CUDA implementation of integration rules within an hp-Finite Element code.

Author: Adrián García [11],
Directors: David Pardo [2*], and Ricardo Celorrio[3+]

[*]Departamento de Matemática Aplicada, UPV/EHU University
[+]Departamento de Matemática Aplicada, Zaragoza's University.

September 4, 2012

[1]garciaprado.adrian@gmail.com
[2]dzubiaur@gmail.com
[3]celorrio@unizar.es

Contents

Abstract

With the introduction in 2006 of CUDA architecture for Nvidia GPUs, a general purpose parallel computing architecture. A new programming model where scientist and engineers had found and answer to their claim of high performance computation with moderate cost. This thesis work begin with the motivations that underline parallel computation and high performance computation. Later it is discussed why CUDA architecture and General Porpoise Graphics Processor Units (GPGPU) had modified the high performance computation world and why it is turning into GPGPU computing.

The author has chosen this new language because of its multiple and countless possibilities focusing in engineering applications. CUDA's novelty it is both a motivation and an obstacle because to the lack of support and documentation specifically to engineering applications.

This work has two main goals. First one is to implement a FEM integration code in a GPU architecture using CUDA language and a Nvidia graphic device. This code performance will be compared to a parallelized CPU code developed by the same author. Second, once the integration has been done the obtained solution is transform into a CRS compress matrix which is the beginning of another work, solve the system described by that particular matrix. The performance of matrix assembly in GPU it is also measured and compared to its respective code in parallelized CPU.

Chapter 1

Introduction.

Since the early days of computers, computer graphics had been essential to make possible a better human-machine interaction. While some applications only need 2D graphics many other start using 3D graphics. This applications whose most representative are computer games, photography& video edition, have evolved in applications who demand a huge computation capability. As this applications require more and more resources standard CPUs were not enough to support their function. That was the main reason to build up a device specifically dedicated to graphics, General Porpoise Graphics Processor Units (GPGPU / GPU) were born.

GPU microprocessor were born to draw points inside a defined screen what is basically linear algebra operations. Within the years GPU advanced and 3D computer graphics became an integral part of many computer applications. The 3D to every application resulted in a huge market of affordable 3D graphics cards. Devices prices get lower and lower and its compute capability higher and higher. Few years ago GPUs outperformed the number of float points operations per second that CPUs were capable, this was the trigger to a new era. GPUs could now be used to high performance applications within a moderate cost. That was particularly beneficial to science and engineering world who is always looking for more compute capabilities.

Science and engineering world turned into this kind of computation obtaining incredible results to many problems and realizing in some other areas that making use of large number of cores with less compute capability it is not always a panacea. The main problem of GPU computing it is closely related to **Amhdal's law** (section 1.1.1). This computational law says. The speedup of a program using multiple processors in parallel computing is limited by the sequential fraction of the program.

In this articles two master students will implement a Finite Element code, which is a solution largely utilized in engineering to simulate structures. And compare the results of running this program in both CPU and GPU.

1.1 High Performance Computing. And Parallelization.

It was 2006 when Intel launched its Core Duo into the market. This was the beginning of a revolution, many years of investigation ended with a single CPU with two cores. Single processor units or mono-cores rapidly died because of the significant computational advantages of utilizing several cores. CPUs had continued evolving and nowadays are available even to commodity computers to home utilization counting on several cores. But is this growth in the number of cores inside a single processor the solution to every problem in high computation problems? The answer is no. If we take a look to *Moore*'s Law, the number of transistor inside a microprocessor doubles every 18 months. Does it mean that microprocessors duplicates its velocity every 18 months? One more time the answer is no. CPU's velocity relies on internal clock frequency inside each microprocessor. Anyway it is a fact that with more microprocessor the program performance increases. This section will make a brief introduction to *Amhdal*'s law. And after introducing *Amhdal*'s law we will have another introduction to the techniques used in CPU parallel computing.

