UNIVERSIDAD DE MÁLAGA
ESCUELA TÉCNICA SUPERIOR DE INGENIERÍA INFORMÁTICA

GRADO EN INGENIERÍA INFORMÁTICA

SUPERCOMPUTACIÓN GRÁFICA APLICADA AL ANÁLISIS DE IMÁGENES CEREBRALES CON NIFTYREG

GRAPHICS SUPERCOMPUTING APPLIED TO BRAIN IMAGE ANALYSIS WITH NIFTYREG

Realizado por
Fco. Nurudín Álvarez González

Dirigido por
Manuel Ujaldón Martínez

Departamento de
Arquitectura de Computadores.

MÁLAGA, SEPTIEMBRE DE 2015

Fecha de defensa:
El secretario del tribunal

Resumen: El procesamiento de imágenes médicas, y especialmente de imágenes cerebrales, tiene un elevado coste computacional. Afortunadamente, técnicas tales como la programación de GPUs nos permiten economizar y liberalizar su uso. En este trabajo estudiamos NiftyReg, una librería de tratamiento de imágenes cerebrales con implementación en GPU mediante CUDA y analizamos distintas posibles formas de optimizar más aún los códigos existentes. Nos centraremos en el aprovechamiento total de la jerarquía de memoria y el uso de la capacidad computacional de la GPU. Las razones que conduzcan a los distintos cambios serán expuestas a modo de hipótesis, las cuales serán probadas de forma empírica según los resultados obtenidos con la aplicación. Finalmente, para cada conjunto de optimizaciones relacionadas estudiaremos la validez de sus resultados en términos tanto del rendimiento como en cuanto a la precisión de las imágenes resultantes.

Palabras clave: GPU, CUDA, GPGPU, NiftyReg, supercomputación, registro de imágenes, imágenes biomédicas, imágenes cerebrales

Abstract: Medical image processing in general and brain image processing in particular are computationally intensive tasks. Luckily, their use can be liberalized by means of techniques such as GPU programming. In this article we study NiftyReg, a brain image processing library with a GPU implementation using CUDA, and analyse different possible ways of further optimising the existing codes. We will focus on fully using the memory hierarchy and on exploiting the computational power of the CPU. The ideas that lead us towards the different attempts to change and optimize the code will be shown as hypotheses, which we will then test empirically using the results obtained from running the application. Finally, for each set of related optimizations we will study the validity of the obtained results in terms of both performance and the accuracy of the resulting images.

Keywords: GPU, CUDA, GPGPU, NiftyReg, supercomputing, image registration, medical imaging, brain imaging

Contents

1 Introduction

1.1 Motivation

In general, medical image processing algorithms and techniques are computationally intensive. This also applies to brain image analysis and registration. The use of GPUs to speed up the bottlenecks that happen in those processes can lead to performance improvements that help towards the liberalisation and generalisation of those techniques.

This project falls under the scope of a joint research project from the University of Málaga and a company located in the Parque Tecnológico Andaluz called Brain Dynamics granted by the Junta de Andalucía. The main motivation of the joint research project is to study, improve and accelerate the business processes of the company associated with neuroimaging and brain image processing.

The objective of our subproject is to analyze and improve the GPU implementa-

tion of a medical processing library (mostly aimed at brain image processing) known as NiftyReg. Most of our focus, then, is set on attempting to apply improvements brought by the latest GPU architectures, in order to to squeeze out performance from already existing GPU code.

1.2 Document structure and methodology

Throughout this document we will start introducing concepts from the most general to the most specific. Three introductory chapters will provide readers with some knowledge about GPUs, their evolution, how they are programmed and the nature of the GPU implementation of the library. Afterwards, two chapters will introduce the changes and optimizations performed, the reasoning behind them and the obtained results. Finally, we will draw conclusions from the set of results and summarize what has been done and what has been obtained in a succinct manner.

Results depicted throughout the document will be, unless noted otherwise, collected from an Ubuntu LTS 14.04.2 machine, using the 7.0 version of the CUDA driver. The hardware used was an NVIDIA GeForce GTX 980 (from the Maxwell generation of NVIDIA GPUs), an Intel(R) Core(TM)2 Duo CPU E8200 @ 2.66GHz processor and 4 GB of System RAM. Similarly, unless noted otherwise, all results in terms of accuracy perform in the same way as the original GPU implementation. The methodology followed to obtain the results is described below.

First, an analysis was performed on the library to study its nature. Using CMake, a multiplatform source project manager with which the library was built to work on, the study begun.

Given the sheer size of the library, the analysis was performed with different degrees of granularity. As a console application whose output is a set of files, the first step was to label the code so that the amount of kernel launches could be parsed from the console output. Upon a first inspection, a more detailed labelling was added to study the amount of time spent per function and the amount of function calls. To aid in this task, a Python program was developed to parse the timing results.

After the first analysis of the library, and considering the highlights and changes on newer GPU architectures, the focus was shifted onto GPU code that appeared to be both computationally expensive and susceptible of being improved. To ensure that the results would represent how the application really performs, and to pave the way for this document, the Python parser was extended so that it would produce reports and plots. The matplotlib library was used for this purpose.

Knowing the improvements and changes over the latest GPU generations and the code, changes were implemented if a performance gain could be expected. Changes that were expected to produce greater improvements were implemented first. When each implementation was finished, its correctness was tested with several different datasets as input by comparing the results obtained with the ones from the original GPU implementation. Comparison was performed via root-mean squared error of the resulting images. When the implementation was matured and corrected, testing was performed to study its impact in performance.

To ensure the testing was uniform, the parsing tool was extended yet again to be able to perform tests autimatically. Several executions with the same input options and data were performed, in order to make ensure the validity of the obtained results.

The results depicted in this document were all obtained using one of the larger images from the dataset provided. Bigger images performed worse in terms of execution time with respect to the CPU than their smaller sized counterparts, so we considered that working on the worst case scenario would provide more meaningful results. Upon being obtained and verified, the results were summarized and converted into plots and graphs, using the plots of the parsing tool as a base. The plots showing the obtained results throughout this document are those obtained programatically, enhanced for the shake of presentation.

The following list summarizes the technologies and tools used:

- **CMake 2.8**
- **CUDA 7.0**
- **GCC 4.8.2**
- **Python 2.7.6**
- **Matplotlib 1.4.3**

The GPGPU movement

In order to understand the principles behind GPU programming and its applications to general problems such as the one covered in this project, we must first introduce the context and evolution of GPUs as a platform. After doing so, we will describe their generalization and evolution, with overviews of several different models and architectures. Finally, some perspective will be drawn to the current state of the art and the expectations for the future.

2.1 The GPU Streaming Processor

Graphics Processing Units (GPU) were conceived as a processor dedicated to graphics, that is, a piece of hardware which frees the CPU from tasks related to graphic processing. One of the reasons for the existence of this dedicated processor is the high computational cost of these tasks, due to the large amount of data to be processed in

short time intervals.

From its inception, the CPU, based on the Von Neumann architecture, has focused on the instructions that manipulate data rather than on the data itself. Because of this, CPUs are not efficient when working on multiple data simultaneously.

The high performance offered by the GPU versus the CPU is due to a large change in the way information was handled historically, from a sequential model, to a data-centric one. In this new model, data were grouped in streams, and it was possible to perform calculations on each of their elements at the same time.

This model, as a programming paradigm, resulted in the development of a processor specialized in streams, referred to as a Streaming Processor.

2.1.1 Advantages and drawbacks

The way GPU processor-based streaming operated is what has mainly defined its advantages and drawbacks.

Its main advantage is scalability, that is, the ability to handle arbitrary workloads using the same architecture. Since this benefit is a result of the architecture itself, the improvement rate is much higher. As a result, GPU performance doubles every six months, much faster than CPUs.

However, it must be pointed out that not all applications benefit from its architecture. Naturally sequential algorithms, which are hard to parallelize, will not perform well. Neither will applications that make heavy use of selection structures in their algorithms.

2.2 From graphics to general purpose

Over the past few years, the use of GPUs to speed up codes that originally ran on a CPU has increased. This transition was mainly caused by the evolution of GPUs from their original approach (rendering graphics) to a flexible and programmable computing architecture (General Purpose GPU or GPGPU).

Despite its relative novelty, it is becoming widely accepted. This interest steems from the continuous evolution of GPUs in the context of GPGPU, as well as the results obtained when compared the CPU architectures.

Since the arrival of the first graphics processing platforms, a number of improvements have followed to build more efficient devices. In the following sections we will examine in a deeper way the most important stages of this evolution.

2.2.1 Starting point

From its inception, the GPU has executed highly parallel algorithms. However, these were initially only responsible for the different stages of the rendering process (graphics pipeline), and, as such, were fixed.

During the 90s, graphical programming gave birth to several APIs (including OpenGL and later DirectX), beginning the normalization of GPU programming. Those frameworks allowed developers to work with the GPU in a more transparent and efficient manner.

As software evolved, hardware companies also modified the graphics pipeline introducing two programmable processors called shaders (processing vertexes and fragments), making GPUs more versatile. However, these early shaders were programmed using assembler. Because of this, new tools that simplified their programming were needed in order for them to become popular.

Thus, in 2002 HLSL (High-Level Shading Language) was born as an initiative started by Microsoft. It was a language with a higher level of abstraction than GPU assemblers, but it still required the programmer to know the architecture of the target GPU.

Afterwards, in late 2002, Cg appeared (C for graphics). It was developed by NVIDIA in collaboration with Microsoft and was very similar to HLSL. The language was based on the C programming language with elements adapted to GPU architectures. Compared with HLSL, Cg had all the features of a higher level language, more functions for the programmer, and also made the code less dependent on hardware.

Finally, GLSL (OpenGL Shading Language) appeared as an alternative from the OpenGL Architecture Review Board. Also based on C, it allowed developers to make

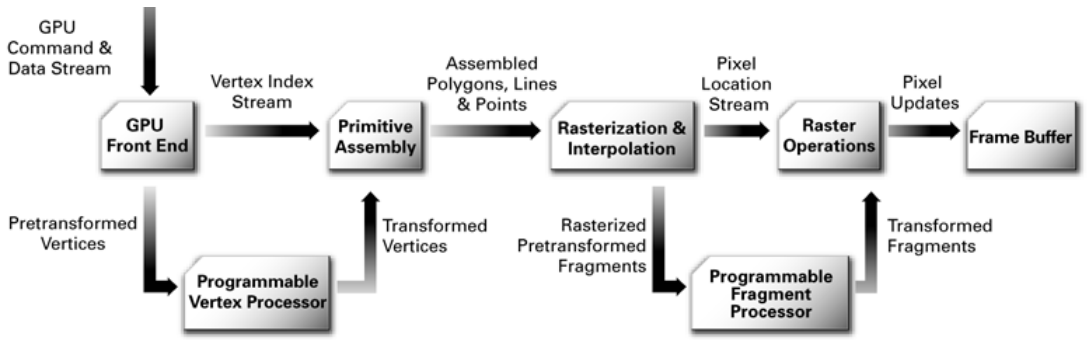

Figure 2.1: *Graphics pipeline after the inclusion of shaders.*

cross-platform applications that took advantage of most of the new features of GPUs. It was initially introduced as an extension to OpenGL 1.4, and officially included in OpenGL 2.0 in 2004.

2.2.2 Towards GPGPU: First steps

In the early 2000s, GPUs were becoming more and more programmable. However, until then they had only been used for programming graphics applications.

It was the scientific sphere that, upon gaining consciousness about the power of GPUs, attempted to solve problems for general-purpose applications. From the conventional implementation of a CPU algorithm, its GPU counterpart required a total rewriting to restructure input data, instructions and operators as the geometry of a spatial problem. That way the problem could be computed by the programmable graphics processors.

Developers had to ensure that no side effects or changes ocurred, as such computations could not happen within the graphics pipeline, which was not designed for it. These tasks required a deep knowledge of the internal architecture, paired with sufficient skills and experience.

Algorithms	**Improvement Factor**
Particle systems Physics simulations Molecular dynamics	2-3x
Database queries Data mining Reduction operations Signal processing	5-10x
Volume rendering Image processing Biocomputing	10-20x
Raytracing 3D visualization	+20x

Table 2.1: *Improvement with respect to the CPU when executing different kinds of intrinsecally parallel algorithms.*

Since 2003, codes taking advantage of GPUs performance began to fluorish. These programs made clear the difference between the CPU and the GPU, which

increased as years went by and as developers gained experience and improved their algorithms. 2.1 shows the improvements that were observed with some of those early implementations.

2.2.3 The arrival of CUDA

In 2003, a team of researchers led by Ian Buck announced the first programming model that allowed to develop on a GPU using a high level language as if it were a general purpose processor. This meant not only a higher level of abstraction and ease when developing code, but also improved performance.

NVIDIA knew their incredibly fast hardware should be accompanied by an equally cutting edge programming model, so they invited the team to join the company to start developing what came to be CUDA. Integrating both hardware and software, NVIDIA released CUDA in 2006 as the first global solution for general purpose computing on GPUs. Some of the improvements included:

- Code readability.
- Programming ease and shorter development time.
- Simpler debugging and code optimization.
- Platform-independent code.
- Complex mathematical operations and accurate results.

The CUDA computing platform provided developers with a environment based on C/C++ alongside with several extensions that allowed programmers to implement parallel applications. Overtime it also offered alternatives that gave programmers the ability to express parallelism using other high level languages (Fortran, Python ...) and open standards (such as OpenACC directives).

	2008	2015
CUDA GPUs	100.000.000	600.000.000
Supercomputers in top500.org	1	75
University courses	60	840
Scientific articles	4.000	60.000

Table 2.2: *Evolution of CUDA.*

The release of CUDA was widely accepted by scientific, academic and developer communities in general. Furthermore, NVIDIA refined the paradigm overtime and

brought a number of improvements that eliminated all the difficulties encountered. In fact, since its arrival day to today, the CUDA platform has been used in more than 600.000.000 GPUs and 60.000 research applications (see 2.2).

