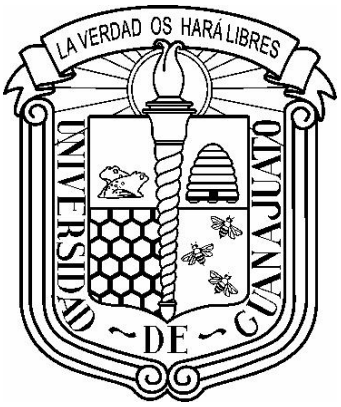

University of Guanajuato

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

January 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 13x 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

List of Figures

Introduction

Commodity graphics processing units (GPUs) are becoming increasingly popular to accelerate scientific applications due to their low cost and potential for high performance when compared with central processing units (CPUs). A large number of contemporary problems and scientific research are being 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 second chapter focus on the use of techniques of heterogeneous computing to gain more performance out the GPUs when applying to a specific task. Also the necessary means how to test the speed-up against the CPU. 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 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). Typically GPUs are know for 3D graphics rendering and video games, 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. This was the beginning of GPU parallel capabilities.

At first the GPUs where only used for general-purpose computing like computer graphics, but in-till resent years the GPU has been used to accelerate scientific research, analytics, engineering, robotics and consumer applications.(GPGPU)[8].

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 examples are Euler Solver 16x seed-up (add Reference seed-up).

In any case, for a given simulation a compromise between speed and accuracy is always made. The current tendency of the CPU relies on increases the clock seeped and adding more cores per unit and be able to work and a parallel manner, because of the there are some limitations[15]

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.

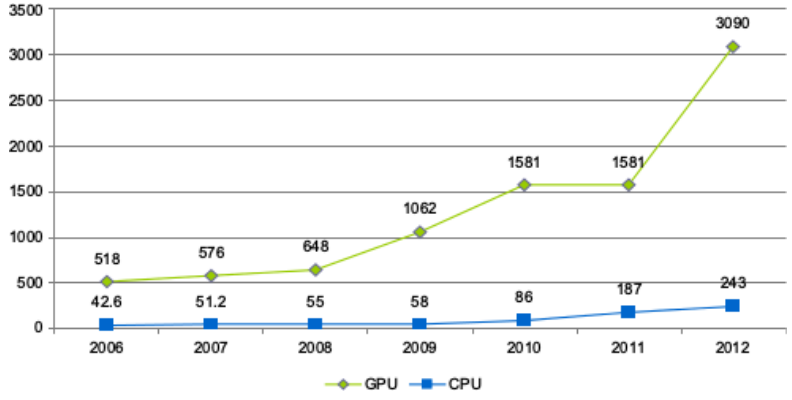

FIGURE 1.1: GPU and CPU peak performance in gigaflops

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 not by the time calculations done 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). [15].

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

The current paradigm of using CPUs for computing growth is unsustainable. the largest supercomputers use around 10 megawatts (MWs) of power, 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 MWs of power, this would require a nuclear power reactor to run it! [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, 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 multi-thread CPU core. On a typical CPU has two, four, six or eight cores. On a GPU as many as N SM core. 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). [4]

As commonly named CUDA cores are the number of procesors in a single NVIDIA GPU chip. For example one of the first GPU capabile 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 sequantial 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. In figure 1.3 different levels of memory can be observer between the host and the device, which differs form the CPU [9].

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 choosed double precision (DP), normally there is a penalty of 2x - 4x seed-up. [22] Libraries such as CUBLAS and CUFFT provides useful information how NVIDIA handles floating point operations under the hood.

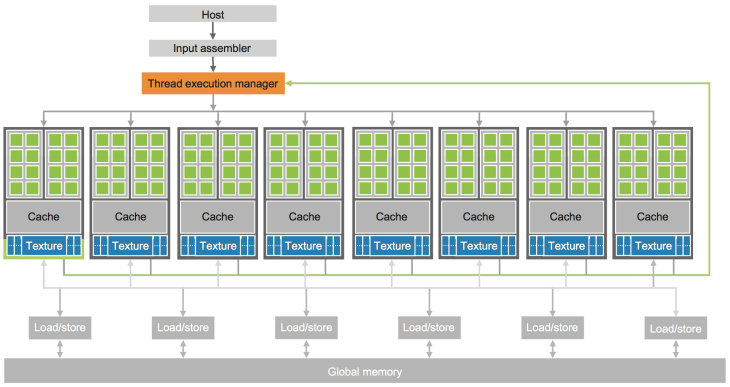

FIGURE 1.3: Memory transfer between the host and device

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 cars. However CUDA has more robust debugging and profiling for GPGPU computing. This two frameworks are developed to be close to the hardware layer, using C programming language. 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.[9]

