercomputer [1].

As mentioned, the GPU exceeds the CPU in calculations per second FLOPS with a low energy consumption. However, the GPU is designed to launch small amounts of data in parallel with only several instructions, and in other words the GPU swap, switch threads very fast and they are extremely lightweight. In a typical GPU system, thousands of threads are waiting for call request to start working. While on the CPU, it runs up-to 24 threads on a hex-core processor. They can execute a single operation on comparatively large set of data with only one instruction, and this can be extremely cost-wise operation on the GPU.

1.2 GPUs as computing units

The GPU, unlike its CPU cousin, has thousands for registers per SM (Streaming Multiprocessor), which are arithmetic processing units. A SM, in other words, is similar to a multi-thread CPU core. A typical CPU has two, four, six or eight cores. GPU there are as many as n SM core. The SMs are configured in such a way that they are able to access memory location close by. We can see this in the Figure 1.2. For a particular calculation, all the stream processors within a group execute exactly the same instruction on a particular data stream, then the data is sent to the upper level, the host (CPU) [6].

CUDA cores are the number of processors in a single NVIDIA GPU chip. For example one of the first GPU capable of running CUDA code was the NVIDIA 9800 GT, which had 112 cores, while the latest high-end GPU GTX 980 has 2048 cores.

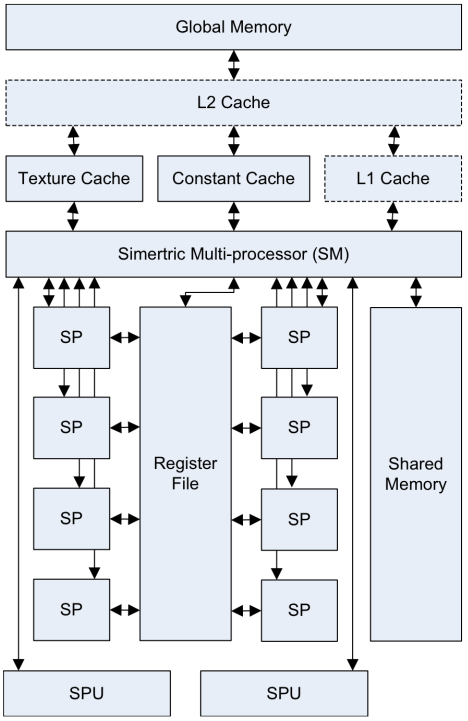

FIGURE 1.2: SM Architecture [6].

Each CUDA core can execute a sequential thread, just like a CPU thread, which NVIDIA calls it Single Instruction, Multiple Thread (SIMT). In addition, all cores in the same group execute the same instruction at the same time, much like classical SIMD (Single instruction, multiple data) processors. SIMT handles conditionals somewhat differently than SIMD, though the effect is much the same, where some cores are disabled for conditional operations. In other words, a single instruction is executed throughout the device.

Being able to efficiently use a GPU for an application requires exposure to the inherent data-parallelism Optimized for low-latency, serial computation. This can be seen in contrast with a CPU, which is optimized for sequential code performance, fast switching registers and sophisticated control logic allowing to run single complex programs as fast as possible, which is not possible on the GPU. Memory management is very important for GPUs. Refers how to allocate memory space and transfer data between host (CPU) and device (GPU). While the CPU memory hierarchy is almost non-existent, on the GPU inherent data is important [16]. Figure 1.3 illustrates the memory hierarchy of the device. In addition the global memory is huge in comparison with the L1/Cache and the texture memory. However, the access to the global memory is slow in comparison to the other. We can observe in figure how the data is sent from the host to the device and vise-versa.

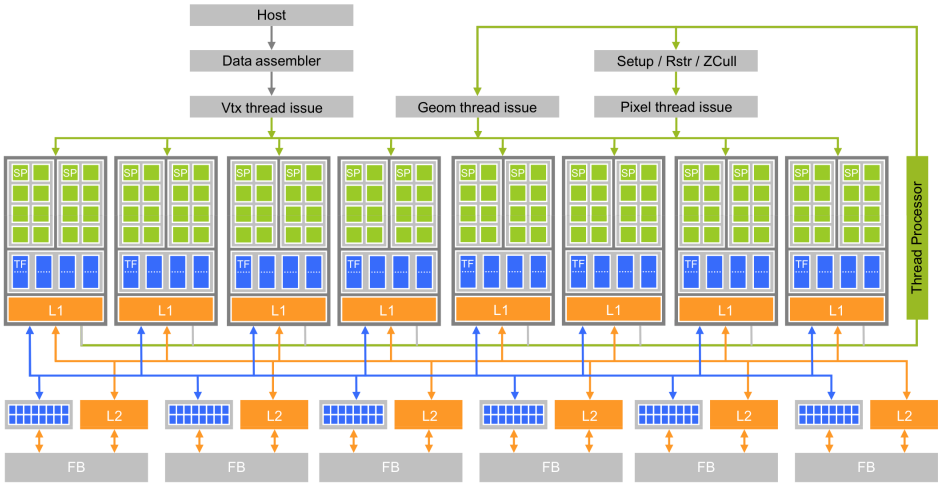

FIGURE 1.3: Unified programmable processor array of the GeForce 8800 GT graphics pipeline [16].

On the GPU, floating point precision and optimization are very important. However, there is a penalty for choosing either performance or precession. All the GPUs are optimized for single precision floating operations, a 24 bit size. NVIDIA also provides a double precision point, size of 53 bits, which is a standard based on IEEE 754 notation. Normally the GPU uses the single precision (SP) by default. If a double precision (DP) is chosen, normally there is a penalty between 2x and 4x speedup[35]. Libraries such as CUBLAS and CUFFT provide useful information how NVIDIA handles floating point operations under the hood.

1.3 Programming on GPUs

There exist among many, two main computing platforms; NVIDIA's Compute Unified Device Architecture (CUDA), and Khronos's Open Computing Language (OpenCL). NVIDIA's CUDA provides the necessary tools, frameworks and library to program parallel application. While, the OpenCL is a open standard framework meaning that is possible to do parallel computing on other GPUs, like on AMD cards. Programmers can easily export their code to others graphics cards. However, CUDA has more robust debugging and profiling for GPGPU computing. The two frameworks are developed to be close to the hardware layer, using the C programming language as primarily programming language. Furthermore, CUDA provides both a low level API and a higher level API. Those who are familiar to OpenCL and CUDA, can easily modify their code to work on either platform [16].

CUDA programming model views the GPU as an accelerator processor which calls parallel programs throughout all the SM [36]. In addition, the CUDA parallel programs are only launched on the device (GPU) and are named as kernels. The kernels are executed across a large amount of threads, which contains the CUDA code. The basic idea of programming on a GPU is simple, the following steps explains the procedure in Figure 1.4.

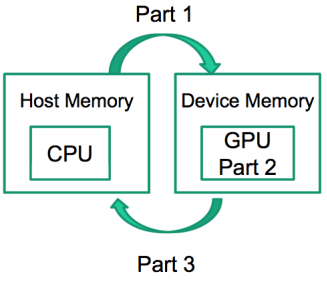

FIGURE 1.4: Programming Cycle between the CPU and GPU [30]

- Create memory(data) for the host (CPU) and devices (GPUs).
- Send the data host memory to the highly parallel device.
- Do something with data on the device, e.g. matrix multiplication,calculation, parallel algorithm.
- Return the data from the device to the host.

The structure of CUDA reflects the coexistence of CPU and GPUs. The CUDA code is a mixture of both host code and device code. Moreover, the device code is an extension of the C compiler with additional namespaces/CUDA keywords for parallel code;

the CUDA compiler is called NVCC. The host code is the standard low level ANSI C language. However, it is possible to program applications in C++, Python and Fortran. While the standard C code has extension marked as .c for source and .h for headers files, the CUDA code has extensions of .cu for source files and .cuh.

Kernels are launched or executed on a large amount of threads in the SM. They can be configured by threads per block and by block per grid. The thread and block configuration is illustrated in the Figure 1.5. A thread is the simplest executing process. It consists of the code of the program, the particular point where the code is being executed [16]. In addition all the threads in a kernel can access the global memory (RAM), Figure 1.3 illustrates the physical position. Moreover, many threads form a block, and many blocks form a grid. CUDA handles the execution of the random-access threads, which take up-to very few clock cycles in comparison to CPU threads.

Each of the threads is able to obtain a implicit variable that identifies its position within the thread block and its grid. The thread access for the x coordinate is showed in the Listing 1.1. This is only the case for 1D block, which is widely used for shared memory access (see chapter 4) [30].

```
int index = blockIdx.x * blockDim.x + threadId.x;
```

Listing 1.1: 1D thread block operation on a CUDA kernel

For the case of a 2D thread block, the Listing 1.2 describe such configuration. In addition, the 2d thread block is the most common thread block configuration. It Is also possible to configure a 3D thread block just by adding the z coordinate to the threadIdx index. However, it is very limited.

```
int i = blockIdx.x * blockDim.x + threadIdx.x;
int j = blockIdx.y * blockDim.y + threadIdx.y;

int index = j * YSIZE + i;
```

Listing 1.2: 2D thread block operation on a CUDA kernel

$$blockIdx.x \times blockDim.x + threadId.x$$

In CUDA, host memory and device memory have separate memory spaces. Both of them have physically a separated location, for example, the RAM for either the CPU or the GPU. Furthermore, the programmer requires to send data from the host memory to the device's global memory (RAM) and vice-versa. The process is illustrated in the Figure 1.3. Memory which is allocated in the device needs to be freed on the device, and the same occurs for the host memory. Moreover, the process is accomplished with similar

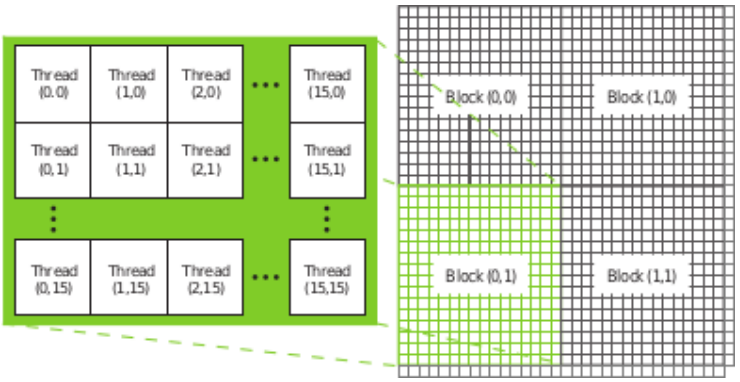

FIGURE 1.5: Thread per block division and block per grid [16].

device operations, free or delete in C/C++. Some of the operations are performed by CUDA's Application Programming Interface (API) on behalf of the programmer [16].

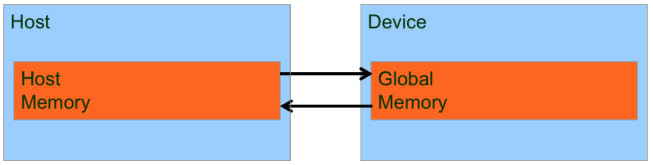

FIGURE 1.6: Memory transfer between CPU and GPU [16]

1.3.1 CPU and GPU multithread comparison

Current CPUs are typically multicore systems, which are capable of parallelizing code fairly easy. In addition, this suggests a parallel system. However, we would require a large infrastructure [30]. For example, if we want to implement a simple vector addition using the CPU cores, we would require to compute a portion of the code on each code, one core the even numbers and the other core the odd numbers, see Figure 1.7. Furthermore, this makes the implementation difficult to scale and require many cores, which are not so easily available after a 8 core system.

CPU CORE 1

```
void add( int *a, int *b, int *c )
{
    int tid = 0;
    while (tid < N) {
        c[tid] = a[tid] + b[tid];
        tid += 2;
    }
}
```

CPU CORE 2

```
void add( int *a, int *b, int *c )
{
    int tid = 1;
    while (tid < N) {
        c[tid] = a[tid] + b[tid];
        tid += 2;
    }
}
```

FIGURE 1.7: CPU Core process [16].

On the GPU we are able to accomplish the same result of the CPU with a less amount of code. The Listing 1.3 demonstrates how to evaluate an addition of two matrices. a

and b, then return the result in the matrix c. The difference between the codes, is that the GPU executes the kernel across all threads configured by the kernel call. Moreover, it enables a highly parallel process with just a couple lines of code.

```
__global__ void add( int *a, int *b, int *c ) {
    int tid = blockIdx.x;
    if (tid < N)
        c[tid] = a[tid] + b[tid];
}
```

Listing 1.3: GPU Kernel threads launch

For example, to execute a kernel with 32 x 32 threads per block and 15 x 4 blocks per grid, we just include the block and the threads dimensions when calling the kernel in the main loop 1.4. Finally, the kernel will spam the CUDA code across all the configured threads in the device.

```
dim3 blocks(15, 4);
dim3 threads(32, 32);
add<<< blocks2, threads, 0 >>>(A, B, C);
```

Listing 1.4: Kernel call with configurable threads per block and block per grid

To conclude, this chapter provided a overview of heterogeneous programming in a modern context. CUDA enhance the C language with parallel computing support. In addition, it is possible to launch millions of parallel threads, oppose to, a few threads on the CPU. The number of GPU cores will continue to increase in proportion to increase in available transistors as silicon process improve. In addition, GPUs will continue to go through vigorous architectural evolution, despite their proof for high performance on data-parallel applications.

Chapter 2

Introduction to Domain Wall Dynamics under Nonlocal STT

This chapter is a brief overview of the theory of spintronics and the study of Domain Wall Dynamics under Nonlocal Spin-Transfer-Torque, which quantitatively test the effects of spin-diffusion, on real Domain Wall (DW) structures, by numerically implementing the Zhang-Li Model on a NiFe soft nanostrip. The numerical method used for the solution is a the method known as Finite Differences in the Time Domain (FDTD) on a 3d cell grid with whose integration is done using a 4th order integration of Runge-Kutta (RK4).

2.1 Theory

The electrons not only carry an elementary unit of charge e, but they also carry an elementary unit of angular momentum. Whenever we produce an electrical current by inducing motions of electrons, it could indeed be viewed as a collection of little magnets that are moving around, see Figure 2.1. In other words, any electron charge transport is simultaneously accompanied by a transport of spin, or magnetic moment carried by these electrons [34].

2.1.1 Spintronics

Spintronics is a new type of electronics that exploit the spin degree of freedom of an electron in addition to its charge [37], see Figure 2.1. The interest is motivated by the quest to understand basic physical principles underlying the electron and spin interactions in materials and possible technological applications. The field of spintronics has attracted

massive interest since the discovery of giant magnectroresistence (GMR) effect in 1988 by Albert Fert and Peter Grünberg who were awarded the 2007 Nobel Prize in physics. The GMR effect has been widely used in hard disk drives (HDD), which have deliver a huge impact on industries and consumer electronics. Spintronics is a promising technology which will complement the present electronics with additional "spin" quantum freedom to charge freedom that is currently used in devices [11].

2.1.2 Spin Transfer Torque

A torque is simply a time rate of change of angular momentum [26]. Hence, spin transfer torque occurs when spins flowing from one layer to another can reorient the magnetization in the layers, see Figure 2.3. The magnetization of the ferromagnet changes the flow of spin angular momentum by exerting a torque on the flowing spins to reorient them, and therefore the flowing electrons must exert an equal and opposite torque on the ferromagnet. This torque that is applied by non-equilibrium conduction electrons onto a ferromagnet is what we will call the spin transfer torque [26].

Spin current which is a flow of spin angular momentum, is generated in addition to the charge current. The spin current normally appears in ferromagnets. However, it should be able to be generated in non-magnets. The simplest method of generating a spin-polarized current in a metal is to pass the current throughout a ferromagnetic material. A common application is the GMR as mentioned before [32].

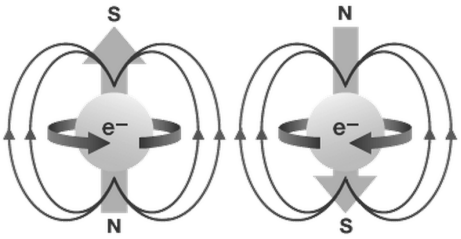

FIGURE 2.1: Electrons not only carries charge, but also spin and magnetic properties [14].

Spin polarized transport occurs naturally in any materials which have a spin imbalance between spin-up and spin-down at the Fermi Level. It occurs at spin-down electrons and is nearly identical, but states are shifted in energy with respect to each other. The Fermi level is the highest energy level which an electron can occupy at the absolute zero temperature. Since the electrons are in the lowest energy state hence the Fermi level is in between the valence band and the conduction band [32].

2.1.3 Domain Wall

An abrupt in magnetization at the boundary of two anti-aligned domains is not a favorable condition. Domain walls form between such domains as means of minimizing the energy of the two anti-aligned domains. Domains walls are transitions layers in which the magnetization changes gradually from on magnetization to another. In other words this refers to the boundaries between regions of uniform magnetization. The gradual change prevents the large increase in exchange energy that would accompany an abrupt change in the magnetization angle. Common domain wall geometric include Bloch walls, Néel walls and vortex walls [9]. In this study only two DW are analyzed the Vortex Wall and the Asymmetric Transverse Wall.

Vortex Wall (VW)

In the case of Vortex wall the magnetization rotates in the plane perpendicular to the domain wall, but the local magnetization is wrapped around a single vortex point, see Figure 2.2.