2.2.4 OpenCL

At the end of 2008, OpenCL was released as an open alternative to proprietary solutions for GPGPU. It was the result of many years of development by an open software consortium. Originally conceived by Apple and developed in conjunction with AMD, IBM, Intel and NVIDIA ; it was retaken by the Khronos Group and converted into an open, royalty-free standard.

Unlike CUDA, OpenCL is defined as a general purpose programming standard in heterogeneous systems that can run on different architectures, such as CPUs, GPUs and FPGAs. OpenCL provides an API for parallel computing and a programming language based on ISO C99 with extensions for data parallelism.

OpenCL operates in a way that is based on a host machine that distributes the workload between all devices in the system, which can be one or more computational units. The latter is divided into multiple processing elements.

Although OpenCL is a valid alternative to CUDA, the distance between both is sometimes tremendous. If the implementation and distribution of work is perfectly ad-

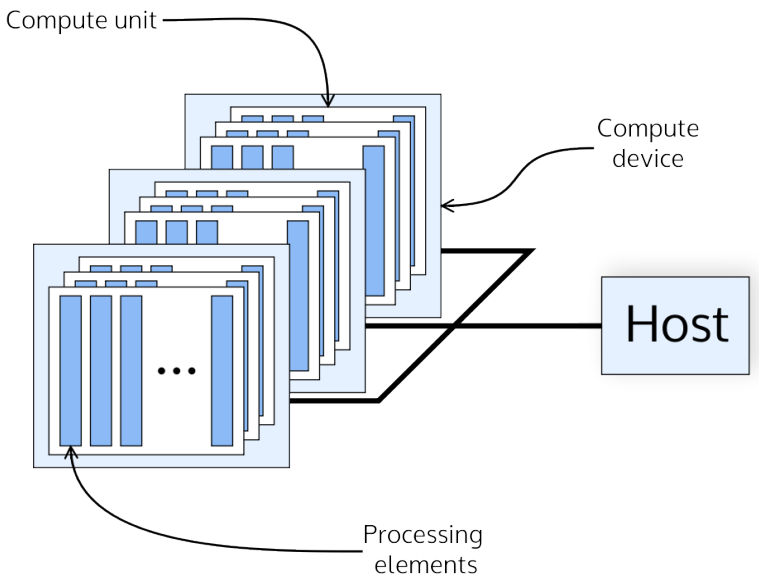

Figure 2.2: *The processing model employed by OpenGL.*

justed to the target architecture, OpenCL performance should not be much lower than that of CUDA. However, and since the main feature of OpenCL is its portability, it generally does not perform as well as CUDA does.

2.2.5 Present and future of GPGPU

GPU programming has evolved enormously and at a steady rate in recent years. However, and due to the evolutive trends of the platform, the obvious following step was to attempt to increase scalability out of the GPU itself.

As a result, computer clusters were designed with devices being interconnected, operating in groups and acting as a single graphics device. This led to a momentum gain in the field of high performance computing for GPUs and the principles of GPGPU, which have since then popularized in that scope.

The obtained improvements were not only limited to the amount of servers and workstations in existance. It also caused a raise in the number of heterogeneous supercomputers, incorporating novel GPU technology (such as GPUs from the latest generations) as coprocessors to carry out part of the computing work. 2.3 shows the evolution of graphics coprocessors in the TOP500 supercomputers list in the last four years.

Switching towards the hybrid model, making use of GPGPU techniques is relatively recent, and as such it still has a long way to go. GPUs offer performance several orders of magnitude greater than the CPU in problems that could be computationally untractable before. They are positioned as an alternative to traditional processors and, without a doubt, will continue to grow and develop in the future; cooperating further with the CPU to allow for a more efficient and complete problem solving.

	June 2011	June 2012	June 2013	June 2014
NVIDIA Fermi	12	53	31	18
NVIDIA Kepler	0	0	8	28
Intel Xeon Phi	0	1	11	21
ATI Radeon	2	2	3	3
IBM Cell	5	2	0	0
Hybrid	0	0	1	4
Total	19	58	54	74

Table 2.3: *Evolution of GPUs in TOP500.*

GPU programming using CUDA

After describing the historical context, evolution and importance of GPGPU, our focus is set on describing a current and popular framework that allows developers to use such techniques. We will introduce the CUDA model designed by NVIDIA , which encompasses both an API (Application Programming Interface) and a hardware platform, and that has been used to implement our work.

Beyond describing the computing platform and the programming model, the evolution of the architecture will also be briefly discussed.

3.1 CUDA (Compute Unified Device Architecture)

CUDA [13] is a parallel computing platform and programming model designed by NVIDIA which allows developers to access the computing power of GPUs to deploy task and

data parallelism. The model is composed of three different levels: software, firmware and hardware.

3.1.1 Software

On the software level, the CUDA model offers a set of different ways to develop applications and write code to be run on GPUs. Among them we can find:

- **Programming APIs:** Allow developers to implement GPU code in their programming language of choice. Although C/C++ are the most common high-level languages to develop CUDA applications, there exist APIs for other languages such as Fortran, Java or Python.
- **Optimized libraries:** There are several libraries prepared so that developers can make full use of them with just a few lines of code, allowing them to make use of GPU-acceleration. (cuBLAS, cuFFT, Thrust, etc.)
- **Compiler directives:** Standard compiler directives, like those of an open initiative called OpenACC, simplify code acceleration by only requiring programmers to identify code sections that can exploit data parallelism. The bulk of the parallelization effort is, thus, left to the compiler but with a performance payoff.

3.1.2 Firmware

NVIDIA offers a computing driver that is compatible with the one responsible for rendering. This driver can be controlled through simple APIs to manage CUDA devices, video memory and other components of the architecture.

3.1.3 Hardware

Finally, CUDA is implemented so that applications can be run on different compatible hardware implementations. This point is expanded upon in section 3.3.

3.2 Programming model

Next, the CUDA programming model is presented, taking C as its baseline language. It extends the C language, supplying tools that serve as an interface for parallel programming on GPUs. In this model the GPU acts like a coprocessor and only executes

a fraction of the code, while the rest is handled by the CPU. This is made transparent to the developer by to the CUDA compiler driver (NVCC) and the separation of CPU and GPU code:

1. **Device code** compiles to *PTX* [1] code files. It is decoupled from the underlying hardware, allowing the developer to ignore the details the platform.
2. **Host code** is sent to C compiler to create object files. Depending on the platform, a different compiler will be used, with GCC as an example in LINUX and CL when using Windows.

Finally, the linker produces a CPU-GPU executable. For NVCC to be able to divide the code, it is necessary to introduce new syntax elements that are used by the programmer to define kernels. Kernels are CUDA-C functions that contain code for one GPU thread only, and will be executed by all GPU threads. Context switch among them is immediate, and they tend to perform only one simple task.

3.2.1 Processing levels

First, and in order to be used from CPU code, kernels have to be declared as `__global__`. To launch a kernel, host code must include a declaration similar to `KernelToLaunch «<G, B, m, s»>`, with **G** and **B** being grid and block sizes, **m** being shared memory size and **s** being the stream to be used. We will describe these arguments more thoroughly, starting with G and B. Threads are identified within kernels as follows:

1. Threads are organized in blocks. Each thread has an identifier that is accessible within the kernel by means of the built-in variable `threadIdx`.
2. Similarly, blocks are grouped within a grid and, like threads, to each block is given a unique identifier within the kernel, `blockIdx`.

Both grid and thread blocks can be 1D, 2D or 3D, and their sizes are set by the programmer under certain constraints. Their dimensions are accessible within the kernel through the variables `blockDim` and `gridDim` respectively. This hierarchy allows CUDA code to be scalable by being able to run on any compatible hardware without a need to recompile for different target sizes.

[1] PTX is a low- level parallel thread execution virtual machine and instruction set architecture (ISA).

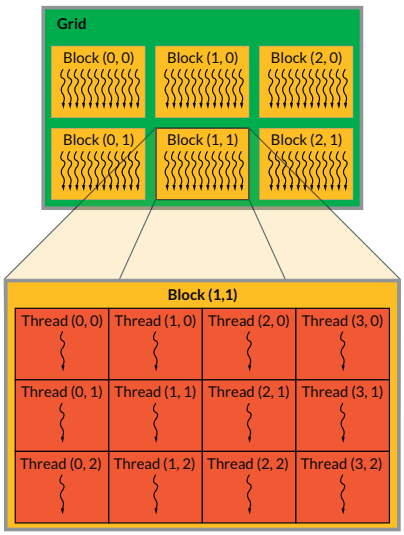

Figure 3.1: *Graphical representation of a grid with six thread blocks each one of 12 threads. NVIDIA Corporation [13]*

In addition to configurable grid and block sizes, threads are also grouped as warps, that up to this day contain 32 threads[2]. They are the atomic execution unit, and are executed in unpredictable order although they can be synchronized if required.

Warps execute one common instruction at a time for all their threads. Because of this, warp divergence caused by branching is a hurdle in terms of performance. When such divergence happens, each branch is serialized disabling threads that do not participate in the running branch. When all execution paths complete, the threads converge back. This serialization only occurs within a warp, and two differents warp are able to execute distinct paths simultaneously. On the other hand, blocks are executed in free order and they cannot be synchronized. Threads are able to comunicate only with others threads in the same block. These details must be carefully handled by the programmer in order to ensure program correctness.

3.2.2 Streams

Since the second generation of CUDA capable GPUs, concurrent execution of different kernels has been made posible by means of streams. A stream is a sequence of kernels that execute in order. Kernels in different streams will execute independently, although CUDA provides functions for synchronize them.

By default all the kernels are executed within the same stream. To create new

[2]The number of threads per warp could change on future generations of NVIDIA GPUs.

streams, CUDA C offers a new data type, `cudaStream_t`, and a new constructor, `cudaStreamCreate()`. The following code is an example of an array with two streams:

```
cudaStream_t stream[2];
for (int i = 0; i < 2; ++i)
  cudaStreamCreate(&stream[i]);
```

Kernels are assigned to a stream through the fourth parameter of the kernel launch call, **s**, we previously discussed. The amount of concurrent streams depends of the generation, (with 16 streams in Fermi and 32 in Kepler). The details of stream concurrency are explained in Section 3.4.3.2.

3.2.3 Processing flow

As already mentioned in section 3.2, in the CUDA model the GPU (device) acts as a coprocessor for the CPU (host) but with its own memory. Because of this, it is necessary to transfer data from host memory to device memory, perform the computation and bring the data back [4]. A diagram of the processing flow can be observed in Figure 3.2:

Although this scheme is still in use, future generations will simplify it by adopting an unified memory architecture for both host and device.

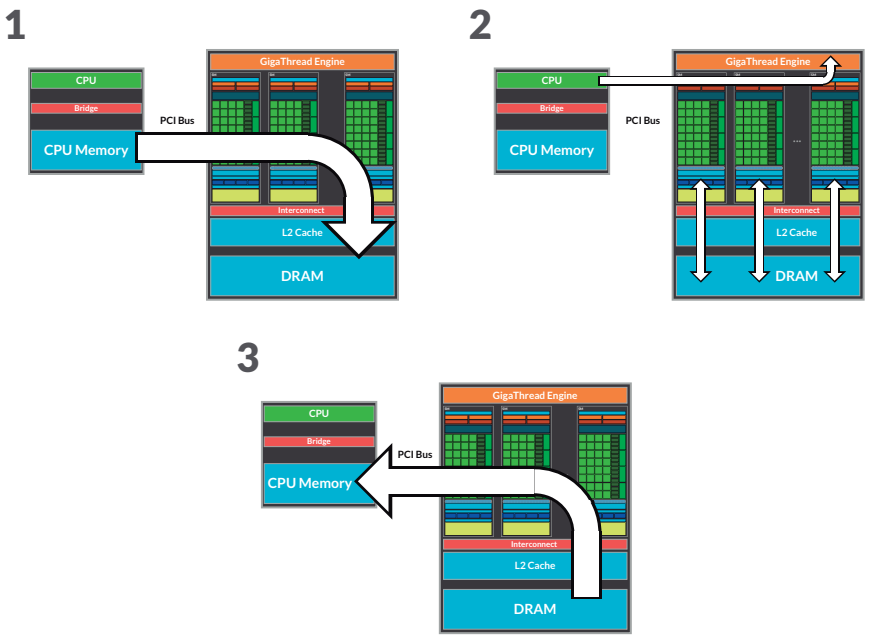

Figure 3.2: *CUDA processing flow.*

3.3 Hardware model

The massively parallel threading model is built upon the CUDA hardware model. Generation after generation, the model is expanded upon but backwards compatibility is maintained. We will discuss the modelin a general way, with the specifics being explained in section 3.4.

The NVIDIA GPU architecture is based on SIMT (Single-Instruction, Multiple-Thread) processing, similar to SIMD (Single Instruction, Multiple Data) but different in that it specifies the execution and branching behaviour of each single thread. This is achieved in hardware by an array of Streaming Multiprocessors (SMs) where each of them contain lots of CUDA cores.

In addition, the computing cores are paired with a memory hierarchy. CUDA GPUs three memory layers on-die for each multiprocessor. In order from faster to slowest:

- **Registers:** fastest of them all and implemented as register banks in each SM for the cores to perform most of their computing work.
- **Shared memory:** slightly slower than registers, shared among threads in the same SM and used as a cache memory managed by the programmer.
- **Read-only memories:** constant and texture caches, both slightly less popular and general purpose oriented.

If on-chip memory is not enough, the GPU can make use of its SGRAM global memory common to all multiprocessors. This memory, of a SGRAM (Synchronous Graphics Random Access Memories) nature, is three times faster than its CPU counterpart. However, it is still 500 times slower than shared memory.

