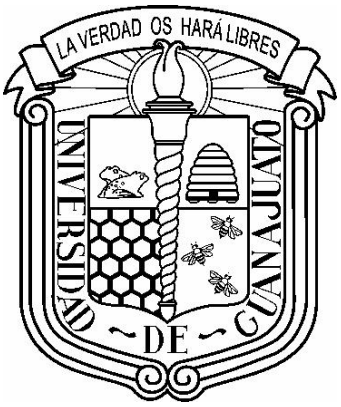

University of Guanajuato

Bachelor's Thesis

Study of Domain Wall Dynamics under Nonlocal Spin-Transfer Torque using heterogeneous computing

Author:
Thomas Sanchez Lengeling

Supervisor:
Dr. Claudio González David

A thesis to obtain the degree of Bachelor of Computacional Systems Engineering

in the

Campus Irapuato Salamanca
Division of Engineering
Department of Electronic Engineering

March 2015

UNIVERSITY OF GUANAJUATO

Abstract

Campus Irapuato Salamanca
Division of Engineering
Department of Electronic Engineering

Bachelor of Computacional 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 Zhang - Li 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 Runge - Kutta integration.

Acknowledgements

The acknowledgements ...

Contents

Abstract i

Acknowledgements ii

Contents iii

List of Figures v

Introduction vii

1 Heterogeneous Computing 1
- 1.1 Motivation . . . 1
- 1.2 GPUs as computing units . . . 4
- 1.3 Programming on GPUs . . . 5
  - 1.3.1 CPU and GPU multithread comparison . . . 8

2 Heterogeneous Performance Analysis and Practices 10
- 2.1 Practices . . . 10
- 2.2 Performance Metrics . . . 12
  - 2.2.1 Timing . . . 12
  - 2.2.2 Bandwidth . . . 12
- 2.3 Memory Handling with CUDA . . . 13
  - 2.3.1 Global Memory . . . 14
  - 2.3.2 Shared Memory . . . 15
  - 2.3.3 Constant Memory . . . 15
  - 2.3.4 Texture Memory . . . 15
  - 2.3.5 Thread Synchronization . . . 16
- 2.4 Concurrent Kernels . . . 17
- 2.5 Kernel Analysis . . . 17
- 2.6 Hardware constraints . . . 18
  - 2.6.1 Thread Division . . . 19
- 2.7 Visual Profiler . . . 20
  - 2.7.1 Profiler Kernel Report . . . 21
  - 2.7.2 Collect Data On Remote System . . . 22

3 Introduction to Domain Wall Dynamics under Nonlocal STT 23
- 3.1 Theory . . . 23

3.1.1 Spintronics . . . 23
3.1.2 Domain Wall . . . 24
3.1.3 Spin Torque in Domain Walls . . . 25
3.2 Domain Wall Dynamics under Nonlocal STT . . . 26
3.2.1 Theoretical Approaches . . . 26
3.3 Numerical Solution . . . 27
3.3.1 Finite differences in the time domain . . . 27
3.3.1.1 Boundary conditions implementation . . . 29
3.3.2 Fourth order Runge and Kutta method . . . 30
3.3.3 Effective Beta . . . 32

4 Implementation of Domain Wall Dynamics under Nonlocal STT 33
4.1 Simulation . . . 33
4.1.1 Data allocation and threads . . . 34
4.1.2 Initial Calculations . . . 36
4.1.3 Numerical Methods . . . 37
4.1.3.1 Finite differences in the time domain . . . 38
4.1.3.2 Zhang and Li Model . . . 40
4.1.3.3 Runge and Kutta . . . 40
4.1.3.4 Final evaluation . . . 41
4.1.4 Calculate effective beta . . . 41
4.2 Validation . . . 42

5 Optimization Results 43
5.1 Supercomputer "Piritakua" . . . 43
5.1.1 Architecture Differences . . . 44
5.2 Optimization . . . 45
5.2.1 Branching . . . 47
5.2.2 Concurrent Kernels . . . 48
5.2.3 Shared Memory . . . 50
5.2.4 Structure of Arrays, SAO . . . 52
5.2.5 Occupancy . . . 53
5.3 Optimization Results . . . 54

6 Conclusions and future work 57

List of Figures

1.1 GPU and CPU . . . 2
1.2 Architecture of a GPU . . . 4
1.3 Host and Device . . . 5
1.4 Programming Cycle . . . 6
1.5 Part of the CUDA's 2D grid . . . 7
1.6 Memory Space GPU and CPU . . . 8
1.7 CPU Thread execution . . . 8

2.1 PCIe Bandwidth . . . 11
2.2 Different memory types . . . 11
2.3 schematic cache hierarchy of a CUDA GPU . . . 14
2.4 Different memory types . . . 14
2.5 Texture Memory . . . 16
2.6 Concurrent Kernels . . . 17
2.7 Visual Profiler metrics graphs and plots . . . 20
2.8 Visual Profiler example . . . 21

3.1 Domain Wall VW, ATW . . . 25
3.2 Domain Wall - Vortex . . . 27
3.3 Domain Wall - Vortex . . . 27
3.4 FDTD grid . . . 28
3.5 Sampled at regular intervals a, Taylor expansion . . . 29
3.6 Euler Method . . . 31
3.7 Fourth-order Runge and Kutta Method . . . 32

4.1 Control flow . . . 35
4.2 2D Flatten array . . . 35
4.3 Grid layout . . . 36
4.4 Laplacian block calculation . . . 39

5.1 Initial GPU results . . . 46
5.2 he execution flow . . . 47
5.3 Initial Streams . . . 48
5.4 Streams kernels Tesla K20 . . . 50
5.5 Waiting time concurrent kernels . . . 50
5.6 Shared Memory Strategy . . . 51
5.7 Array of structures (AOS) . . . 52
5.8 Structure of Arrays (SAO) . . . 52
5.9 Structure of Arrays (SAO) . . . 55

5.10 Optimization results with the Profiler 55
5.11 Structure of Arrays (SAO) 55

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 benefit 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 exploration and study of the role of accelerator hardware like the use of the GPUs on physical computing, more specific in the area of spin-diffusion effects within a continuously variable magnetization distribution.

The work begins with a overview of the current trends in computing, focusing our attention specifically on GPUs, on how they differ from the CPUs and common programming practices that uses heterogeneous computing.

The next chapter is overview of the CUDA code implementation of Dr. Claudio's work "Domain Wall Dynamics under Non-local Spin-Transfer Torque".

The second chapter focus on optimization techniques and practices, which the GPU is able .

Also the necessary means how to test the speed-up against the CPU.

The forth chapter are the results collected by applying optimization techniques to the initial CUDA code implementation. The outcome is compared by launching the code in-to several GPUs nodes. Finally the last chapter of the thesis is a conclusion of the work and future research.

Chapter 1

Heterogeneous Computing

Heterogeneous computing refers 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 know 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 essential part of personal computer since the early use. Over the course of 30 years the graphics architecture has evolve form drawing a simple 3D scene to be 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 has been developed to increase the graphics pipeline and the render output. This lead 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, allowed various customization of the rendering process. However only allowed some modifications of the GPU output. Specific adjustment were extremely complicated 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 in rending video. In addition graphics shift from the CPU to the GPU. This was the beginning of GPU parallel capabilities [16].

At first the GPUs where only used for general-purpose computing (GPGPU) like computer graphics, but in-till resent years the GPU has been used to accelerate scientific research, analytics, engineering, robotics and consumer applications [11].

GPUs are attractive for certain type of scientific computation as they offer potential seed-up 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 [17].

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. Because of this there are limitations [22].

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.

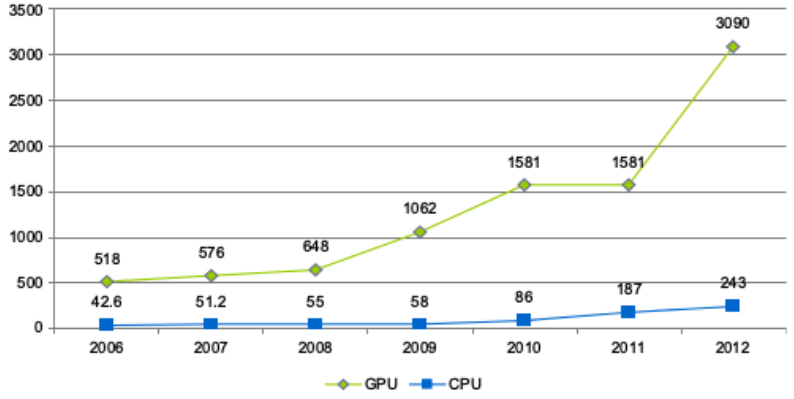

FIGURE 1.1: *GPU and CPU peak performance in gigaflops*

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 seed-up can be described by Amdahl's Law in terms of the fraction of parallelized work (f) [22].

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

The current paradigm of using CPUs for computing is growth unsustainable. In 2012, Japan among the countries with elite supercomputers, builded the machine "K Computer", with 705,024 multi-core CPUs, it can achieve 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 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 can archive an astonishing 24 petaflops theoretical peak performance. Moreover, with a power consumption of 8.2 MW. They demonstrated that 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 [20].

As said 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, in other words the GPU swap, switch threads very fast and they are extremely lightweight. In a typical system, thousands of threads are waiting to work. While the CPU only run 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. Although this can be extremely cost-wise operation on the GPU.

1.2 GPUs as computing units

A insight of the architecture of GPU can give a idea of why it outperforms the CPU on various benchmarking.

The GPU, unlike its CPU cousin, has thousands for registers per SM (streaming multiprocessor), this are arithmetic processing units. An SM can thought of like a multithread CPU core. On a typical CPU has two, four, six or eight cores. On a GPU as many as N SM core. The SM are configured in such a way that they are able to access memory located near 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) [5].