The CUDA programming model views the GPU as an accelerator processor which calls parallel programs throughout all the SMI. This programs are only called on the device and are called kernels, which launch a large amounts of threads to execute CUDA code. The basic idea of programming on a GPU is simple. We can observer this in the figure 1.4

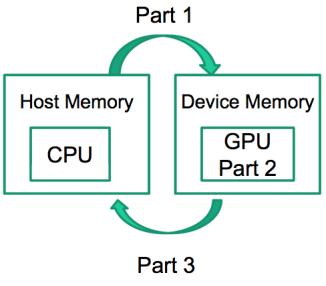

FIGURE 1.4: Programming Cycle between the CPU and GPU

- 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. The CUDA C compiler is called NVCC. The host code is the standard low level ANSI C language. The device code is marked is CUDA keywords for identifying data-parallel functions and has a extension file .cu.

When a kernel is launched, executed by a large amount of threads, where they are organized as a one, two or three dimensional grid of thread blocks. A thread is the simplest executing process. It consists of the code of the program, the particular point where the code is being executed. [9]. 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. The threads per block can be observer in figure 1.5. All the threads in a kernel can access the global memory, figure 1.3.

Each of the threads can be access by implicit variable that identifies its position within the thread block and its grid. In a case of 1-D block. [19]

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

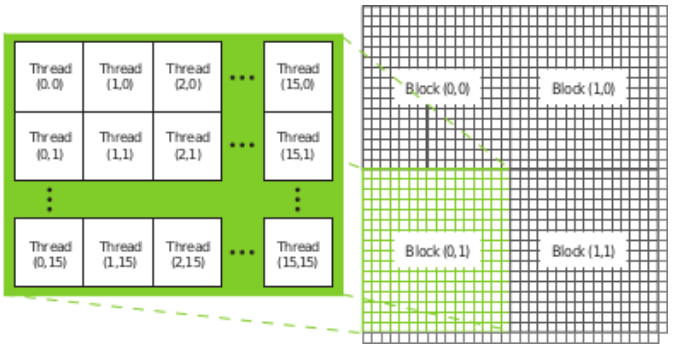

FIGURE 1.5: Part of a 2D CUDA's thread grid, divided in blocks, each block with its own respective threads.

In CUDA. host and device have separate memory spaces. This can bee seen on the host and device with the DRAM(Dynamic random-access memory) data. For example

a NVIDIA GTX 660m comes with 2GB of memory, which is the global memory for the device. As told the host and device allocates data. The programmer needs to send data from the host memory to the device's global memory. We can see this in the figure **??**. Once the memory is transfer back to the host, is completely necessary to free the memory from the device and host. This is typically done with free or delete on C/C++. The CUDA's Application Programming Interface (API) functions performs this activities on behalf of the programmer. [9]

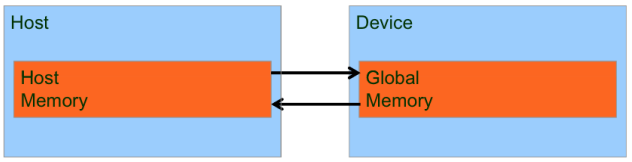

FIGURE 1.6: Separate memory spaces for the CPU and GPU

1.3.1 Vector Addition Example

A simple example of a vector addition to show the comparison between the GPU and CPU, input, two list of number which is sum up each corresponded element to produce a final output with the addition of both list. Figure 1.7 shows this process. [19]

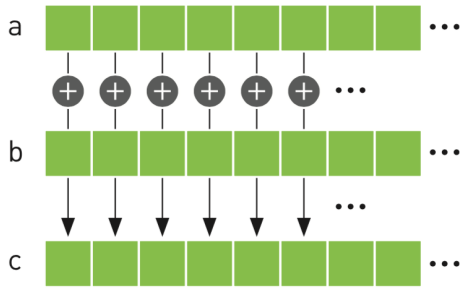

FIGURE 1.7: Simple Vector Addition Example

1.3.1.1 CPU Code

This first example illustrates the CPU code executed in a single thread. The code is straight forward to understand. First create the memory for each array, A, B and C with size N. Then calculate the sum of the two vectors with the function *add*. As we can see in the function *add*, we use the while loop to go through each element of the arrays A and B, which are added into a single array C.

```
#include <iostream>

#define N 100

void add( int *a, int *b, int *c );

int main()
{
    int A[N], B[N], C[N];

    //fill the arrays with values
    for(int i = 0; i < N; i++){
        A[i] = 1;
        B[i] = i;
    }

    add(A, B, C);

    //Display the results
    for (int i = 0; i < N; i++) {
        std::cout << A[i] << ", " << B[i] << ", " << C[i] << std::endl;
    }

    return 0;
}

void add( int * A, int * B, int * C )
{
    int index = 0;

    //go through each index of the arrays and make the operation
    while(index < N){
        C[index] = A[index] + B[index];
        index++;
    }
}
```