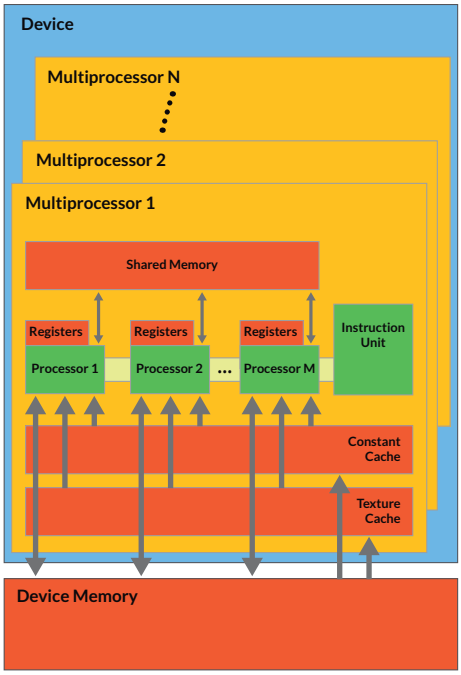

Figure 3.3: *CUDA hardware model. NVIDIA Corporation [13]*

3.4 Evolution of the architecture by generations

To identify the different architecture models, NVIDIA assigns a version number to each device generation. This number, called CUDA Compute Capability (or C.C.C.), is used by applications at runtime to determine which hardware features and/or instructions are available on the present GPU. The C.C.C. is formed by two numbers (x.y):

- x is the major version number and it determines the generation: 1 for Tesla, 2 for Fermi, 3 for Kepler and 5 for Maxwell.
- y is the minor version number and represents the incremental improvement to the core architecture.

The main features of the different generations are explained below.

3.4.1 The first generation: Tesla (G80 and GT200)

Tesla was the first CUDA capable GPU generation, launched in 2006. It unified the vertex shader with the pixel shader, and allowed them to be used for GPGPU by changing the pipeline from a lineal model to a loop pipeline.

Architecture	G80	GT200	GF100	GK110 (k20)	GK110 (k20x)	GK110 (k40x)	GK210 (k80)	GM204 (GTX 980)
C.C.C.	1.0	1.2	2.0	3.5	3.5	3.5	3.7	5.2
Launch year	2006	2008	2010	2012-13	2013	2013-14	2014	2015
TPC	8	10	4	-	-	-	-	-
SM [SM/TPC]	16 [2]	30 [3]	16 [4]	13	14	15	(2x) 15	16
Int and fp32 [cores/SM]	128 [8]	240 [8]	512 [32]	2496 [192]	2688 [192]	2880 [192]	(2x) 2880 [192]	2048[128]
Fp64 [cores/SM]	0[0]	30 [1]	256 [16]	832 [64]	896 [64]	960 [64]	(2x) 960 [64]	64 [4]
LSU [cores/SM]	0 [0]	0 [0]	256 [16]	416 [32]	448 [32]	480 [32]	(2x) 480 [32]	512 [32]
SFU [cores/SM]	32 [2]	60 [2]	64 [4]	416 [32]	448 [32]	480 [32]	(2x) 480 [32]	512 [32]
Warp Scheduler per SM	1	1	2	4	4	4	4	4
32-bit register per SM	8K	16K	32K	64k	64k	64k	128k	64k
Shared memory per SM	16KB	16KB	16KB	16KB + 32KB	16KB + 32KB	16KB + 32KB	16KB + 32KB + 48KB	96KB (2x48KB)
L1 Cache per SM	None	None	+ 48KB	+ 48KB	+ 48KB	+ 48KB	+ 80KB + 96KB + 112KB	None
L2 Cache	None	None	768KB	1.5MB	1.5MB	1.5MB	(2x) 1.5MB	2MB

Table 3.1: *Summary table with the main features of several models on each hardware generation.*

Each G80 GPU has 8 Thread Processing Clusters (TPC), which in turn have two SMs with 8 cores each. This means that there are 128 scalar processing cores, that in addition support dual-issuing of MAD and MUL operations. G80 GPUs have 8K 32bit registers and 16Kb of shared memory per SM. Figure 3.4 shows a diagram of the architecture.

Overtime, NVIDIA improved the Tesla architecture with the GT200 GPU. The main enhancements are listed below:

- **An increase in the amount of cores.** The number of TPC blocks was raised from 8 to 10, with an increase in the amount of SMs per TCP to three. Due to this, GT200 GPUs had 240 cores.
- **More threads per chip.** The software limitation on G80 only allows 768 threads per SM whereas the GT200 accepts up to 1024 threads.
- **Doubled register file size.** The register bank is doubled from the previous architecture, increasing the amount of registers to 16K per SM.
- **Double-precision floating-point support.** One core for fp64 operations is added in each SM.

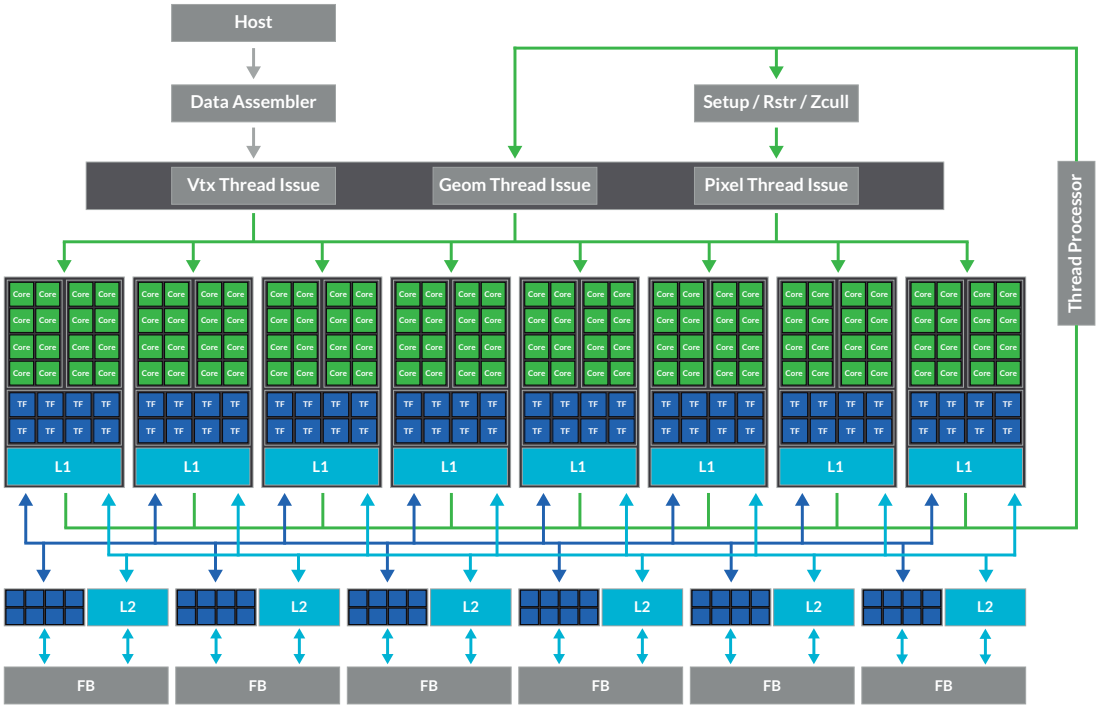

Figure 3.4: *GeForce 8800 GTX (G80) block diagram. NVIDIA Corporation [9, pg. 02]*

- **Improved shared memory.** Hardware memory access coalescing was added to improve memory access performance.

Figure 3.5 shows the three SMs in a TCP of a GT200, revealing that in this case the increase in the number of cores is produced by an increment of TCP units instead of a higher number of cores per SM.

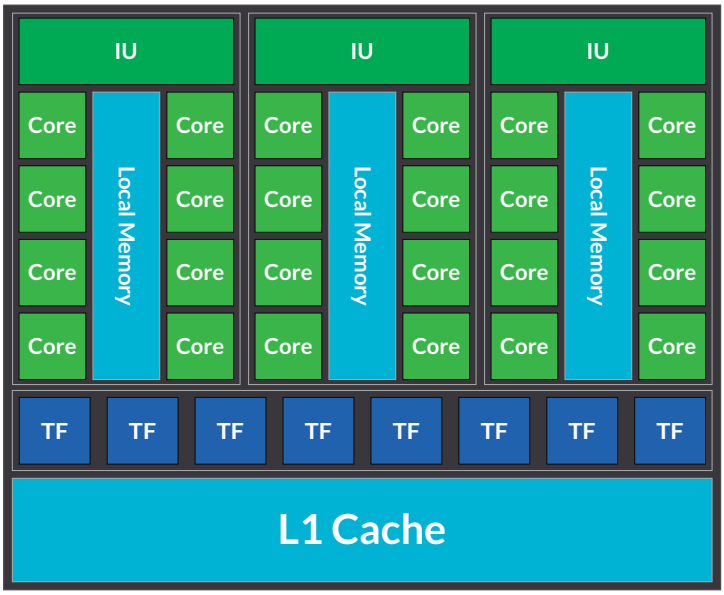

Figure 3.5: *Thread Processing Cluster of the GT200 GPU. NVIDIA Corporation [10, pg. 13]*

3.4.2 The second generation: Fermi (GF100)

With Fermi, the TCP disappears and NVIDIA makes a new hardware block, called the Graphics Processing Cluster (GPC), that encapsulates all key graphics processing units. Inside of this hardware block there are four stream multiprocessors.

In opposition to the intergenerational enhancements implemented with Tesla, with Fermi NVIDIA decided to reduce the number of SMs and increase the number of cores per multiprocessor. Thus, Fermi has three different types of cores:

1. **Integer and floating point units**. 32 cores per SM redesigned for optimized 64-bit int operation. These cores are used for both simple and double precision calculations[3].

2. **Load/Store units**. For Load/Store operations 16 cores are incorporated allowing source and destination addresses to be calculated for sixteen threads per clock.

3. **Special Functions Unit (SFU)**. Four cores are added for quick calculation of complex functions such as sin, cos, reciprocal and root (with an accuracy tradeoff)

[3]Fermi can only run 16 fp64 operations at a time.

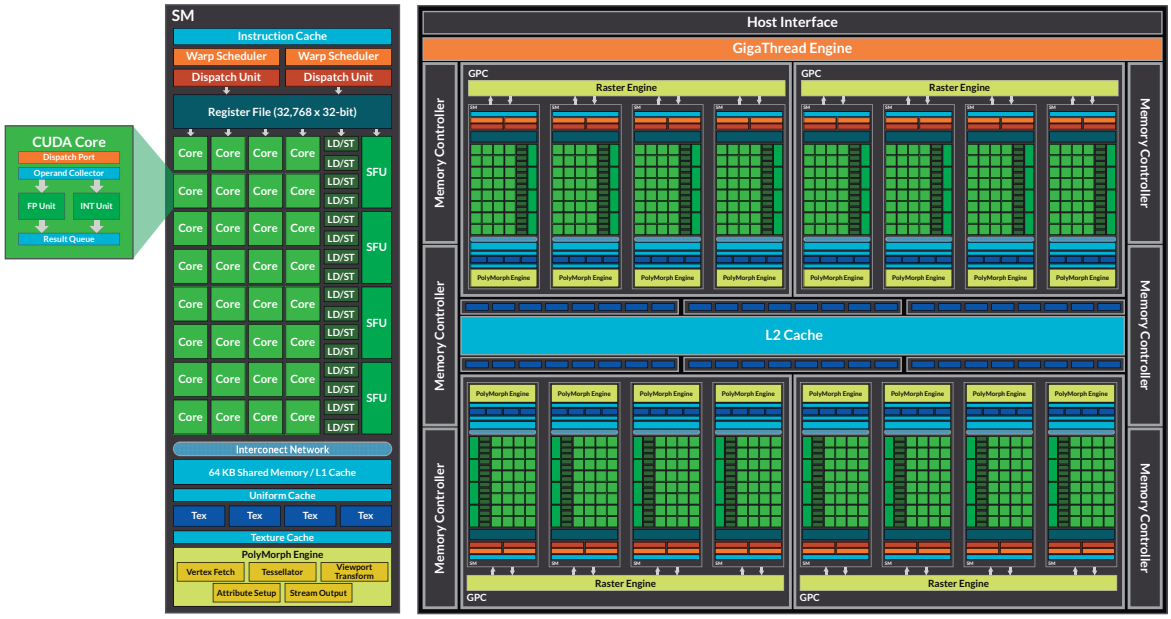

Figure 3.6: *GF100 block diagram and Stream Multiprocessor detail[11, pg. 11 and 16].*

In addition, GF100 GPUs have two warp schedulers, with an instruction dispatch unit in each of them. This configuration allows to launch two warps concurrent and independently, and due to this the schedulers do not need to check for dependencies from within the instruction stream.

One of the main improvements over the previous generation is the memory hierarchy. Each SM in Fermi has 64KB of on-die memory that it is configurable in two different modes: 16KB of shared memory and 48KB of L1 cache or vice versa. The first mode helps optimising algorithms where data addresses are not known beforehand, while the second works best mode for algorithms with well defined memory accesses. Moreover this generation incorporates 768KB of L2 cache common to all stream processors. In the left side of Figure 3.7 we can find a diagram of this hierarchy.

3.4.3 The third generation: Kepler (GK110 and GK210)

Following the trend introduced by Fermi, Kepler also increased the number of cores per SM and reduced the amount of multiprocessors. Even though the GK110 was not the first chip implementing the Kepler architecture, this section is focuses on the GK110 and beyond as they are the most widely used models of it.

The quantity of cores per SM is the same in the different incremental improvements to the architecture, although the number of stream multiprocessors changes

from one to another. Table 3.1 shows the different versions and their main features.

SMs in Kepler (called SMXs) have 192 single precision CUDA cores, and each core has fully pipelined floating-point and integer arithmetic- logic units. In addition, these SMs increase the double-precision computation capacity with 64 dedicated units. Moreover, GK110 GPUs have 32 LD/ST units per SM, doubling the amount of load and store units available in the Fermi architecture. Finally, each SM has 32 Special Function Units (SFU).

Four warp schedulers with two instruction dispatch units in each each can be found in every SMX. This allows up to eight warps to be issued and executed concurrently.