As commonly named 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: *Architecture of a NVIDIA GeForce GTX 580*

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 word a single instruction is executed throughout the device.

Being able to efficiently use a GPU for an application requires to expose 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. This 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 [12]. The 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 observed in the figure how the data is sent from the host to the device and vi-versa.

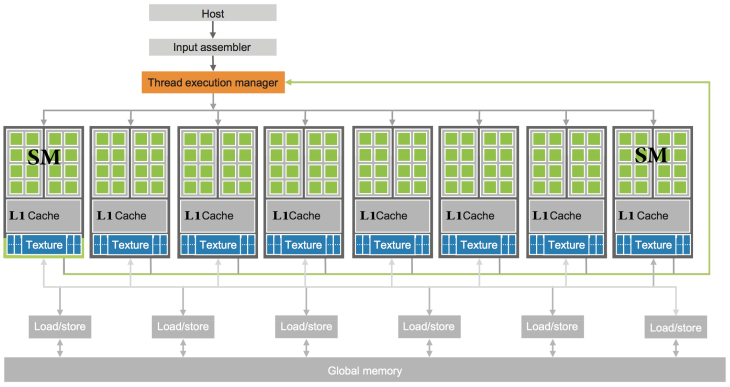

FIGURE 1.3: *Memory transfer between the host and device*

On the GPU precision and optimization are very important but there is a penalty for choosing performance or precession. All the GPUs are optimized for single precision floating operations, 24 bit size, Also provides double precision point, size of 53 bits. This is using the standard notation IEEE 754. Normally the GPU uses single precession(SP) by default, if chosen double precision (DP), normally there is a penalty of 2x - 4x speedup[32]. Libraries such as CUBLAS and CUFFT provides 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 programs parallel computing, but for there GPUs. While OpenCL is a open standard framework meaning that is possible to do parallel computing on other GPUs, like on AMD cards. Programmers can easily port 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 [12].

CUDA programming model views the GPU as an accelerator processor which calls parallel programs throughout all the SM [33]. 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 1.4.

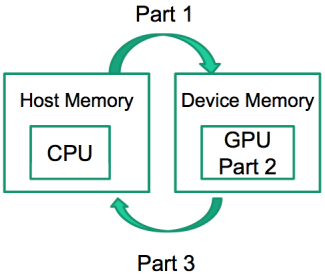

FIGURE 1.4: *Programming Cycle between the CPU and GPU*

- 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 namaspaces/CUDA keywords for parallel code, the CUDA compiler is called NVCC. The host code is the standard low level ANSI C language. However, 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 [12]. 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 can be access by 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 code 1.1. This is only the case for 1D block, which is widely used for shared memory access (see chapter 4) [26].

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

LISTING 1.1: *1D block*

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

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

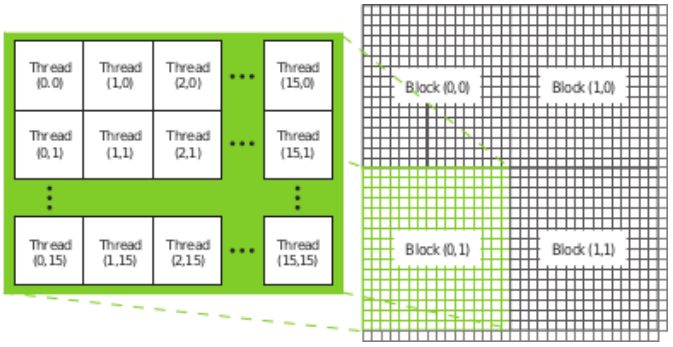

FIGURE 1.5: *Part of a 2D CUDA's thread grid, divided in blocks, each block with its own respective threads.*

In CUDA, host memory and device memory (all types) have separate memory spaces. Both of them have physically a separated location, for example, the RAM for either the CPU o the GPU. Furthermore, the programmer requires to send data from the host memory to the device's global memory (RAM) and vi-versa. The process is illustrated in figure **??**. Memory which is allocated in the device needs to be freed on the device,

the same occurs for the host memory. Moreover, the process is accomplish with similar devices 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 [12].

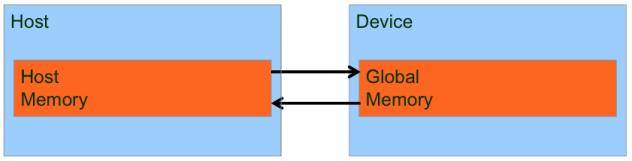

FIGURE 1.6: *Separate memory spaces for the CPU and GPU*

1.3.1 CPU and GPU multithread comparison

The current CPUs are typically multicore systems, which are capable of parallelizing code fairly easy. Suggest a parallel system. However we would require a large infrastructure [26]. 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 evens and the other core the odds, 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 thread execution*

On the GPU we are able to accomplish the same result of the CPU with lest amount of code. The code 1.3 demonstrate how to evaluate a 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, 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 parallel capabilities*

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.

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

LISTING 1.4: *kernel call*

To conclude, this chapter provided a overview of heterogeneous programming in a modern context. CUDA enhance the C language with parallel computing support. Which is possible to launch enormous amounts of parallel threads, oppose of 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 demonstration high performance on data-parallel applications [12].

Chapter 2

Heterogeneous Performance Analysis and Practices

When working with GPUs hardware new challenges emerges. How can we make the best usage of the GPU hardware. In the conventional CPU model, we have what is called linear or flat memory model. In addition this 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 a opposite memory location, all of this can be access as a contiguous block [5]. Meanwhile, on the GPU there are exceptions, their exists different memory hierarchies which dramatically change the performance output. By allocation the optimal memory types, speedup and increase throughput can be accomplished. To ensure optimization, some analysis should be done, like comparing latency, memory hierarchies and data bandwidth between CUDA kernels. The debug of parallel CUDA code can be accomplish using the NVIDIA's Visual Profiler. This chapter demonstrated techniques, practices and methods to debug and analyzed parallel process.

2.1 Practices

There are three rules for developing high performance GPGPU (General-purpose on the GPU) program, which are based on NVIDIAs GPU standards. [7]

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

As we know the GPUs are plugged into the PCI Express bus of the host computer, in other words the CPU. The PCIe bus has extremely slow bandwidth compared with the GPU. This is why is important to store the data on the GPU and keep it busy. In addition minimize the data transfer from the host and back to the device. We can see this in the table 2.1. CUDA enables the GPU to carry out petaFLOP performance in a single device [5]. In addition they are fast enough to compute a large amount of data. To accomplish such high performance, each CUDA Kernel needs to use all the available resources of the GPU. Furthermore, avoid wasting compute cycles. Finally if a single Kernel doesn't use all of the available bandwidth, multiple kernels can be launched at the same time on a single GPU, which are streams [7]

	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 2.1: *PCIe bus and GPU bandwidth comparison*

The practices should be taken in consideration to identify the portions of code where it would be beneficial for improving GPU acceleration.[18]

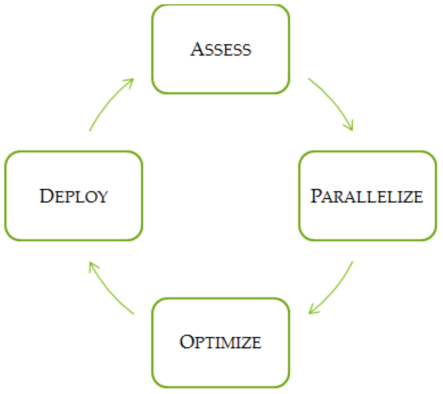

FIGURE 2.2: *Different memory type and penalties usage*

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

Increase parallelization from the original code, could be either 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, 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 implementation before carrying out a complete change.

2.2 Performance Metrics

There are many possible approaches to profiling the 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 giving kernel. Throughput is how many operations completed per cycle.

2.2.1 Timing

Timing a launched kernel can be done on either the GPU or the CPU. Is important to remember that the CPU and GPU are not synchronized. So its 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 [18]. 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. Note that the timing are measured on the GPU clock, so the timing is independent from the OS [5].

2.2.2 Bandwidth

The bandwidth refers to the rate at which data can be transferred between host and device and vi-versa. The bandwidth is one of the most important factors for testing performance o 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

base on the hardware specifications that is available by NVIDIA. This is calculated using the following formula:

$$theoreticalbandwidth = (clockrate * (bit - wide - memory - interface/8) * 2)/10^9$$

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

$$(1107 * 10^6 * (512/8.0) * 2)/10^9 = 141.6Gb/sec$$

The GTX 280 has a theoretical bandwidth of $141.6Gb/sec$. The effective bandwidth is calculated by timing specific program activities and by knowing how data is accessed by the application [18].

$$effective - bandwidth = ((Br - Bw)/109)/time$$

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[25].

In practice the difference between theoretical bandwidth and effective bandwidth indicated how much bandwidth is wasted on accessing memory and calculations. If the effective bandwidth is low compared to the theoretical bandwidth is one indication that there is not enough work being done in the GPUs. In addition there a several solutions; analyze the code to make more parallelize instructions, execute more computational instructions on the GPUs, finally analyze the number of threads per block that are executing on execute kernels.

2.3 Memory Handling with CUDA

In this section four types of memory handling are going to be explained, global memory (device memory), shared memory, texture memory and constant memory. The figure 2.3 illustrates physically the position of the different memory types inside the device chip.

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

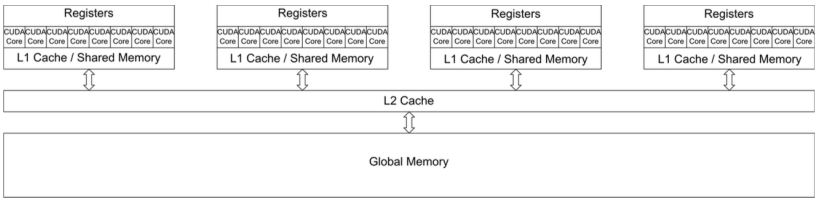

FIGURE 2.3: *The schematic cache hierarchy of a CUDA GPU with 4 Streaming Multiprocessors and 8 CUDA Cores each.*

core locations. Moreover, the memory access is very slow in comparison to the shared memory [5].