Asymmetric Transverse Wall (ATW)

The transverse wall has a reflection symmetry about a line perpendicular to the strip axis, and a lack of symmetry about the center line of the strip. However, the asymmetric transverse wall, is the absence of that symmetry, see Figure 2.2.

FIGURE 2.2: Vortex Wall (VW) and Asymmetric Transverse Wall (ATW) [5].

2.1.4 Spin Torque in Domain Walls

Domain walls are the foundation for various spintronics devices that use magnetic momentums, in other words, spin of electronics, the used of the spin degree of freedom. see Figure 2.3 illustrates a micromagnetic model of the domain wall trapped in a nanowire. The domain wall can be pushed along the wire in a controllable manner by applying an external magnetic field or by passing an electrical current through the wire [15].

The energy of the incoming carrier is no the only factor that determines whether or not it passes to the other side of domain wall, the spin also must be taken into account. Since each spin orientation experiences a different potential, simulation of such properties is necessary.

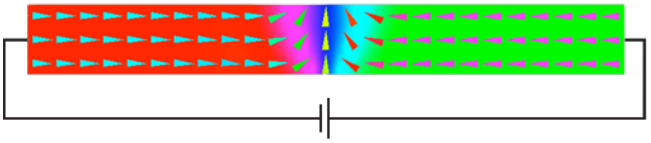

FIGURE 2.3: Domain Wall in a nanowire while passing a current

Spin Torque induced domain wall motion opens up a host of possibilities for applications. The success of spintronics untimely depends on out ability to precisely control the polarization of electrons transported within the actual thin film structure [27]. Advances in spintronics recognized by 2007 Nobel Prize in Physics have enabled over the last decade advances in computer memory, in hard drives. This is a metal based structures which utilize magnetoresistive effects to save and read data from a magnetic disk [32].

In 2008 Parkin, designed a new type of memory storage, the racetrack memory. In addition, the new technology uses Spin Torque to store data. The racetrack memory stores bits along a single ferromagnetic wire. To write and read information, a current is applied along the wire that moves the bits to writing or reading unit [24].

2.2 Domain Wall Dynamics under Nonlocal STT

The motion of domain walls due to spin transfer torque (STT) of electrons has been studied theoretically and experimentally. Furthermore, moving magnetic domain walls using electric currents via spin-torque effects is one the recent developing in spintronics. We analyze a moving domain wall on a soft nanostrip, because it concentrates all of the magnetization non-uniformity, which acts as a built-in detector for spin torques. The inclusion of STT into micromagnetics has up to now been performed with local terms that express the STT as a function of only the local magnetization [5].

2.2.1 Theoretical Approaches

The inclusion of STT into micromagnetics has up to now been performed with local terms that express the STT as a function of the local magnetization only. The magnetization dynamics is described by the classical Landau-Lifshitz-Gilbert (LLG) equation [5], expanded to a STT variable.

$$\frac{\partial \vec{m}}{\partial t} = \gamma_0 \vec{H}_{eff} \times \vec{m} + \alpha \vec{m} \times \frac{\partial \vec{m}}{\partial t} - \vec{T} \tag{2.1}$$

The idea of incorporating spin torque into the LLG equation has been incorporated into a model proposed by Zhang-Li in 2004 [19]. The LLG equation 2.1 incorporated effects of a spin-polarized current in a magnetic system, and the resulting spin transfer. They develop a form for the spin torque based on the spatial variation of the magnetization, an appropriate approach for domain walls. Then in 2005 the same authors Zhang-Li extended this idea working out the difference between the adiabatic and non-adiabatic torque contributions. [38].

$$\frac{\partial \delta\vec{m}}{\partial t} = D_0 \bigtriangledown^2 \delta\vec{m} - \frac{1}{\tau_{sd}}\delta\vec{m} \times \vec{M} - \frac{1}{\tau_{sf}}\delta\vec{m} + (\vec{\mu}\cdot\vec{\bigtriangledown})\vec{M} \tag{2.2}$$

The previous Equation 2.2 is referred as the Zhang-Li Model. It represents a non-adiabatic spin torque, along the presence of spin diffusion. Spin diffusion is a process by which magnetization is exchanged spontaneously between spin, which spin is able to accumulate in metals. The associated diffusion current flows in all directions, giving rise to nonlocal effects. The diffusion term of the Equation 2.2 which carriers drift-diffusion equation implies that the spin density does not depend solely on the local magnetization, which gives rise of nonlocal magnetics effects [5].

Amongst the rapidly growing variety of proposed and developed spin structures, nonlocal spin detection devices, where measurement and current excitation paths are spatially separated, have recently gained a prominent position [37].

2.2.2 Experiment

We quantitatively test the effects of spin diffusion, on domain walls structures. Therefore, we numerically solve the Zhang-Li model along a soft nanostrip of NiFe, using the Equation 2.2. Zhang-Li's research initially solves analytically the diffusion Equation 2.2, However, it ignores the term of spin diffusion. Therefore we numerically solve the solution of the mode including the spin diffusion term [38].

The sample considered is a 300 nm wide and 5 nm tick NiFe soft nanostrip. The dimensions are widely used for experimental practice. Two domain walls are used a Asymmetric Transverse Wall (ATW) and a Vortex Wall (VW). ATW maps of magnetization components of non equilibrium spin accumulation under a uniform current density with $D = 0, 1$ and 10 nm^2/ps, see Figure 2.4.

Vortex Wall (VW) same as for Asymmetric Transverse Wall (ATW), has noticeable effects of the diffusion constant around the vortex core, which is the smallest feature of the wall, illustrated in Figure 2.5.

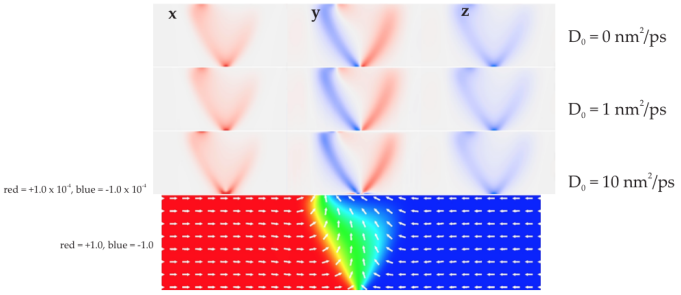

FIGURE 2.4: Asymmetric Transverse Wall (ATW) results [5].

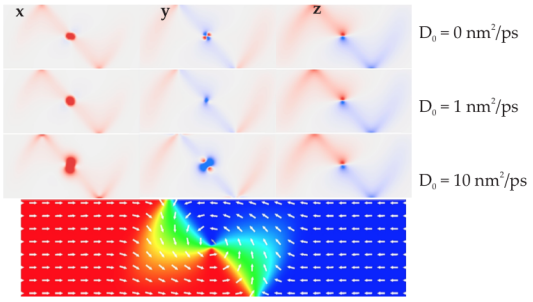

FIGURE 2.5: Vortex Wall results [5].

2.3 Numerical Solution

The Zhang-Li Model 2.2 is physically real, however, computationally expensive. Therefore we need to simulate the physical behavior. The numerical methods used for the solution is a the method known as Finite Differences in the Time Domain (FDTD) whose integration is done using a 4th order Runge-Kutta integration.

2.3.1 Finite differences in the time domain

The finite difference in the time domain (FDTD) method is able to solve complicated problems. However, it is generally computationally expensive. Solutions may require a large amount of memory and computation time [31]. FDTD, amongst otherd, is a numerical analysis technique use for approximating solutions for differential equations. The method belongs in the general class of grid-based differential numerical modeling methods. Demonstration of the numerical methods id in this section[7].

The FDTD method essentially uses a weighted summation of functions values at neighboring points to approximate the derivate at a particular point, in this case a point in a 3d grid. The result for each cell is based on the results from the cell and its neighbors at the previous time-frame, illustrated in Figure 2.6.

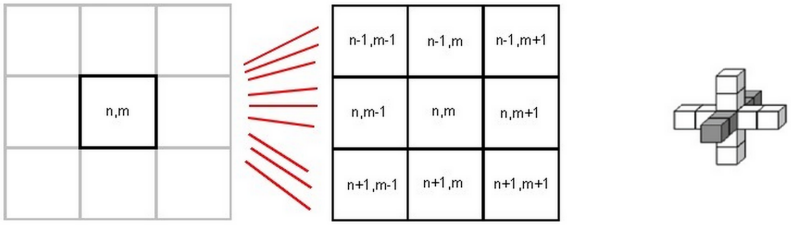

FIGURE 2.6: The result for each cell is based on evaluating the derivate cell neighbors [7].

The magnetization is sampled on a uniform rectangle mesh at points $(x_0 + i\bigtriangledown_x, y_0 + j\bigtriangledown_y, z_0 + k\bigtriangledown_z)$. The computational cell is centered about the sample point with dimensions. $\bigtriangledown_x \times \bigtriangledown_y \times \bigtriangledown_z$ [7].

Looking at the Equation 2.2, we need a method to calculate the first and second derivate. With the Taylor expansion we are able to perform such calculation. The Second order Taylor expansion readily yields expressions for the first and second central derivates. First and second-order derivates of the magnetization components in order to define the divergence of the magnetization $(\nabla \cdot m)$, and the components of the exchange field $(\nabla^2 m)$, respectively. The magnetization components along boundaries also need to be evaluated in order to define surface charges $(m \cdot n)$. Boundary conditions need to be incorporated in the evaluated of the effective field without loss of accuracy.

Consider a regular, differentiable one-dimension scalar function $f(x)$ sampled at regular intervals of a, see Figure 2.7. Second order Taylor expansion readily tiles expressions for the first and second central derivates that are widely used in numerics, namely $\frac{df}{dx} = \frac{f_{i+1} - f_{i-1}}{2a}$ and $\frac{d^2 f}{dx^2} = \frac{f_{i+1} - 2f_i + f_{i-1}}{a^2}$ [7].

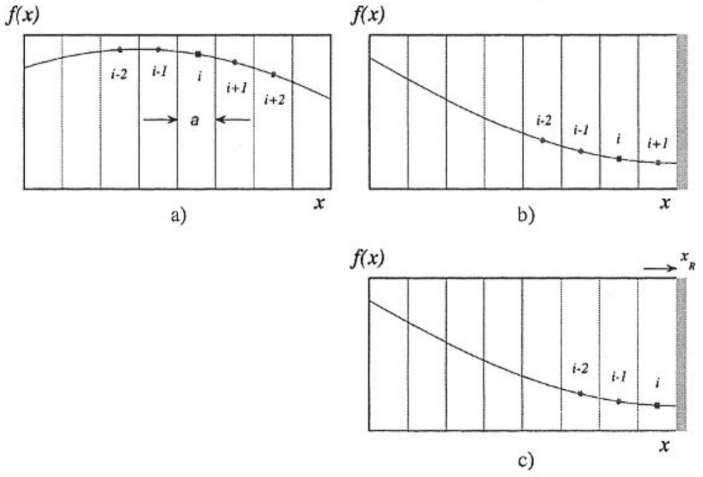

FIGURE 2.7: Sampled at regular intervals a, (a) Function of inside the grid. (b) Mesh points second to closest to boundary. (c) Mesh points closet to boundary

However, the numerical derivation of the structure of a simple Bloch wall using such expressions soon reveals that second order Taylor expansion leads to restricted accuracy. Fourth order expansion has actually been found to prove much superior [7].

Taylor expansion of the function $f(x)$ around $x = x_i$ yields where $f^{(k)}(x_i) = f(x)$ if $k = 0$

$$f(x) = \sum_{k=0}^{\infty} \frac{(x - x_i)^k}{k!} f^{(k)}(x_i) = \sum_{k=0}^{\infty} \frac{(x - x_i)^k}{k!} f^{(k)}$$

Applying the previous equation to nearest and next nearest neighbor to grid point i and truncation the the 4th order yields a set of four equations. The set of linear equations provide numerical estimates for the first, second, third and fourth derivatives of f at any given point i. The general form of the first and second derivate based on second nearest neighbors expansion reads:

$$f_i^{(1)} = \frac{f_{i-2} - 8f_{i-1} + 8f_{i+1} - f_{i+2}}{12a} \tag{2.3}$$

$$f_i^{(2)} = \frac{f_{i-2} + 16f_{i-1} - 30f_i + 16f_{i+1} - f_{i+1}}{12a^2} \tag{2.4}$$

The equation 2.3 for the second derivate based on second nearest neighbors expansion solves for the Laplacian operator in the Zhang-Li Model equation 2.2. However, points close to the edges need to be evaluated for better precision.

2.3.1.1 Boundary conditions

Expressions 2.3 are valid when the grid point becomes closet or next-to-closet to the boundary of the magnetic box. However, specific accuracy preserving close to the boundary needs to be worked out. The general present approach is to replace equations that are missing because of the lack of grid points outside the magnetic volume by equations including explicit reference to boundary conditions [7].

Consider first a point second to closet to bound, 2.7,b. Grid point $i + 1$ is missing for this particular geometry. However, defining x_R as the right boundary coordinate along the x axis. The $f^{(1)}(x_R)$ to be know along the boundary to be replace by the derivate of Taylor's expansion [7].

$$f^{(1)}(x) = \sum_{k=0}^{\infty} \frac{(x - x_i)^{k-1}}{(k-1)!} f^{(k)}(x_i) \tag{2.5}$$

Using 2.7-b. $x_R - x_i = 3a/2$ becomes [7].

$$\begin{bmatrix} -2a & \frac{(-2a)^2}{2!} & \frac{-(2a)^3}{3!} & \frac{(-2a)^4}{4!} \\ -a & \frac{(-a)^2}{2!} & \frac{(-a)^3}{3!} & \frac{(-a)^4}{4!} \\ a & \frac{(a)^2}{2!} & \frac{(a)^3}{3!} & \frac{(a)^4}{4!} \\ 2a & \frac{(2a)^2}{2!} & \frac{(2a)^3}{3!} & \frac{(2a)^4}{4!} \end{bmatrix} \begin{bmatrix} f_i^{(1)} \\ f_i^{(2)} \\ f_i^{(3)} \\ f_i^{(4)} \end{bmatrix} = \begin{bmatrix} f_{i-2} - f_i \\ f_{i-1} - f_i \\ f_{i+1} - f_i \\ f^{(1)}(x_R) \end{bmatrix} \tag{2.6}$$

Similarly, for a point closet to boundary, graph b in Figure 2.7, the grid points i + 1 and i + 2 are missing. The two first equation of 2.6 need now to be replaced by a single equation, while the two remaining equations need to be truncated to the third order. Review graph c in 2.7, the minimal set of equations now reads [7].

$$\begin{bmatrix} -2a & \frac{(-2a)^2}{2!} & \frac{-(2a)^3}{3!} \\ -a & \frac{(-a)^2}{2!} & \frac{(-a)^3}{3!} \\ 1 & \frac{(+a)}{2} & \frac{(+a/2)^3}{2!} \end{bmatrix} \begin{bmatrix} f_i^{(1)} \\ f_i^{(2)} \\ f_i^{(3)} \end{bmatrix} = \begin{bmatrix} f_{i-2} - f_i \\ f_{i-1} - f_i \\ f^{(1)}(x_R) \end{bmatrix} \tag{2.7}$$

In both cases, the second derivatives and the matrices of $f^{(1)}(x_R)$ are demonstrated in the reference [7]. CUDA code implementation of the Laplacian boundary condition is discussed in3.

The main advantages of the finite difference approach are: easy to implement, simplicity of meshing, efficient evaluation of the magnetization energy, and the accessibility of higher order methods. The main disadvantage of this approach is the sampling curved boundaries within a rectangular mesh, resulting in some what discrete approximation. In addition, it could produce a significant error in the evaluation.

2.3.2 Fourth order Runge and Kutta method

Modern numerical algorithms for the solution of ordinary differential equations are based on the method of the Taylor series. Algorithm such as the Runge-Kutta method are constructed so they give an expression depending of the parameter (h), in other works, each step is used as an approximate solution, which uses the first terms of the Taylor series. The method is able to accurately solve a wide range of problems, but it is

generally computationally expensive. Solutions require large amount of memory and computational time [10].

There exist several other computational numeric methods to solve such equations. Methods such as the Euler integrator, the Midpoint Method and the Runge-Kutta fourth order (RK4) integrator method can solve differential equations. However, they differ in the numerically approximation and computation time. The RK4 is used for this simulation because its numerically more accurate when compared to the others methods.

The RK4 method differs widely from the Euler method and the Midpoint method. The Euler method is the simplest, the derivative at the starting point of each interval is extrapolated to find the next function value, illustrate in Figure 2.8. The Euler method only has first order accuracy while the RK4 fourth order integrator [10].

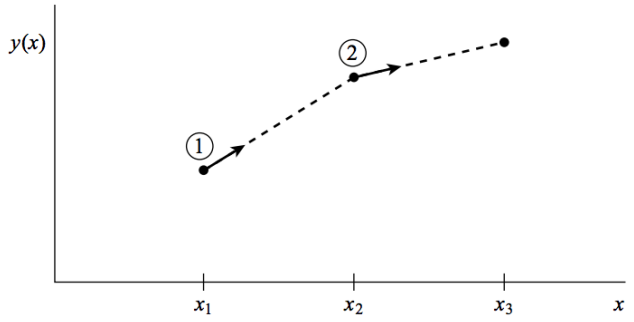

FIGURE 2.8: Euler Method, Is the simplest approximate to solver differential equation or numerically solve equations.