1.1.1 Amhdal's law.

It was 1967 when Gene Ahmdal proposed the following observation. The speedup of a program using multiple processors in parallel computing is limited by the time needed for the sequential fraction of the program. Amdahl's law is a model for the relationship between the expected speedup of parallelized implementations of an algorithm relative to the serial algorithm, under the assumption that the problem size remains equal when parallelized. In case a part of the algorithm can not be parallelized, the program speedup will improve only because of the part that is possible to run in parallel. And the improvement will depend in how much of that algorithm can be parallelized. More technically, the law is concerned with the speedup achievable from an improvement to a computation that affects a proportion P of that computation where the improvement has a speedup of S.

$$\frac{1}{(1-P)+\frac{P}{S}} \tag{1.1.1}$$

1.1.2 Parallel computing.

Parallel computing is a form of computing where many cores execute the same program at a once. It has been widely used long time ago but it was not until 2006 that processors with multiple cores were accessible to non

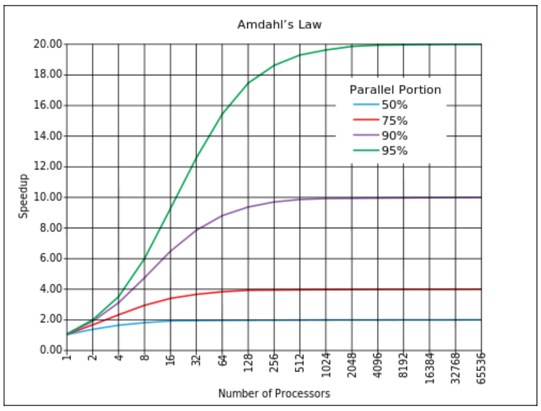

Figure 1.1: The speedup of a program using multiple processors in parallel computing is limited by the sequential fraction of the program. For example, if 95% of the program can be parallelized, the theoretical maximum speedup using a parallel algorimth would be 20x as shown in the diagram.

professional users. Before 2006 parallel programs had to run in physically separated computers, this still is a technique widely used nowadays.

Carry out the computation simultaneously turns into an improvement of required time to execute the program. First approximation to parallel computers can be divided in two groups. Processes that run parallel inside a single machine and processes that require several computers to perform the task, to this group belong computer cluster, grids and MMP's. Closer this article main goal are multi-core, multi-processor machines and symmetric multiprocessing techniques. Who are inside the category of parallel execution using only one computer.

Multi-core processor.

This are nowadays processors, they count with several cores, inside a single silicon component with two or more cores. The improvement in performance gained by the use of a multi-core processor depends on the software algorithms used and their implementation. Possible gains are limited by the fraction of the algorithm that can be run in parallel. Best case, so-called embarrassingly parallel problems may realize speedup factors near the number of cores, or even more if the problem is split up enough to fit within each core's cache, avoiding the use of main system memory which is much slower. But this is not usual neither real, this problems exist mostly as laboratories examples.

Symmetric multiprocessing.

When two or more identical processors are connected via bus to a single shared main memory and are controlled by a single OS. This is a very challenging type of parallel programming because it requires two distinct modes of programming, one for the CPUs themselves and one for the interconnect between the CPUs. Main advantage comes because each processor can make different calculus, making possible to run in parallel different parts of the program at a time.

General Purpose Graphics Processor Units (GPGPU).

This is by far the newest paralelization technique and is the one that it is going to be used along this work. To implement a program which objective is to simulate a 3D grid with hp-Finite Elements. GPUs are coprocessors that have been heavily optimized for computer graphics processing. Computer graphics processing is a field dominated by data parallel operations—particularly linear algebra matrix operations. In the early days, GPGPU programs used the normal graphics APIs for executing programs. However, several new programming languages and platforms have been built to do general purpose computation on GPUs like CUDA for Nvidia GPUs and Stream SDK for ATI GPUs.

1.2 Why CUDA.

Gaming industry and high-definition 3D graphics application make the Graphic Processor Unit to evolve into a highly parallel, multithreaded, manycore processor with tremendous computational power and very high bandwidth. With the advent of multicore processors software pieces should now be written so as to exploit these resources as much as possible. This parallel programming era has arrived because of customers great demand on more and more applications on their PCs, laptops and on their portable gadgets. Users want better GUI (Graphics User Interface), HD quality video, faster virus scanners, real time network security systems, better realism in video games and faster access to data base. Moreover, the engineering and scientific community is, for example looking for deeper insights into the biological cells at molecular level.

Aiming the scope to scientific and engineering high performance application CUDA technology offers an extraordinary base line to develop high performance simulations involving calculations with hundreds of GFLOPS (Giga Floating Point Operations Per Second). Thanks to hundreds of cores in modern GPUs and software architecture, developers can think of exploiting these valuable resources and develop applications that run 100 times and even faster. It is remarkable to say that GPU architectures has the most advantage relation of GFLOP's per Dollar and also one of the best GFLOP's per watt all of this inside a small socket which gives one of the best ratio GFLPO's per square millimetre. This is incredibly important when talking about maintenance and cost of GPUs.