The figure 2.4 illustrates each memory type has it's bandwidth penalty of used as well as the latency in computer cycles. Different memory types can be used in different applications to maximize the memory used. The Shared Memory is very limited so it cannot be handler across all situations. Furthermore, when implementation wrong memory type on the device there can be latency penalty and bandwidth drop, instead of having a gain in performance.

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 2.4: *Different memory type and penalties usage*

2.3.1 Global Memory

Understanding how to efficiently use global memory is essential in CUDA memory management. Focusing on data reuse within the SM and caches avoids memory bandwidth limitations. Global memory on the GPU is designed to quickly stream memory blocks of data into the SM.

- Get the data on to the Device, keep it there.
- Give the GPU enough workload, this using 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 on the device, and it can be anything from 1 byte to 8GB, depending on the GPU. Also the memory is visible to all the threads

of the grid. Any thread can read and write to any location of the global memory, The memory is always allocated with *cadaMalloc*. And only global memory can be passed to the kernels and are called with _global _. [7]

2.3.2 Shared Memory

CUDA C compiler treats variables differently than a typical variable, it creates a copy of the variable for each block that is launched on the GPU, now every thread in that block can access the memory, this is why is called shared memory. This memory reside physically on the GPU, because the memory is very close the cache, the latency is typical very low [26]. One thing comes to mind, if the threads can communicate with others threads, so there should be 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 access we need to synchronize, when thread A finish writing then Thread B can access it. This is typical case when shared memory with synchronize thread is needed [5]. Shared memory is magnitudes faster to access than global memory, essentially is like a local cache for each threads 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.

2.3.3 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 [7]. A simple analogue is the *#define* or *const* attribute in the C++ programming language, the variable performs like a variable that cannot be modified. On CUDA this is exactly the same, 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[26].

2.3.4 Texture Memory

Like 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 id used for computer graphics applications, but it can also be use for HPC. The main idea of this read-only memory is that threads are likely to read from address "near' the address they nearby threads.[26]

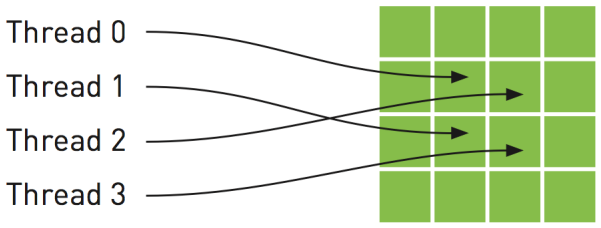

FIGURE 2.5: *Mapping of threads into a two dimensional array of texture memory*

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 table:

- 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.

2.3.5 Thread Synchronization

This refers to synchronizing threads operations. For efficiency, a pipeline can be created by queuing a number of kernels to keep the GPGPU busy for as long as possible. Further, some form of synchronization is required so that the host can determine when the kernel or pipeline has completed [7]. 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 complete.
- 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. Will wait for all threads to finish there job [7].

Threads synchronization can also be accomplish inside of the kernels calls. The idea is the same, the kernel will wait until all the threads have completed there task. Threads synchronization is general used when loading data into shared memory.

2.4 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. As the figure 5.3 illustrates.

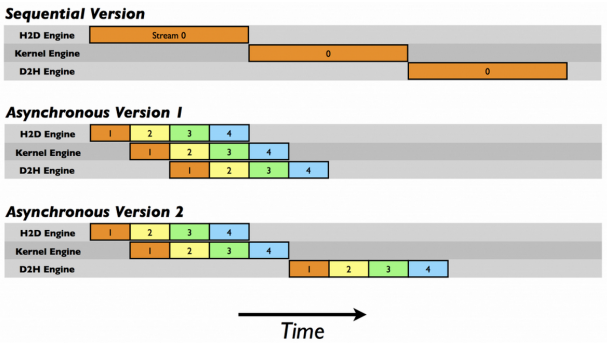

FIGURE 2.6: *Overlapping kernel execution using CUDA streams*

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 launch 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)[12].

By 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. We need to careful about two things. First, the host memory involved needs to be allocated, since 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 out 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 [26].

2.5 Kernel Analysis

As said before, kernels are the essential part of CUDA programming, threads are launched automatically throughout each thread blocks of the device. Furthermore it millions of

threads execute the same code parallel, which is When execution the

Memory Bandwidth Bound
: This refers when the code/application is limited by memory access. Most GPUs card have 1GB- 6GB of memory, this is used to process the data on the GPU, while the CPU has massively amount of memory available for use. A solution to this is to reuse the data, change the type of memory used in the GPU. A multi-GPU approach, launching kernels in several GPUs at once. This will dramatically increase the amount of memory in the application.

Compute Bound
: Refers to the computation time execution, in other works calculations done in the device, under the assumption that theres is enough memory for the calculations. what is actually the analysis time operations on the kernels. Theoretical bandwidth vs effective Bandwidth can measure performance for a compute-bound Kernel. Therefore its possible to increase the FLOPS per device.

Latency Bound
: Is one whose predominate stall reason is due to memory fetches. This is actually the saturating the global memory, or any type, but still have to wait to get the data into the kernel. Physically can be the data being sent from one part of the Device to the other. Also depends the time required to perform an operation, and are 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.

The performance of relatively simple kernels, which perform computations across a large number of data elements, is more a function of the GPU's memory system performance than the processing performance. It can be beneficial for such memory-bound kernels to decrease the amount of memory access required by increasing the complexity of the computation. [5]

2.6 Hardware constraints

This refers to the limit how many threads per block a kernel launch can have. If exceed this values they kernel will never run. The threads per block really depends of the hardware capabilities. The compute capabilitiesThe compute capability of a device is represented by a version number, also sometimes called its "SM version". This version

number identifies the features supported by the GPU hardware and is used by applications at runtime to determine which hardware features and/or instructions are available on the present GPU.[19] In a roughly summarized as:

- Each block cannot have more than 512/1024 threads in total. (Capability 1.x or 2.x-3.x)
- The Maximum dimensions of each block are limited to [512,512, 64]/[1024, 1024, 64](compute 1,1.2)
- Each block cannot consume more than to 8k, 16k, 32K registers total
- Each block cannot consume more than 16kb/48kb of shared memory

SM Resources, improve performance of an application by trading one resource usage for another. [18]

Another inefficiency that can cause low performance to the applications is the number transfers memory calls between the CPU and GPU. The GPU communicates with the CPU via a *PCIe* bus, in addition all of the massive FLOPS per second that can be achieve cannot actually be sent to CPU. The GPU should be filled with the enough workload at the beginning of the application and at the end only return the memory to the CPU.

2.6.1 Thread Division

There a hardware limitations in how much threads per block a kernel can handle. Launching a kernel with the hardware constrains of the device will only ensure us that the kernel will actually be executed in the device, Nonetheless, not 100% optimal and the results can be incorrect. Furthermore, is necessary to launch kernels with the amount of threads per block base on the hardware settings. The block size will determine how faster the code will run. However, not the biggest block will run faster, depends con the problem and the data set. By Benchmarking the application, is possible to find the optimal configuration that best fits the problem. One thing to keep in mind, thread blocks should be a multiple number of SMs, with this idea is possible to obtain optimal thread block configuration.

2.7 Visual Profiler

Is a hard task to keep track of each individual thread. This becomes difficult for debugging 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 archive maximum performance on the GPUs. The Profiler provides metrics in the form of plots and graphs, which describes opportunities to fully utilize the compute and data movements capabilities of the GPU, as well of each kernel launch in the application. See Figure 2.7.

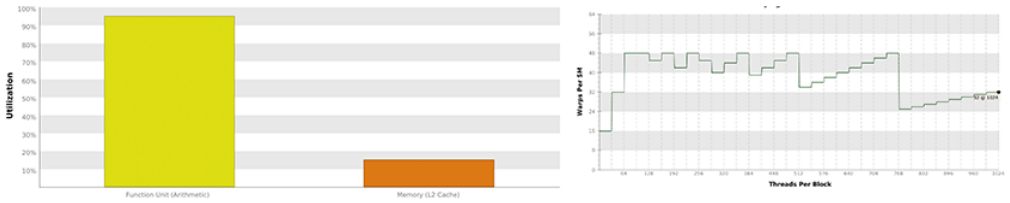

FIGURE 2.7: *Profiler provides optimization metrics*

NVIDIA's profiling tools comes in various flavors; 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 situations where profiling a region of the application is helpful [19].

- Analyze data initialization and movement in the CPU and GPU, as well as evaluating CUDA calls.
- The application operates in phases, where a 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 is possible to collect data from a portion of the iterations.

The Visual Profiler provides a step-by-step optimization guidance, where is possible to evaluate the GPU usage, examine individual kernels and analyze timeline of the application which the profiler shows memory movements and usage, CUDA calls, number of threads and performance. The figure 2.8 shows, each Kernel has its own percentage of execution time of the overall application [18].

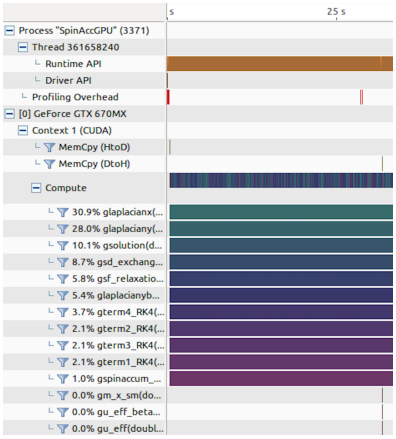

FIGURE 2.8: *Visual Profiler kernel execution*

2.7.1 Profiler Kernel Report

The profiler will execute several times the application for it to collect data from each kernels. This enables to precisely optimize phases of the application[26]. The profiling tools can verify how long the application spends executing each kernel as well the number of used blocks and threads. Through this is possible to obtain various memory throughput measures, like global load throughput and global store throughput, 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 time to collect data about each kernel. The information obtained by each kernel can be sum-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 is limiting the performance respectively.