RK4 goes as follows:

$$y_{n+1} = y_n + 1/6K_1 + 1/3K_2 + 1/3K_3 + 1/6K_4 \tag{2.8}$$

where

$$\begin{aligned} K_1 &= h\dot{f}(x_n, y_n) \\ K_2 &= h\dot{f}(x_n + h/2, y_n + k_1/2) \\ K_3 &= h\dot{f}(x_n + h/2, y_n + k_2/2) \\ K_4 &= h\dot{f}(x_n + h, y_n + k_3) \end{aligned} \tag{2.9}$$

As the equations shows, in each step, the derivative is evaluated four times, once at the initial point, twice at trial midpoints, and once at a trial endpoint. From these four values, the final value is calculated, just like the Equation 2.8.

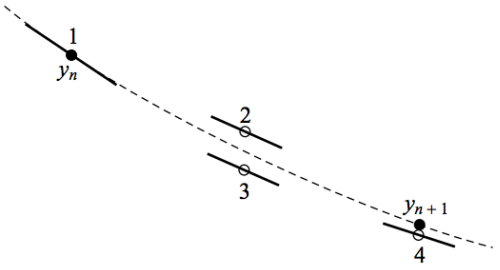

FIGURE 2.9: Fourth order Runge and Kutta method, at each iteration the derivative is evaluated four times.

In conclusion, the simultaneous solution of the diffusive Zhang-Li Model 2.2 has uncovered a qualitatively new feature of the spin-transfer torque effect in the presence of spin diffusion. Namely the dependence of the steady-state DW velocity on DW structure [5]. We quantitatively test the effects of spin diffusion, on real domain walls structures for ATW and VW. This is done by numerically solve the Zhang-Li model along a soft nanostrip of NiFe. The numerical methods used to solve such model as mentioned is the FDTD on a 3D cell grid with whose integration is done using RK4.

Chapter 3

Implementation of Domain Wall Dynamics under Nonlocal STT

This chapter is the study of the heterogeneous computing implementation of the Domain Wall Dynamics under Nonlocal Spin-Transfer Torque. We use the massively parallel capabilities of a single GPU to numerically solve a mathematical equation, known as the Zhang-Li Model. The numerical method used for the solution is method known as Finite Differences in the Time Domain (FDTD) whose integration is done using the 4th order Runge-Kutta. The integration is done on a 3D grid space which outputs the magnetization data of the Vortex Wall, Asymmetric Transverse Wall and a single value, the effective beta.

3.1 Simulation

We numerically solve the mathematical equation Zhang-Li Model 2.2. The numerical solution is done by applying the FDTD whose integration is done using the 4th order Runge-Kutta. The integration is done by using a 3D grid space of 57,600 cells. The sample considered is a 300nm wide in y direction and 5nm thick in z direction. Furthermore, the sample is a soft nanostrip composed of NiFe, a material and size widely used in experiments. When using this size the asymmetric transverse wall 2.4, and the vortex wall have nearly equal energies. The numerical mesh size is 3 x 3 x 5 nm^3, and the calculation box has a length (x direction) of 1,200 or 3,172 nm. The table 3.1 illustrates the mesh information and calculation box for the simulation [5]. In addition, Table 3.2 shows the constant values for the numerical solution for the equation 2.2 such as μ, D_0, τ_{sd} and τ_{sf} and the stop condition value, β_{diff}.

Cell size	Value	Calculation box	Value
NX	480	TX	1200.0
NY	120	TY	300.0
NZ	1	TZ	5.0

TABLE 3.1: Mesh size and calculation box

The implementation is divided into two sections, the host and the device, see Figure 3.1. The host section is in charge of I/O data, allocating the CPU and GPU data and calculating the β_{diff} value. The device is numerically solved in the Zhang-li model.

The simulation begins by reading the magnetization values from a data file. Then, the data set allocates the initial magnetization matrices as well as the static values. Afterwards, the Runge-Kutta algorithms begins. see Algorithm 1. The algorithm loop only breaks when the β_{diff} value converges to $1.0e^{-}9$. The simulation is configured to integrate 50,000 times the Zhang-Li Model before calculating the β_{diff}.

Diffusion parameters	Value	Runge-Kutta 4th	Value
μ	1	time step (dt)	$25.0e^{-6}$
D_0	$1.0e^{3}$ nm mm^2/ns	Max time	1.0
τ_{sd}	$1.0e^{-3}$ ns	β_{diff}	$1.0e^{-9}$
τ_{sf}	$25.0e^{-3}$ ns	Iterations	50,000

TABLE 3.2: Diffusion parameters and Runge-Kutta 4th

3.1.1 Data Allocation and Threads

The first section of the implementation begins with allocation of data into several matrices for both host memory and device memory The input data is the 3D and 2D magnetization information of 57,600 cells. The data being read is stored in three temporary 2D matrices. each one for x, y, and the z coordinate. Next step, the 2D matrices are flattened into three continuous memory blocks, as shown in Figure 3.3. The device code 3.1 flattens the 2d index into a continuous linear 1D index. All the matrices are flatten due that CUDA manage more efficiently continuous blocks of memory and is faster to access each element.

```
int i = blockIdx.x * blockDim.x + threadIdx.x + 2;
int j = blockIdx.y * blockDim.y + threadIdx.y;
 // map the two 2D indices to a single linear, 1D index
int index = j * grid_width + i;
```

LISTING 3.1: Kernel flatten 2d - 1d

NVIDIA's GPUs handle more efficiently square matrices and values. Therefore, we need to map the input data set to square matrices. However, for a given architecture,

there is an optimal matrix dimension and number of threads per matrix which balances occupancy and instruction level parallelism. The matrix size used for the implementation is 512 x 128 which are the nearest square values of 480 x 120, review Listing 3.2). The number of threads per square matrix depends on the number of threads available for each hardware configuration. The current implementation used 16 x 16 threads per block.

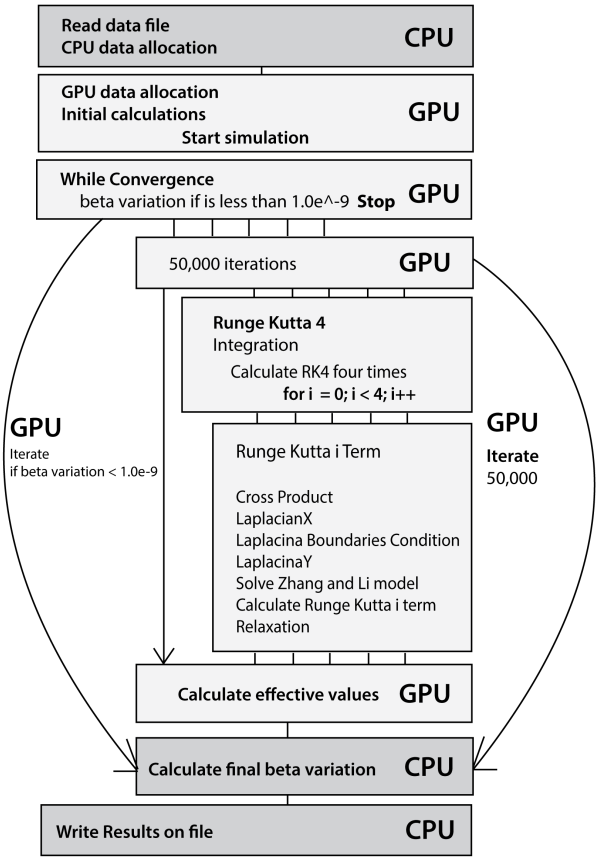

FIGURE 3.1: Control flow of the simulation for the CPU and GPU.

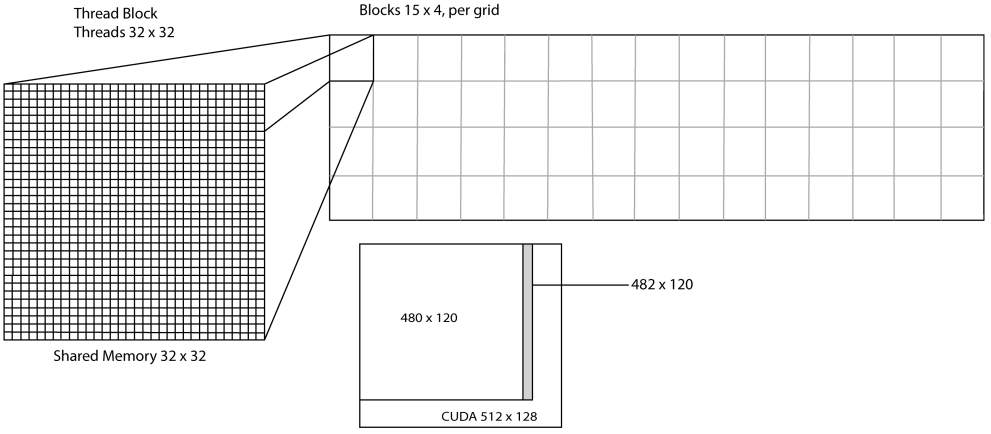

FIGURE 3.2: Memory allocation in terms of the blocks per threads and grids.

Matrix size X	480	Device allocation X	512
Matrix size Y	120	Device allocation Y	128

TABLE 3.3: Matrix allocation size

The magnetization data is stored in three matrices x, y, and z, each one of them with a capacity of 56,700 values, in other words, 480 times 120. Base on that information we want to calculate the optimal number of grids that will ensure a complete use of the hardware resources. The number of blocks per grid corresponds to dividing the dimensions of the array by the number of threads, see the last two operations in the Listing 3.2.

```
NXCUDA = (int)powf(2,ceilf(logf(NX)/logf(2)));
NYCUDA = (int)powf(2,ceilf(logf(NY)/logf(2)));

//Setup optimum number of blocks
XBLOCKS_PERGRID = (int)ceil((float)NX/(float)XTHREADS_PERBLOCK);
YBLOCKS_PERGRID = (int)ceil((float)NY/(float)YTHREADS_PERBLOCK);
```

LISTING 3.2: Device capacity calculation and number of block per grid

Depending on the hardware properties, each GPU is able to allocate different number of threads per block and as well as different shared memory sizes. The shared memory in this implementation relies on the number of threads per block. In addition, the number of blocks depends on the input matrix and the number of threads, Examine Figure 3.2. More information about the optimal number of threads per block for the current simulation can be found Chapter 5.

	Fermi	Kepler
Threads per block X	16	32
Threads per block Y	16	32
Number of blocks X	30	15
Number of blocks Y	8	4
Shared memory	16 * 16	32 * 32

TABLE 3.4: Threads, blocks size

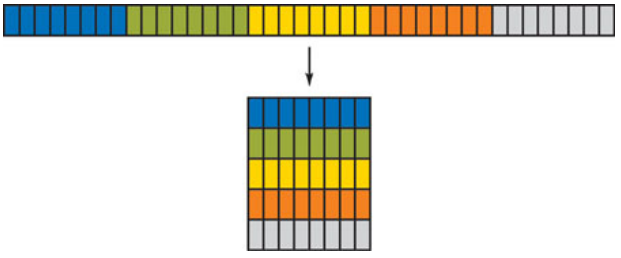

FIGURE 3.3: Conversion of a 2D array to a single continuous block of memory

3.1.2 Initial Calculations

Before the evaluation of the RK4 terms, we need to evaluate static matrices that will not change over the course of the simulation 1. In The pre calculation stage only the term $\delta\vec{m}$ of the Zhang-Li model is evaluated2.2. The $\delta\vec{m}$ term is the non-equilibrium spin density domain wall at rest [5], see Listing 3.3.

```
//Compute x, y and z component of source term
gsource << <blocks, threads >> >(...);
gsource << <blocks, threads >> >(...);
gsource << <blocks, threads >> >(...);

//Project source term on magnetization components by computing
//a cross product twice
gm_x_source << <blocks, threads >> >(...);
gm_x_source << <blocks, threads >> >(...);
```

Listing 3.3: Initial calculations

The kernel `gsource` of the Listing 3.3 is only launches at the beginning of the application. From that operation we obtain the matrix `sm`, which is used to compute the Zhang-Li model in the RK4 integration section.

As previously mentioned, the CUDA matrices are square sizes, 512 x 128. However, the input data set is 480 x 120. Therefore, we need to limit the number of threads inside every kernel. The issue is solve by including a simple `if` inside of every kernel call, see Listing3.4. Yet, we also need to include a minor adjustment of two indices in the x direction. The adjustment is due to the mesh boundary condition in the FDTD method, see Figure 3.4.

```
__global__ void rungeKuttaTerms(...)
{
    if (i > 1 && i < NX + 2 && j >= 0 && j < NY){
        //calculations
    }
}
```

Listing 3.4: Limits the threads executing inside a kernel

3.1.3 Numerical Methods

The following algorithm illustrates the necessary operations for the RK4 integration process (algorithm 1). As previously mentioned, the algorithm only resides on the device. Furthermore, the host only communicates with the device in two sections: sending input data or matrices that were allocated in the host to the device and the second part, when

the algorithm finishes calculating the b_{eff} term, and the value is sent back to the host. Finally, the algorithm breaks when the b_{eff} term reaches a numeric error of $1.0e^{-9}$.

Data: deltam, sfrelax, sdex, sm, laplacian
Result: deltam
data initialization;
while $\beta_{diff} < 1.0e^{-9}$ **do**
    Runge and Kutta 4th;
    for $i = 1; i <= 4; i+ = 1$ **do**
        sdex ← crossProduct(deltam, mag); calculate cross product
        FDTD with boundary condition
        laplacian ← laplacianXYBoundary(deltam);
        evaluate Zhang-Li model
        zhangLi ← solveZhang(sfrelax, sdex, Laplacian, sm);
        RK4 evaluation
        rkterm(i) ← rktime(i, solveZhangLi, dt);
        if $i == 4$ **then**
            deltam ← rk4(zhangLi, tmp, dt, rkterm(1),rkterm(2), rkterm(3),rkterm(4))
        else
            deltam ← rk4(zhangLi, tmp, dt)
        end
        evaluate RK4 term
        deltam ← rk4(zhangLi, tmp, dt)
        sfrelax ← relaxation(deltam, tau)
        if $i == 4$ **then**
            tmp ← copy(rkterm(4));
        end
    end
    β_{diff} = calculate(temp, beta);
end

Algorithm 1: Runge and Kutta 4th integration implementation

3.1.3.1 Finite differences in the time domain

The finite differences method requires the domain of interest to be broken down into small regions. Such subdivision of space is known as mesh, grid or cell division. The grid is divided into a 512 x 128 space, which is mapped to the GPU hardware 3.1. Furthermore, the number of threads per is block is synchronized to the cell division, but as well to the number of SM available in the device, see Tables 3.3 and 3.4. To numerically solve the Zhang-Li Model, we need to determinate and evaluate the first

and second derivate. The proposed solution is based on the second nearest neighbors 2.3. Moreover, the nearest neighbors method evaluates the derivative of the input data, see Figure 3.4.

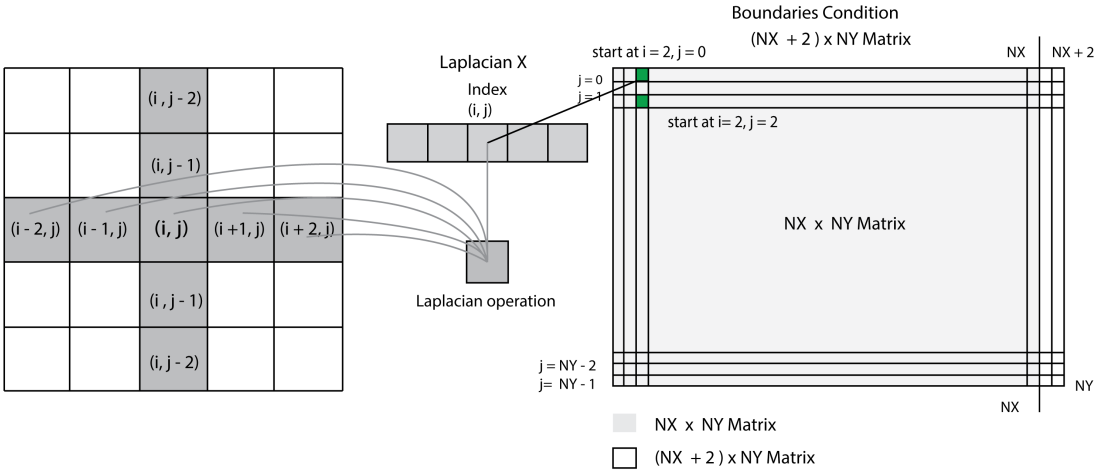

FIGURE 3.4: Laplacian XY block calculation and boundaries condition

The calculation of the nearest neighbors expansion is done by evaluation a neighborhood of $[-2, 2]$ values in the x direction. The index begins at $i = 2$ and finishes at $NX + 2$. The same occurs for the Laplacian in the Y direction, however, begins at $j = 2$ and finish at $NY - 2$. The implementation 3.5 is based on the nearest neighbors Equation 2.3.

```
int i = blockIdx.x * blockDim.x + threadIdx.x + 2;
int j = blockIdx.y * blockDim.y + threadIdx.y;
// map the two 2D indices to a single linear, 1D index
int idx = j * grid_width + i;

lapyX[idx] = - deltamX[idx + 2] / 12.0 + 4.0 * deltamX[idx + 1] / 3.0
             - 5.0 * deltamX[idx] / 2.0
             - deltamX[idx - 2] / 12.0 + 4.0 * deltamX[idx - 1] / 3.0;

lapyY[idx] = - deltamY[idx + 2] / 12.0 + 4.0 * deltamY[idx + 1] / 3.0
             - 5.0 * deltamY[idx] / 2.0
             - deltamY[idx - 2] / 12.0 + 4.0 * deltamY[idx - 1] / 3.0;

lapyZ[idx] = - deltamZ[idx + 2] / 12.0 + 4.0 * deltamZ[idx + 1] / 3.0
             - 5.0 * deltamZ[idx] / 2.0
             - deltamZ[idx - 2] / 12.0 + 4.0 * deltamZ[idx - 1] / 3.0;
```

LISTING 3.5: Laplacian evaluation for x, y and z coordinate

However, the calculations of the nearest neighbors is only valid for values of points inside a grid of [-2, 2] × [-2, 2] × [-2, 2]. We need to calculate the boundary condition whose points are in the edge of the grid. In other words, this is close to the magnetization boundaries.