Kepler follows a memory hierarchy akin to that of Fermi, although texture memory was made accessible for GPGPU as a 48KB read only memory. Furthermore, all memory layers were improved:

- **Register Bank**. The amount of 32-bit registers per multiprocessor increased up to 64K.
- **Shared Memory and L1 cache**. In addition to the shared memory configuration modes that were seen in Section 3.4.2, a new mode is added in this generation: 32KB shared among L1 cache and shared memory.
- **L2 cache**. The amount of memory in this layer is doubled to 1536KB. In addition, the L2 cache on Kepler offers twice the bandwidth per clock available on Fermi. [12]

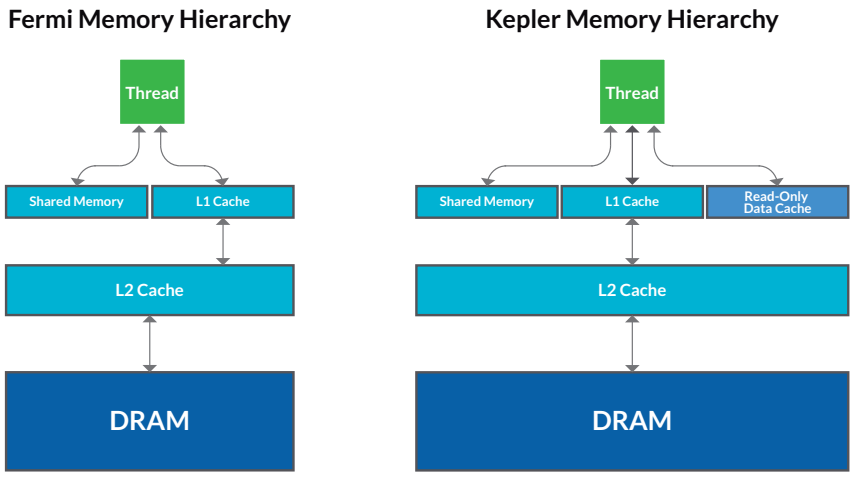

Figure 3.7: *Fermi and Kepler memory hierarchy. NVIDIA Corporation [11, pg. 19] and [12, pg. 13]*

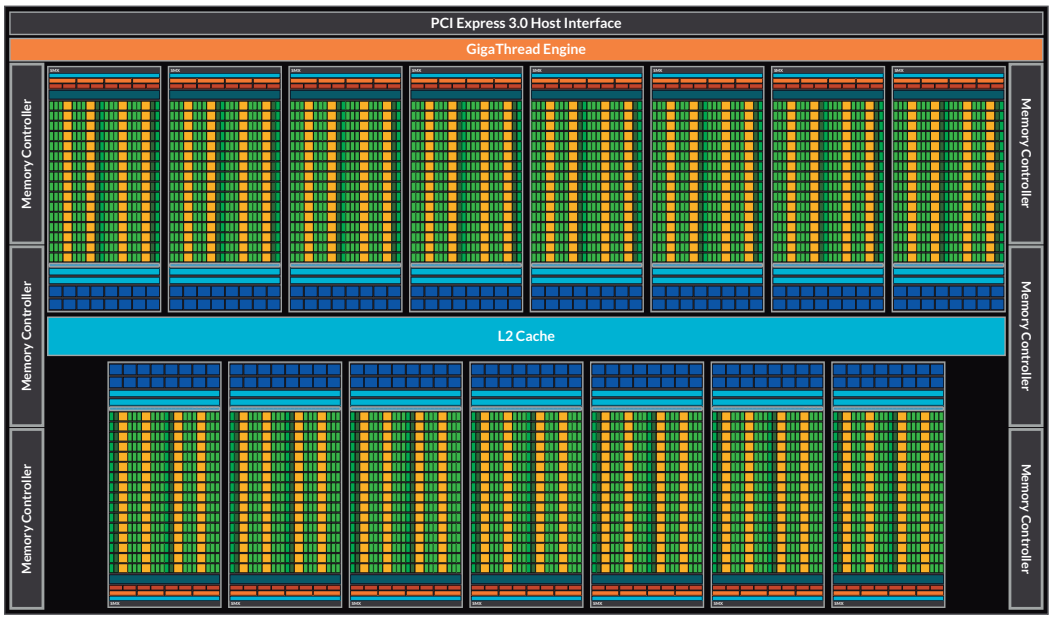

Figure 3.8: *Kepler GK110 full chip block diagram. NVIDIA Corporation [12, pg. 06]*

Both GK210 and GK110 implement the features depicted in Section 3.4.3.1 and Section 3.4.3.2. Both are models in the Kepler architecture, but GK210 GPUs have more resources on-chip than their GK110 counterparts. Thus, both chips share the same amount of cores per SMX but GK210 GPUs have 128K 32-bit registers per SMX and 128KB of shared memory/L1 cache with the available configurations shown below:

- 112KB shared memory + 16KB L1 cache
- 96KB shared memory + 32KB L1 cache
- 48KB shared memory + 80KB L1 cache
- Reversed values for the previous configurations

3.4.3.1 Dynamic Parallelism

Prior to the release of GK110 GPUs, those acted as a coprocessor for the CPU, capable of producing large speed-up factors but without any autonomy of their own. Dynamic parallelism allows the GPU generate and process new work by itself. This way, the CPU does not need to be interrupted in order to launch new kernels and create events and threads to control dependencies, synchronize the results and control task scheduling

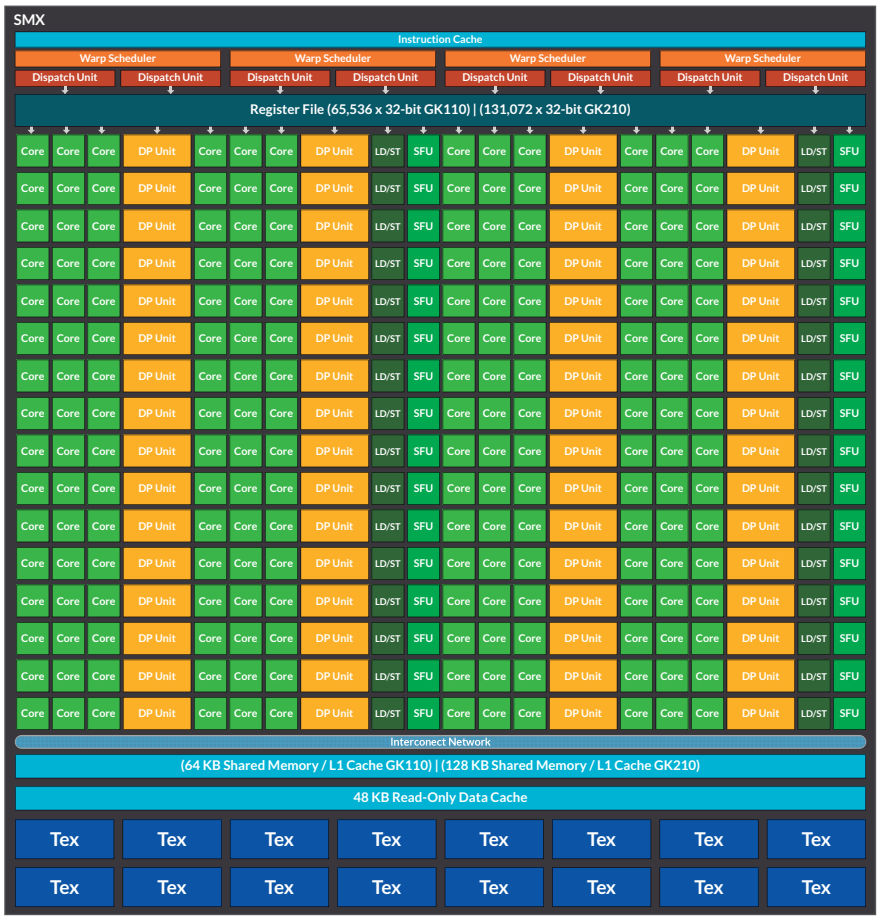

Figure 3.9: *SMX with 192 single-precision CUDA cores, 64 double-precision units, 32 SFU and 32 LD/ST units. NVIDIA Corporation [12, pg. 08]*

[1]. Figure 3.10 depicts how dynamic parallelism is performed, helping to minimize the back and forth communication between CPU and GPU and improving performance.

By means of dynamic parallelism, the programmer is able to develop recursion from within the GPU. Because of this, algorithms that were impossible to implement on Fermi GPUs, such as Quicksort, adaptative grids or variable length loops, can be implemented on Kepler.

On Fermi GPUs, only the host could send a grid to the CUDA Work Distributor (CWD), which would then distribute blocks among the different SMs. Kepler GPUs, however, include a Grid Management Unit (GMU) in charge of both device and host grids. This component processes grids from both host and device and sends them to the CWD. The work distributor, which admits up to 32 grids, then sends blocks to occupy the SMs. Furthermore, the GMU can pause the dispatching of new grids if there happens to be a two-way link. Figure 3.11 shows Fermi and Kepler workflows.

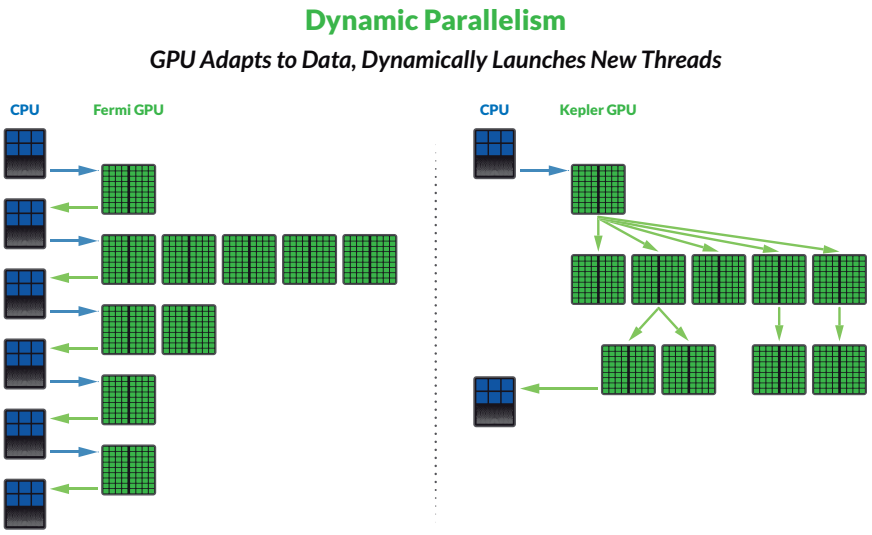

Figure 3.10: *With Dynamic Parallelism the GPU can deploy data parallel tasks by itself. NVIDIA Corporation [12, pg. 15]*

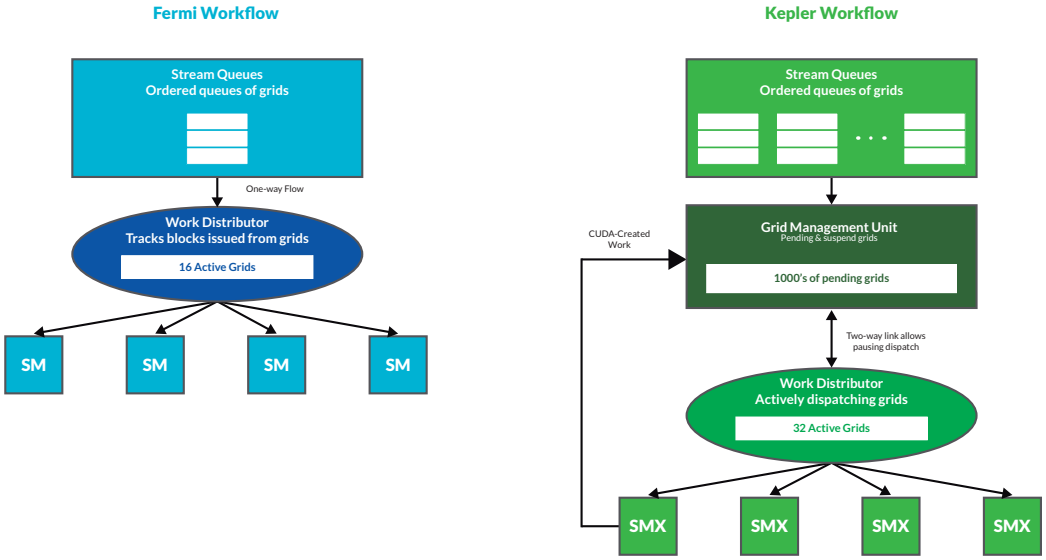

Figure 3.11: *Fermi (left side) and Kepler (right side) workflow. NVIDIA Corporation [12, pg. 19]*

3.4.3.2 Hyper-Q

On Fermi up to 16 streams could be launched concurrently, but the underlying implementation uses single queue, and thus, only the end of a stream and the start of other could be executed at the same time. On Kepler, up 32 streams can be executed concurrently, due to the fact that each stream is managed independently on a different hardware queue. Moreover, this also allows to execute streams in parallel with respect to other stream coming from the same or a different CUDA program, MPI process or POSIX thread.

3.4.4 The fourth generation: Maxwell (GM204)

Figure 3.12: *Depiction of the SMMs found in Maxwell. NVIDIA Corporation [14, pg. 8]*

Following the current trends in computer architecture, the Maxwell architecture is focused on maximising the performance per watt consumed. Thus, NVIDIA reorganized internal components of the multiprocessors (called SMMs in Maxwell), so that they are split in four parts as shown in Figure 3.12. Each processing block of CUDA cores contains:

1. **32 int and floating points units** (128 per SMM).
2. **1 double precision unit** (4 per SMM).
3. **8 Load/Store units** (32 per SMM).
4. **8 Special Functions Unit (SFU)** (32 per SMM).

In addition, each of the four splits contains a warp scheduler, which is capable of dispatching two instructions per warp on every clock cycle. This configuration aligns with warp size, making it easier to use efficiently.

3.4.4.1 Memory improvements

The memory hierarchy was modified from Kepler. Now shared memory does not share the block with the L1 cache. Instead, the L1 caching function has been moved to be shared with the texture caching function. The size of shared memory has increased to 96KB although this is limited to 48KB per thread block [7]. The size of the L2 cache is 2MB on GM204 GPUs.

Another improvement implemented on Maxwell is memory compression. To reduce DRAM bandwidth demands, Maxwell GPUs can now make use of lossless compression techniques as data is written into memory. The bandwidth savings from this compression appear when clients, such as the Texture Unit, later read the data.