2. **Instructions and Memory Latency**
   Instruction and memory latency 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.

2.7.2 Collect Data On Remote System

As mention before, 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 the data is collected is possible to access the data using the Visual profiler, which enables a GUI and more compressive information about the application. There are two ways to perform 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 [19].

Finally, this chapter gives a overview of practices and performance studies for GPGPU. In addition a better understanding of the hardware and memory management, as well as hardware limitation, which determinate the best usage of the GPUs. As we can see NVIDIA's profiling tools is useful to analyze different stages of our application, moreover to determined which parts of the CUDA code is better to optimize from others to gain more performance.

Chapter 3

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. This is a quantitatively test the effects of spin-diffusion, on real Domain Wall (DW) structures, by numerically implementing the Zhang - Li model into 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 Runge-Kutta integration (RK4).

3.1 Theory

Now as we know, each electron not only carries an elementary unit of charge e, but also carries 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. In other words, any electron charge transport is simultaneously accompanied by a transport of spin, or magnetic moment carried (passing electron) by these electrons [31].

3.1.1 Spintronics

Spintronics is a new type of electronics that exploits the spin degree of freedom of an electron in addition to its charge [34]. The interest is motivated by the quest to understand basic physical principles underlying the electron and nuclear spin interactions

in materials and by 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 brought a huge impact on industries and consumer electronics. Spintronics is a promising technology which will complement the present electronics with addition "spin" quantum freedom to charge freedom that is currently used in devices [10].

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 can also be generated also 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 [29].

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 and spin-down electrons is often nearly identical, but the states are shifted in energy with respect to each other. Fermi level is the term used to describe the top of the collection of electron energy levels at absolute zero temperature [29].

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[34].

3.1.2 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 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[8]. In this study only two DW are analyzed Vortex Wall and Asymmetric Transverse Wall.

Vortex Wall (VW)

In the case of Vortex wall the magnetization rotates in the place perpendicular to the domain wall, but the local magnetization is wrapped around a single vortex point. See figure 3.1.

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, asymmetric transverse wall, is the absence of such symmetry, figure 3.1.

FIGURE 3.1: *Vortex Wall (VW) and Asymmetric Transverse Wall (ATW)*

3.1.3 Spin Torque in Domain Walls

Domain walls are the basis for various spintronics devices that uses magnetic momentums, in other words spin of electronics, the used of the spin degree of freedom.

Spin Torque induced domain wall motion opens up a host of possibilities for applications. Advances in spintronics recognized by 2007 Nobel Prize in Physics have enable 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 [29].An interesting application using this idea is new design for a different type memory disk drive called racetrack memory by Parkin in 2008[21]. 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.

It deals with the spin-transfer torque in the case of a continuously varying magnetization. In this case the spin-transfer torque acts on inhomogeneous magnetization patterns, such as domain walls or magnetic vortices. Thus, also the magnetic processes in a racetrack memory and gyrating magnetic vortices driven by spin-transfer torque can be described.

The spin-depended, spatially varying potential inherent on a domain wall has interesting consequence for the transport of carries. 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. Sin each spin orientation experiences a different potential.

This has simulated research into domain wall (DW) dynamics, particularly those resulting from interactions with current passing through the DW via the phenomenon of spin momentum transfer (SMT) [29].

The success of spintronics untimely depends on out ability to precisely control the polarization of electrons transported within the actual thin film structure [24].

3.2 Domain Wall Dynamics under Nonlocal STT

3.2.1 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, expanded with a STT variable 3.1.

$$\frac{\partial \vec{m}}{\partial t} = \gamma_0 \vec{H}_{eff} \times \vec{m} + \alpha \vec{m} \times \frac{\partial \vec{m}}{\partial t} - \vec{T} \tag{3.1}$$

This novel idea of incorporating spin torque into the LLG equation has itself been incorporated into a model proposed by Zhang - Li in 2004 [15]. The LLG equation 3.1 is 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, as especially 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. Which lead to an even longer magnetization dynamics equation[35] [8].

$$\frac{\partial \delta \vec{m}}{\partial t} = D_0 \bigtriangledown^2 \delta \vec{m} - \frac{1}{\tau_{sd}} \delta \vec{m} \times \vec{M} - \frac{1}{\tau_{sf}} \delta \vec{m} + (\vec{\mu} \cdot \vec{\bigtriangledown}) \vec{M} \tag{3.2}$$

The equation 3.2 is referred to Zhang - Li model. Represents the non-adiabatic spin torque.

Therefore, a simultaneous solution of the diffusive Zhang and Li model together with the magnetization dynamics equation has uncovered a qualitatively new feature of the spin-transfer torque effect in the presence of spin diffusion. Spin diffusion is a process by which magnetization can be exchanged spontaneously between spin.

The sample that is considerate is a 300 nm wide and 5 nm tick NiFe soft nanostrip. This dimensions are widely used for experimental use.

Asymmetric Transverse Wall (ATW), maps of magnetization components of non equilibrium spin accumulation under a uniform current density with $D = 0,1$ and $10nm^2/ps$. Figure 3.2.

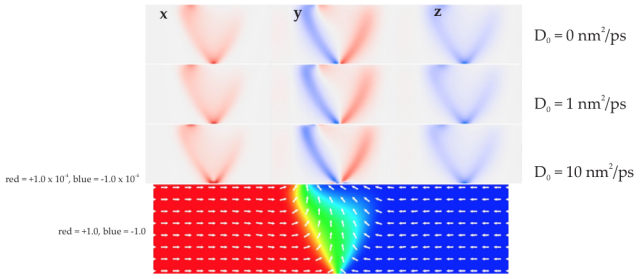

FIGURE 3.2: *Domain Wall - Vortex*

Vortex Wall (VW) same as for ATW, we point out the noticeable effect of the diffusion constant around the vortex core, which is the smallest feature of the wall. Figure 3.3.

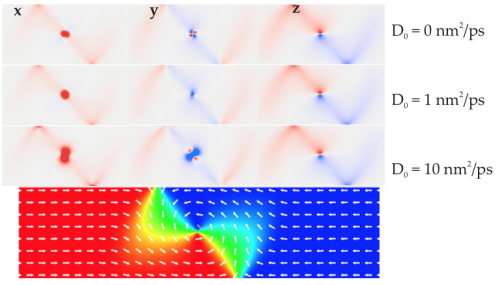

FIGURE 3.3: *Domain Wall - Vortex*

3.3 Numerical Solution

We Quantitatively test the effects of spin diffusion, on real Domain walls structures, this is done by numerically solve the Zhang-Li model into micro-magnetics, using the equation 3.2. The equation is physically realistic however computationally expensive.

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 Runge-Kutta integration.

3.3.1 Finite differences in the time domain

The finite difference in the time domain (FDTD) method can solve complicated problems, but it is generally computationally expensive. Solutions may require a large amount of memory and computation time. FDTD is a numerical analysis technique use for approximating solutions to the associates system of differential equations. The

method belongs in the general class of grid-based differential numerical modeling methods [6].

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, figure 3.4.

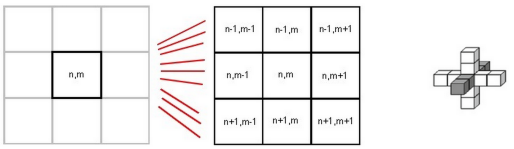

FIGURE 3.4: *The result for each cell is based on evaluating the derivate cell neighbors.*

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. $\bigtriangledown_x \times \bigtriangledown_y \times \bigtriangledown_z$ [6].

Looking at the equation 3.2, we need a method to calculate the first and second derivate. With the Taylor expansion we can perform such calculation. The Second order Taylor expansion readily yields expressions for the first and seconds 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, a, see figure 3.5. Second order Taylor expansion readily tiles expressions for the first and seconds central derivates that are widely used in numerics, namely $\frac{df}{dx} = \frac{f_{i+1} - f_{i-1}}{2a}$ and $\frac{d^2 f}{dx^2} = \frac{f_{i+1} - 2f_i + f_{i-1}}{a^2}$.

However, the numerical derivation of the structure of a simple Bloch wall using such expressions soon reveals that second order Taylor expansion ledes to restricted accuracy. Fourth order expansion as actually been found to prove much superior. [6]

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

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

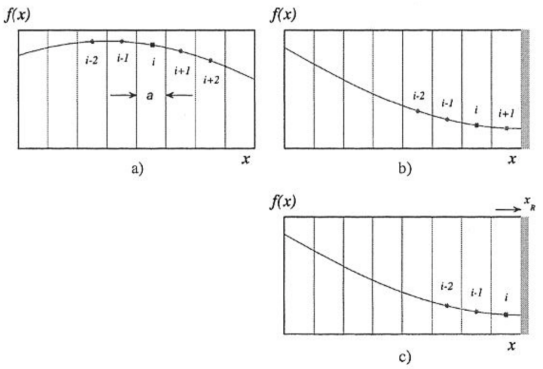

FIGURE 3.5: *Sampled at regular intervals a, (a) Function of inside the grid. (b) Mesh points second to closest to boundary. (c) Mesh points closet to boundary*

Applying the previous equation to nearest and next nearest neighbor to grid pint 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:

$$f_i^{(1)} = \frac{f_{i-2} - 8f_{i-1} + 8f_{i+1} - f_{i+2}}{12a} \tag{3.3}$$

$$f_i^{(2)} = \frac{f_{i-2} + 16f_{i-1} - 30f_i + 16f_{i+1} - f_{i+1}}{12a^2} \tag{3.4}$$

The equation 3.3 for the second derivate based on second nearest neighbors expansion solves for the laplacian operator in the Zhang -Li Model equation 3.2. However, points close to the edges need to be evaluated for great precession.