The Matrices 2.6 and 2.7 are used for evaluating the values near the matrix borders. The matrices have boundaries at $j = 0$ and $j = NY - 1$ for all values in the i cell. The matrix 2.7 is evaluated. For the condition $j = 1$ and $j = NY - 2$ for all i values, we used the matrix 2.6. Listing 3.6 demonstrates how boundary condition was implemented. To obtain the correct values, first the Laplacian in the y direction is calculated; afterwards, the boundary condition in the x direction is evaluated.

```
__global__ void glaplaciany(...){} //Compute laplacian in Y direction

__global__ void glaplacianBoundaries(...){
    if (i > 1 && i < NX + 2 && j == 0){
        // Update Laplacian Boundaries Equation 3.10
    }
    else if (i > 1 && i < NX + 2 && j == 1){
        // Update Laplacian Boundaries Equation 3.9
    }
    else if (i > 1 && i < NX + 2 && j == NY - 2){
        // Update Laplacian Boundaries Equation 3.9
    }
    else if (i > 1 && i < NX + 2 && j == NY - 1){
        // Update Laplacian Boundaries Equation 3.10
    }
}
__global__ void glaplacianx(...){} //Compute laplacian in X direction
```

LISTING 3.6: Evaluation of Laplacian X, Y with boundary condition

The three CUDA kernels `glaplaciany`, `glaplacianx` and `glaplaciany` are evaluated for each x, y and z coordinate within a 2d CUDA grid space. The three coordinate sum up to 172,800 cell points to be calculated.

3.1.3.2 Zhang and Li Model

The Zhang-Li Model is solved by using the code in the Listing 3.8. Furthermore, the Zhang-Li equation is used as input function for the RK4 integration process 2.8. For the first term of the Zhang-Li Model, the `sfrelax` matrix is calculated in the relaxation process at the end of the RK4 integrator. The next term `sdex` is computed in the Cross Product process, which is done before the each RK4 calculation. The matrix `sm` is only calculated once at the initial calculation process, see Listings 3.3. The last term `lapl` matrix is evaluated in the Laplacian process, see Algorithm RK4 1.

```
        sfrelax[idx] = -deltam[idx] / tau_sf;
        sdex[idx] = -(deltam[idx] * m[index] - deltam[idx] * m[idx]) / tau_sd;

        //Evaluate Zhang - Li Method
        solveZhangLi[idx] = sfrelax[idx] + sdex[idx] + lapl[idx] - sm[idx];
```

LISTING 3.7: Zhang-Li evaluation

The output matrix `solveZhangLi` is used as integration input for the 4th order Runge and Kutta method. The Zhang-Li evaluation is done four times, once per Runge and Kutte term.

3.1.3.3 Runge and Kutta

The 4th order Runge and Kutta method is implemented by using four CUDA kernels, each kernel evaluates the derivative with a different time increment, Equation 2.9. The first three RK4 terms are slightly same; they differ from using the previous evaluated term 3.8. The final 4th term is the weighted average of the previous increments 3.9.

```
rk1[idx] = dt * deltam[idx];  //rk term 1
deltam[idx] = temp[idx] + 0.5 * rk1[idx];

rk2[idx] = dt * deltam[idx]; //term 2
deltam[idx] = temp[idx] + 0.5 * rk2[idx];

rk3[idx] = dt * deltam[idx]; //term 3
deltam[idx] = temp[idx] + 0.5 * rk3[idx];
```

Listing 3.8: Runge and Kutta 1st, 2nd and 3rd terms

```
rk4[idx] = dt * deltam[idx]; //final term rk4
double diff = rk1[idx] + 2.0 * (rk2[idx] + rk3[idx]) + rk4[idx];
deltam[idx] = temp[idx] + diff / 6.0;
```

Listing 3.9: 4th term of the Runge and Kutta integration

The Runge and Kutta integration is grouped up in two `for` cycles, described in Listing 3.10. The inner `for` cycle evaluates the RK4 for the x, y, z coordinate. The outer `for` cycle evaluates four times the Runge and Kutta integration.

```
for(int term = 0; term < 4; term++){
    for(int coord = 0; coord < 3; coord++){
        gsd_exchange<<<blocks, threads>>>(term, coord);
        glaplacianx<<<blocks, threads>>>(term, coord);
        glaplacianyboundaries<<<blocks, threads>>>(term, coord);
        glaplaciany<<<blocks, threads>>>(term, coord);
        gsolution<<<blocks, threads >>>(term, coord);
        gterm_RK4<<<blocks, threads >>>(term, coord);
    }
}
```

Listing 3.10: Summarize of Runge and Kutta 4th Integration

The simulation is set to 50,000 iterations of the RK4 integrator, In each iteration the algorithm evaluates if the current value will diverge or converge to a valid result, see Figure 3.1. It is highly possible for the simulation to converge to a result and finish the simulation. Because of this, we need to validate the output data.

3.1.4 Calculate Beta Diffusion

Kernels 3.11 are only launched at the end the RK4 integration, which outputs the β_{diff}. After each RK4 step, the beta diffusion adds ups all accumulated values from previous steps. Moreover, the β_{diff} values is output of the numerically solution of the effects of spin diffusion, on domain wall structures. Therefore, by numerically solving the Zhang-Li Model including the spin diffusion term.

```
gm_x_sm << <blocks, threads >> >(...);
gu_eff << <blocks, threads >> >(...);
gu_eff_beta_eff << <blocks, threads >> >(...);
gbeta_diff << <blocks, threads >> >(...);
```

Listing 3.11: Calculate beta diffusion

Although, we do not know when the RK4 algorithm will end, the simulation will stop when β_{diff} reaches the minimum of 10^{-9}. Afterwards then the magnetization data will be written. Moreover, the application is able to write the magnetization results for the VW or for the ATW. Furthermore, the magnetization data are written into two separated data files, which contain the spin effective data and the spin accumulation data.

3.2 Validation

Because the CUDA framework is a highly parallel system, it is fairly easy to obtain erroneous data from the calculations. Even setting up the threads per block incorrectly is possible to get a data set that is wrong, or results that converge. When making changes to the code, it is necessary to validate the new code.

The validation is done by comparing the output of the simulation with a valid data set. The output of the validation application tells us the error factor of the current data with the valid set. So for each data set there is a threshold value, that can tell if that is close enough to the results, an example of the validation performed.

According to the results, the new code should not produce errors in the spin.dat greater than 7.0^{-17}, in other words valid code do not lead to differences greater than the precision expected from computations with double precision 1.0^{-16}. In the case of eff data the errors are in the order of 1.0^{-11} and no greater than 6^{-11}. For the variation the precision is expected to be within the double precision range of 1^{-16}.

For the initial results we used the GeForce GT 650M, with 384 CUDA core at 745 MHz and 2GB GDDR5 of memory (Reference 3.5). The values obtained from the first GPU

were used as standard values for future validation testing. The speedup and timing are used for comparing optimal performances is found in Chapter 5.

GeForce GT 650M 384 CUDA			
Data set	Simulation time	Speedup	Diffuse beta
upVW magnetization	377590.3 ms	1.00x	4.848728452719814e-02
ATWpm magnetization	377409.2 ms	1.00x	4.054674178687585e-02

TABLE 3.5: Calculation results

In conclusion, the numerical solution of the Zhang-Li Model using the FDTD method is mostly done on the GPU, while the host only executes I/O operations. In addition, the simulation numerically solves the differential equations of the Zhang-Li Model using the 4th order Runger-Kutta integration. Moreover, each RK4 term evaluates the Laplacian 2.3 with boundary conditions 2.7 2.6. Finally, the results of the simulation are validated within the double precision range.

Chapter 4

Heterogeneous Performance Analysis and Practices

While working with GPUs, new challenges emerge, such as how can we make the best use of the millions of threads using the GPU hardware. In the conventional CPU model, we have what is called linear or flat memory model, which appears to the programmer as a single contiguous address space. Furthermore, the CPU can directly address all the available memory, in other words, there is almost no efficiency penalty in creating global data, local data, or even access data that is located on an opposite memory location; All of this can be accessed as a contiguous block [6]. Meanwhile, on the GPU there are exceptions; there exists different memory hierarchies which dramatically change the performance. By allocating the optimal memory types, speedup and increase throughput can be accomplished. To ensure optimization, some analysis should be made, such as comparing latency, memory hierarchies and data bandwidth between CUDA kernels. Debugging of parallel code is possible using the NVIDIA's Visual Profiler. This chapter demonstrates techniques, practices and methods to debug and analyze parallel process on the GPUs.

4.1 Practices

There are three rules for developing high performance GPGPU (General-purpose on the GPU) program, which are based on NVIDIAs GPU standards [8].

1. Get the data on the GPU device and keep it there
2. Process all the data en the GPU, give it enough work to do.

3. Focus on data reuse within the GPU context, to avoid memory bandwidth limitations

The GPUs are plugged into the PCI Express bus of the host computer. The PCIe bus has extremely slow bandwidth compared with the GPU. This is why it is important to store as much data as possible in the GPU and keep it busy, but as well minimize the data transfer from the host and back to the device. The process is illustrated in the Figure 4.1. CUDA enables the GPU to carry out petaFLOP performance in a single device [6]. Moreover, the GPUs are fast enough to compute a large amount of data in parallel. To accomplish such high performance; each CUDA kernel needs to use all the available resources of the GPU, avoid wasting compute cycles and minimize thread branching.

	Bandwidth (GB/s)	**Speedup over PCIe Bus**
PCIe x16 v2.0 bus (one-way)	8	1
GPU global memory	160 to 200	20x to 28x

FIGURE 4.1: PCIe bus and GPU bandwidth comparison [6]

The practices should be taken into consideration to identify the parts of code where it would be beneficial for improving GPU acceleration [22].

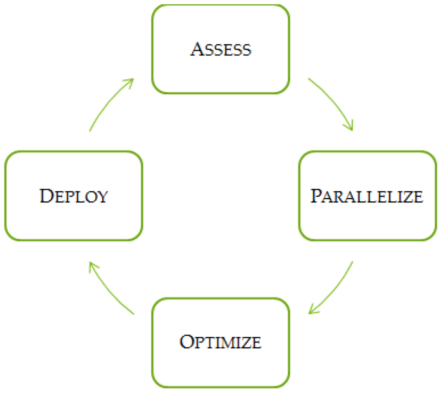

FIGURE 4.2: GPU application practices [22].

Asses

The first step is to locate the part of the code where the majority of the execution time occurs. The programmer can evaluate memory bottlenecks for GPU parallelization.

Parallelize

To increase parallelization from the original code, this could be done either by adding GPU-optimized libraries such as cuBLAS, cuFFT, or including more amount of parallelism exposure though the use of CUDA code.

Optimize

The developer can optimize the implementation performance through a number of considerations, such as overlapping kernel executing, kernel profiling, memory handling and fine-tuning floating-point operations.

Deploy

Compare the outcome with the original expectation. Determinate the potential speedup by accelerating a given section. First a partial parallelization should be implemented before carrying out a complete change.

4.2 Performance Metrics

There are many possible approaches for profiling CUDA code, but in all cases the objective is the same: identify the kernel or kernels in which the application is spending most of its execution time and increase the throughput by a given kernel. Throughput is how many operations are completed per cycle.

4.2.1 Timing

Timing a launched kernel should be done on either the GPU or the CPU. However, the GPU and CPU are not synchronized, and events are able to block multiple threads at any given moment. Although it is necessary to synchronize the CPU thread with the GPU kernels launches. CUDA provides the required functions to synchronize the CPU with the GPU calling immediately before starting the timer [22]. CUDA is able to handle timers within the GPU, which records times in a floating-point value in milliseconds. This is done with `cudaEventRecord()`, just by including `start` and `stop` in the function inputs. Moreover, the timing is measure on the GPU clock, therefore, the timing is independent from the OS and CUDA [6]. Lastly, the timing results of the various stages of the simulation is found in chapter 5.

4.2.2 Bandwidth

Bandwidth refers to the rate at which data can be transferred between host and device and vice-versa. The bandwidth is one of the most important factors for testing performance on the GPUs. Choosing the right type of memory could dramatically increase performance and bandwidth. There are two main bandwidth types to indicate performance: theoretical bandwidth and effective bandwidth. The theoretical bandwidth is

based on the hardware specifications available by NVIDIA. The bandwidth is calculated using the following:

$$\texttt{theoretical bandwidth} = \frac{(clockrate * (512/8.0) * 2.0)}{10^9}$$

For example the NVIDIA GeForce GTX 280 uses DDR RAM with a memory clock rate of 1,105 Mhz and a 512-bit-wide memory interface

$$\frac{(1107 * 10^6 * (512/8.0) * 2.0)}{10^9} = 141.6 \text{ Gb/sec}$$

The GTX 280 has a theoretical bandwidth of $141.6 Gb/sec$. The effective bandwidth is calculated by timing specific program activities and by knowing how data is accessed by the application [22].

$$\texttt{effective-bandwidth} = \frac{((Br - Bw)/109.0)}{time}$$

Where Br is the number of bytes read per kernel, Bw is the number of bytes written per kernel and t is the elapsed time given in seconds [29].

In practice the difference between theoretical bandwidth and effective bandwidth indicates how much bandwidth is wasted on accessing memory and calculations. If the effective bandwidth is low compared to the theoretical bandwidth, this is an indication that there is not enough work being done in the GPUs. In addition, there a several solutions: analyze the code to accomplish more parallelize instructions, execute more computational instructions on the GPUs, bind memory blocks, in other words pin the initial memory block on the CPU with the final memory block on the CPU.

In the chapter 5 we analyze the bandwidth and timing for each NVIDIA GPU used to optimize the application. However, bandwidth information is only available when transferring data from the CPU to the GPU or from the GPU to the CPU.

4.3 Memory Handling with CUDA

In this section four types of memory handling will be explained: global memory (device memory), shared memory, texture memory and constant memory. Figure 4.3 illustrates physically the position of the different memory types inside the device chip.

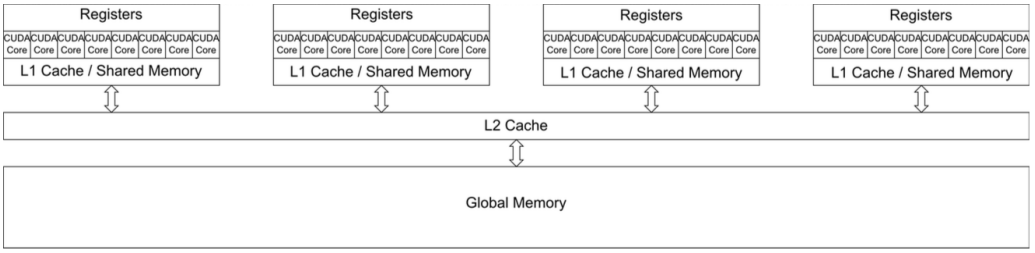

FIGURE 4.3: The schematic cache hierarchy of a CUDA GPU with 4 Streaming Multiprocessors and 8 CUDA Cores each [6].

Global memory is very large in comparison to the shared memory, which is on the L1 cache. However, the global memory is far away from the registers and from the CUDA core locations. Moreover, the memory access is very slow in comparison to the shared memory [6].

The Figure 4.4 illustrates the five different memory types that are available in CUDA. But more interesting are the bandwidth penalty and the latency in computer cycles for each one of them. Moreover, different memory types can be used in different applications to maximize performance, hence memory usage. The Shared Memory is very limited so it cannot be handler across all situations. Furthermore, when implementing a wrong memory type on the device there are possibilities for latency penalties and bandwidth drop, instead of having a performance gain.

Storage Type	Registers	Shared Memory	Texture Memory	Constant Memory	Global Memory
Bandwidth	~8 TB/s	~1.5 TB/s	~200 MB/s	~200 MB/s	~200 MB/s
Latency	1 cycle	1 to 32 cycles	~400 to 600	~400 to 600	~400 to 600

FIGURE 4.4: Different memory type and penalties usage [6]

4.3.1 Global Memory

Understanding how efficiently to use global memory is essential part of CUDA memory management. Focusing on data reuse within the SM and caches is needed to avoid memory bandwidth limitations. Global memory on the GPU is designed to quickly stream memory blocks of data into the SM. However, global memory tends to be slow compared with there types of memory [8].