Listing 1.1: CPU Vector Addition

We can notice if we set N to be a large number, the function *add* could take a large amount of time to execute. But the example only illustrates the used of the CPU as a single core, however nowadays CPUs commonly have around 4-8 cores. To be able to execute the previous code on all the cores available in the CPU, threads are needed to be implemented. But you would need reasonable amount of code and debugging to make that happen. Also is a complicated task to schedule all the threads in the CPU.

1.3.1.2 GPU Code

We can accomplish the same operation very similar in the GPU with CUDA. First create CPU and GPU memory with there corresponded code. Send the CPU memory to the device, make calculations on the highly parallel GPU, finally return the results the CPU.

```
#include <iostream>

#define N 100

// CUDA KERNEL
__global__ void add( int *a, int *b, int *c );

int main()
{
    int a[N], b[N], c[N];
    int *dev_a, *dev_b, *dev_c;

    // allocate the memory on the GPU
    cudaMalloc( (void**)&dev_a, N * sizeof(int) ) );
    cudaMalloc( (void**)&dev_b, N * sizeof(int) ) );
    cudaMalloc( (void**)&dev_c, N * sizeof(int) ) );

    //allocate the memory on the CPU
    for(int i = 0; i < N; i++){
        A[i] = 1;
        B[i] = i;
    }

    //calculate the vector addition in the GPU
    add<<<N,1>>>( dev_a, dev_b, dev_c );

    //copy back the result from the GPU to the CPU
    cudaMemcpy( c, dev_c, N * sizeof(int), cudaMemcpyDeviceToHost ) );


    //Display the results
    for (int i = 0; i < N; i++) {
        std::cout << A[i] << ", " << B[i] << ", " << C[i] << std::endl;
    }

    cudaFree( dev_a );
    cudaFree( dev_b );
    cudaFree( dev_c );

    return 0;
}
```

Listing 1.2: GPU Vector Addition

As we can see the function *cudaMalloc* and *cudaFree* are very similar to the C code functions *malloc* and *fee* for allocating memory and deleting memory.

CUDA automatically spams the threads to it correspondend block, so we only need to acces the index of the block and pass it to the index arrays. To parrallel code will stop in-till the block index reaches the number of elements of the arrays, N.

```
void add( int * A, int * B, int * C )
{
    // handle the data at this index if (tid < N)
    int index = blockIdx.x;
    if(index < N)
        c[index] = a[index] + b[index];
}
```

Listing 1.3: GPU Vector Addition

This chapter provided a quick 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. [9]

Chapter 2

Heterogeneous Performance Analysis and Practices

When working with GPUs hardware 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. This appears to the programmer as a single contiguous address space. 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. [4] Meanwhile on the GPU there are expectations, their exists different memory hierarchies which dramatically change the performance output. By allocation the optimal memory types, seed-up and increase throughput can be accomplished, but also analyzing. To ensure optimization, some analysis should be done, like comparing latency, memory hierarchies and data bandwidth between CUDA kernels, The study of the performance of the CUDA code can be done by using NVIDIA's Visual Profiler.

2.1 Practices

There are three rules to follow for creating a high performance GPGPU program.[6]

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

GPUs are plugged into the PCI Express bus of the host computer. The PCIe bus has extremely slow bandwidth compared with the GPU. This is why is important to store the data on the GPU and keep it busy. And minimize the data transfer to the host and back to the device. We can see this in the following table 2.1. Because CUDA-enable GPUs can carry out petaFLOP performance, they are fast enough to compute large amount of data. So each Kernel launch needs to use all the available resources of the GPU and avoide wasting compute cycles. 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. For example a DP vectors require 8 bytes of storage per vector element this will double the bandwidth requirement. So is important to take advantage of the memory usage, take advantage of the memory types, use less memory copies between the GPU. [6]

	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

Some practices should keep in mind to rapidly identify the portions of code where it would be beneficial for GPU acceleration. [13]