3.3.1.1 Boundary conditions implementation

Expressions such as 3.3 are valid when the grid point becomes closet or next-to-closet to the boundary of the magnetic box. Specific accuracy preserving, expansion need to be worked out. The general principal in the 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. [6]

Consider first a point second to closet to bound, 3.5-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

$$f^{(1)}(x) = \sum_{k=0}^{\infty} \frac{(x - x_i)^{k-1}}{(k-1)!} f^{(k)}(x_i) \tag{3.5}$$

Using 3.5-b. $x_R - x_i = 3a/2$ becomes.

$$\begin{bmatrix} -2a & \frac{(-2a)^2}{2!} & \frac{-(2a)^3}{3!} & \frac{(-2a)^4}{4!} \\ -a & \frac{(-a)^2}{2!} & \frac{(-a)^3}{3!} & \frac{(-a)^4}{4!} \\ a & \frac{(a)^2}{2!} & \frac{(a)^3}{3!} & \frac{(a)^4}{4!} \\ 2a & \frac{(2a)^2}{2!} & \frac{(2a)^3}{3!} & \frac{(2a)^4}{4!} \end{bmatrix} \begin{bmatrix} f_i^{(1)} \\ f_i^{(2)} \\ f_i^{(3)} \\ f_i^{(4)} \end{bmatrix} = \begin{bmatrix} f_{i-2} - f_i \\ f_{i-1} - f_i \\ f_{i+1} - f_i \\ f^{(1)}(x_R) \end{bmatrix} \tag{3.6}$$

Similarly, for a point closet to boundary, reference 3.5-c, grid points i + 1 and i + 2 are missing. The two first equation of ... need now to be replaced by a single equation, whilst the two remaining equations need to be truncated to the third order. For the geometry illustrated in 3.5-c, the minimal set of equations now reads.

$$\begin{bmatrix} -2a & \frac{(-2a)^2}{2!} & \frac{-(2a)^3}{3!} \\ -a & \frac{(-a)^2}{2!} & \frac{(-a)^3}{3!} \\ 1 & \frac{(+a)}{2} & \frac{(+a/2)^3}{2!} \end{bmatrix} \begin{bmatrix} f_i^{(1)} \\ f_i^{(2)} \\ f_i^{(3)} \end{bmatrix} = \begin{bmatrix} f_{i-2} - f_i \\ f_{i-1} - f_i \\ f^{(1)}(x_R) \end{bmatrix} \tag{3.7}$$

In both cases, and second derivatives and fully determined provided $f^{(1)}(x_R)$ be known along the boundary. For further reference [6]. The implementation to solve the laplacian with boundaries conditions check chapter 4.

The main advantages of the finite difference approach are ease of implementation, 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 with a rectangular mesh, resulting in some what discrete approximation, which ca produce significant error in the computation.

3.3.2 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 the step as an approximate solution of the first terms of the Taylor series. [27] The method can accurately solve a wide range of problems, but it is generally computationally expensive. Solutions require large amount of memory and computational time.

There exist several other computational numeric methods to solver 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, see figure 3.6. Euler method only has first order accuracy while the RK4 its fourth order integrator [23].

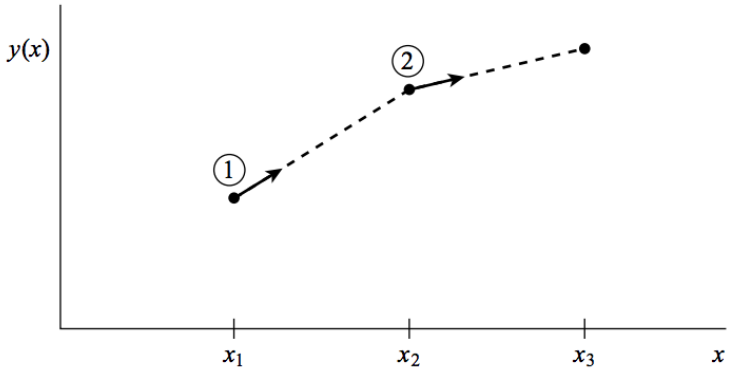

FIGURE 3.6: *Euler Method, Is the simplest approximate to solver differential equation or numerically solve equations.*

RK4 goes as follows:

$$y_{n+1} = y_n + 1/6K_1 + 1/3K_2 + 1/3K_3 + 1/6K_4 \tag{3.8}$$

where

$$\begin{aligned} K_1 &= h\dot{f}(x_n, y_n) \\ K_2 &= h\dot{f}(x_n + h/2, y_n + k_1/2) \\ K_3 &= h\dot{f}(x_n + h/2, y_n + k_2/2) \\ K_4 &= h\dot{f}(x_n + h, y_n + k_3) \end{aligned} \tag{3.9}$$

As the equations shows, 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 **??**.

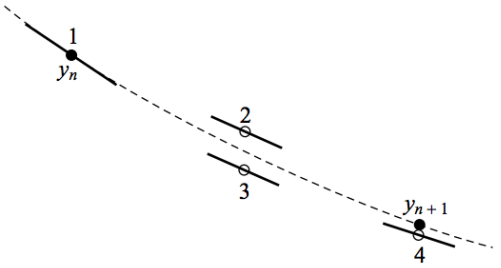

FIGURE 3.7: *Fourth-order Runge and Kutta method, Each step the derivative is evaluated four times.*

3.3.3 Effective Beta

We show that and effective non-adiabatic parameter β_{diff} dependent on the DW structure, provides

$$(\beta\upsilon)^* = \frac{\delta\vec{m} \cdot \partial_x\vec{m}}{\tau_{sd}\|\partial_x\vec{M}\|^2} \tag{3.10}$$

–fill final calculation.

The procedure is (i) compute the non-equilibrium spin density $\delta\vec{m}$ with the DW at rest, solve equation 3.2 to convergence or directly its time-independent version. (ii) compute the β_{diff} distribution from equation 3.10 and finally compute β_{diff} by averaging with the $|\partial_x\vec{m}|^2$ weight function.

Therefore, a simultaneous solution of the diffusive Zhang and Li model together with the magnetization dynamics equation 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. In summarize, we quantitatively test the effects of spin diffusion, on real Domain walls structures, this is done by numerically solve the Zhang-Li model into micro-magnetics. 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 4

Implementation of Domain Wall Dynamics under Nonlocal STT

This chapter is the study of the heterogenous implementation of 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 a the 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.

4.1 Simulation

The simulation consist of the integration of the equation 3.2, know as the Zhang and Li model for $1ns$ in a grid of 57,600 cells. The sample considered is a 300nm wide in y direction and 5nm thick in z direction NiFe (nickeliron alloy) soft nanostrip, a material and size widely uses in experiments with this characteristics. Using this size the asymmetric transverse wall 3.2, 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 4.1 illustrates mesh information and calculation box for the simulation [4]. In addition the table 4.2 shows the constant values for the numerical solution of the equation 3.2 such as μ, D_0, τ_{sd} and τ_{sf}.

The simulation is divided into two calculations parts, the host code and the device code. As the figure 4.1 illustrates the data flow of the simulation. Each step of the simulation

Mesh size	value	calculation box	value
Cell NX	480	Box TX	1200.0
Cell NY	120	Box TY	300.0
Cell NZ	1	Box TZ	5.0

TABLE 4.1: *Mesh size and calculation box*

is going to explain in the following section with more detail. First the initial values are read from a data file, which are the initial magnetization data coordinates. Then the data set is used to calculate the initial magnetization matrices for the simulation, the initial matrices are only calculated once. Afterwards, the simulation begins with the RK4 integration, which numerically solves the Zhang-Li model 3.2. The simulation is configured to integrate 50,000 times the Zhang-Li Model before calculating the final effective Beta value. The simulation stops if the effective beta convergence to 1.0e-9. The figure 4.1 shows the control flow of the simulation. But also illustrates the workload on the host and the device. As we can see the heavy computation operations are done on the GPU side. While on the CPU only minor intense computation are done such as I/O data, memory allocation and final beta variation.

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

TABLE 4.2: *Diffusion parameters and Runge - Kutta 4th*

4.1.1 Data allocation and threads

First section of the implementation starts by allocation data into several matrices for both host memory and device memory. The initial magnetization data can be either self generated by the application or by reading a file which contains information related to the magnetization. In both cases the data is divide into two blocks of data. Both blocks have 57600 (480 x 120) rows of information The first 57600 rows contains initial magnetization x, y coordinates. The next block of 57600 rows is the initial magnetization in x, y, and z. The data being read is stored on a three temporary 2D matrix, that corresponds to the x, y, and z coordinate. In addition the three matrices are flatten into three continuous memory blocks, as showed in figure 4.2. The device code 4.1 flattens the 2d index into a single linear 1D index.

```
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;
```

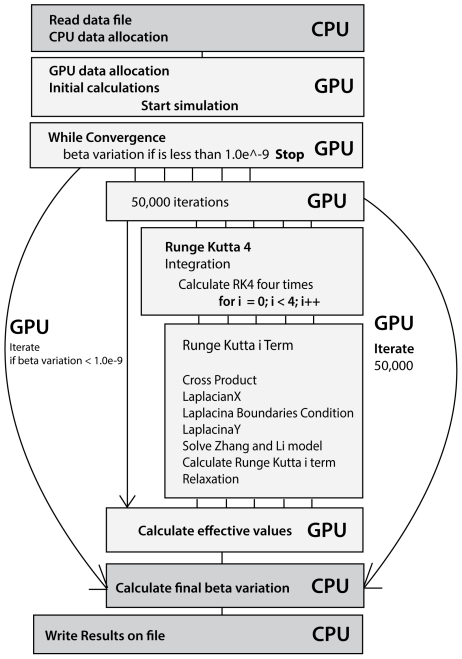

FIGURE 4.1: *Control flow of the simulation*

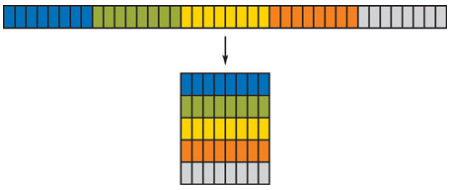

FIGURE 4.2: *Converting 2D array to a single continuous block of memmory*

LISTING 4.1: *Kernel Flatten*

To ensure optimal memory allocation on the GPU side is best to assign square matrices to the device. The calculation for this operation is done using the first two operations in 4.2.