- Get the data on to the Device: keep it there.
- Give the GPU enough workload: this uses all the resources available from the GPU.

- Focus on data reuse within the GPGPU to avoid memory bandwidth limitations.

In other words the global memory resides in the device, and it should be anything from 1 byte to 8GB, depending on the GPU RAM available. Furthermore, the memory is visible to all the threads of the grid. Every thread at a given location is possible to read and to write as global memory. The memory is always allocated with the keyword *cadaMalloc*. In addition, the global memory is only used by passing it to the kernel called the keyword _global _. Global memory is widely used for the current implementation [8].

4.3.2 Shared Memory

The CUDA C compiler treats variables differently than a typical c language variable. The compiler creates a copy of the variable for each block that is launched on the GPU. Now every thread in that block has access to the memory, hence, shared memory. This memory resides physically on the GPU, because the memory is very close the cache. The latency is typical very low [30]. One thing comes to mind, if the threads can communicate with others threads, there should be a way to synchronize all the threads. A simple case should be if thread A writes a value into the shared memory, and Thread B wants to be accessed we need to synchronize. When thread A is finished writing then thread B can access it. This is typical case when shared memory with synchronize thread is needed [6].

Shared memory is much faster to access than global memory, and essentially it is like a local cache for each thread of a block. While the shared memory is limited to 48K a block, the global memory is the amount of DRAM on the device. The duration of the shared memory on the device is the lifetime of the thread block. Using _shared _in-front of the data type will innovate shared memory.

Shared memory is widely used for applications where the kernels access a great amount of global memory. In addition, using shared memory eliminates the use of clock cycles per kernel which increases performance on a single kernel call. For the current application we used extensively shared memory, eliminating the use of global memory. More information about the process ca be found in Chapter 5.

4.3.3 Constant Memory

Constant memory is an excellent way to store and broadcast read-only data to all the threads on the GPU. One thing to keep in mind is that the constant memory is limited to

64KB [8]. A simple analogy is the `#define` or `const` attribute in the C++ programming language. The variable performs like a value that cannot be modified. On CUDA this is exactly the same, and the value can only be read and not written. Furthermore, the value will not change over the course of a kernel execution and only the host can write the constant memory [30]. Constant memory is able to reduce calls for static global memory. In chapter 5 we discuss improvements speedup improvements by using this type of memory.

4.3.4 Texture Memory

Similar to constant memory, texture memory is another variety of read-only memory that can improve performance and reduce memory traffic when reads have certain access patterns. Traditionally texture memory is used for computer graphics applications, but it can also be used for HPC. The main idea of this read-only memory is that threads are likely to read from address 'near' the address of the nearby threads [30].

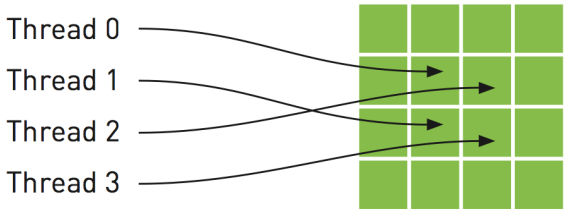

FIGURE 4.5: Mapping of threads into a two dimensional array of texture memory [16]

The texture memory in a form works like the GPU graphics texture: when you want to use the texture bind with some sort of data is necessary and when you finish using it, unbind the texture from the data. The usage can be summarized in the following list:

- Allocate global memory in the Host.
- Create texture reference and bind it to memory object.
- On the device obtain the reference from the texture.
- Use Texture memory operations on the device
- When the work is done on the texture, unbind the texture reference on the host.

The texture memory is not used on the current implementation, for obvious reasons: it is a read only memory. For the numerically methods we need to constantly read and write blocks of memory.

4.3.5 Thread Synchronization

Thread Synchronization refers to orderly execute thread operations. For efficiency, a pipeline can be created by queuing a number of kernels to keep the GPGPU busy for as long as possible. Furthermore, some form of synchronization is required so that the host is able to determine when the kernel or pipeline has been completed [8]. Commonly used synchronization mechanisms are:

- Explicitly calling `cudaThreadSynchronize()`, which acts as a barrier causing the host to stop and wait for all queued kernels to be completed.
- Performing a blocking data transfer with `cudaMemcpy()` as `cudaThreadSynchronize()` is called inside `cudaMemcpy()`.

The basic unit of work on the GPU is a thread. It is important to understand from a software point of view that each thread is separate from every other thread. Every thread acts as if it has its own processor with separate registers and identity. It will wait for all threads to finish their job [8].

Thread synchronization is also possible inside kernel calls. The idea is the same: the kernel will wait until all the threads have completed their task. When more threads are synchronized, they schedule more work, hence, better performance and more workload. Thread synchronization is generally used when loading data into shared memory. The implementation of such process is in Chapter 5, section optimizations.

4.4 Concurrent Kernels

Kernels are executed in a sequential form with parallel instructions. In addition, with CUDA's streams is possible to launch several kernels in parallel, in other words, overlap kernel in the same launch sequence. Figure 5.3 illustrates this.

A stream in CUDA is a sequence of operations that execute on the device in the order in which they are issued by the host code. Every kernel is launched on the default stream zero. Hence, to overlap kernel execution, non-default streams should be used for every kernel launch. To accomplish concurrent kernels, streams should be pinned to a non-default stream (non zero)[16].

Using two or more CUDA streams, we can allow the GPU to simultaneously execute a kernel while performing a copy between the host and the GPU. However, we need to be careful about two issues. First, the host memory involved needs to be allocated. Since

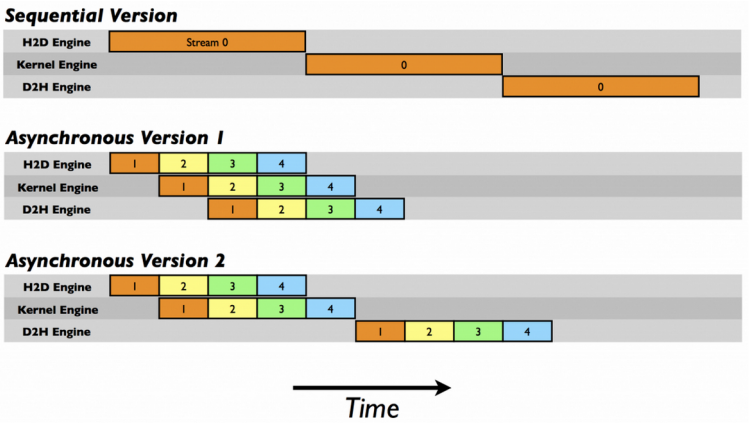

FIGURE 4.6: Overlapping kernel execution using CUDA streams

we will queue our memory copies, we need to synchronize those copies. Second, we need to be aware that the order in which we add operations to our streams will affect our capacity to achieve overlapping of copies and kernel execution. The general guideline involves a breadth-first, or round robin, to assign work and queue work to the kernels [30].

The order of kernel executing affects the operations of the streams, and moreover, the application performance. In the current application, we carefully examined the order of kernel executing. More about how to implement concurrent kernels for the simulation is found in Chapter 5.

4.5 Kernel Analysis

Kernels are the essential part of CUDA programming;, threads are launched automatically throughout each thread per blocks of the device. Furthermore, millions of threads execute the same code in parallel. However, the parallel code can be bound by three factors memory, compute and latency [6].

Memory Bandwidth Bound
: Refers to the application's limitations for memory access. Most GPUs cards have 1GB-6GB of memory, which is used to process the data on the GPU. Different solutions are: reuse data, use different GPU memory types, or implement a multi-GPU approach to increase the memory.

Compute Bound
: Refers to the computation time execution: in other words, calculations done on the device, under the assumption there is enough memory for the calculations.

Therefore, it is the number of operations per cycle in the kernel. Theoretical bandwidth vs. effective Bandwidth can measure performance for a compute-bound Kernel. Therefore, it is possible to increase the FLOPS per device.

Latency Bound

Is one whose predominate stall and it is due to memory fetches. This is actually the saturating the global memory, or any type, but it still has to wait to get the data into the kernel. Physically, it is data being sent from one part of the device to the other. Moreover, depends on the time required to perform an operation. It is counted in cycles of operations. A way to reduce the latency is to increase the number of parallel instructions (more calls per thread), in other words more work per thread and fewer threads. However, this is not always possible.

Depending on the problem, the application can be bound by the previous three factors. In the next chapter we will explain how and why the current implementation is bounded by memory, compute and latency.

4.6 Hardware constraints

The hardware capabilities, limits how many threads per block a kernel launch is able to have, but as well as the version of CUDA that the hardware is able to execute. The compute capabilities of a device represents by a version number, called -"SM version" or CC for short. The version number identifies the features supported by the GPU hardware and is used by the applications at runtime to determine which hardware features and/or instructions are available on the present GPU [23].

Another inefficiency, that can cause low performance in the CUDA application, is the number transfers memory calls between the CPU and the GPU. The GPU communicates with the CPU via a *PCIe* bus as mentioned before. In addition, all of the massive FLOPS per second that are computed on the GPU are not able to be sent back to the CPU, because, of the physical connection between the GPU and CPU. Ideally, the GPU should have a large workload as possible before returning data back to the CPU. However, this is not always possible, a technique to increase more throughput is to pin/bind the memory in the host. Another method is to send as much workload as possible in a single kernel call and by using the maximum the GPU hardware capabilities [22]. For the current implementation, CPU and GPU are relatively low, only a few times communication is done by the device and the host.

4.6.1 Thread Division

There are several hardware limitations in how many threads per block a kernel can handle. Launching a kernel with the hardware constraints of the device will only ensure us that the kernel will actually be executed on the device, Nonetheless, not 100% optimal and the results can be incorrect. Furthermore, it is necessary to launch kernels with the right amount of threads per block based on the hardware settings. The block size will determine how fast the kernel will execute. However, not the biggest block will run faster, relays on the problem and the data set. By benchmarking the application, it is possible to find the optimal configuration that best fits the problem. One thing to keep in mind is that thread blocks should be a multiple number of SMs. With this idea, it is possible to obtain optimal thread block configuration. Review Chapter 5 for the optimal thread configuration for the current simulation. The optimal number of threads per block did not occur on the maximal available threads per block of the devices.

4.7 Visual Profiler

It is a challenging task to keep track of each individual thread even more, a million threads. It becomes difficult to debug highly parallel applications. The NVIDIA's Visual Profiler is a profiling tool that can be used to measure performance and find potential opportunities for optimization in order to achieve maximum performance on the GPUs. The Profiler provides metrics in the form of plots and graphs, which illustrate instances of the GPU, such as kernel calls, data transfer, kernel executing time, memory dumps and others. See Figure 4.7.

NVIDIA's profiling tools comes in various ways; a standalone profiler through the visual profiler compiler nvvp, integrated in a GUI NSight Eclipse Edition as NSight command (Visual Profiler), and as a command-line profiler though nvprof command. Each one has its disadvantages and advantages. The command-line profiler is useful for remotely access, where a GUI is not available, while the NSight can show graphs, plots and timeline of the application. The Profiler support CUDA applications as well as openCL applications. However, there are exceptions.

The Visual Profiler, by default, will execute the entire application, nonetheless typically only some parts of application only need performance optimization. This enables to determine kernels, code where critical performances is needed. The common situation where profiling a region of the application is helpful [23].

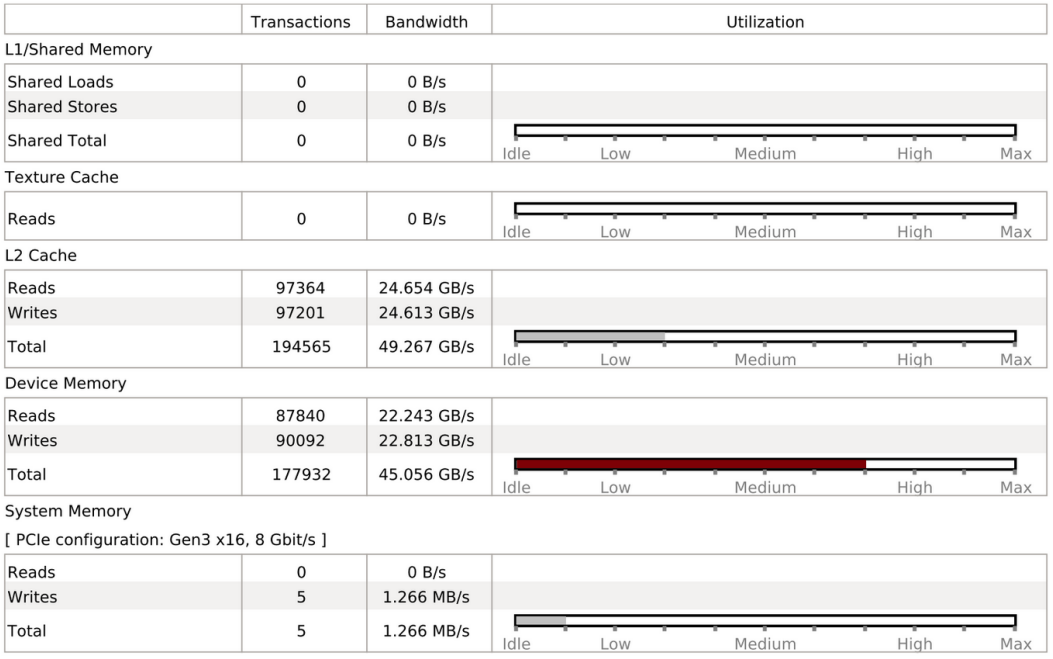

FIGURE 4.7: Profiler provides optimization metrics necessary to improve the application.

- Analyze data initialization and movement in the CPU and GPU, as well as evaluating CUDA calls.
- The application operates in phases, where an algorithm operates throughout each region. The application can be optimized independently from other phases of the code.
- The application contains algorithms that operate though a large number of iterations. In this case it is possible to collect data from a portion of the iterations.

The Visual Profiler provides a step-by-step optimization guidance, where it is possible to evaluate the GPU usage, to examine individual kernels and to analyze timeline of the application which the profiler shows memory movements and usage, CUDA calls, number of threads and performance. Figure 4.8 shows that each kernel has its own percentage of execution time of the overall application [22].

4.7.1 Profiler Kernel Report

The profiler will execute several times the application for it to collect data from each kernel. This enables it to precisely optimize phases of the application[30]. The profiling tools can verify how long the application spends executing each kernel as well

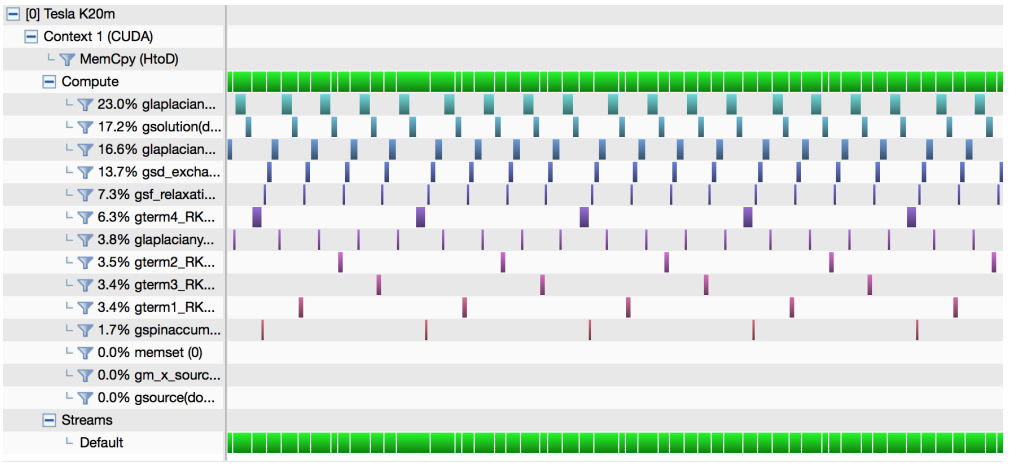

FIGURE 4.8: Visual Profiler kernel execution, and timeline execution

the number of used blocks and threads. Through this it is possible to obtain various memory throughput measures, like global load throughput and global store throughput. They indicate the global memory throughput requested by the kernel and therefore corresponding to the effective bandwidth mentioned in the last section.

As we know, the profiler executes the application several times to collect data about each kernel. The information obtained by each kernel can be summed-up in-to a report that can be exported in a pdf file, which has the following information.

1. **Compute, Bandwidth, or Latency Bound**
   The performance determines if the kernel is bounded by computation, memory bandwidth, or instructions/memory latency. It shows how it is limiting the performance respectively.

2. **Instructions and Memory Latency**
   Both limit the performance of a kernel when the GPU does not have enough work to keep busy. The performance of latency-limited kernels can often be improved by increasing occupancy. Occupancy is a measure of how many warps the kernel has active on the GPU, relative to the maximum number of warps supported by the GPU.

3. **Compute Resources**
   GPU compute resources limit the performance of a kernel when those resources are insufficient or poorly utilized. Compute resources are used most efficiently when instructions do not overuse a function unit.

4. **Floating-Point Operation Counts**
   Floating-point operations executed by the kernel can be either single precision or double precision.

5. **Memory Bandwidth**
   Memory bandwidth limits the performance of a kernel when one or more memories in the GPU cannot provide data at the rate requested by the kernel.

The profiling report was used in the current implementation to optimize the CUDA code. More about the results can be found in Chapter 5.