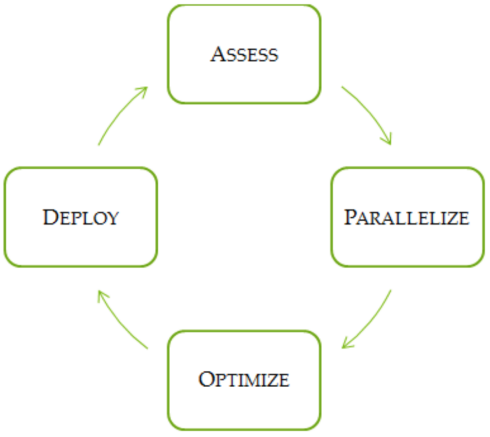

FIGURE 2.2: Different memory type and penalties usage

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: to identify the function or functions 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.[13]. CUDA also can handle timers within the GPU, and record 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. [4].

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 memory 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. [13]

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

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. [18]

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. There a several solutions, analyze the code to make more parallelize instructions, execute more computational instructions on the GPUs, analyze the number of threads per block that are executing on execute kernels .

2.3 Visual Profiler

NVIDIA's Visual Profiler will execute your application, examining special performance counters built into the GPU. The profiler will launch several times the kernels to collate precise information of the high-performance application. Also is possible to analyze memory request inside the application[19]. The Profiler can verify how long the application spends executing each kernel as well the number of blocks and threads.Through 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. The Visual profiler is very useful to indicate how much load and work is being done on the GPU, it also information about the memory throughput that can be helpful to indicate if the kernel is being actually optimized. [13]

2.3.1 Kernel Analysis

Through the NVIDIAs Profiler the kernels are invoked several times to gather optimal results. The an Once they information about how to optimize each kernel depending on several results. Also the profilers

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. [4]

2.4 Memory Handling with CUDA

In this section four types of memory handling are going to be explained, shared memory, global memory (device memory) and finally host memory. In figure 2.3 each memory type has it's bandwidth penalty of used and latency in cycles. Each one can be used in different applications to maximize the memory used. The shared Memory is very limited so it cannot be handler for all the kernels, when performed wrong on the device there is a huge latency and bandwidth penalty, instead having a gain in performance [4].

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 2.3: Different memory type and penalties usage

2.4.1 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 0GB 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 _.

[6]

2.4.2 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.[19]. 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. [4] 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 _to the kernel call invocate shared memory.

2.4.3 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. [6]. A simple analogue is the *#define* or *const* attribute in the c++ programming language, the variable performed like a variable that cannot be modified. On CUDA is excitability the same, the value can only be read and not written, the value will not change over the course of a kernel execution and only the host can write the constant memory.[19]

2.4.4 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.[19]

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.

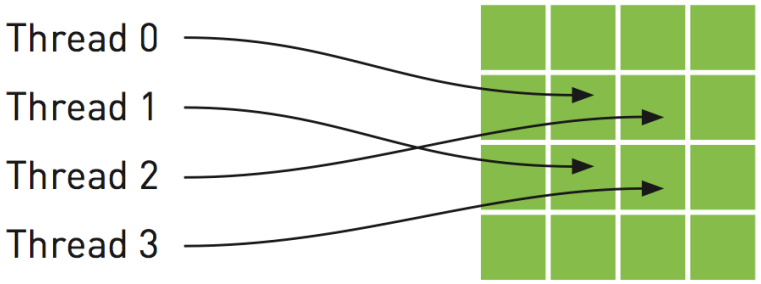

FIGURE 2.4: Mapping of threads into a two dimensional array of texture memory

- 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.

2.4.5 Thread Synchronization

This refers to synchronizing threads operations,

Will wait for all threads to finish there job.

2.5 Performance Issue

Refers..

2.5.1 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. 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. [13]

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, by this all 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 it to the CPU.

2.5.2 Thread Division

The hardware has its limits in how much thread per block a kernel can handle. Launching a kernel with the hardware constrains for above can only ensure that the kernel will actually be executed in the device, not a optimize set of threads per block. For this is necessary launch kernel with the amount of threads per block base on the hardware contains that will optimize the performance of the GPU. The impact of the block size that is choosed impacts on how much faster the code will run. By Benchmarking, is possible to find what configuration is the best for the problem. One thing to notice is that thread blocks should be a multiple number of SMs, with this idea is possible to obtain optimal thread block configuration.

Chapter 3

Introduction to Domain Wall Dynamics and a Implementation with CUDA

This chapter is a overview of the theory and experiments behind Dr. Cluadio's work "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 [3]. The implementation takes advantage of the highly parallel process capabilities of the GPU.