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

TABLE 4.3: *Matrix allocation size*

The magnetization data is stored on three matrices x, y, and z, with size of 56,700 values, in other words 480 times 120. Base on this information we want to calculate the optimal number of grids that will ensure complete use of the hardware resources. The number of blocks per grid corresponds dividing the dimensions of the array by the number of threads. The last two operations in 4.2.

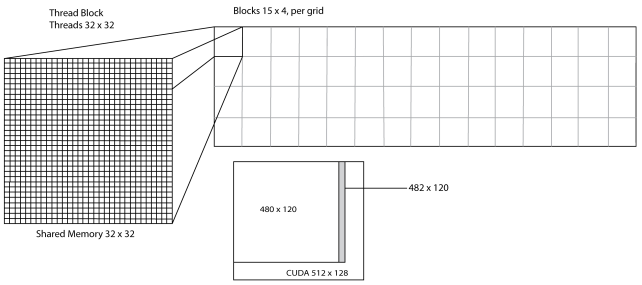

FIGURE 4.3: *Memory allocation in terms of the blocks per threads and grids*

```
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 4.2: *Device capacity calculation and number of block per grid*

Depending on the hardware properties, each GPU can allocate different number of threads per block and as well as a different shared memory size. The Shared memory in this implementation relays on the number of threads per block. In addition the number of blocks depends on the input matrix and the number of threads. See figure 4.3.

	Fermin	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 4.4: *Threads, blocks size*

4.1.2 Initial Calculations

In this initial step the $\delta\vec{m}$ is calculated from the equation 3.2, which is the non-equilibrium spin density [4] domain wall at rest. The Following equations are evaluated in the first step of the magnetization 4.3.

```
    //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 4.3: *Initial calculations*

In 4.3 the kernel *gsource* evaluates once the matrix *sm*, which is used to compute the Zhang-Li model in the RK4 integration.

///shift error // in implementation

As mention before the matrices for CUDA are square matrices 512 x 128, however the data set is 480 x 120. Because of the matrices size difference we need to limit the number of threads execution of the CUDA kernels. To limit such threads branching with a simple if within the kernels solves this issue 4.4. Due to the boundaries condition of the FDTD a small shift of 2 indices in the x direction is necessary.

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

LISTING 4.4: *Laplacian X using global memory*

4.1.3 Numerical Methods

micromagnetic simulation and demonstrate how to advance the configuration of the system through time. we do not know for how long we need to integrate the system until it stops in a local energy minimum configuration.

This step is where all the computational intense operations are performed. The algorithm 1 demonstrates the operations necessary for the RK4 integration process.

Data: deltam, sfrelax, sdex, sm, laplacian
Result: deltam
data initialization;
while $b_{eff} < 1.0e^{-9}$ **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
end

Algorithm 1: Runge and Kutta 4th integration implementation

4.1.3.1 Finite differences in the time domain

The finite differences method requires the domain of interest to be broken down into small regions. Such a subdivision of space is known as a mesh or grid, cell division is explained in the table 4.1, whose implementation in CUDA threads and blocks are done with 4.3 and 5.5. Based on the equations from chapter three 3.3. The first and second derivate are based on seconds nearest neighbors expands are calculated. The base idea is showed on the figure . The equation 3.3 is evaluated for three coordinates x, y and z.

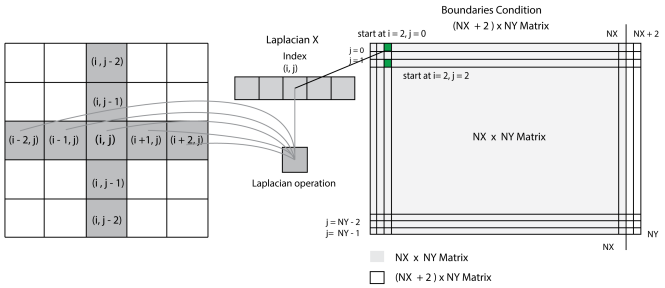

FIGURE 4.4: *Laplacian XY block calculation and boundaries condition*

As the Figure 4.4 demonstrate the calculation the nearest neighbors expansion by evaluation a neighborhood of $[-2, 2]$ values in the x direction. In addition the index begins at $i = 2$ and finishes at $NX + 2$. The same occurs for the laplacian in the Y direction, however it begins at $j = 2$ and finish at $NY - 2$. The implementation 4.5 uses the equation from chapter 3 3.3 as previous mention. The calculation 4.5 is done for each three x, y and z coordinates.

```
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;

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 4.5: *Laplacian X using global memory*

However the calculation of the nearest neighbors only work for values points inside inside a grid of [-2, 2] in x, y, and z. In addition we need to calculate the boundaries values for those values that are in the edge of the grid, such as the beginning points and the end points. To solve the boundaries issue we use two square matrices matrices from chapter three, The equation 3.7 for boundaries at $j = 0$ and $j = NY - 1$ for all values in the i cell, and the equation 3.6 for the condition $j = 1$ and $j = NY - 2$ for all i values. The code 4.6 demonstrate how the implementation was done, to achieve the correct values first the laplacian in the y direction is calculated, then the boundary condition, and finally the in the x direction.

```
__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 4.6: *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. In addition the three coordinate sum up to 172,800 cell points to be calculated.

4.1.3.2 Zhang and Li Model

The Zhang - Li Model 3.2 is solved using code 4.8. Furthermore the equatin is used for function f in the RK4 integration equation 3.8. The first term of the equation *sfrelax* is calculated in the relaxation process at the end of the RK4 integrator **??**. Then the term *sdex* is evaluated in the Cross Product process, which is done before the RK4 calculation. The matrix *sm* is only calculated once at the initial calculations process, in the kernel *gsource* 4.3. Finally the *lapl* matrix is evaluated in the Laplacian X, Y and boundary condition.

```
        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 4.7: *Runge and Kutta 4th Terms*

The output *solveZhangLi* of the Zhang - Li Model evaluation 4.7 (last term), past to the calculation of the i term of the Runge Kutta Integration.

4.1.3.3 Runge and Kutta

Intuitively the equation 3.8 is implemented by using four CUDA kernels, each kernel calculates the step of the integration. The first part of 4.8 computes the Runge and Kutta's 1st, 2nd, and 3rd term 3.9. The final fourth term, is the sum of the previous 3 terms, the last two lines of code in 4.8 show the calculation.

```
    rk1[idx] = dt * deltam[idx];  // Terms k1, k2, k3
    deltam[idx] = temp[idx] + 0.5 * rk1[idx];
```

```
    rk4[idx] = dt * deltam[idx]; //final k4
    deltam[idx] = temp[idx] + (rk1[idx] + 2.0 * (rk2[idx] + rk3[idx])
                                + rk4[idx]) / 6.0;
```

LISTING 4.8: *Runge and Kutta 4th Terms*

4.1.3.4 Final evaluation

The integration can be sum up as two *for* cycles. The first *for* calculating the RK4 and the second one each x, y, and z coordinate 4.9.

```
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 4.9: *Summarize of Runge and Kutta 4th Integration*

4.1.4 Calculate effective beta

The final procedure of the simulation calculates the effective beta. ... The following kernels are launched only when the RK4 integration is done, for this implementations is 50,000 iterations of RK4 4.1.

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

LISTING 4.10: *Calculate effective beta*

The final step of the simulation is calculating the beta variation value. This value tell us....

Once the effective beta diverges to 1^{-9} the simulation stops and the magnetization data is written. Depending on the application configuration is possible to write either the magnetization results for the VW or for the ATW. The magnetization data is written into two separated data files, which contains the effective data .eff and the spin accumulation data .spin.

4.2 Validation

Because CUDA framework is highly parallel system is fairly easy to obtain erroneous data from the calculations, even setting up the threads per block incorrectly is possible to get data set that is wrong, or results that don't diverge. When making changes to the code, its 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 the that is close enough to the results. A example of the validation performed.

According to our results, new code shouldn't produce errors in the spin.dat data greater than $7.0e^{-17}$, in other words valid code don't 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^{e-11} and no greater than 6^{-11}. For the diffuse beta variation the precision expected to be within the double precision range of 1^{-16}.

The results 4.5 are examples of the simulation output.

Data set	Simulation time	Diffuse beta
upVW magnetization	377590.3ms	4.848728452719814e-02
ATWpm magnetization	377409.2ms	4.054674178687585e-02

TABLE 4.5: *Calculation results*

To conclude, the simulating at its core uses the RK4 for integration which uses as function the Zhang-li model equation 3.2. In addition, to solve such differential equations the FDTD method is used. As we can expected the simulation is computational expensive, the RK4 3.8 evaluates fourth times the Zhang-Li equation. Moreover, each term evaluates numerically the laplacian 3.3 and boundaries conditions 3.7 3.6. We showed procedure of numerically solving the equations of chapter 3.

Chapter 5

Optimization Results

This chapter is the results of the CUDA code implementation launched on a single GPU device. The test were performed on various GPUs architectures, which, has different hardware characteristics. The application is analyzed by the 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 3. Lastly the code is executed remotely on the supercomputer "Piritakua" at the Department of Multidisciplinary Studies Yuriria, University of Guanajuato.