4.7.2 Collect Data On Remote System

As mention before, it is possible to collect data from a remote system where a GUI is not available, using the command-line nvprof. Remote profiling is the process of collecting profile data from a remote system that is different than the host system at which that profile data will be viewed and analyzed. Once all the profiling results are collected it is possible to access the information using a local Visual profiler. It enables a GUI and more compressive information about the application. There are two ways to perform a remote profiling. To use nvvp remote profiling you must install the same version of the CUDA Toolkit on both the host and remote systems. It is not necessary for the host system to have an NVIDIA GPU [23]. For the current application, remote profiler was used. However, the server did not have an external monitor or virtual. Therefore, it was not possible to obtain all the profiling analysis, thus we used a local laptop for profiling.

Finally, the chapter gives an overview of practices and performance studies for GPGPU and a better understanding of the hardware and memory management on the GPU. In addition, the hardware limitation are able to determinate the best usage of the GPUs. The NVIDIA's profiling tool is useful to analyze different stages of our application. Therefore, is possible to determinate elements of the CUDA code where is better to optimize from others, thus, gain an improvement in performance and reduction of executing time.

Chapter 5

Optimization Results

In this chapter we present the results of the CUDA code implementation launched on a single GPU device. The tests were performed on various GPUs architectures. The application is analyzed on various stages using NVIDIA's Visual Profiler. In addition, the CUDA kernels were evaluated in performance, execution time, occupancy and concurrent kernels. Furthermore, the results, are analyzed and optimized using the schemes from Chapter 4. The code is executed remotely on the supercomputer Piritakua at the Department of Multidisciplinary Studies at Yuriria, Guanajuato of the University of Guanajuato. The last section is an overview of all the optimization results achieved in the physics simulation.

5.1 Supercomputer Piritakua

The experiments were carried out using the supercomputer Piritakua. Dr. Claudio from the University of Guanajuato is in charge of maintaining and administrating the GPU Cluster. The supercomputer is located at Yuriria, a small town in the center of Mexico. Nonetheless, it is possible to access to cluster and make test from any part of the world.

The cluster has the CentOS (Community Enterprise Operating System) 64 bits as operating system. The OS is a free operating system and one of the most popular GNU Linux distribution for web servers and supported by RHEL (Red Hat Enterprise Linux) [3]. The specifications of cluster are 5.1.

The host code was executed on only two types of CPUs: on an eight core Intel i7-3630QM and on a high-end eight core Intel Xeon CPU E5620. In addition, the Xeon was used in all the simulation test, except when testing on the GeForce 670mx. Lastly, the code was

Processor	Number	Cores	RAM
Server Dell Intel(R) Xeon(R) E5620 2.4 GHz	1	8	12 GB
Server HP Proliant SL 350s Gen3 Intel(R) Xeon(R) X5650 2.67 GHz	2	24	32 GB
Server HP Proliant SL 250s Gen8 Intel Xeon E5-2670 2.60 GHz	3	48	104 GB

TABLE 5.1: CPU technical specifications

executed on laptop to show the performance comparison between a lightweight GPU and a server based GPU.

The device code was executed using the CUDA Toolkit 5.5 version. The main advantage of the 5.5 version are the improvements in MPI(Message Passing Interface) and HyperQ. The MPI implementation was not used for the current simulation. However, we obtained benefits from using the HyperQ when applying concurrent kernels to the implementation.

When accessing Piritakua remotely, it is possible to use all the GPUs nodes available on the cluster. The specifications of the GPU connected to the back-end are as follow, CC stands for compute capability.

Model	Core	RAM	DP GF	SP GF	Bandwidth	GHz	CC	Power
Tesla K20m	2496	5GB	1,170	3,520	208GB/s	0.73	3.5	225W
Tesla M2070	448	6GB	515	1,030	150GB/s	1.15	2.0	225W
Tesla C2050	448	2.5GB	512	1,030	144GB/s	1.15	2.0	238W
GeForce 580	512	1.5GB	520	1,154	192.2GB/s	1.5	2.0	244W
GeForce 670mx	960	3GB	520	1,154	67.2GB/s	0.6	3.0	-

TABLE 5.2: GPU technical specifications

The code was launched on all Piritakua's GPUs and on the GeForce GTX 670m. The "m" stands for the mobil graphic cards. In addition, the 670m card is designed for less power usage, but high graphics power. It has more cores than some Tesla models, however, they have less Bandwidth than standard versions. The 670m card was used as comparison between the laptop GPUs and the high-end desktop/servers GPUs.

5.1.1 Architecture Differences

NVIDIA's GPU have constantly been improved over the years. Nonetheless, most programming paradigms have stayed the same. There are currently three main architectures: Fermi, Kepler and Maxwell. For example, a new GPU a streaming processor is able to handle up-to 2048 threads at a time, but the maximum block size stayed at 1024. Another example is the use of Shared Memory. Maxwell has 64KB dedicated Shared

Memory. The maximum amount of Shared Memory per Block is 48KB for all three architectures [13]. Table 5.3 provides information on the models.

There are two GPU architectures where the implementation was launched: the Fermi and the Kepler. The Tesla K20m and the GeForce 670mx are based on the Kepler GPU architecture. The Tesla M2070, M2050 and the GeForce GTX 580 were implemented on the Fermi architecture. The Kepler architecture is newer than the Fermi. More information about the architectures is provided in the Table 5.3. The Maxwell architecture was not used for the simulation. However, it is showed for future reference.

Name	Fermi		Kepler		Maxwell
Compute Capability	2.0	2.1	3.0	3.5	5.0
Single Precision Operation per Clock/SM	32	48	192		128
Double Precision Operation per Clock/SM	4/16[1]	4	8	8/64[2]	1[3]
Max Number of Threads per SM / SM	16		32		
Max Number of Registers per Thread/SM	1536		2048		
Max Number of Threads per Block	1024				
Active Thread Blocks per SM / SM	8		16		32
Max Warps per Multiprocessor/ SM	48		64		
Registers / SM	32K		64K		
Level 1 Cache	16/48 KB		16/32/48 KB		64 KB
Shared Memory / SM	16/48 KB		16/32/48 KB		64 KB
Warp Size	32				

TABLE 5.3: GPU Architecture Specifications

5.2 Optimization

The cluster has two main architecture types Fermi and Kepler. Furthermore, the initial results of 1.00x speedup were tested on a Kepler architecture, the GeForce GT 650M. The Figure 5.1 illustrates the executing time for all the GPUs in the server.

We used NVIDIA's Visual Profiler to obtain kernel metrics of the Tesla K20m, see Table 5.4. The output is organized by kernel importance and performance during the simulation. The Laplacian kernel evaluation (`glaplacinay`, `gLaplacianx` and `gLapBoundaries`) uses up-to 44.37% of the overall simulation. The *gsolution* kernel, which solves Zhang-Li Model 2.2 consumes up-to 14.04%. The RK4 integration only exhausts a minor part of the overall simulation. However, the `gSolution`, `gsdExchange` and Laplacian calculation are part of the RK4 integration, which overall is about 99%.

The throughput was not reviewed on the current application. Only two stages of the simulation transfer of the CPU data occurs: on the initial stage where the CPU data is sent to the GPU, and the final stage where the data is retrieved from the GPU

The optimization focus is to give the GPUs as much work as possible, using at the fullest the GPU hardware capabilities. In addition, reducing the overall performance time of each kernel by eliminating the computational hover-head process on the highest consumed kernel 5.4.

Time%	Time	Calls	Avg	Min	Max	Kernel
23.50	3.6s	26521	137.5us	96.0us	597.1us	`gLaplaciany`
17.04	2.6s	26521	99.7us	57.0us	561.1us	`gSolution`
16.75	2.6s	26522	98.0us	62.8us	400.6us	`gLaplacianx`
13.37	2.0s	26522	78.2us	40.8us	453.8us	`gsdExchange`
7.22	1.1s	26522	42.2us	23.4us	326.0us	`gsfRelaxation`
6.22	965.2ms	6630	145.6us	79.2us	722.6us	`gTerm4RK4`
4.12	640.3ms	26522	24.1us	21.8us	138.7us	`gLapBoundaries`
3.41	529.2ms	6630	79.8us	41.6us	478.8us	`gTerm2RK4`
3.36	520.8ms	6630	78.5us	41.5us	372.2us	`gTerm3RK4`
3.35	519.5ms	6631	78.3us	41.1us	372.2us	`gTerm1RK4`

TABLE 5.4: Kernel time executing and on the Tesla K20

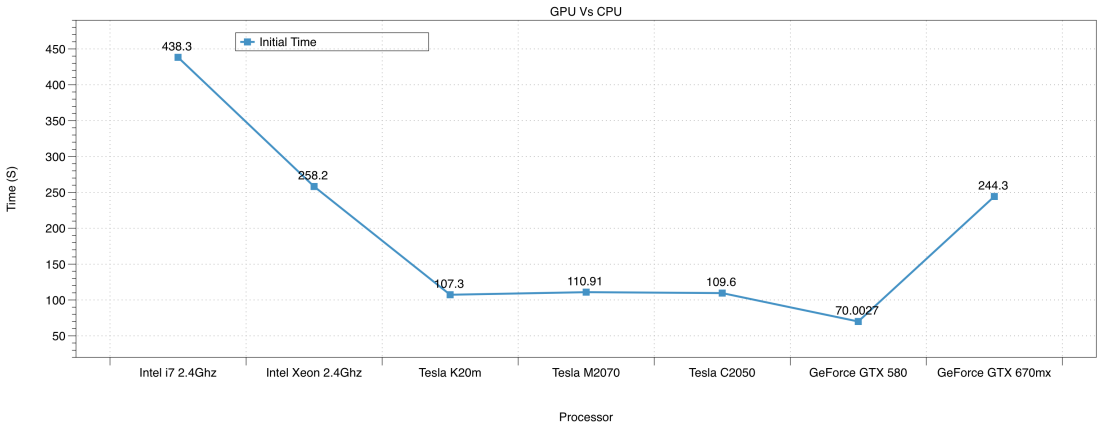

FIGURE 5.1: Initial implementation benchmarking on several different GPUs nodes, with a initial speedup of 1.0x

Figure 5.1 illustrates the GeForce GTX 580 which is the card with the least amount of execution time, and the Tesla K20m is the fastest amongst the Tesla Cards.

The following sections are the optimization techniques and methods applied to the application as well as comparing the performance between the initial implementation and the modified versions. The optimization is broken down into five steps; Branching, Occupancy, Concurrent Kernels, Shared Memory and Structure of Arrays. Branching refers to how kernels and threads are executed in the application. Occupancy refers to the number of threads per devices being used. Concurrent Kernels is the execution of several kernels at once. Shared Memory is using as much shared memory as possible. Finally, Structure of Arrays is the modification of the memory allocation in the device.

5.2.1 Branching

CUDA follows the Single Instruction Multiple Thread architecture, which means, all threads on the device execute the same code. Each thread is able to operate on its own data and has its individual address counter. Moreover, threads are free to use a different path. Each thread launches the same operation at the same time. However, they have to wait for all the threads in the kernel to finish their task. In other words, some threads can finish their job before groups of threads are executing their tasks. Therefore, a thread within a warp/block branches differently the other threads get deactivated [13]. The method is described in the following code 5.1 and illustrated in Figure 5.2.

```
__global__ void kernel(int* out){
idx = threadIdx.x;
int result;
if(idx == 0){
    result = foo();
} else {
    result = bar();
    out[idx] = result;
}
```

LISTING 5.1: Branching threads

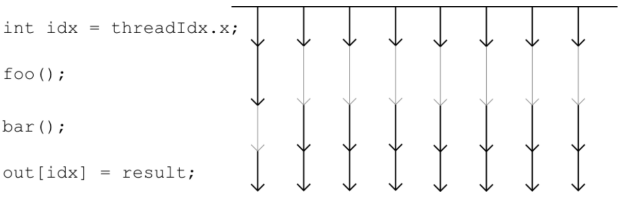

FIGURE 5.2: The execution flow of a branching code, with a warp size of eight. Black arrows are active threads, and the grey ones are disabled.

The branching problem occurred in the section where boundary condition for Laplacian was being analyzed 5.2. Only a single kernel was used to check the boundary condition. In addition, a bottleneck occurred. The implementation gets the job done with only one kernel. However, a minor part of the threads are only working, which is a waste of computation resources and energy.

```
__global__ void glaplaciany(...); //Compute Laplacian in Y direction
__global__ void glaplacianx(...); //Compute Laplacian in X direction

__global__ void glaplacianyboundaries(...){
    if (i > 1 && i < NX + 2 && j == 0){
        // Update Laplacian Boundaries
    }
    else if (i > 1 && i < NX + 2 && j == 1){
        // Update Laplacian Boundaries
    }
    else if (i > 1 && i < NX + 2 && j == NY - 2){
```

```
        // Update Laplacian Boundaries
    }
    else if (i > 1 && i < NX + 2 && j == NY - 1){
        // Update Laplacian Boundaries
    }
}
```

LISTING 5.2: Branching problem in the Laplacian boundary condition evaluation

To solve the branching issue, we included more work on the laplacian boundary condition kernel. The new kernel evaluates the boundary condition in a single kernel. Therefore, this refers to eliminating branching threads, more importantly, reducing global memory calls, see Listing 5.3.

```
__global__ void glaplacianyboundaries(...){
    if (i > 1 && i < NX + 2 && j == 0){
        // Update Laplacian Boundaries
    }
    else if (i > 1 && i < NX + 2 && j == 1){
        // Update Laplacian Boundaries
    }
    else if (i > 1 && i < NX + 2 && j == NY - 2){
        // Update Laplacian Boundaries
    }
    else if (i > 1 && i < NX + 2 && j == NY - 1){
        // Update Laplacian Boundaries
    }
    glaplaciany(...); //Compute Laplacian in Y direction
    glaplacianx(...);  //Compute Laplacian in X direction
}
```

LISTING 5.3: More workload on a single kernel execution

The technique was applied to all parts of the code. Therefore, this eliminated inactive threads and activated threads for more computational process. The technique increased the occupancy percentage of active threads within the kernels. The results of the modified version are in the final section of the chapter.

5.2.2 Concurrent Kernels

Initially, each kernel was launched on the default steam zero. Therefore, every kernel was consequently launched in a serial way. Figure 5.3 illustrates such results using the NVIDIA's Visual Profiler. Each kernel that is being launched is not able to run simultaneously, because, each kernel needs previous data to compute the next data. In other words, the kernels are not independent from each other. Therefore, we changed the implementation to be able to launch parallel kernels.

FIGURE 5.3: Kernels running on the default Stream zero.

Kernels by default cannot run in parallel with others kernels. Furthermore, CUDA does not provide an automatic parallel kernel executing. In addition, the programmer needs to tell the CUDA compiler that some portion of the code or kernel should be run in parallel. However, the compiler does not always know when to use concurrent kernels and this depends on the hardware capabilities and as well the number of threads per block and the number of SM available. If the compiler finds available space to run another kernel simultaneously, it will do so.

For example, the kernel `gsolution` from the Listing 5.4 computes the Zhang-Li Model for x, y, z coordinates, which extensively uses the global memory of the device. To accomplish concurrent kernels, the streams should be able to access memory blocks that are pinned to a specific stream. Therefore, each memory block, which correspond to x, y, z coordinates is mapped to three independent streams. Furthermore, all the matrices corresponding to the coordinate x are mapped to the stream 1, coordinate y to stream 2 and coordinate z to stream 3.

```
__global__ void zhangLiComplete(...)
{
    deltamX[index] = sfrelaxX[index] + sdexX[index] + laplX[index] - smX[index];
    deltamY[index] = sfrelaxY[index] + sdexY[index] + laplY[index] - smY[index];
    deltamZ[index] = sfrelaxZ[index] + sdexZ[index] + laplZ[index] - smZ[index];
}
```

LISTING 5.4: Evaluation of x, y, z coordinates of the Zhang-Li Model in a single kernel.

The single kernel `zhangLi` from the Listing 5.4 is divided into three separated kernels. In addition, the new generic kernel is able to launch parallel kernels, see Listing 5.5. Instead of running one big kernel, three individual kernels are able to launch simultaneously. With the new implementation, shared memory is no feasible. Perviously this was not possible, due to that the matrices were pinned to different locations of the GPU hardware.

```
__global__ void zhangLi(  MatrixCoordinate )
{
    int i = blockIdx.x * blockDim.x + threadIdx.x + 2;
    int j = blockIdx.y * blockDim.y + threadIdx.y;
    int index = j * NXCUDA_CONST + i;

    if (i > 1 && i < NX + 2 && j >= 0 && j < NY)
        deltam[index] = sfrelax[index] + sdex[index] + lapl[index] - sm[index];
}
```

LISTING 5.5: Evaluation of individual coordinates of the Zhang-Li Model.

We applied the current method to all of the kernels which was possible to separate into three kernels calls. Some kernels were not viable to be separated, such as the cross product, which uses memory allocations from different streams.

```
for (int i  = 0; i < 3; i++)
    gsolution<<<blocks, threads, 0, stream[i]>>>(spinAccXYZ[i]->getDev_deltam(),
                                                 spinAccXYZ[i]->getDev_sfrelax(),
                                                 spinAccXYZ[i]->getDev_sm(),
                                                 spinAccXYZ[i]->getDev_sdex(),
                                                 spinAccXYZ[i]->getDev_lapl());
```

LISTING 5.6: Evaluate Zhang-Li Model.