3.1 Theory

Contrarily to charge, spin accumulate in metals, The associated diffusion current flows in all directions, giving rise to nonlocal effects, Beyond transport properties, conduction electrons spin resonance and spin pumping provide further testimonies for non-locality in spin transport. These works all refer to samples consisting in piecewise uniform layers or blocks, magnetic or not. Of special significance to the present work in the non-collinear geometry where a spin current with polarization transverse to the magnetization exists, whose absorption in the vicinity of the surface of a magnetic layer creates a torque on the magnetization, known as spin transfer torque (SFF), [3]

Spintronics is a new type of electronics that exploits the spin degree of freedom of an electron in addition to its charge. [24]

[17]

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 geometrias include Bloch walls, Néel walls and vortex walls. 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 [7]. This can seen in figure .

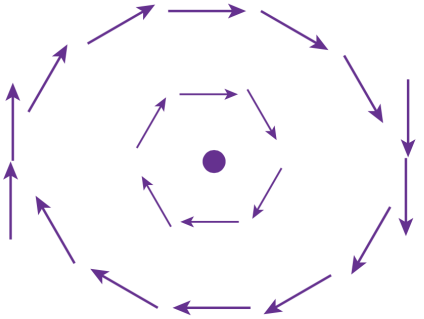

FIGURE 3.1: Domain Wall - Vortex

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. With this it is expected that electronics technology and devices will be faster, compacter and more energy-saving. A interesting application using this idea is new design for a different type memory disk drive called racetrack memory by Parkin in 2008[14]

Spin-transfer torque is a torque that exerts on a magnetization by conduction electron spins, in other words the angular momentum transferred from spins to magnetic moment [25]. 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) [21]

The study spin-diffuse effect within a continuously variable magnetization distribution, integrating with micromagenectis with diffuse model of Zhang and LI [3]

We Quantitatively test the effects of spin diffusion, on real Domain walls structures, this is done by numerically solve the Zhang-Li model [25] into micro-magnetics. The Zhang Li model refers to:

which is the following equation.

3.1.1 Experiment

Base on the work of Dr. Claudio [3]

At first we investigate the steady-sate velocity regime of DWs in NiFe soft nanostrips. applying current densities similar to those reported in experiments. The results that we are going to obtain

Experimentally measured spin-diffusion parameters are used, we want to the solution of.

$$\frac{\partial \delta \vec{m}}{\partial t} = D \triangle \delta \vec{m} + \frac{1}{\tau_{sd}} \vec{m} \times \delta \vec{m} - \frac{1}{\tau_{sf}} \delta \vec{m} - u \partial_x \vec{m} \tag{3.1}$$

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.

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.

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 magnetoresisite effects to save and read data from a magnetic disk. [21]

Some application include racetrack technology by the IBM fellow scientific Parkin [14]

Base on this study numeric applications have been unfold.

3.2 Numerical Methods

Numerical Methods [5]

The differential evaluation in one-dimension, The Second order Taylor expansion readily yields expressions for the first and seconds central derivates

$$\frac{df}{dx} = \frac{f_{i+1} - f_{i-1}}{2a}$$

and

$$\frac{d^2f}{dx^2} = \frac{f_{i+1} - 2f_i + f_{i-1}}{a^2}$$

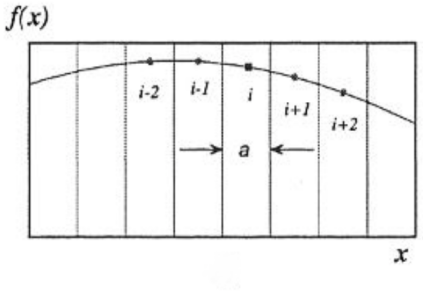

FIGURE 3.2: Sampled at regular intervals a

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

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

Applying the previous equation to nearest and next neares neigborar to grid pint i and tructation tht the 4th order yields a set of four equations:

$$\begin{bmatrix} -2a & \frac{(-2a)^2}{2!} & \frac{-(2a)^3}{3!} & \frac{(-2a)^4}{4!} \\ -a & \frac{(-a)^2}{2!} & \frac{(-a)^3}{3!} & \frac{(-a)^4}{4!} \\ a & \frac{(a)^2}{2!} & \frac{(a)^3}{3!} & \frac{(a)^4}{4!} \\ 2a & \frac{(2a)^2}{2!} & \frac{(2a)^3}{3!} & \frac{(2a)^4}{4!} \end{bmatrix} \begin{bmatrix} f_i^{(1)} \\ f_i^{(2)} \\ f_i^{(3)} \\ f_i^{(4)} \end{bmatrix} = \begin{bmatrix} f_{i-2} - f_i \\ f_{i-1} - f_i \\ f_{i+1} - f_i \\ f_{i+2} - f_i \end{bmatrix} \tag{3.2}$$