5.1 Supercomputer "Piritakua"

The experiments are carried out using the supercomputer Piritakua. The massive GPU cluster was design and built by Dr. Claudio from the University of Guanajuato Multidisciplinary Studies Yuriria. The GPU cluster is located at a small town of Mexico, Yuriria. The supercomputer at the Front-end has a eight core Intel Xeon at 2.4 Ghz, at the back-end several GPU are connected, one NVIDIA Tesla K20, two Tesla M2070 and a GeForce GTX 580.

The cluster has a GNU LINUX distribution installed, the CentOS 64 bits version 6.4. CentOS stands for Community Enterprise Operating System which is free operating system and one of the most popular GNU Linux distribution for web servers and as well is supported by RHEL (Red Hat Enterprise Linux) [2]. The specifications of the front-end cluster are 5.1.

The CUDA Code was launched on only two CPUs, a laptop with a eight core intel i7-3630QM and a high-end CPU Xeon Phi 7120p from the cluster. In addition the Xeon

Processor	Number	Cores	RAM
Server Dell Intel Xeon E5620 2.4 GHz	1	8	12 GB
Server HP Proliant SL 350s Gen3 Intel Xeon 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
CPU Xeon Phi 5110p	1	8	8 GB
CPU Xeon Phi 7120p	1	8	16 GB

TABLE 5.1: *CPU specifications*

Phi was used for all the experiments for the Cluster's GPUs. The Xeon Phi 720p is capable of achieving f 1.2 teraflops of double precision floating point instructions with 352 GB/sec memory bandwidth at 300 W. The code was executed on laptop to show the performance comparison between a lightweight GPU and a server based GPU.

When accessing "Piritakua" remotely 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 external GeForce GTX 670m, located on a laptop. The "m" stands for the mobil graphic cards. In addition the 670m card is design for less power usage, but with high graphics power, it even has more cores than some Tesla models, However, this types of cards has way more less Bandwidth than standard versions. The 670m card was used as comparison between laptop GPUs and high-end desktop/servers GPUs.

5.1.1 Architecture Differences

Architecture dependent technical differences of NVIDIA GPUs. During the CUDA development a lot of internal features have been improved, but most paradigms for the programmer stayed the same. For example a streaming processor can now handle 2048 threads at a time, but the maximum block size stayed at 1024. This results in a 100%

theoretical occupancy for block sizes of 1024 compared to 66% of Fermi. 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 [9].

There are two GPU architectures that CUDA 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 on the Fermi architecture. The Kepler architecture newer than the Fermi. More information about the architectures in the table 5.3. The Maxwell architecture wasn't uses for benchmarking test. However, it is showed for future reference and analysis.

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 CUDA code was launched on each one of "Piritakua"'s GPUs. As we know the supercomputer has different GPUs, as well as several different architectures. Furthermore the initial results in time execution of the implementation can been seen in figure 5.1.

Using the NVIDIA's Visual Profiler we obtain kernel metrics of the Tesla K20m. The output is organized by kernel performance throughout the simulation. As we can notice the laplacian evaluation; *glaplacinay*, *gLaplacianx* and *gLaplacianY Boundaries* uses up-to 44.37% of the overall simulation. The *gsolution* kernels which solves Zhang and Li model 3.2 uses 14.04%. The terms calculation for the RK4 integration only uses 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 optimization focus on giving the GPUs as much work as possible, using at the fullest the GPU hardware capabilities. In addition reducing the performance time of most time consuming kernels in the table 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	$gLaplacianYBoundaries$
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 executing and time int the Tesla K20*

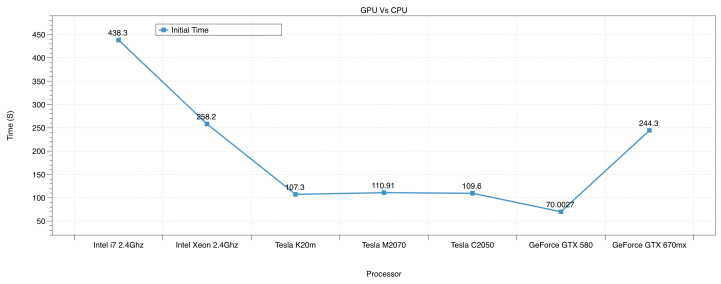

FIGURE 5.1: *Initial results of the implementation running on several different GPU nodes*

As we can see in figure 5.1 the GeForce GTX 580 is the card with the least amount of time execution, and the Tesla K20m the fastest amongst the Tesla Cards.

———include some initial performance metrics with the profiler

The following sections are the optimization techniques or methods applied to the application. In addition demonstrating the comparison between the initial implementation and the modified version. Five versions of the code are describe; Branching, Occupancy, Concurrent Kernels, Shared Memory and Structure of Arrays. Branching refers how the threads are executing and their executing flow in the application. Occupancy number of threads per devices being used. Concurrent Kernels execution several kernels at once. Shared Memory, using as much shared memory as possible. Finally Structure of Arrays modifying the memory allocation in the device.

5.2.1 Branching

CUDA follows the Single Instruction Multiple Thread architecture. This means that there are running several threads executing the same code. Each thread can operate on its own data and has its own address counter . They are free to use each data dependent path. But also each thread is executing the same operation at the same time. When a thread within a warp branches differently the other threads get deactivated[9]. This can be described by the following code 5.1 and the illustration 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: *Threads Branching*

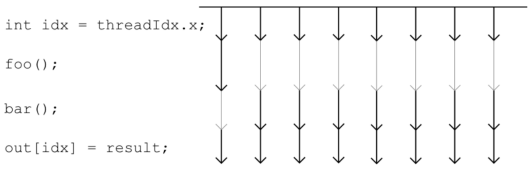

FIGURE 5.2: *The execution flow of a branching code, with warp size 8. Black arrows are active threads, and the grey ones are disabled.*

The branching problem occurred in the section where boundary condition for laplacian was being analyzed 5.2. Only a single kernel was used to checking bounding. In addition a bottleneck occur. The implementation gets the job done. However, a minor part of the threads are working, which is a waste of computation resources.

```
__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 is to include more work on the laplacian boundaries calculation 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*

This technique was applied to all parts of the code, eliminated inactive threads that were not performing computational operations to active ones. The results of the modified version consult the Optimization Results section.

5.2.2 Concurrent Kernels

Initially each kernel was launched in the default steam zero. The figure 5.3 illustrates such result using the NVIDIA's Visual Profiler. Each kernel that is being launched cannot run simultaneously because the next kernel launches needs to compute data, in other words the kernels are not independent from each other.

FIGURE 5.3: *Kernels running in the default 0 Stream.*

Kernels by default cannot run in parallel with others kernels. Furthermore CUDA doesn't provide an automatic parallel kernel executing. In addition the programmer needs to tell the CUDA compiler that some portion of the code or kernel can run in parallel. However, the compiler cannot always execute concurrent kernels, 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 *gsolution* 5.4 kernel computes the Zhang an Li model for x, y, z coordinates, which uses extensively the global memory of the device. To achieve concurrent kernels the streams need to access memory blocks that are pinned to a stream. So each memory block corresponding x, y, z are mapped to 3 streams. Furthermore, all the matrices corresponding to the coordinate x are mapped to the stream 1, y to stream 2 and z to stream 3.

```
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 and Li model in a single kernel.*

The CUDA code 5.4 is divided into a single kernel 5.5. In addition this new generic kernel can be launch in parallel. Instead of running one big kernel, three individual kernels are launched simultaneous. Dividing each kernel is possible to implement shared memory though each kernel which otherwise wasn't possible.

```
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 and Li model*

This same method is applied to every kernel that can be separated into three kernels calls. Some kernels cannot be separate such as the cross product, because the product uses pinned memory block from the other streams. The figure 5.4 shows the results of concurrent kernels in the Tesla K20.

```
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 and Li model.*

Concurrent kernels demonstrates a very promising technique to achieve a huge performance increment in application. However, there are some downsides to this implementation; correctly synchronize kernels, waiting time and hardware resources [18]. The

s

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

timeline of the application **??** illustrates the waiting time between kernels 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, appears approximate 45,00 times. Furthermore, currently branching the kernel execution process can eliminate this issue.

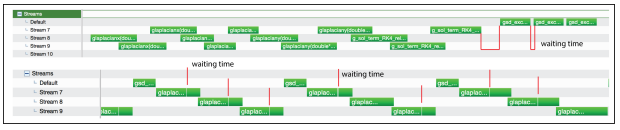

FIGURE 5.5: *Waiting time between each concurrent kernel execution*

5.2.3 Shared Memory

As we know shared memory is faster than global memory, However, shared memory is very limited. To be able to implement shared memory in the kernels, we needed the kernels separated in their x, y and z coordinate as mentioned in the previous section. In addition this allows us to implement shared memory across each kernel, otherwise wouldn't be possible.

The idea behind shared memory is to reduce the amount of global memory calls, which has about 400-600 clock cycles, while the shared memory only 1-32 cycles 2.4. The share memory implementation is accomplish by allocating the data from the thread block into a temporary array, which is the shared memory.In addition the kernel can make calculation on this temporary array and write the values onto the global memory. The code of such implementation is illustrated in 5.7. As we know there is no guaranty that threads will execute at the same order. Using *__syncthreads*() will wait until all threads have completed there task, in this case loading global memory into shared memory. Then we performed operations on the shared memory, and finally save the values in the global memory. Shared memory was used in all the occasions were kernels used extensively 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 which is located near the value of interest. In this case in region of 4x4. The figure 5.6 illustrates which part the block is for allocating shared memory and global memory. The global memory is used for the boundary conditions of the block, while the shared memory for all the values inside the block.