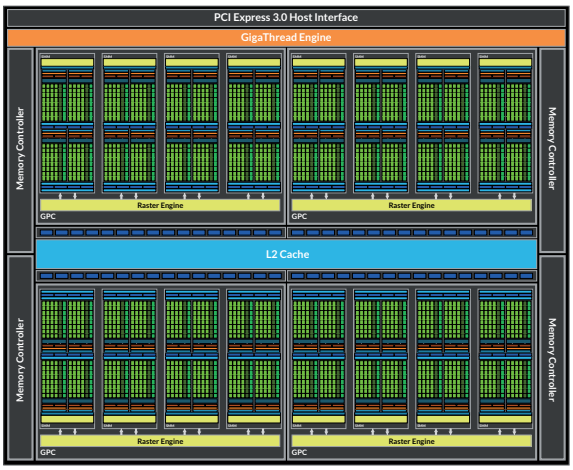

Figure 3.13: *GM204 architecture diagram. NVIDIA Corporation [14, pg. 6]*

3.4.4.2 Atomic operations

Maxwell extends atomic operations, introducing native shared memory operations for 32-bot integers and native shared memory 32 and 64-bit compare-and-swap (CAS) operations. The latter can be used to implement other atomic functions with reduced overhead when compared to Fermi and Kepler atomics. These changes allow programmers to implement list and stack data structures shared by the threads of a block in a much more efficient manner. [14]

NiftyReg and NifTK: Brain Image Processing

As the last introductory note prior to presenting our work, we introduce the medical image processing library we have worked upon. Given the nature of the platform, our description can only be given in terms of a black box point of view and is, as such, kept as brief as possible.

We will introduce the platform within which the library is contained, and then we will describe the architecture of the library from the perspective of a programmer.

4.1 Overview

NiftyReg is an open source image registration software, which can be used both as a library and as a standalone application, mainly used for brain image registration. It is composed of a series of executables and algorithms, some of which include a CUDA/GPU implementation, each of them with different sets of operations and variables. NiftyReg is contained within the scope of NifTK, a platform containing a set of applications and packages related to brain image processing.

NiftyReg and its superproject, NifTK, were developed by the Translational Imaging Group (TIG) at the University College London (UCL). The platform is composed of several modules, aimed to perform image segmentation, registration, visualization and reconstruction tasks. Some of the applications already include some degree of GPGPU/CUDA based optimization. Within NifTK we can find:

- **NiftyReg:** set of programs to perform rigid, affine and non- linear registration of medical images. Some of the algorithms include CPU and a GPU based implementations.
- **NiftySim:** a finite-element modelling package. NiftySim contains CUDA and C++ implementations of the Total Lagrangian Explicit Dynamics (TLED) algorithm and several non-linear constitutive models.
- **NiftySeg:** an open-source software package for image segmentation, bias field correction and cortical thickness estimation.
- **NiftyRec:** a package for 3D Stochastic Emission Tomographic Reconstruction.
- **NiftyView:** a graphical user interface that serves as an entry point to the above packages.

4.2 Structure of GPU-based NiftyReg

We will now go into details, describing NiftyReg in terms of its source code. In order to perform our work on the library, we first had to understand (in black box terms) how the code is structured and built. Since NiftyReg already had a set of algorithms implemented in CUDA, an analysis on such implementation had to also be performed.

The purpose of NiftyReg as a whole is to perform medical image registration with an algorithm created by the developers [8], under foundational papers that can be found in [17] and [16]. Readers interested in the image processing algorithms might find [21] of use. The implementation and execution flow follows a pipe line where registration

steps are performed on different levels, building a resulting registered image from one image used as the reference and another used as the floating image.

The code is built so that certain classes manage subalgorithms and data structures, with the CPU implementation making use of parallelism when possible. The GPU implementation extends the CPU classes and is constrained by the sequentiality of the original implementation. Since the CUDA implementation dates from the early days of the programming model, it appears logical to have kept consistency in terms of the steps and processes carried out. However, with the addition of streams and kernel parallelism, such design choices and rigid structure limits the level of achievable concurrency. Precision wise, the GPU implementation mostly makes use of single precision floating point values. This is, again, a product of its time.

The algorithm as a whole can use different approximation measurement metrics to perform the registration, and are set up either when NiftyReg is used as alibrary or as a standalone application. Our work focus, given that no particular interest was given to any of them in the context of the project, will be set on the initial, preset values and modules common to the whole implementation.

Memory optimizations on NiftyReg's GPU (CUDA) implementation

5.1 Memory bound vs. compute bound code

A practical classification for code behaviour in the context of GPU programming when performance becomes a major concern entails to distinguish between memory bound and compute bound applications [18]. A memory-bound code places the bottleneck on the memory system, mostly for memory operations to predominate versus arithmetic intensity. Compute-bound codes are the other way around. The choice of the hardware platform for the code to run and the software optimizations to be performed are both driven by that feature.

Around 75% of scientific codes are considered memory bound, where a wise

handling of the memory hierarchy has to be carried out to achieve a high performance implementation. In general, memory accesses have to be minimized, particularly those targeting slow memories.

5.2 Memory organization in NiftyReg

Firstly, and taking into account the importance of using the memory system correctly, the original GPU implementation of NiftyReg will be described. We must clarify that this implementation does not cover all the functionalities from its CPU counterpart.

In the library, memory management and organization belongs to those earlier days of the CUDA model, and as such hides a great potential to further exploit and push the capabilities of newer hardware. The way memory is managed up as:

- **Global memory arrays**: most common way of sharing and retrieving information with CUDA kernels. The allocation of those arrays is structured in a way that they can easily be updated to make use of the unified memory model.

  That model is not yet implemented in hardware and, thus, will unlikely lead to performance benefits with the current generation of GPUs. However, the library will benefit from the upgrade in the near future, both in terms of performance and functionality, providing a code which is more readable and easier to maintain.

- **1-dimensional textures**: used over global memory arrays, presumably due to better performance on older platforms. Given the current trends in GPU computing regarding memory management and how caching works on the texture memory, it does not seem sustainable in the long run.

- **3-dimensional textures**: used to manage the different medical images to perform the registration. Both when performing registration over 3D voxel volumes or 2D surfaces, they are represented as 3D textures.

  3D textures are used for two reasons: (1) allowing the program to interpolate values among voxels and (2) to exploit texture memory caching. The performance benefits, however, might fade away with the further generalization of GPU architectures.

5.3 Tested changes and attempted optimizations

5.3.1 1-dimensional textures. Usage and replacements.

Following from the previous analysis and under the assumption that the use of 1D textures does not translate itself into performance gains and knowing that the L1 and texture caches are unified in the Maxwell architecture [15], it was chosen as the first attempted change.

To make use of textures in CUDA, they must be assigned data. Those data may be bound to the texture as either a cudaArray object or by the use of normal data vectors stored in global memory. An example of this process, in the context of NiftyReg, can be found in `_reg_resampling_gpu.cu` for the host source code and `_reg_resampling_kernels.cu` for the device. The following excerpts (from the same files) depict each of the different tasks described:

```
//Bind deformationField to texture
NR_CUDA_SAFE_CALL(cudaBindTexture(0,
                                  deformationFieldTexture,
                                  *deformationFieldImageArray_d,
                                  activeVoxelNumber*sizeof(float4)))

//Bind voxel mask to texture
NR_CUDA_SAFE_CALL(cudaBindTexture(0,
                                  maskTexture,
                                  *mask_d,
                                  activeVoxelNumber*sizeof(int)))
```

Initially, the data is bound. Both `deformationFieldImageArray_d` and `mask_d` are arrays properly stored in device memory, whose types are float4 and int, respectively. The textures they are bound to have been declared at compile time as follows:

```
texture<float4, 1, cudaReadModeElementType> deformationFieldTexture;
texture<int, 1, cudaReadModeElementType> maskTexture;
```

After binding the data to the different textures, kernels that rely on them for fetching data can be launched. The use of textures implies no changes on the kernel launch whatsoever. For instance, the `reg_resampleImage3D_kernel` kernel is launched by simply executing:

```
reg_resampleImage3D_kernel <<< G1, B1 >>> (*warpedImageArray_d);
```

Inside the kernel, texture data will be accessed. In the case of the NiftyReg library, data from 1D textures are accessed using the `tex1Dfetch` function, which does not perform any sort of filtering. It is due to this fact that 1D textures can be converted to global memory arrays in a simple way. Examples of data fetching from two different textures are:

```
__global__ void reg_resampleImage3D_kernel(float *resultArray)
{
    ...
    const int tid2 = tex1Dfetch(maskTexture, tid);

    //Get the real world deformation in the floating space
    float4 realdeformation = tex1Dfetch(deformationFieldTexture,
                                        tid);
    ...
}
```

Finally, the texture may be unbound until it is required again. The process is analogous to binding them, as shown here:

```
NR_CUDA_SAFE_CALL(cudaUnbindTexture(deformationFieldTexture))
NR_CUDA_SAFE_CALL(cudaUnbindTexture(maskTexture))
```

Given that 1D textures are understood and that they can be replaced by an identical counterpart, the steps taken to do such thing will follow.

First, we must note that texture binding and unbinding will no longer be necessary, and that way, we will modify the code to replace them by pointers to the global memory arrays passed as arguments when the kernel is launched. The host code should resemble the following code snippet:

```
void reg_resampleImage_gpu(nifti_image *floatingImage,
                           float **warpedImageArray_d,
                           cudaArray **floatingImageArray_d,
                           float4 **deformationFieldImageArray_d,
                           int **mask_d,
                           int activeVoxelNumber,
                           float paddingValue)
{
    ...
```

```
//Bind deformationField to texture
//NR_CUDA_SAFE_CALL(
//    cudaBindTexture(0,
//                    deformationFieldTexture,
//                    *deformationFieldImageArray_d,
//                    activeVoxelNumber*sizeof(float4)))

//Bind voxel mask to texture
//NR_CUDA_SAFE_CALL(
//    cudaBindTexture(0,
//                    maskTexture,
//                    *mask_d,
//                    activeVoxelNumber*sizeof(int)))
    ...

NR_CUDA_SAFE_CALL(cudaMemcpy(floatingRealToVoxel_d,
                             floatingRealToVoxel_h,
                             3*sizeof(float4),
                             cudaMemcpyHostToDevice))
    ...

reg_resampleImage3D_kernel  <<  G1, B1  >>
                            (*warpedImageArray_d,
                              floatingRealToVoxel_d,
                             *deformationFieldImageArray_d,
                             *mask_d);
    ...

//NR_CUDA_SAFE_CALL(cudaUnbindTexture(deformationFieldTexture))
//NR_CUDA_SAFE_CALL(cudaUnbindTexture(maskTexture))

    ...
```

Code that is no longer required has been commented out to allow for an easier comparison. In these terms, the texture declarations for kernels to access will no longer be needed and they may be removed. Furthermore, the kernels using them will need to be modified in order to add the newly added pointers to global memory as arguments. Texture fetching will be replaced by common array accesses. The results of these changes on device code are depicted below:

```
//texture<float4, 1, cudaReadModeElementType> deformationFieldTexture;
//texture<int, 1, cudaReadModeElementType> maskTexture;

__global__ void reg_resampleImage3D_kernel(float *resultArray)
{
    ...
    const int tid2 = maskTexture[tid];
     //const int tid2 = tex1Dfetch(maskTexture, tid);

    //Get the real world deformation in the floating space
    float4 realdeformation = deformationFieldTexture[tid];
    //float4 realdeformation = tex1Dfetch(deformationFieldTexture,
    //                                    tid);

    ...
}
```

Our hypothesis appeared to be wrong, with the use of global memory over 1D textures appearing about twice as slow. We then considered that it could be the result of read-only texture memory versus memory that the compiler cannot identify as read only. As such, we attempted to make use of pointer aliasing (via the const and restrict keywords). The results are shown in Figure 5.1. R1, R2 and R3 are, respectively, executions with the original library using 1D textures, with the modified library using global memory only and, finally, with global memory using pointer aliasing. The bars representing minimum, average and maximum execution times overlap. Standard deviation among different executions is represented as the range line centered between average and maximum execution times. The shown kernels are just a sample as we have removed 1D textures all across the library with similar results.

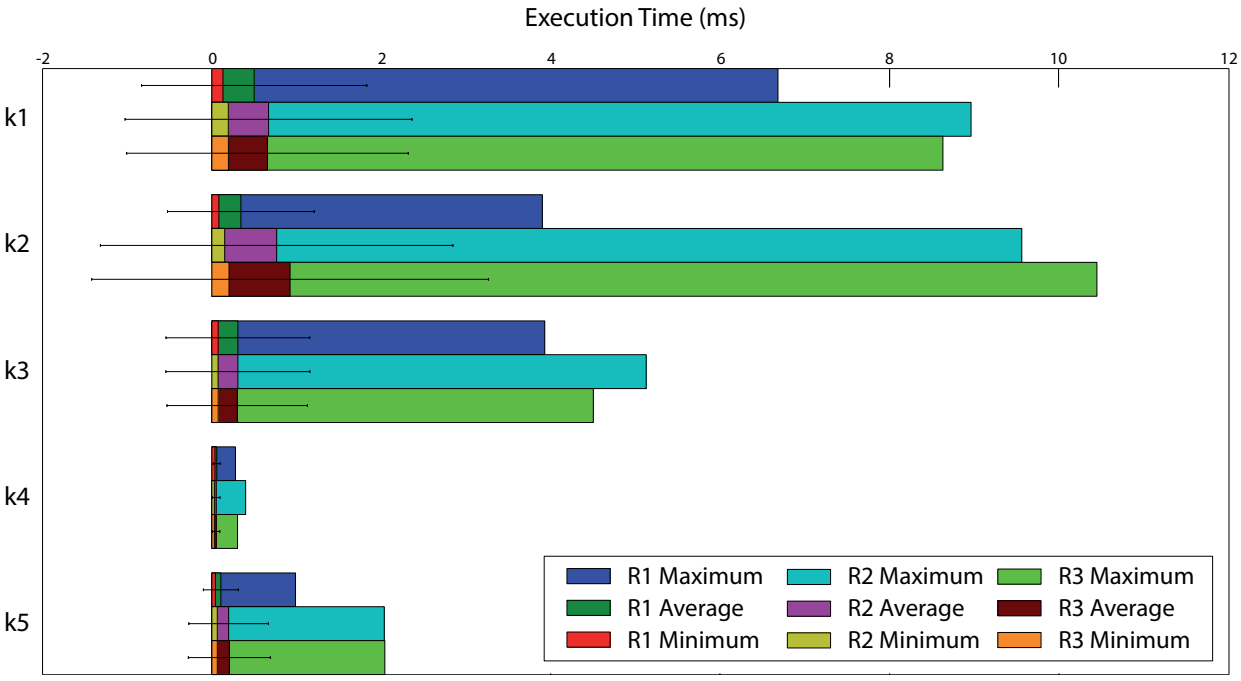

Figure 5.1: *Performance results for our texture conversion. Times are shown in milliseconds for the minimum, average and maximum of the three versions R1, R2 and R3 on a complete set of kernel launches within several NiftyReg executions. k1:* `reg_spline_getApproxDeformationField3D`*; k2:* `_reg_ApplyConvolutionWindowAlongX_kernel`*; k3:* `reg_resampleImage3D_kernel`*; k4:* `reg_spline_getApproxSecondDerivatives3D`*; k5:* `reg_splinegetApproxBending EnergyGradient3D`.