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:

$$f_i^{(1)} = \frac{f_{i-2} - 8f_{i-1} + 8f_{i+1} - f_{i+2}}{12a}$$
$$f_i^{(2)} = \frac{f_{i-2} + 16f_{i-1} - 30f_i + 16f_{i+1} - f_{i+1}}{12a^2}$$

$$\begin{bmatrix} -2a & \frac{(-2a)^2}{2!} & \frac{-(2a)^3}{3!} & \frac{(-2a)^4}{4!} \\ -a & \frac{(-a)^2}{2!} & \frac{(-a)^3}{3!} & \frac{(-a)^4}{4!} \\ a & \frac{(a)^2}{2!} & \frac{(a)^3}{3!} & \frac{(a)^4}{4!} \\ 1 & \frac{(3a)}{2} & \frac{(3a/2)^3}{3!} & \frac{(3a/2)^4}{4!} \end{bmatrix} \begin{bmatrix} f_i^{(1)} \\ f_i^{(2)} \\ f_i^{(3)} \\ f_i^{(4)} \end{bmatrix} = \begin{bmatrix} f_{i-2} - f_i \\ f_{i-1} - f_i \\ f_{i+1} - f_i \\ f_{i+2}(x_R) \end{bmatrix} \tag{3.3}$$

[5]

3.2.1 Finite Differences in the Time Domain

Modern numerical algorithms for the solution of ordinary differential equations are also base on the method of the Taylor series. Each algorithm such as 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.

3.2.2 Runge and Kutta

There exist several computational numeric methods to solver such equations, methods like Euler, Midpoint Method and Runge-Kutta integrator method can solve this type of equations. The RG4 this method is used for the simulation because its numerically more accurate when compared to the others.

The RG4 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 3.3. Euler method only has first order accuracy while the RG4 its fourth order integrator.

RK4 goes as follows:

$$y_{n+1} = y_n + 1/6K_1 + 1/3K_2 + 1/3K_3 + 1/6K_4 \tag{3.4}$$

where

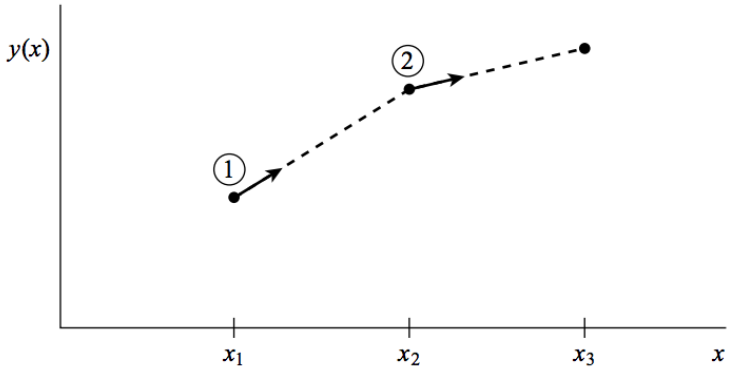

FIGURE 3.3: Euler Method, Is the simplest approximate to solver differential equation or numerically solve equations.

$$K_1 = h\dot{f}(x_n, y_n)$$
$$K_2 = h\dot{f}(x_n + h/2, y_n + k_1/2)$$
$$K_3 = h\dot{f}(x_n + h/2, y_n + k_2/2)$$
$$K_4 = h\dot{f}(x_n + h, y_n + k_3)$$

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 following equation 3.4

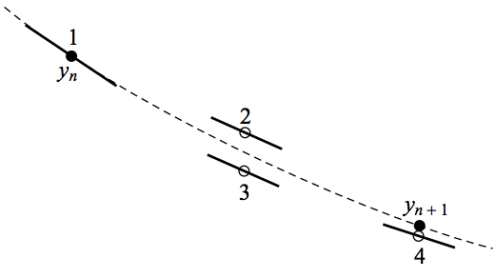

FIGURE 3.4: Fourth-order Runge and Kutta method, Each step the derivative is evaluated four times.

[16]

3.2.3 Kernels

The GPU implementation. the application reads

At initialize the applications first it allocates all the CUDA and c arrays.