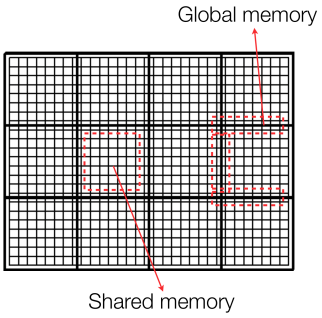

FIGURE 5.6: *Shared Memory Strategy for Laplacian evaluation*

The code 5.8 demonstrate how calculate the Laplacian with the equation 3.3 with the usage of shared memory. As we can see first we load all the global memory onto a a temporary array, the shared memory. The we performed the calculation on the shared memory as mentioned before.

```
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*

This technique look very promising and reducing global memory. However, great amount of time is spent on loading data onto the shared memory array, The results of such implementation is 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 two different ways of laying out your data in memory. This is illustrated in figure 5.7 and 5.9 respectively. AOS, grouping properties of an object together and making an array of those objects in memory, whereas a structure of arrays would be a single structure in which you make an array for each property. The structure of arrays can allow for better cache utilization, easier to access continues data, making better use of each read you make from memory, providing a more effective route to memory.

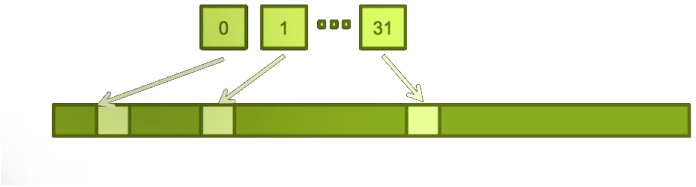

FIGURE 5.7: *AOS memory layout*

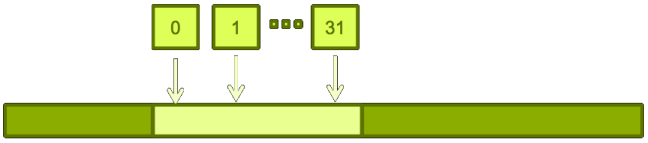

FIGURE 5.8: *SAO memory layout*

The initial implementation the x, y, z data was allocated in separated blocks. Furthermore when accessing blocks of the the same coordinates, the register access the data as the figure5.7.

```
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 this issue a custom class GPUMatrix was programmed, which contained all the matrices for the device. Moreover, the classes allocated the data for each Matrix and free the memory automatically when the simulation is over. The was allocated in a form that easier for the device to access common elements, for example when doing operations only on the x coordinate, the kernel physically access matrices that are near by.

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

LISTING 5.10: *SOA implementation*

5.2.5 Occupancy

Firstly, we increase the use of constant memory in the device, this helps redundant evaluations of variables which speedups the operations per clock cycle in the device. In addition constant memory changes can be appreciate in the code 5.11. Also matrix calculation for the boundary conditions 3.7 and 3.7 were implemented using constant Memory.

```
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*

——include occupancy results using The Visual Profiler.

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. The table 5.5 illustrate the different threads per block configuration on the GeForce GTX 580. The initial configuration for the Fermin 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 Fermin 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.

Threads per Block	Shared Memory	speedup	time
8x8	8x8	7.217x	52318.3
16x16	8x8	7.625x	49517.3
16x16	16x16	7.978x	47329.2
32x32	16x16	7.356x	51333.4
32x32	32x32	Failed	

TABLE 5.5: *Threads per block configuration on the Fermin architecture*

5.3 Optimization Results

This section is the overview of the optimization results compared with each version of the code. The figure 5.6 and 5.7 illustrates the the time execution and the speedup respectively. The Final version of the code is the Occupancy. Moreover the greatest performance occurred on the GeForce GTX 580 Card with a 8.0x speedup.

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*

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	1.662	1.349x	2.084	2.163x

TABLE 5.7: *Speedup performance*

The initial Visual Profiler test 5.1 illustrated us that the laplacian evaluating obtain about half of the overall simulation time. However, in the final optimization 5.10 the laplacian was reduced from 44.37% to 26.24 % execution time. The same occur for the Runge and Kutta term evaluation. Furthermore, the $gsdExchangeFull$ incremented from 13.37%. 23.35%, which is not necessary good. The increment is due to shift in stream operators, the $gsdExchangeFull$ is being process in the default stream cero, while the others on stream one through three.

—include final thread occupancy in the table.

The Tesla K20m was the only GPU which in every code modification it did not lose performance over the course of the optimization process. However, the other GPUs

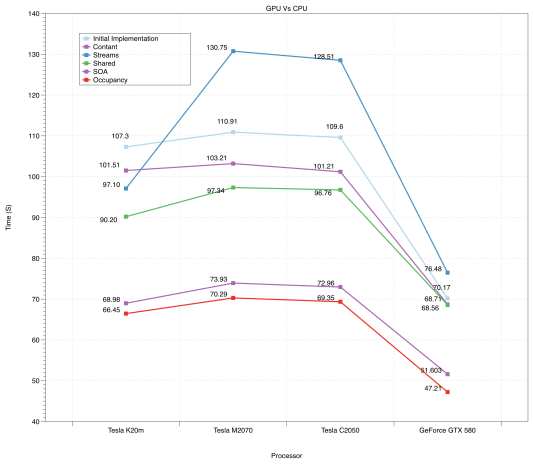

FIGURE 5.9: *SAO memory layout*

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 5.10: *Final optimization results using NVIDIA's profiler, on the Tesla K20m*

drop performance in the stream optimization stage. This 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 enable room to implement shared memory across the kernels. As the figure 5.2 illustrates the Tesla K20m is the only GPU card with CC of 3.5. This is important to mention, because the card has access to Hyper-Q. moreover this improvement can synchronize automatically the kernels for them to run in a concurrent manner.

The SOA optimization overall improved dramatically the performance of the application, obtaining a 1.2x - 2.0x speed up in all GPU cards 5.11.

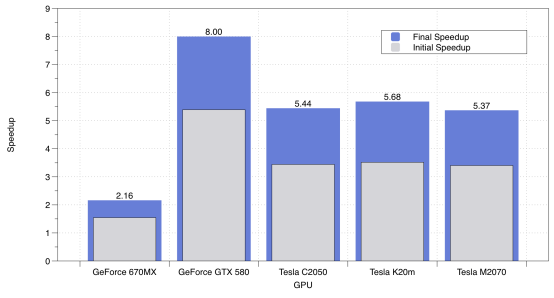

FIGURE 5.11: *SAO memory layout*

We could expected the newest card Tesla k20m would obtain the most speedup and lest time overall, certainly because it has more CUDA cores, the highest compute capabilities. Furthermore, if falls behind the GeForce 580 with about 2.5x speedup difference. However, the Tesla K20 only had a difference of 0.3x speedup compared with the Teslas Cards. The GeForce 5.2 compared with the others GPUs specifications has most Processor clock (GHz), which can make more mathematical calculations per cycle.

Finally, various techniques and practices from chapter 2 we used to archive speedup and an increase performance. We incremented the used 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 occur in the newest NVIDIA card, the K20m, which is the most expensive of all the GPUs. The actual improved occur in the GeForce 580 (the more GHz of all GPUs) with a 2.32x speed up difference.

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 do the 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 that do push the barrier of what is computationally feasible, from speedups of 1.5x to 20x using GPUs[17].

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, does raise 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, is the overall direction of the industry towards higher parallelism, as single threaded performance has reached a local limit, 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 is adopting multi-threaded CPUs and concurrent GPU type algorithms.

Spintronics. In particular we are involved in designing new magnetic materials for spin-devices and modeling and understanding of spin-transport at molecular and atomic scale. By computer simulation is possible to predict their output. Furthermore prove the theoretical experiments with high precision on relative inexpensive computers. By

using the highly parallel capabilities of the GPU it was possible dramatically reduce the computation time from around 400s on the CPU to 41s on a GPU.

Through the optimization we achieved a maximum speedup of 8.0x on not the newest device, the K20m. However, on the GPU with more clock cycles (GHz), the GeForce GTX 580. I believe that this result is very important. Because the newest device, the K20m is 10 times more expensive than the 580. The optimization was focused on giving more workload on the GPUs, this is more performance per SM and threads per block. Which did not work well on the Tesla card, but it did on the GeForce Cards. In addition the 580 is not the newest GeForce card available in the market. The GeForce 980 is expected to have a 40% increase in performance over the 500 series.

The simulation experiment was performed on a relative small data set. However, it is possible to increase the data set, in other word the magnetization data. To archived the same level of performance but with a longer overall computing time. Future test on the magnetization data can be performed on newer GeForce cards and Teslas, such as the GeForce 980 or 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:

3D Memory

Stacks DRAM chips into dense modules with wide interfaces, and 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 result: 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, allowing 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, theyll put the power of GPUs into more compact form factors than ever before.

Mobile and embedded Devices

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

Cloud Computing

NVIDIA is pushing the limits of bring computer graphics to the cloud, with the idea of everybody can access high quality graphics though the network with any kind of device. We can expect a beta test for developers mid 2015.

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

Bibliography

[1] 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.

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

[3] 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.

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

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

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

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

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

[9] T. H ö rmann. Gpu-optimised implementation of high-dimensional tensor applications. Master's thesis, Institut für Informatik, Technische Universität München, Dec. 2014.

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

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

[12] 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.

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

[14] 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.

[15] Z. Li and S. Zhang. Domain-wall dynamics and spin-wave excitations with spin-transfer torques. *Phys. Rev. Lett.*, 92:207203, May 2004.

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

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

[18] nVidia. *CUDA C Best Practices Guide*, Oct. 2014.

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

[20] N. L. Oak Ridge. Titan. https://www.olcf.ornl.gov/titan/, 2013, Cited January 2015.

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

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

[23] W. H. Press, 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.

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

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

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

[27] J. B. Schneider. Understanding the finite-difference time-domain method, www.eecs.wsu.edu/ schneidj/ufdtd, 2010.

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

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

[30] 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.

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

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

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

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

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