Kernel	Metric	R1	R2	R3
reg_spline_getDeformationField3D	Minimum:	0.13	0.19	0.20
	Average:	0.50	0.67	0.66
	Maximum:	6.69	8.97	8.63
	Launches:	8450	8450	7850
_reg_ApplyConvolutionWindowAlongX_kernel	Minimum:	0.08	0.15	0.20
	Average:	0.34	0.77	0.92
	Maximum:	3.90	9.57	10.45
	Launches:	800	800	750
reg_resampleImage3D_kernel	Minimum:	0.07	0.07	0.07
	Average:	0.31	0.31	0.30
	Maximum:	3.93	5.13	4.50
	Launches:	8450	8450	7850
reg_spline_getApproxSecondDerivatives3D	Minimum:	0.04	0.03	0.03
	Average:	0.06	0.05	0.05
	Maximum:	0.28	0.40	0.30
	Launches:	8450	8450	7850
reg_spline_getApproxBendingEnergyGradient3D_kernel	Minimum:	0.04	0.06	0.06
	Average:	0.11	0.20	0.21
	Maximum:	0.99	0.21	2.04
	Launches:	800	800	750

Table 5.1: *Summary of the obtained results of R1, R2 and R3.*

5.3.2 3-dimensional textures. Usage and replacements.

After obtaining those results, we decided that a similar analysis on 3D textures would make little sense, as no significant performance gains could be expected. However, the kernels were still studied, since we had believed that the use of 3D texturing was due to a need to interpolate values among voxels. In the contrary, we discovered that in most kernels the coordinates used were shifted and always placed at the center of the voxels, yielding the value of the voxel alone. Furthermore, access to the different voxels was linearized in most cases.

We will now proceed in a similar fashion to the one followed to describe the usage and replacement of 1D textures in terms of their 3D counterparts. For the shake of succintness, only tasks and sentences deemed as different enough will be described.

Following the same scheme, data must be bound to the texture. This process (performed on the host side, in `_reg_resampling_gpu.cu`) is, itself, very similar to the one described for 1D textures. However, the filtering mode, addressing and normalization options are also set up. These are not specific to 3D textures (and could be applied to any kind of textures), but in the context of NiftyReg it is only 3D textures that use them.

```
// Bind the required textures
referenceTexture.normalized = true;
referenceTexture.filterMode = cudaFilterModeLinear;
referenceTexture.addressMode[0] = cudaAddressModeWrap;
referenceTexture.addressMode[1] = cudaAddressModeWrap;
referenceTexture.addressMode[2] = cudaAddressModeWrap;
cudaChannelFormatDesc channelDesc = cudaCreateChannelDesc<float>();
NR_CUDA_SAFE_CALL(cudaBindTextureToArray(referenceTexture, *reference_d, ↩
    channelDesc))
```

Data is bound to a texture which, in this case, is declared to have 3 dimensions. Kernel launchers, as in the previous case, do not have to be modified.

```
texture<float, 3, cudaReadModeElementType> referenceTexture;
```

The data is accessed using the coordinates of the desired voxel. In the particular case of NiftyReg, these coordinates are integers shifted by 0.5 on all axes, returning the exact value of the voxel. Since the coordinates have been normalized, they ought to be relative to the length of each dimension.

```
__global__ void reg_getSquaredDifference3D_kernel(float *squaredDifference↩
    )
{
    ...

    const int z = index/(imageSize.x*imageSize.y);
    const int tempIndex = index - z*imageSize.x*imageSize.y;
    const int y = tempIndex/imageSize.x;
    const int x = tempIndex - y*imageSize.x;

    float difference = tex3D(referenceTexture,
                             ((float)x+0.5f)/(float)imageSize.x,
                             ((float)y+0.5f)/(float)imageSize.y,
                             ((float)z+0.5f)/(float)imageSize.z);

    ...

}
```

We believed that filtering and normalization could produce performance benefits if removed on kernels that required neither. Upon testing, however, performance remained intact. It was decided, then, to attempt to focus on the arithmetic intensity of the kernels instead, expecting to be able to find performance improvements.s

Exploiting the computing power of the GPU in NiftyReg

6.1 GPU computing power: underlying ideas

Once memory management is reasonably handled, optimization efforts must focus on making full use of the computing power of the GPU. In these terms, kernels must be adjusted in order to make use of Thread Level Parallelism (TLP) and Instruction Level Parallelism (ILP). Furthermore, the effect of load/store operations must be alleviated, aiming to improve the amount of operations per loaded datum. Finally, specific CUDA functions, devised to perform certain operations within less cycles, must also be used when attempting to reach peak performance [20].

In CUDA, thread level parallelism is given by the architecture, which guarantees

that threads will be independent with the exception of inter-block synchronization. The potential of TLP has to be obtained by selecting the most appropiate block size for the operations that are required.

Instruction level parallelism refers to available parallelism among independent instructions. CUDA devices exploit ILP in order to both hide latencies and obtain higher performance by interleaving operations that do not depend on one another.

The amount of ILP that can be deployed is, first and foremost, dependant on the problem that has to be solved. However, there are several heuristics that can be tried to exploit ILP in order to make full use of the compute capabilities of the GPU. In the context of NiftyReg, the following were applied:

- **Static branch precalculation**
- **Loop unrolling**
- **Instruction reordering**

The CUDA library contains certain functions mostly designed for graphics processing where speed is preferred, which are performed faster at the cost of being less precise. Furthermore, there are also CUDA intrinsics such as the family of FMA instructions, which allow the device to carry out a fused multiply add operation with a single instruction and greater precision. The usage of both fast math functions and the architecture-specific instructions improves efficiency with few changes.

6.2 Improving computation performance in NiftyReg

6.2.1 Optimizations for the _reg_tools kernels

With the intention to optimize computing bottlenecks, we profiled NiftyReg in order to decide which kernels should be our main aim. As a result, we focused on the _reg_ApplyConvolutionWindowAlong kernels located in reg_tools_kernels.cu. The three of them are very similar from one another and, because of that, a successful optimization on any of the three is relatively simple to apply to the others. Our changes and speedups will be with respect to the original implementation of the _reg_ApplyConvolutionWindowAlongX kernel, though similar results and optimization schemes apply on the Y and Z axes. A simplified depiction of the kernels can be found below, in C-like pseudocode:

```
__global__ void _reg_ApplyConvolution...(float4 *smoothedImage,
                                         int windowSize,
                                         float4 *gradientImageTexture,
                                         float *convolutionKernelTexture){
    if(tid < c_VoxelNumber){
        ...
        // Index and coordinate calculations
        ...
        float4 finalValue;
        float3 Y, t, c;
        for(int i=0; i<windowSize; i++){
            if(-1<x && x<imageSize.x){
                float4 gradientValue = tex1Dfetch(gradientImageTexture, ↩
                    index);
                windowValue = tex1Dfetch(convolutionKernelTexture, i);

                // Kahan summation
                Y = gradientValue * windowValue - c;
                t = finalValue + Y;
                c = (t - finalValue) - Y;
                finalValue = t;
            }
            index++; x++;
        }
        smoothedImage[tid] = finalValue;
    }
    return;
}
```

After studying the kernels and what they usually received as parameters, convolution windows with sizes ranging from 20 to 80 elements were observed. Because of this, we focused on optimizing the loop. Our first approach was to precalculate the inner if to reduce the amount of instructions within the loop and remove the cost of performing the branching jumps. To do so, minimum and maximum possible values within the structure for a given thread were computed and used as the indexes on the for loop, as follows:

```
int low= max(x, 0);
int high=min(x+windowSize, imageSize.x);
int windowIndex=low-x;
index += windowIndex;
for(;; low < high; low++){
    float4 gradientValue = tex1Dfetch(gradientImageTexture, index++);
    windowValue = tex1Dfetch(convolutionKernelTexture, windowIndex++);
    ...
}
```

Branch precalculation did not result in an speedup, though we suspected it could produce better results than the original implementation in case loop unrolling was performed (by exploiting ILP). As such, it was not completely ruled out and was, rather, left as a parallel effort.

Afterwards we studied the Kahan [6] summation being performed. Kahan summation is a standard way of performing a sum of many numbers, reducing rounding errors in the process.

However, the sequentiality of the algorithm and dependence among instructions partially hinders GPU performance (which is unable to exploit ILP). In an attempt to improve performance while mantaining precision, an `error-accumulative` version of the Kahan summation was implemented, known as Pichat [6] summation. This allowed us to reorder operations as shown below, allowing for instruction interleaving in order to exploit ILP:

```
float4 finalValue;
float3 Y, t, c;
for(;; low<high; low++){
    float4 gradientValue = tex1Dfetch(gradientImageTexture,index++);
    windowValue = tex1Dfetch(convolutionKernelTexture,windowIndex++);

    // Accumulative Kahan summation
    // (t and Y can be interleaved now)
    t = finalValue;
    Y = gradientValue * windowValue;
    finalValue += Y;
    c += (finalValue - t) - Y;
}
finalValue -= c;
```

The changes resulted in a 1.5x speedup (keeping the if) and 1.22x (removing the if) with respect to the original implementation. A slight precision loss was empirically observed after the change. Round up error depends, however, in the data to be processed, so given the subtle change we believe it will not be significant in the general case. Using Pichat summation also resulted in more kernel calls, decreasing the positive effect of the changes.

Observing the PTX code for the body of the loop, which is depicted below, hinted us that using fused multiply add operations could be useful in this context.

```
BB2_3:
    ...
    ld.global.f32   %f31, [%rd18];
    ld.global.v4.f32    {%f32, %f33, %f34, %f35}, [%rd17];
    mul.f32     %f39, %f32, %f31;
    mul.f32     %f40, %f33, %f31;
    mul.f32     %f41, %f34, %f31;
    add.f32     %f63, %f6, %f39;
    add.f32     %f60, %f5, %f40;
    add.f32     %f57, %f4, %f41;
    ...
    setp.lt.s32 %p3, %r56, %r8;
    mov.f32     %f56, %f57;
    mov.f32     %f59, %f60;
    mov.f32     %f61, %f63;
    mov.f32     %f62, %f61;
    @%p3 bra    BB2_3;
```

Since the original focus of the kernels seemed to be precision over performance, we implemented a FMA-based version of Pichat summation (v5). The code changes are depicted below:

```
float4 finalValue;
float3 t, c;
for(;; low<high; low++){
    float4 gradientValue = tex1Dfetch(gradientImageTexture,index++);
    windowValue = tex1Dfetch(convolutionKernelTexture,windowIndex++);

    // Accumulative Kahan summation with FMA ops
    t = finalValue;
    finalValue = fmaf(gradientValue, windowValue, finalValue);
    c -= fmaf(gradientValue, windowValue, t.x - finalValue.x);
}
finalValue -= c;
```

An speedup of 1.5x (keeping the if) with respect to the original implementation was obtained. The speedup without the if was lower than its FMA-less counterpart, being 1.15x only. The same effect as before was observed on the number of kernel launches, which increased and hindered the performance gains from the optimization. Finally, and going against our expectations, precision did not improve from our previous implementation.

In order to produce a faster version without a loss in precision (observed with Pichat summation), we implemented the original Kahan summation algorithm with FMA as depicted below:

```
float4 finalValue;
float3 t, c;
for(;; low<high; low++){
    float4 gradientValue = tex1Dfetch(gradientImageTexture,index++);
    windowValue = tex1Dfetch(convolutionKernelTexture,windowIndex++);

    // Kahan summation
    t = finalValue;
    finalValue = fmaf(gradientValue, windowValue, finalValue) + c;
    c = fmaf(gradientValue, windowValue, t - finalValue.x);
}
```

Performance with respect to the original implementation was 1.3x keeping the if and 1.13x without it. In this case, however, kernel launches decreased (indicating a more rapid convergence towards the result). Precision changed again, being worse in this case when compared to the CPU but closer to the original GPU implementation (that was to be optimized). In these terms, the hypotheses about the treated data being the main driving factor in terms of the precision yielded on the result seemed to hold.

Finally, and in order to test all the possible combinations, we implemented the kernel performing the sum with only FMA operations and without any error correction algorithms. This was done under the assumption that the increased accuracy of the FMA operations would be enough to overcome the rounding errors being carried over through the sum. The resulting implementation is shown below:

```
float4 finalValue;
for(;; low<high; low++){
    float4 gradientValue = tex1Dfetch(gradientImageTexture,index++);
    windowValue = tex1Dfetch(convolutionKernelTexture,windowIndex++);

    finalValue = fmaf(gradientValue,
                      windowValue,
                      finalValue);
}
```

Although the speedup in terms of kernel execution times was not very high (1.21x with if and 1.13x without), the FMA implementation resulted in a higher application speedup due to a reduced amount of kernel executions (which was the lowest among all the attempted changes). What is more, it was also more precise than the other schemes, which explained the change in kernel launches. Our previous concerns about the data-dependency in terms of the accuracy of the results still apply, and ought to be considered when deciding which scheme to use.