To allocate a big chunk of memory in the Device

```
cudaMalloc
```

And C with

```
malloc
```

In the initialization function it also reads the magnetization data from a especific file in this specific case from "upVW-magn-2.5nm.data"

The initial values for the simulation are

NX	480
NY	120
NZ	1
TX	1200.0
TY	300.0
TZ	5.0
u	1
D	1.0eˆ3 nmˆ2/ns
tau sd	1.0eˆ-3 ns
tau sf	25.0e-3 ns

3.2.4 Threads

The number of threads that are allocated within each kernel is a 2d grid.

Threads per block X	32
Threads per block Y	32

Depending on the hardware configuration, each GPU can allocate different threads per block. To make a homogeneous grid space for each GPU a simple calculation is made.

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


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

if((int)powf(2,ceilf(logf(NZ)/logf(2))) < 1)
    NZCUDA = 1;
else
        NZCUDA = (int)powf(2,ceilf(logf(NZ)/logf(2)));

//Setup optimum number of blocks
XBLOCKS_PERGRID = (int)ceil((float)NX/(float)XTHREADS_PERBLOCK);
printf("XBLOCKS_PERGRID = %i\n",XBLOCKS_PERGRID);

YBLOCKS_PERGRID = (int)ceil((float)NY/(float)YTHREADS_PERBLOCK);
```

The calculations are divided into two parts, the CPU code and the GPU code. All the intense computation and simulation is done on the GPU, while on the CPU only minor calculations are done, such as I/O data.

3.2.5 CPU

In the $initial_calculations$ functions it calculates the terms on magnetization components

the read magnetization data. This function basically reads data from a .dat file and allocates the memory for each blocks, it reads

The file is divide into two blocks of data, the first block of 57600 rows are the coordinate X and the coordinate Y. Then the next 57600 rows by 3 columns are the magnetization data. Base on the information read the matrices of data is created.

here two data sets are created. The coordinates point data (x, y) and the magnetization data (x, y, z).

Afther this initilization data, the next step is to send this data, that is actually read on the CPU (host) to de GPU(device).

First we print the Initial and final coordinates that read, this is to ensure that the values ared sucuefully.

3.2.6 GPU

Array are created on the on the Host and sento the Device using

In the type it can be either cudaMemcpyHostToDevice or cudaMemcpyDeviceToHost, depeding, if the memory thats is being copied is sent to the host or to the device.

```
cudaMemcpy(dst, src, size_in_bytes, type);
```

after initialization the coordinates points an the magnetization data in the device are done, does values are sent to the GPU, with the function cudaMemcpy() and value set to cudaMemcpyHostToDevice.

In the Initialization of the calculations most of the arrays are filled up with values base on the data read from the .dat magenetization.

```
__global__ void gsource(double *sm_out, double *matrix_in, double u, int grid_width);

__global__ void gm_x_source(double *tempx, double *tempy, double *tempz,
                            double *mx, double *my, double *mz,
                            double *sm_x, double *sm_y, double *sm_z,
                            int grid_width);
```

The function gsource

Makes the following calculation of the $double * matrix\_in$ or m

$$out[i] = (m[i-2] - 8.0 * m[i-1] + 8.0 * m[i+1] - m[i+2]) * \frac{u}{12 * deltaX} \tag{3.5}$$

where

$$deltaX = \frac{TX}{NX}$$

This calculation is done for the arrays read from the .dat file, for dev_mx, dev_my and dev_mz and are saved in a temporary arrays dev_sm_x, dev_sm_y, and dev_sm_z.

The method.

gm_x_source calculates the coss producto of the array m_{xyx} and sm_{xyz}, this is done twice.

THis data is saved on the arrays $dev_s m_{xyz}$,

Afther launching this two kernels the initial setup is done, the next step is the actual simulation using runge and kutta integrator.

3.2.7 KG4

As seed in Runge and Kutta section, this method is implementation to numerically solve the differential equation. Intuitivelly the implementation on CUDA code is done with 4 kernls, where each kernel calculates respectivelly the order of the integrator. In the last term calculation is where all the magic occurr, the sum of the previous 3 calculated terms.

```
__global__ void gterm1_RK1( . . .);
__global__ void gterm2_RK2( . . .);
__global__ void gterm3_RK3( . . .);
__global__ void gterm4_RK4( . . .);
```

Between each term calculation of RG4 laplacian calculation kernels are launched.

```
__global__ void glaplacianx( . . . );
__global__ void glaplacianyboundaries( . . . );
__global__ void glaplaciany( . . . );
```

The final kernel is launched $voidgterm4_RK4()$ obtain the array $deltam_{xyz}$, which is the final result of the RK4 integrator. This array is sent to the last step.

3.2.8 effective values

When the rg4 integretaor is done effective values are calculatated, this values sirve the porpese of calculation the.