FIGURE 5.4: Concurrent kernels in the Tesla K20 using NVIDIA's Visual Profiler.

In Chapter 3 we mentioned that the input data set was 480 x 120 and mapped to a CUDA square grid of 512 x 128. Therefore, limiting the number executing threads within each kernel. Because of this, the extra threads were able to launch a different kernel in parallel with the current one, see Figure 5.4.

Concurrent kernels demonstrate a very promising technique to achieve a huge performance increment in the current simulation. In theory it is possible to have multiple kernels executing at the same. However, there are some downsides to the implementation; correctly synchronize kernels, waiting time and hardware resources are among the problems [22]. The timeline of the application 5.5 illustrates the waiting times between kernel execution. However, the waiting time are very small time steps between 0.01ms and 0.01ms, but waiting time occurs for each step of the RK4 and appears approximate 45,00 times. Furthermore, branching the kernel execution process should eliminate the issue.

5.2.3 Shared Memory

Shared memory is faster than global memory but it is very limited, see Chapter 4. To be able to implement shared memory in the kernels, we needed the kernels to be separated in their x, y and z coordinated, as mentioned in the previous section. In addition, this allows us to implement shared memory across each kernel, otherwise it would not be

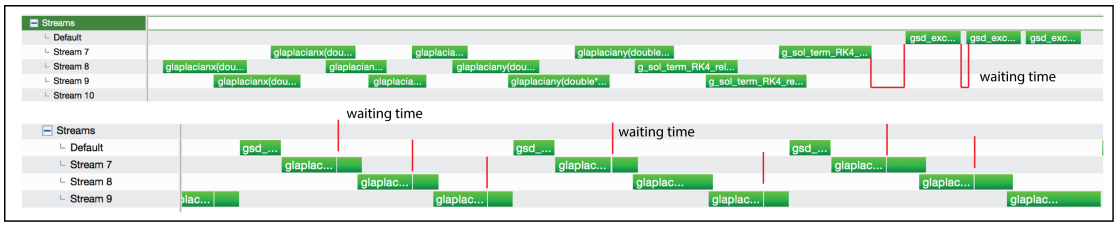

FIGURE 5.5: Waiting time between each concurrent kernel execution

possible. Shared memory was applied in all the kernels where global memory was used extensively, eliminating the number clock cycles per thread.

The idea behind shared memory is to reduce the amount of global memory calls, which has about 400-600 latency clock cycles, while the shared memory only has 1-32 latency clock cycles 4.4. The shared memory implementation is accomplished by allocating the data from the thread block into a temporary array, in other words the shared memory. In addition, the kernel is able to perform calculations on the temporary array and write the values onto the global memory. The implementation code is illustrated in the Listing 5.7. There is no guarantee that threads will execute at the same order. Using `__syncthreads()` will wait until all threads have completed their task which is in this case loading global memory into the shared memory array. Chapter 4 section thread synchronization, has more information about thread synchronization and shared memory. Once all the operations on the shared memory array are finished, the final part is to write the shared memory values back to the global memory.

```
int i = blockIdx.x * blockDim.x + threadIdx.x;
int j = blockIdx.y * blockDim.y + threadIdx.y;
int index = j * NXCUDA_CONST + i;

if (i > 1 && i < NX + 2 && j >= 0 && j < NY){
     int cacheIdx = threadIdx.y * blockDim.x + threadIdx.x;
     __shared__ double deltamS[THREADS_SHARED * THREADS_SHARED];

     //load memory into shared memory
     deltamS[cacheIdx] = operationGlobal(globalMemory);
     __syncthreads();

     //copy back the shared memory to global memory
     deltam[index]  = deltamS[cacheIdx];
}
```

LISTING 5.7: Shared memory

To calculate the Laplacian, we need to access a great amount of global memory, located near the value of interest. In this case this is region of 4x4x4 grid space. Figure 5.6 illustrates what part of the block is used for allocating shared memory and global memory.

The global memory is used for the boundary conditions of the block, while the shared memory is for all the values inside the block.

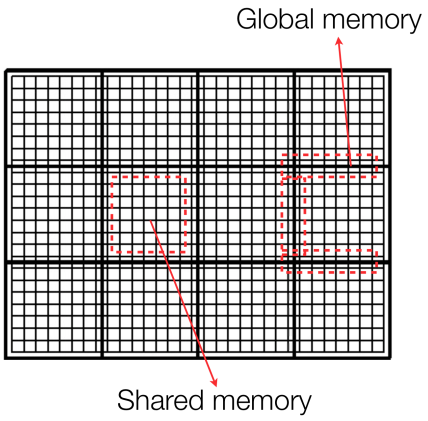

FIGURE 5.6: Shared Memory Strategy for the Laplacian evaluation

Listing 5.8 demonstrates how to calculate the Laplacian from Equation 2.3 with shared memory. Firstly, we load all the global memory into a temporary array, the shared memory. Then we performed the calculation on the shared memory as mentioned before. The last step is to return data to the global memory.

```
if (i >= 0 && i < NX && j >= 0 && j < NY){
    __shared__ double lapS[ THREADS_SHARED * THREADS_SHARED];
    lapS[sIdx] = deltam[Index];
    __syncthreads();

    if (threadIdx.x >= 2 && i < threadIdx.x - blockDim.x -2){ //shared
       lapy[idx] = - lapS[sIdx + 2] / 12.0 + 4.0 * lapS[sIdx + 1] / 3.0
                   - 5.0 * lapS[sIdx] / 2.0
                   - lapS[sIdx - 2] / 12.0 + 4.0 * lapS[sIdx - 1] / 3.0;
    else{ //global memory
        lapy[idx] = - deltam[idx + 2] / 12.0 + 4.0 * deltam[idx + 1] / 3.0
                    - 5.0 * deltam[idx] / 2.0
                    - deltam[idx - 2] / 12.0 + 4.0 * deltam[idx - 1] / 3.0;
    }
}
```

LISTING 5.8: Laplacian evaluating using shared memory with boundaries condition

The current approach seems very promising for reducing global memory. However, a great amount of time is spent on loading data onto the shared memory array, In consequence, this delays threads executing and results in a decrease in performance. Therefore, fast allocating shared memory data is the optimal solution to ensure the optimal use of this type of memory. The results of such implementation are in the following section, optimization results.

5.2.4 Structure of Arrays, SAO

AoS and SoA refer to Array of Structures and Structure of Arrays respectively. These two terms refer to the two different ways of laying out data in the device or host. This is illustrated in Figure 5.7 and 5.8 respectively. The AOS approach, groups up properties of an object together and makes an array of those objects in the memory. The SOA would be a single structure in which you make an array for each property. The structure of arrays can allow for better cache utilization, make easier to access continues data, make better use of each read you make from memory, provide a more effective route to memory.

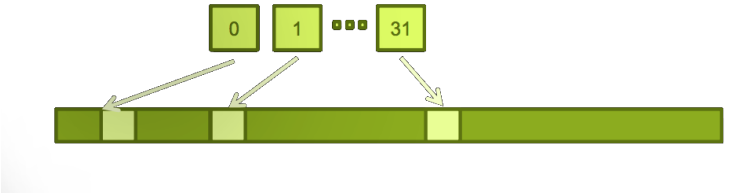

FIGURE 5.7: AOS memory layout

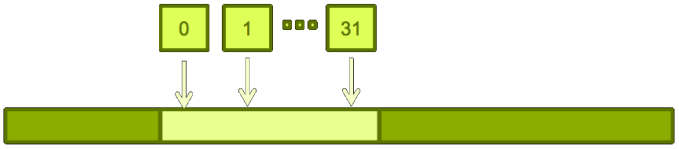

FIGURE 5.8: SAO memory layout

The initial implementation of the x, y, z data was allocated in separated blocks. Furthermore, when accessing blocks of the same coordinates, the register access the data. Review Listings 5.9 and 5.10.

```
deltam_x = (double **)calloc(NYCUDA, sizeof(double *));
deltam_y = (double **)calloc(NXCUDA, sizeof(double *));
deltam_z = (double **)calloc(NXCUDA, sizeof(double *));
```

LISTING 5.9: AOS implementation

To solve the issue, a custom class GPUMatrix was programmed. The class contained all the matrices for the device. Moreover, the classes allocated the data for each Matrix and freed the memory automatically when the simulation was over. This was allocated in a structure that was easier for the device to access common elements. For example, when evaluating operations only on the x coordinate, the kernel physically accesses matrices that are near by and eliminate unnecessary shift in registers or hierarchy memory access.

```
GPUMatrix<T> *dev_deltam;
```

```
GPUMatrix<T> *dev_sdex; //Exchange term
GPUMatrix<T> *dev_sfrelax;
GPUMatrix<T> *dev_m;
```

LISTING 5.10: SOA implementation

The current approach eliminates unnecessary data shift in registers, and it is able to stack more values per registers. In theory, there are more computational time per threads. The results of such implementation are illustrated in the last section of the chapter. With the new implementation, the code becomes more reusable for future improvements.

5.2.5 Occupancy

Firstly, we increased the use of constant memory in the device. thus, eliminating redundant evaluations of variables and operations. The results are an increase in performance and more computational workload on each thread. In addition, constant memory modifications are illustrated in the Listing 5.11.

The matrices 2.7 and 2.7 evaluates the boundary conditions, which do not change over the course of the application, They are only calculated at the initial stage of the application. Therefore, it is possible to implement those matrices using constant memory and thus, eliminating unnecessary calls in the device.

```
gsource << <blocks, threads >> >(u_val, dev_sm_z, dev_mz, NXCUDA);

sfrelax_y[index] = -deltam_y[index] / tau_sf;

DELTAX = (double)TX / (double)NX;
```

LISTING 5.11: Constant Memory changes

The different numbers of threads per block and as well the number of blocks per grid can dramatically increase or decrease the performance of the application. Table 5.5 illustrates the different threads per block configuration on the GeForce GTX 580.

Threads	Shared	speedup	time	Occupancy
8x8	8x8	7.217x	52318.3	56.6%
16x16	8x8	7.625x	49517.3	86.6%
16x16	16x16	7.978x	47329.2	100.0%
32x32	16x16	7.356x	51333.4	66.6%
32x32	32x32	Failed		

TABLE 5.5: Threads per block configuration and occupancy on the Fermi architecture

Using NVIDIA's Profiler made it possible to obtain the occupancy percentage of threads the device. The initial configuration for the Fermi and the Kepler was 32x32 threads

per block for global memory and 16x16 threads per block for the shared memory. We found, that the optimal configuration for the Fermi cards was 16x16 threads per block and as well for the shared memory and for the Kepler cards was 32x32 threads per block for both memory types.

5.3 Optimization results

This section is an overview of the optimization results compared with the initial CUDA implementation. Each version of the code is compared with the first test results, which were done using the GeForce GT 650M. Figures 5.9 and 5.10 illustrate the time execution and the speedup in Tables 5.6 and 5.7. The final version of the code is the Occupancy. Moreover, the greatest performance occurred on the GeForce GTX 580 Card with 8.0x speedup 5.10.

GPU	Original	Constant	Streams	Shared	SAO	Occupancy
Tesla K20m	107322.7	101513.4	97106.0	90201.7	68988.2	66456.0
Tesla M2070	110912.3	103212.4	130754.1	97343.4	73938.1	70299.3
Tesla C2050	109635.1	101212.4	128516.6	96762.0	72964.5	69358.1
GeForce GTX 580	70002.7	68712.2	76481.9	68567.1	51603.7	47213.2
GeForce 650m	244372.9	237371.9	227237.8	279804	181217.4	174419

TABLE 5.6: GPU Optimization time

Table 5.6 displays the overall executing time for all the version of the code, original, constant, streams, shared memory, SAO and Occupancy. We can see the comparison of the initial time and the final. There is a difference between 40s and 50s time decrease. The time reduction is relatively low. There is no large difference in waiting 100s or 66s for a simulation to be completed. However, if we increase the data set to five decimal points, the simulation could take up-to a couple of hours or days to process, thus with an the optimal implementation we will reduce the time by a factor of eight.

Finally, the speedup comparison in Table 5.7 and Figure 5.10 illustrates how much performance increase we could obtain in a new simulation using the new implementation.

GPU	Original	Constant	Streams	Shared	SAO	Occupancy
Tesla K20m	3.517x	3.718x	3.888x	4.186x	5.473x	5.682x
Tesla M2070	3.403x	3.534x	2.888x	3.879x	5.107x	5.371x
Tesla C2050	3.442x	3.571x	2.938x	3.902x	5.175x	5.444x
GeForce GTX 580	5.391x	5.521x	4.937x	5.551x	7.317x	8.0x
GeForce 670MX	1.544x	1.598x	1.662x	1.349x	2.084x	2.163x

TABLE 5.7: GPU Speedup performance

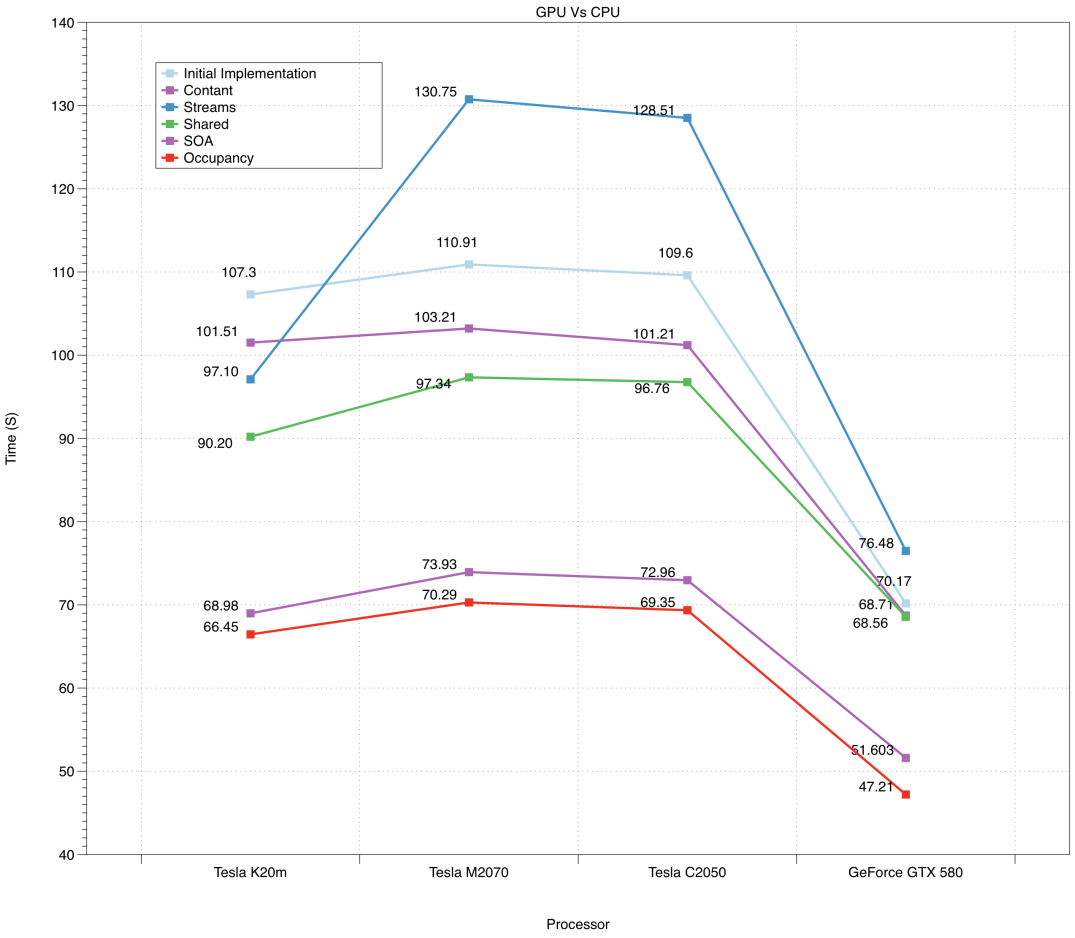

FIGURE 5.9: Overall simulation time

Based on the initial profiling in Table 5.4, the Laplacian evaluation consumed about half of the overall simulation time. However, on the final optimization results 5.11, the Laplacian was reduced from 44.37% to 26.24 % on execution time. But more importantly the importance of the kernel was reduced, in other words more computational workload on the other kernels. The same occurred for the Runge and Kutta term evaluation. The speedup mainly occurred in the Occupancy version of the code, see Table 5.7. Furthermore, the `gsdExchangeFull` incremented from 13.37%. 23.35%, which is not necessary good. The increment in time is due to shift in streams operators. The `gsdExchangeFull` is processed in the default stream zero, while the others kernels launched concurrently in a different stream. Table 5.11 illustrates the final profiling results using NVIDIA's profiling tools.