To complete our study, we decided to apply loop unrolling over the main loop, in an attempt to squeeze performance as much as possible. We hypothesized that the optimal unroll factor would be 20, since all observed kernel windows seemed to have a size of a multiple of 20 plus 1. Since unrolling is equivalent semantically to common loops, the amount of kernel launches did not vary. However, kernel execution times changed, with most kernels seemingly performing better with 5 unrolls over 20. The best performing implementation pre-unrolling (FMA only with ifs) reached higher speedups (1.36x) with 5 unrolls that, paired with the lower amount of kernel launches, resulted in NiftyReg running 1.1x times faster globally.

Figure 6.1 and Figure 6.2 show the performance and precision changes with respect to the original implementation. Only X-axis kernel data is shown, though similar results were observed for Y-axis and Z-axis kernels.

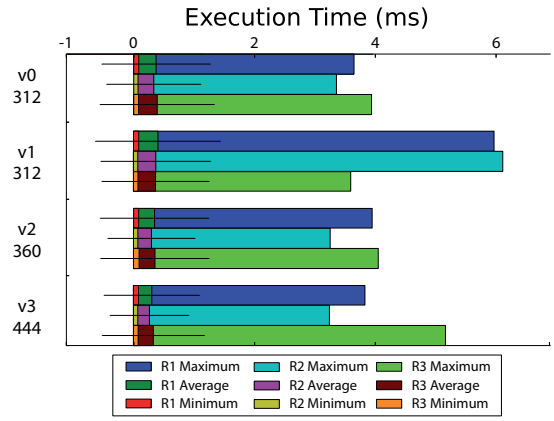

Figure 6.1: *Performance comparison with respect to the original implementation.*

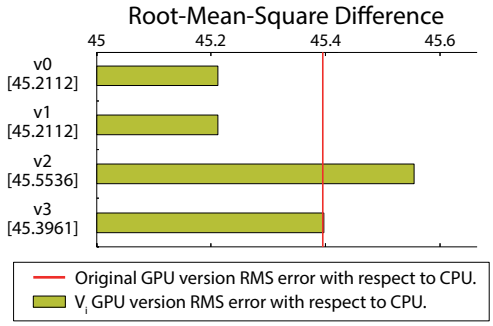

Figure 6.2: *Accuracy comparison with respect to the original implementation.*

In summary, and given the obtained results, the FMA only implementation seems preferable on all grounds. However, when the aim is accuracy, it could be the case that either FMA-Kahan and FMA-Pichat perform better in the general case. A detailed overview of the results can be found in Table 6.1. All speedups are relative to the original implementation. The table identifiers refer respectively to the 4 better performing kernels, with those being FMA-only with if (v0), FMA-only without if (v1), FMA-Kahan with if (v2) and Pichat with if (v3).

Optimization	Unroll factor	Kernel time	Launches	GPU error	CPU error	Kernel speedup	Weighted speedup
Original	0	0.45	372	0.00	45.39	1.00x	1.00x
	5	0.39				1.15x	1.15x
	20	0.46				0.98x	0.98x
Pichat with if	0	0.30	444	20.86	45.40	1.50x	1.26x
	5	0.27				1.67x	1.40x
	20	0.34				1.32x	1.11x
Fma with if	0	0.37	312	20.89	45.21	1.22x	1.45x
	5	0.33				1.36x	1.63x
	20	0.39				1.15x	1.38x
Pichat+fma with if	0	0.30	444	20.86	45.40	1.50x	1.26x
	5	0.27				1.67x	1.40x
	20	0.27				1.67x	1.40x
Kahan+fma with if	0	0.34	360	17.97	45.55	1.32x	1.37x
	5	0.30				1.50x	1.55x
	20	0.35				1.29x	1.33x
Original without if	0	0.48	372	0.00	45.39	0.94x	0.94x
	5	0.40				1.13x	1.13x
	20	0.36				1.25x	1.25x
Pichat without if	0	0.37	444	20.86	45.40	1.22x	1.02x
	5	0.31				1.45x	1.22x
	20	0.30				1.50x	1.26x
Fma without if	0	0.40	312	20.89	45.21	1.13x	1.34x
	5	0.38				1.18x	1.41x
	20	0.36				1.25x	1.49x
Pichat+fma without if	0	0.39	444	20.86	45.40	1.15x	0.97x
	5	0.30				1.50x	1.26x
	20	0.33				1.36x	1.14x
Kahan+fma without if	0	0.40	360	17.97	45.55	1.13x	1.16x
	5	0.34				1.32x	1.37x
	20	0.38				1.18x	1.22x

Table 6.1: *Summary table containing the RMS error of the results obtained on GPU with the ones obtained on the original GPU and CPU implementations, average execution times (in ms), amount of kernel launches, kernel time speedup and composite speedup (kernel time * amount of launches) for the reg_ApplyConvolutionWindowAlongX kernel.*

6.2.2 Optimizations for the _reg_localTransformation kernels

Upon achieving reasonable speedups in the _reg_ApplyConvolutionWindowAlongAxis family of kernels, we profiled the application again looking for other bottlenecks. Two kernels seemed to be particularly heavy in terms of time spent on them, both reg_spline _getDeformationField3D and reg_spline_getApproxSecondDerivatives3D on the set of _reg_localTransformation kernels.

After a closer inspection of the _reg_localTransformation kernels, we noticed a shared-memory filling pattern. The code, in order to atenuate the effects of accessing data stored in global memory in older architectures, initialized shared memory with values that were preset on a few functions. Since it did not seem to produce any kind of improvements in terms of performance, it was replaced with an equivalent implementation using constant arrays.

As performance remained unchanged, no code will be shown. It must be noted, however, that both maintainability and readability improved, as the resulting code was simpler and relied only on basic constructs (data typed as const __device__).

Given the size of the code, we aimed to find small performance improvements that would apply to all kernels. Most of them made use of vector types with operators and functions defined in NiftyReg. Because of this, and knowing that certain equivalent operations using FMAs had a faster performance (as depicted below), we aimed to find include fused multiply add operations where possible:

```
// A * B - C (not using FMA)
Y.x = gradientValue.x * windowValue - c.x;
/* Resulting PTX code (trimmed)

    mul.f32         %f39, %f31, %f38;
    sub.f32         %f40, %f39, %f51;
*/
```

```
// A * B - C (using explicit FMA)
Y.x = fmaf(gradientValue.x, windowValue, -c.x);
/* Resulting PTX code (trimmed)

    neg.f32         %f39, %f51;
    fma.rn.f32      %f40, %f31, %f38, %f39;
*/
```

Testing different possible operand combinations, we hypothesized that the compiler is not always able to detect the presence of an implicit FMA operation when using vector types. Because of this, implementing FMA operations explicitly seemed like a reasonable optimization method. An example of the changes is shown below:

```
// Original implementation (reg_spline_getApproxBendingEnergy3D_kernel)
float4 XX = secondDerivativesTexture[index++];
XX=XX*XX;
float4 YY = secondDerivativesTexture[index++];
YY=YY*YY;
float4 ZZ = secondDerivativesTexture[index++];
ZZ=ZZ*ZZ;
float4 XY = secondDerivativesTexture[index++];
XY=XY*XY;
float4 YZ = secondDerivativesTexture[index++];
YZ=YZ*YZ;
float4 XZ = secondDerivativesTexture[index];
XZ=XZ*XZ;

penaltyTerm[tid]=
        XX.x + XX.y + XX.z +
        YY.x + YY.y + YY.z +
        ZZ.x + ZZ.y + ZZ.z +
            2.f*(XY.x + XY.y + XY.z +
                 YZ.x + YZ.y + YZ.z +
                 XZ.x + XZ.y + XZ.z);
```

```
// Alternate FMA based implementation (↩
   reg_spline_getApproxBendingEnergy3D_kernel)
float4 XX = secondDerivativesTexture[index++];
float4 YY = secondDerivativesTexture[index++];
float4 ZZ = secondDerivativesTexture[index++];
float4 XY = secondDerivativesTexture[index++];
float4 YZ = secondDerivativesTexture[index++];
float4 XZ = secondDerivativesTexture[index];

penaltyTerm[tid]=
               fmaf(fmaf(XY.x, XY.x, fmaf(XY.y, XY.y, fmaf(XY.z, XY.z,
                    fmaf(YZ.x, YZ.x, fmaf(YZ.y, YZ.y, fmaf(YZ.z, YZ.z,
                    fmaf(XZ.x, XZ.x, fmaf(XZ.y, XZ.y, XZ.z * XZ.z)))))))),
                    2.0f,
                    fmaf(XX.x, XX.x, fmaf(XX.y, XX.y, fmaf(XX.z, XX.z,
                    fmaf(YY.x, YY.x, fmaf(YY.y, YY.y, fmaf(YY.z, YY.z,
                    fmaf(ZZ.x, ZZ.x, fmaf(ZZ.y, ZZ.y, ZZ.z * ZZ.z)))))))
                    );
```

We later noticed that FMA placement depended on whether compilation was performed in debug mode or not. PTX code obtained in release mode contained both inlined operators and FMA instructions. A comparison between the PTX produced by the original version and ours is shown below:

```
// PTX code for the original implementation
// All data is loaded and then operations are performed
ld.global.v4.f32        {%f1, %f2, %f3, %f4}, [%rd6];

...

ld.global.v4.f32        {%f38, %f39, %f40, %f41}, [%rd6+80];
fma.rn.f32      %f45, %f1, %f1, %f8;
fma.rn.f32      %f46, %f3, %f3, %f45;
fma.rn.f32      %f47, %f9, %f9, %f46;

...

fma.rn.f32      %f59, %f39, %f39, %f58;
fma.rn.f32      %f60, %f40, %f40, %f59;
fma.rn.f32      %f61, %f60, 0f40000000, %f52;

...

st.global.f32   [%rd8], %f61;
```

```
// PTX code for the FMA based implementation
// Data loads and operations are interleaved

ld.global.v4.f32        {%f1, %f2, %f3, %f4}, [%rd6+80];
mul.f32         %f8, %f3, %f3;
fma.rn.f32      %f9, %f2, %f2, %f8;
fma.rn.f32      %f10, %f1, %f1, %f9;

...

ld.global.v4.f32        {%f11, %f12, %f13, %f14}, [%rd6+64];
fma.rn.f32      %f18, %f13, %f13, %f10;
fma.rn.f32      %f19, %f12, %f12, %f18;
fma.rn.f32      %f20, %f11, %f11, %f19;

...

st.global.f32   [%rd8], %f62;
```

In this case, the resulting code for our implementation did not allow for full ILP deployment since load and arithmetic operations appear in a dependent sequence. Since our aim was to obtain code that performed such interleaving (which was, instead, produced by the compiler), we deemed these changes as both time consuming and not useful.

Given the few improvements obtained, our efforts pivoted towards exploiting spatial locality on kernels where it was possible. First, we studied the reg_spline_get ApproxBendingEnergyGradient3D kernel. The memory access pattern appeared to be very regular, consisting on a 3x3x3 cube being traversed per thread. Furthermore, the basis value matrices, which weight the voxels of the cube depending on their position, are either fully or inverse-sign symmetrical. The code, simplified for clarity, is shown below:

```
// Thread, coordinate and related computations
int x,y,z;
...

// Iterate over the 3D cube
for(int c=z-1; c<z+2; ++c){
    for(int b=y-1; b<y+2; ++b){
        for(int a=x-1; a<x+2; ++a){
            if(-1<a && -1<b && -1<c && a<gridSize.x && b<gridSize.y && c<↩
               gridSize.z){
                unsigned int indexXYZ = 6*((c*gridSize.y+b)*gridSize.x+a);

                gradientValue += secondDerivativesTexture[indexXYZ++] * ↩
                   xxBasis3D[coord]; // XX
                gradientValue += secondDerivativesTexture[indexXYZ++] * ↩
                   yyBasis3D[coord]; // YY
                gradientValue += secondDerivativesTexture[indexXYZ++] * ↩
                   zzBasis3D[coord]; // ZZ
                gradientValue += secondDerivativesTexture[indexXYZ++] * ↩
                   xyBasis3D[coord]; // XY
                gradientValue += secondDerivativesTexture[indexXYZ++] * ↩
                   yzBasis3D[coord]; // YZ
                gradientValue += secondDerivativesTexture[indexXYZ++] * ↩
                   xzBasis3D[coord]; // XZ
            }
            coord++;
        }
    }
}
...

nodeGradientArray[tid] += gradientValue;
```

It seemed that shared memory could be used to avoid computing one of the 3x3 slices (located one position ahead in the x axis) by computing the first two and sharing the third one with the thread two positions behind. An implementation for such scheme was developed, taking into account the need for a different traversal on the basis.

The first changes (shown below) included manually unrolling the loops to work on each X-axis slice, and storing the first slice and computing the third (taking into account the negative symmetry on some of the basis matrices).

```
// Compute the first (x-1) slice
int a = x-1;
int coord = 0;
for(int c=z-1; c<z+2; ++c){
    for(int b=y-1; b<y+2; ++b){
        if(-1<a && -1<b && -1<c && a<gridSize.x && b<gridSize.y && c<gridSize.z){
            unsigned int indexXYZ = 6*((c*gridSize.y+b)*gridSize.x+a);

            gradientValue    += secondDerivativesTexture[indexXYZ++] * xxBasis3D[coord]; // XX
            gradientValue    += secondDerivativesTexture[indexXYZ++] * yyBasis3D[coord]; // YY
            gradientValue    += secondDerivativesTexture[indexXYZ++] * zzBasis3D[coord]; // ZZ
            gradientValueNeg += secondDerivativesTexture[indexXYZ++] * xyBasis3D[coord]; // XY
            gradientValue    += secondDerivativesTexture[indexXYZ++] * yzBasis3D[coord]; // YZ
            gradientValueNeg += secondDerivativesTexture[indexXYZ++] * xzBasis3D[coord]; // XZ
        }
        // Since we work over the X axis, we traverse the basis matrices on Y and Z
        coord+=3;
    }
}
// Store on shared memory, separating those
// that are neg-sign symmetrical so the value can be computed
shMem[threadIdx.x]             = gradientValue;
shMem[threadIdx.x+blockDim.x] = gradientValueNeg;

// Compute our total sum
gradientValue += gradientValueNeg;
```

Similar code to compute the center slice (a = x and coord = 9) was added. Then, in the case that the thread was at the edge of the image or was the last thread in the block, the last slice was computed. This had to be done due to the fact that sharing data was not possible in this case. In the other case, and as depicted in the following code, the slice is taken from shared memory and added to the total sum:

```
// Wait for the threads to send their data
__syncthreads();

// Add the next slice, if it has been computed (accounting for neg. ↩
   symmetry)
if(x < nextX && z == nextZ && y == nextY && threadIdx.x + 2 < blockDim.x)
    nextSubGradientValue=shMem[threadIdx.x+2] - shMem[threadIdx.x+2+↩
       blockDim.x];

gradientValue += nextSubGradientValue;
```

We expected those efforts to translate into reasonable performance gains, as we avoided 9 global memory accesses per thread. However, after finishing our implementation and measuring performance, the results were not very noticeable. Figure 6.3 depicts the obtained results with each of the schemes and variations (loop unrolling, shared memory constant removal) taken into account.