```
__global__ void gm_x_sm( . . . );
__global__ void gu_eff( . . . );
__global__ void gu_eff_beta_eff( . . . );
__global__ void gbeta_eff( . . . );
__global__ void gbeta_diff( . . . );
```

The last kernel $voidgbeta\_diff(...)$; is where the two final arrays are obtain, which then are sent to the CPU for the final calculation.

The final calculation is just the sum of all the elements of $beta_diff_num$ and $beta_diff_den$, there divided. This final single values tells us...

This is the final step of the simulations this is where $beta\_diff$ is obtained.

The final data is saved

3.2.9 time

When the simulation is done, it will repeat the process intil the values converges.

3.3 Validation

The validate the code, that is obtained from the simulation

Once obtain the results from the simulation, the results are written into two separated data files; .eff and .spin. depending of the configuration of the application is possible to obtain the uVW or the. Because CUDA framework is highly parallel system is farly easy to obtain errenois data from the calculations, even setting up the threads per block incorrectly is possible to get data set that a wrong, or results that don't diverge. It is necessary that when finishing making changes to the code validating the results with a valid data set is done.

The validation is done by checking the output 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.

//validation float error precision

3.4 Help Kernels

3.5 Data Flow

The initial data flow of the kernels goes as follow, Fi

Chapter 4

Optimization Results

This chapter is the results of the CUDA code implementation launched on several different GPUs nodes. The test are performed on various GPUs architectures, which, has different hardware characteristics. Each GPU node is analyzed using the NVIDIA's Visual Profiler, in addition the CUDA kernels are evaluated in performance; throughput, bandwidth, executing and parallel time. Furthermore the results, are analyzed and optimized using the schemes from chapter 3. Lastly the code is executed remotely on the supercomputer "Piritakua" of the University of Guanajuato, Campus Yuriria.

4.1 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 Campus Yuriria. The GPU cluster is located at a small town of Mexico, Yuriria. The supercomputer at the Front-end has a 8 core Intel Xeon at 2.4 Ghz, at the back-end several GPU are connected, one NVIDIA Tesla K20, two Tesla M2070 and a GTX 580.

The specifications of the front-end cluster.

Processor	Number	Cores	RAM
Servidor Dell Intel Xeon E5620 2.4 GHz	1	8	12 GB
Servidores HP Proliant SL 350s Gen3 Intel Xeon X5650 2.67 GHz	2	24	32 GB
Servidores 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

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 Phi was used for all the test for the Cluster's GPUs.

Talk about the Xeon Phi.

When accessing "Piritakua" remotely is possible to use all the GPUs available on the cluster. The specifications of the GPU connected to the front-end are as follow.

Model	Cores	RAM	DP	SP	Bandwidth
Tesla K20m	2496	5GB	1.17 Tflops	3.52 Tflops	208 GB/s
Tesla M2070	448	6GB	515 Gflops	1030 Gflops	150 GB/s
Tesla C2050	448	2.5GB	512 Gflops	1030 Gflops	144 GB/s
GeForce GTX 580	512	1.5GB	520 Gflops	1,154 Gflops	192.2 GB/s
GeForce GTX 670MX	960	3GB	520 Gflops	1,154 Gflops	67.2 GB/s

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 card version. 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.

There are two GPU architectures that code was launched, the Fermi and the Kepler. The Tesla K20m is base on "Kepler" GPU architecture and Tesla M2070, Tesla M2050 and GeForce GTX 580 on the Fermi architecture. The Kepler is a newer architecture than the Fermi. The big difference between the is the number of CUDA cores per SM.

4.1.1 Experiment detail

4.2 Results

The CUDA code was launched on each GPU of the Piritakua supercomputer. As we know the supercomputer has different GPU, as well as several architectures and different number of CUDA cores.

4.2.1 Initial Test

4.2.1.1 Visual profiler

The visual profiler.

The visual profiler was used on Laptop with GeForce GTX 670m with the intel eight core i7-3630QM.

the

4.2.2 Obtimized

4.2.3 Subsection 2

4.3 Main Section 2

Chapter 5

Conclusions and future work

5.1 Main Section 1

Appendix A

Appendix Title Here

Write your Appendix content here.

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] 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.

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

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

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

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

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

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

[9] 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.

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

[11] 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.

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

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

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

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

[16] 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.

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

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

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

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

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

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

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

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

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