Tesla K20m was the only GPU which in all the code modifications did not lose performance over the course of the optimization process. However, the other GPUs dropped performance in the stream optimization stage. The stream process is where each kernel was divided into three separated kernels. Doing this, we were able to calculate the x, y, z coordinates independently. In addition, this enabled room to implement shared memory

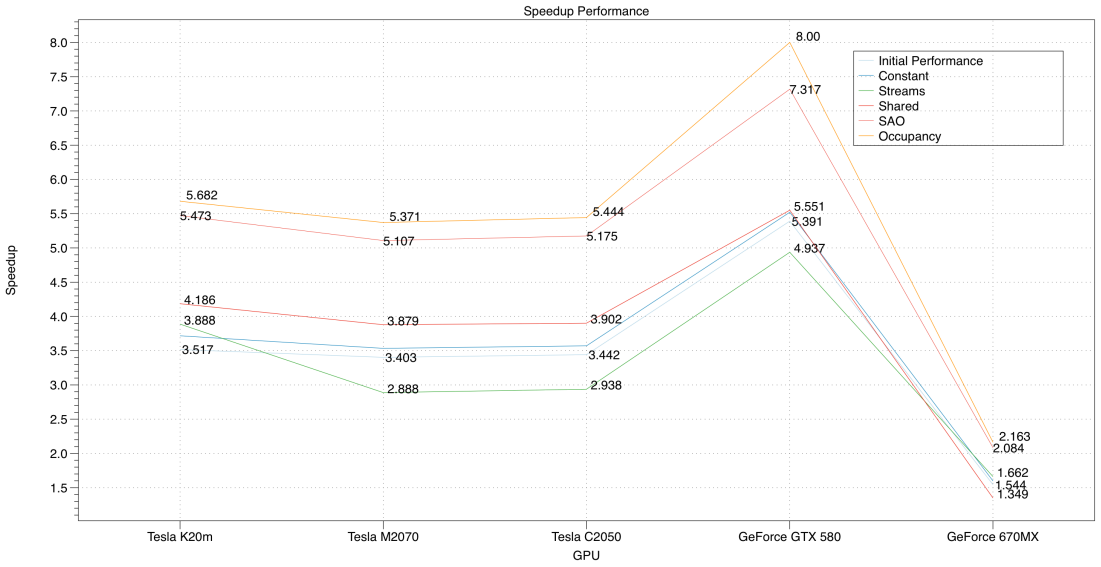

FIGURE 5.10: Speedup performance output.

Time %	Time	Calls	Avg	Kernel
35.24	21.33s	216330	98.5us	`gSolTermRK4Relaxation`
26.24	15.88s	288441	55.0us	`glaplacianXYBroundaries`
23.35	14.13s	160000	88.3us	`gsdExchangeFull`
15.17	9.18s	72108	127.2us	`gSolTerm4RK4Relaxation`

FIGURE 5.11: Final optimization results using NVIDIA's profiler, on the Tesla K20m

across possible kernels. The GPU table specifications 5.2 illustrates that the Tesla K20m is the only GPU card with CC of 3.5. Therefore, it has access to Hyper-Q technology. which is able to synchronize automatically concurrent kernels, just by activated a steam process to the kernel.

The SOA optimization, improved overall dramatically the performance of the application, obtaining a 1.2x - 2.0x speedup in all GPU cards, see Table 5.7 and Figure 5.10. The final version, Occupancy, improved up-to 0.7x speedup on the 580 GPU. However, the Teslas cards improved only 0.2x-0.25x speedup. The speedup difference, 0.5x, is due to the process cycle of the GeForce GPU.

We expected that the newest card, the Tesla k20m, would obtain the highest speedup overall, certainly because it has more CUDA cores and the highest compute capabilities. However, it falls behind the GeForce GTX 580 with about 2.5x speedup difference, see Table 5.7. In addition, the Tesla K20 performance results only have a difference of 0.3x speedup compared with the other Teslas cards, see Figures 5.10 and 5.12. If we compare the GTX 580 card with the others GPUs specifications, it has the most Processor clock (GHz) and more mathematical calculations per cycle, see Table 5.2.

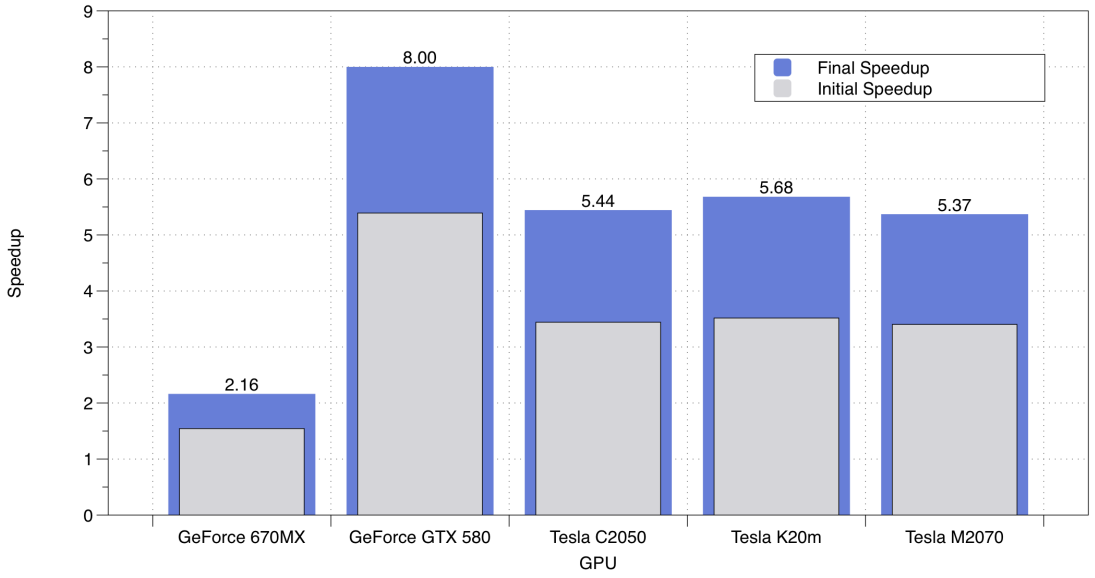

FIGURE 5.12: Optimization speedup overview

The results demonstrate the increase workload and occupancy on the device as well the increment in computational process per thread and per kernel.

Finally, the techniques and practices from Chapter 4 were used to archive speedup and increase performance on the initial CUDA implementation. Methods used were: increased the use of constant memory, shared memory, changed the memory allocating access, analyzed thread branching and finally analyzed kernel occupancy. The highest performance of all GPUS did not occurr on the newest NVIDIA card, the K20m, thus, this was the expensive of all the GPUs available in the cluster. The actual improvement appeared on the GeForce 580 (the more GHz of all GPUs) with a 2.32x speedup difference in comparison with the Tesla K20m, see Table 5.10. If the problem data set is to be scaled, for example, to a simulation of eight days, the speedup performance of 8.0x will drastically decrease the time execution in only one day. Moreover, it is possible to execute more simulations on the initial time step.

Chapter 6

Conclusions and Future Work

GPUs definitely have a place in the world of computational physics and other similar applications. Their use allows to carry out teh same work with less energy and more science with less resources. They make HPC computing clusters affordable for small research groups. The true limit test of this new technology will be if it is actually used to advance new science. In the field of computational physics studies, this pushes the barrier of what is computationally feasible, from speedups of 1.5x to 20x using GPUs [21].

Acceptance has been slow due to many factors, GPUs are sometimes seen as a fad or a niche. The specialized skill set and effort required for GPU programming along with the risk of spending money to setup a GPU cluster, raises a concern for productivity and viability of this technology. Adopting this technology requires abandoning legacy codes and smart optimizations that have been developed over the years. A wrong choice may result in wasted time and effort.

What is certain is at the moment, it is the overall direction of the industry towards higher parallelism. Single threaded performance has reached a local limit, where all types of processors are seeking more performance out of parallelism. This means that a large portion of the work needed to parallelize a code for a certain parallel architecture will most probably be applicable to another parallel architecture as well. From the literature and my experiences, one can observe that in order to achieve good results in programming with GPUs it is necessary to take a heterogeneous approach to coding. That refers to adopting multi-threaded CPUs and concurrent GPU type algorithms.

Spintronics, in particular involves designing new magnetic materials for spin-devices and modeling and understanding of spin-transport at molecular and atomic scale. Manipulating magnetic domain walls to store and transfer information is envisioned to enable

high-density, low-power, non-volatile, and non-mechanical memory. Promising for future systems for example the racetrack memory by Parkin at IBM, where DWs can be moved by applied magnetic fields and/or by currents via the spin transfer torque effects such as the simulation proposed by this research. However, most of the technologies and experiments are still in development. Furthermore, there are several obstacles to be overcome to enable these technologies.

Using computer simulation is possible to predict the outcome of the theoretical approach. In this when reproducing the effects of spin diffusion by numerically implementing the Zhang-Li Model into micormagnetics, we apply a current to a regime of DWs in a NiFe soft nanostrips. Furthermore, it provid the theoretical experiments with high precision on relative inexpensive computers. By using the highly parallel capabilities of the GPU it was possible to dramatically reduce the computation time of the simulation from around 400s on the non-threaded CPU version to 41s on a GPU. Therefore, it is now possible to execute ten times more simulations on the same time frame.

Through the optimization we achieved a maximum speedup of 8.0x. The result did not occur on the newest device, the K20m, but on the mid-range GPU, the GeForce GTX 580, which has more clock cycles (GHz) than the others cards. The newest device, the K20m, is ten times more expensive than the 580 card. The Tesla cards are mainly designed for server used, multiple users, while the GeForce is designed for high performance graphics, high workload on a single user. The optimization approach was focused on giving more workload on the GPUs, more performance per SM and increasing the work of threads per block. Lastly, the GTX 580 is not the newest GeForce card available on the market. The new GeForce 980 is expected to have a 40% increase in performance over the 500 GTX series.

The simulations were performed on a relative small data set. Moreover, it is possible to increase the data set, in other words, a bigger magnetization data. This overall will reduce the execution time by a factor of 8.0, increasing the number of simulations in the same time frame. Future test are likely to be performed on a newer GPU architectures, such as the Maxwell, for example, on the new GeForce 980 or on the Tesla K80.

The current thread is to push the hardware capabilities and performance along with Mooers' Law, despite these issues there are some trends in the hardware industry that will make working with GPU easier and more widespread within a HPC context. The following are some examples:

3D Memory

A possibility is stacking DRAM chips into dense modules with wide interfaces, and this brings them inside the same package as the GPU. This lets GPUs get data

from memory more quickly boosting throughput and efficiency allowing us to build more compact GPUs that put more power into smaller devices. The results are: several times greater bandwidth, more than twice the memory capacity and quadrupled energy efficiency.

NVLink

Todays computers are constrained by the speed at which data can move between the CPU and GPU. NVLink puts a fatter pipe between the CPU and GPU which, allows data to flow at more than 80GB per second, compared to the 16GB per second available now.

Pascal Module

NVIDIA has designed a module to house Pascal GPUs with NVLink. At one-third the size of the standard boards used today, they will put the power of GPUs into more compact form factors than ever before.

Mobile and Embedded Devices

With the new Tegra K1 it is possible to do supercomputing on the level of mobile devices, achieving upto 1 TFlops of performance. Embedded systems with CUDA capabilities have the possibility to integrate high performance algorithms to such small platforms.

Cloud Computing

NVIDIA is pushing the limits of brining computer graphics to the cloud. The idea is for everybody to have access to high quality computer graphics.

Using heterogenous computing is possible to dramatically decrease computational time on CPU applications that are not feasible with the current CPU paradigm. They also offer room for new types of computational simulations in a reasonable time frame.

To conclude, I offer my personal perspective on GPU computing and I think the importance of using accelerator hardware is an economic and environmental issue and part of the future. The environmental aspect of doing computing is often overlooked, but an ever increasing important one to be considered. As heavy computer users we will have to take responsibility for our electricity use. The benefit of less energy use is clear with more computational power. These ideas will provide definite benefits for future use.

Bibliography

[1] Titan, oak ridge and nacional laboratory. https://www.olcf.ornl.gov/titan/, 2013, Cited January 2015.

[2] J. A. Anderson, C. D. Lorenz, and A. Travesset. General purpose molecular dynamics simulations fully implemented on graphics processing units. *J. Comput. Phys.*, 227(10):5342–5359, May 2008.

[3] CentOS. Centos project. http://www.centos.org/, 2015, Cited January 2015.

[4] S. Che, M. Boyer, J. Meng, D. Tarjan, J. Sheaffer, S.-H. Lee, and K. Skadron. Rodinia: A benchmark suite for heterogeneous computing. pages 44–54, 2009.

[5] D. Claudio-Gonzalez, A. Thiaville, and J. Miltat. Domain wall dynamics under nonlocal spin-transfer torque. *Phys. Rev. Lett.*, 108:227208, Jun 2012.

[6] S. Cook. *CUDA Programming: A Developer's Guide to Parallel Computing with GPUs*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 1st edition, 2012.

[7] J. Fagerberg, D. C. Mowery, and R. R. Nelson. *Handbook of magnetism and advanced magnetic materials*, volume 2. Wiley-Interscience, Chichester, Sep 2007.

[8] R. Farber. *CUDA Application Design and Development*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 1st edition, 2011.

[9] E. Golovatski. *Spin Torque and Interactions in Ferromagnetic Semiconductor Domain Walls*. BiblioBazaar, 2012.

[10] W. H., S. A. Teukolsky, W. T. Vetterling, and B. P. Flannery. *Numerical Recipes in C (2Nd Ed.): The Art of Scientific Computing*. Cambridge University Press, New York, NY, USA, 1992.

[11] P. Haney and T. U. of Texas at Austin. Physics. *Spintronics in Ferromagnets and Antiferromagnets from First Principles*. University of Texas at Austin, 2007.

[12] A. Harju, T. Siro, F. Canova, S. Hakala, and T. Rantalaiho. Computational physics on graphics processing units. 7782:3–26, 2013.

[13] T. Hörmann. Gpu-optimized implementation of high-dimensional tensor applications. Master's thesis, Institut für Informatik, Technische Universität München, Dec. 2014.

[14] IBM. The application of spintronics. http://www-03.ibm.com/ibm/history/ibm100/us/en/icons/spintronics/, 2013, Cited January 2015.

[15] G. Karapetrov and V. Novosad. Domain walls riding the wave. *Physics*, 3:96, Nov 2010.

[16] D. B. Kirk and W.-m. W. Hwu. *Programming Massively Parallel Processors: A Hands-on Approach*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 1st edition, 2010.

[17] R. Landaverde, T. Zhang, A. K. Coskun, and M. Herbordt. An investigation of unified memory access performance in cuda. 2014.

[18] K.-J. Lee, M. Stiles, H.-W. Lee, J.-H. Moon, K.-W. Kim, and S.-W. Lee. Self-consistent calculation of spin transport and magnetization dynamics. *Physics Reports*, 531(2):89 – 113, 2013. Self-consistent calculation of spin transport and magnetization dynamics.

[19] Z. Li and S. Zhang. Domain-wall dynamics and spin-wave excitations with spin-transfer torques.

[20] J. Nickolls and W. J. Dally. The gpu computing era. *IEEE Micro*, 30(2):56–69, mar 2010.

[21] NVIDIA. Popular gpu-accelerated applications. http://www.nvidia.com/docs/IO/64497/NV_GPU_Accelerated_Applications.pdf, 2012, Cited January 2015.

[22] NVIDIA. *CUDA C Best Practices Guide*, 2014.

[23] NVIDIA. Cuda documentation. http://docs.nvidia.com/cuda/#axzz3ORV92FoV, 2014, Cited January 2015.

[24] S. S. Parkin, M. Hayashi, and L. Thomas. Magnetic domain-wall racetrack memory. *Science*, 320(5873):190–194, 2008.

[25] D. A. Patterson and J. L. Hennessy. *Computer Architecture: A Quantitative Approach*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 1990.

[26] D. C. A. Ralph and M. D. Stiles. Spin transfer torques, 711.

[27] C. Richard, M. Houzet, and J. Meyer. Andreev current induced by ferromagnetic resonance. *Phys. Rev. Lett.*, 109:057002, Jul 2012.

[28] F. Robert. What itll take to go exascale. *Science*, 335(January):394–396, 2012.

[29] G. Ruetsch and M. Fatica. *CUDA Fortran for Scientists and Engineers: Best Practices for Efficient CUDA Fortran Programming*. Elsevier Science, 2013.

[30] J. Sanders and E. Kandrot. *CUDA by Example: An Introduction to General-Purpose GPU Programming*. Addison-Wesley Professional, 1st edition, 2010.

[31] J. B. Schneider. Understanding the finite-difference time-domain method howpublished= www.eecs.wsu.edu/~schneidj/ufdtd, year = 2010.

[32] E. Tsymbal and I. Zutic. *Handbook of Spin Transport and Magnetism*. Taylor and Francis, 2011.

[33] S. O. Valenzuela. Nonlocal electronic spin detection, spin accumulation and the spin hall effect. *International Journal of Modern Physics B*, 23(11):2413–2438, 2009.

[34] C. Wang. Characterization of spin transfer torque and magnetization manipulation in magnetic nanostructures, 2012.

[35] N. Whitehead and A. Fit-florea. Precision and performance: Floating point and ieee 754 compliance for nvidia gpus.

[36] N. Wilt. *The CUDA Handbook: A Comprehensive Guide to GPU Programming*. Pearson Education, 2013.

[37] V. Zayets. Spin and charge transport in materials with spin-dependent conductivity. *Phys. Rev. B*, 86:174415, Nov 2012.

[38] S. Zhang and Z. Li. Roles of nonequilibrium conduction electrons on the magnetization dynamics of ferromagnets. *Phys. Rev. Lett.*, 93:127204, Sep 2004.