Although the amount of kernel launches increased slightly, the overall benefit of the changes are clear: kernel execution times are reduced by a 2x factor. The amount of error when comparing the result of our implementation to the CPU version of NiftyReg also decreased. This, however, and as been echoed throughout our description of the results obtained, does not mean that in the general case the results should be better. The way in which data is accessed affects rounding error buildup, and different images and implementations will differ in the amount of error on a per-case basis. A graph comparing the amount of error with respect to v0 between v1, v2 and v3, v4 is found in Figure 6.4.

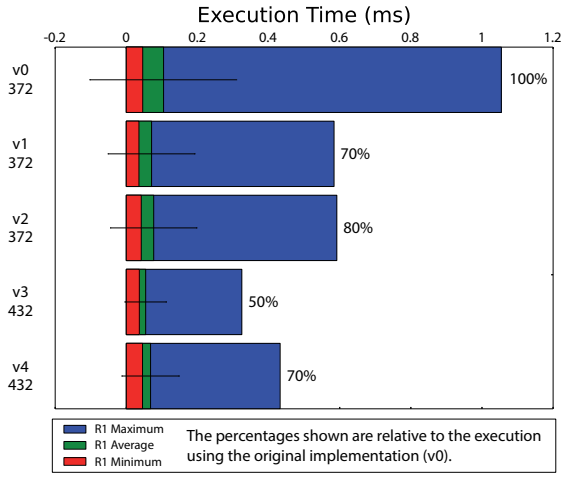

Figure 6.3: *Performance results for the set of performed changes on the reg_spline_get ApproxBendingEnergyGradient3D kernel. Each of the different versions is accompanied by the number of kernel launches that NiftyReg performs. v0 corresponds to the original implementation. v1 to the original implementations with constants moved to constant memory, with v2 being v1 after loop unrolling. v3 corresponds to our final implementation, exploiting basis matrix symmetry and shared memory; v4 is the unrolled version of v3.*

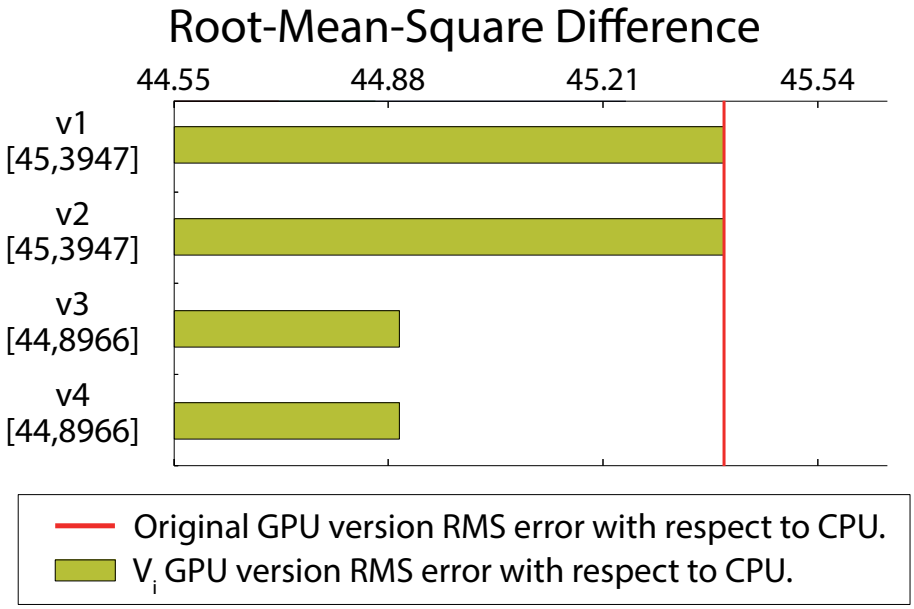

Figure 6.4: *RMS error results for the set of performed changes on the reg_spline_get ApproxBendingEnergyGradient3D kernel.*

7 Conclusions

The implementation of our hypetheses and changes, as we have been explaining through the two previous chapters, impacted performance in different ways.

In regards to memory management, the results suggest that NiftyReg does not benefit from replacing data stored as CUDA textures as global memory arrays (using the latest iteration of the CUDA model, Maxwell). The improvement that we expected, based on the linearized use of texture memory and the misuse of the features given by textures, was not present. Pointer aliasing on the global memory arrays to provide the compiler additional information indicating that data would be read-only did not result in an improvement in performance either. We believe that it might be due to the fact that texture caching patterns are still more advanced than their general counterparts, even with L1 cache and texture memory being unified in the aforementioned generation. Future iterations of the CUDA model (which diverges further and further from classical graphics processing to GPGPU) might produce performance benefits in NiftyReg when

not using textures. More work will be needed, particularly on upcoming platforms implementing new memory architectures (such as the physical implementation for unified memory using stacked RAM).

Whilst focusing on computational intensity, we obtained a wide set of results and conclusions, related to both performance and accuracy. When working on convolution kernels, and in line with what we expected, we found that the original summation scheme (Kahan summation) could be replaced with either Pichat summation or FMA summation without any sort of rounding error correction. In the end, using FMA operations alone proved to be the best solution as it resulted in the highest accuracy with respect to the CPU implementation and the best performance. We noted that accuracy results could be particularly dataset-dependent (although, by nature of how it is performed, we believe that FMA-only implementations would still outperform other schemes).

Delving further into those kernels, and against our first intuition, the optimal unrolling factor for the summation over the convolution window was found to be 5. This went against our initial reasoning, which assumed higher unrolling factors (of 20 in particular) would perform better due to the size of the windows being multiples of 20 plus 1. Similarly negative results were found with respect to if removal, which did not improve performance despite our belief that it would allow the kernel to better deploy instruction level parallelism.

Following from the impact of the FMA implementation in the convolution kernels, we attempted to study if the nvcc compiler did not make full use of FMA operations in the Local Transformation kernels. We found a case in which FMA operations were not used (that was, in fact, multiply and subtract), but in general the compiler outdid our efforts.

Finally, we attempted to obtain performance gains by exploiting the symmetrical nature of a problem, which allowed for data precalculation and sharing for threads within a block. By unrolling, restructuring and breaking up a volume when calculating a gradient taking into account the symmetry of the basis matrices, we reduced the kernel computation time to around half of the original time.

7.1 En español.

La implementación de nuestros cambios e hipótesis, tal y como se ha ido exponiendo a lo largo de los dos últimos capítulos, tuvo distintas formas de afectar al rendimiento total.

En lo referido a la gestión y organización de la memoria, los resultados sugieren que NiftyReg no se beneficia al reemplazar el uso de texturas CUDA con arrays en memoria global (usando la última iteración del modelo CUDA, Maxwell). La mejora que esperábamos, basándonos en el uso linearizado de la memoria de texturas y el mal uso de las características que éstas proporcionan, no se logró. El etiquetado de punteros a los arrays en memoria global para proporcionar información al compilador sobre los datos, de solo lectura, tampoco resultó en una mejora de rendimiento. Entendemos que dicho resultado se puede deber a que los patrones de cacheado de texturas siguen siendo más avanzados que sus homólogos de carácter general. Versiones futuras del modelo CUDA (que diverge progresivamente desde el procesamiento tradicional de gráficos a un GPGPU puro) podrían producir mejoras de rendimiento en NiftyReg cuando no se haga uso de la memoria de texturas. Será necesario más trabajo, particularmente en futuras plataformas que implementen nuevas arquitecturas de memoria (tales como implementaciones del modelo de memoria unificada usando stacked RAM).

Focalizándonos en la intensidad computacional obtuvimos un amplio conjunto de resultados y conclusiones, relacionados tanto con el rendimiento en tiempo como con la precisión. A partir del trabajo sobre los kernels de convolución, y siguiendo la línea de lo esperado, encontramos que el método original de sumatorio (sumatorio Kahan) podía ser reemplazado tanto por el método de sumatorio Pichat como por un método sin corrección de errores usando únicamente operadores FMA. Al final concluímos que el uso de operaciones FMA sin ningún tipo de corrección adicional era la mejor solución, dado que produjo tanto una mejora en término de los tiempos de ejecución como un decremento del error en comparación con la versión original en CPU. Remarcamos, a modo de punto a considerar, que los resultados en términos de precisión pueden ser particularmente dependientes del conjunto de datos a tratar (aunque, por la naturaleza de cómo se realiza, creemos que usar FMAs únicamente seguiría siendo el método ganador).

Profundizando más en estos kernels, y contra nuestra primera intuición, encontramos que el factor de desenrollado de bucle óptimo sobre la ventana de convolución era 5. Este resultado iba a la contra de nuestro razonamiento inicial, que asumía que factores de desenrollado mayores (de 20 en particular) producirían un mayor beneficio debido a que el tamaño de las ventanas era de múltiplos de 20 más 1. De forma similar encontramos que el precálculo y la eliminación de ifs no produjo mejoras sustanciales pese a nuestra creencia de que permitiría al kernel desplegar de forma más efectiva el paralelismo a nivel de intrucción.

Debido al impacto positivo de la implementación basada en FMAs en los ker-

nels de convolución, intentamos estudiar si el compilador nvcc no las aprovechaba completamente en los kernels de Local Transformation. Encontramos un caso en que las operaciones FMA no eran detectadas (que era, en realidad, multiplicación y resta) pero en general el compilador producía código superior.

Finalmente, intentamos obtener mejoras de rendimiento mediante gracias al aprovechamiento de la simetría propia del problema, que permitía precalcular y compartir datos entre hilos de un bloque. Mediante desenrollado de bucles, reestructuración del código y reordenación de un volumen a la hora de calcular un gradiente teniendo en cuenta la simetría de las matrices de las bases, reducimos el tiempo de computación hasta ser alrededor de la mitad del original.

8 Bibliography

[1] Antonio Ruiz, Manuel Ujaldón. Exploiting Kepler Capabilities on Zernike Moments. 2015.

[2] Chris McClanahan. History and Evolution of GPU Architecture. 2010.

[3] Christos Kyrkou. Stream Processors and GPUs: Architectures for High Performance Computing. .

[4] Mark Harris. Introduction to CUDA C, 2013.

[5] Ian Buck. *Stream Computing on Graphics Hardware*. PhD thesis, September 2006.

[6] John Michael McNamee. A Comparison Of Methods For Accurate Summation. *ACM SIGSAM Bulletin*, 38, March 2004.

[7] Mark Harris. Maxwell: The Most Advanced CUDA GPU Ever Made, 2014. URL http://devblogs.nvidia.com/parallelforall/maxwell-most-advanced-cuda-gpu-ever-made/.

[8] Marc Modat, Zeike A. Taylor, Josephine Barnes, David J. Hawkes, Nick C. Fox, and Sebastien Ourselin. Fast free-form deformation using the normalised mutual information gradient and graphics processing units. *Med Phys*, pages 278–284, 2010.

[9] NVIDIA Corporation. NVIDIA GeForce 8800 GPU architecture overview. Technical report, November 2006.

[10] NVIDIA Corporation. NVIDIA GeForce GTX 200 GPU architectural overview. Technical report, May 2008.

[11] NVIDIA Corporation. NVIDIA GF100 Whitepaper. Technical report, 2010.

[12] NVIDIA Corporation. NVIDIA's Next Generation CUDA Compute Architecture: Kepler GK110/210 Whitepaper. Technical report, 2014.

[13] NVIDIA Corporation. CUDA C Programming Guide, 2015. URL http://docs.nvidia.com/cuda/cuda-c-programming-guide.

[14] NVIDIA Corporation. NVIDIA GeForce GTX 980 Whitepaper. Technical report, 2015.

[15] NVIDIA Corporation. Maxwell Tuning Guide: 1.4.2.1. Unified L1/Texture Cache, 2015. URL http://docs.nvidia.com/cuda/maxwell-tuning-guide/#l1-cache.

[16] Roche A. Subsol G. Pennec X. Ourselin, S. and N. Ayache. Reconstructing a 3d structure from serial histological sections. *Image and Vision Computing*, 19(1-2): 25–31, 2001.

[17] Sonoda L. I. Hayes C. Hill D. L. G. Leach M. O. Rueckert, D. and D. J. Hawkes. Nonrigid registration using free-form deformations: Application to breast mr images. *IEEE Transactions on Medical Imaging*, 18(8):712–721, 1999.

[18] Samuel Williams, Andrew Waterman, David Patterson. Roofline: an insightful visual performance model for multicore architectures. *Communications of the ACM*, 52:65–76, April 2009. doi: 10.1145/1498765.1498785.

[19] Stephan Soller. GPGPU origins and GPU hardware architecture. 2011.

[20] Vasily Volkov. Better performance at lower occupancy. UC Berkeley Lecture, 2010. URL http://www.cs.berkeley.edu/~volkov/volkov10-GTC.pdf.

[21] Barbara Zitová and Jan Flusser. Image registration methods: a survey. *Image and Vision Computing*, 21:977–1000, 2003.