Thus there is a great pressure on designers to develop applications which should run many times faster than CPUs applications. Above all the compatibility with the C programming language turns the learning curve very steep, and the hardware abstraction provided by CUDA makes the programmer's life easier than ever before. The programmer does not need to be aware of the graphics APIs and can use C programming language to launch thousands of threads running in parallel on hundred's of cores.

The speed of the GPU is increasing at a a much higher rate as compared to the CPU (see below) making the GPUs as a co-processor for handling large number of calculations per second demanded by the customers.

1.2.1 GPU Efficiency.

GPUs are emerging as potential compute devices, with vendors such as Nvidia claiming orders of magnitude better performance than conventional solutions. In part, this advantage was due to rather different power budgets. GPUs plateaued at roughly 250-300W, whereas few commodity CPUs exceeded 130W. Similarly, high-end GPUs tended to max out the available area, while CPUs were constrained to less area. To make a reasonable comparison between different CPUs and GPUs analysis is focused on efficiency and normalized performance by the area and power consumption. Few years ago GPUs were immature the comparison showed that GPU and CPU were more or less at the same level of computing capabilities. Some GPUs were less efficient that CPUs on the same process technology while other GPUs were far more efficient.

The comparison now in 2012 is much more strait forward. GPUs are a much more acceptable part of the computing landscape and are at a midpoint in terms of maturity. Although there are still a number of factors to take into account.

- GPUs and CPUs typically require additional chips to make a complete system. To obtain a good efficiency factor in GPU a good CPU its needed, and to obtain a good efficiency in CPU a good bother board and RAM memory are essentials.
- GPU programming models are still restrictive compared to CPUs.
- GPU power often includes board-level components such as memory and VRMs; CPUs do not.
- Server processors tend to invest considerable power and area into coherency, memory capacity and large caches to enable scalability, whereas

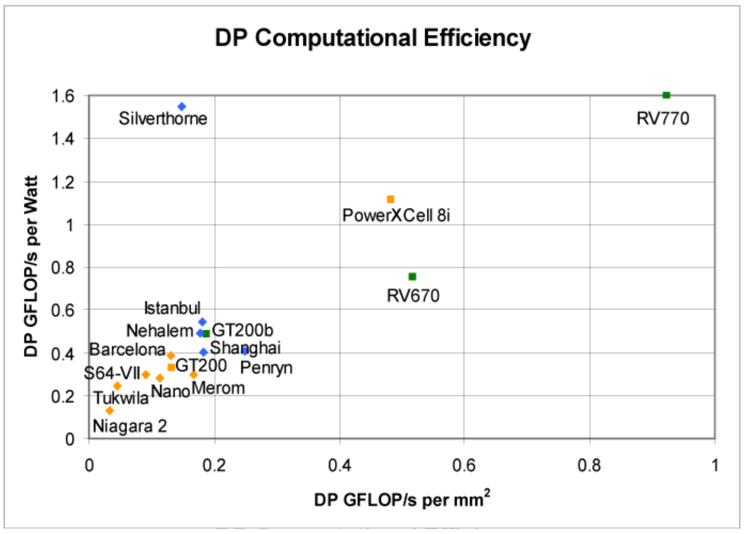

Figure 1.2: 2009 performance per watt and performance/mm2 of silicon for various CPUs and GPUs. To help make more sense of all this data, and highlight key differences, GPUs are marked with squares and CPUs with diamonds. 65nm products are shown in orange, 55nm in green and 45nm in blue.

client CPUs and GPUs do not.

- Most importantly, theoretical compute power does not translate into actual workload performance. CPUs have a much higher utilization than GPUs.

Looking at figure 1.3 we can see that the best throughput processor (Fermi) has a 68% area and 77% power advantage compared to the best CPU (Ivy Bridge), despite using an older process technology. And there is also remarkable IBM's Blue Gene/Q conclusively demonstrates that a CPU designed for throughput can match and even exceed the power efficiency of GPUs. There is still a gap in terms of area efficiency, but smaller than the data suggests given that Blue Gene/Q includes a large cache and robust interconnects that are not found in a GPU.

In late 2012 and early 2013, there should be a number of new products that change the overall picture. New processor technology will reach to GPUs with high performance 32/28nm microprocessors. Realistically, these new products should widen the gap between CPUs and throughput processors making super computing available to commodity PCs.

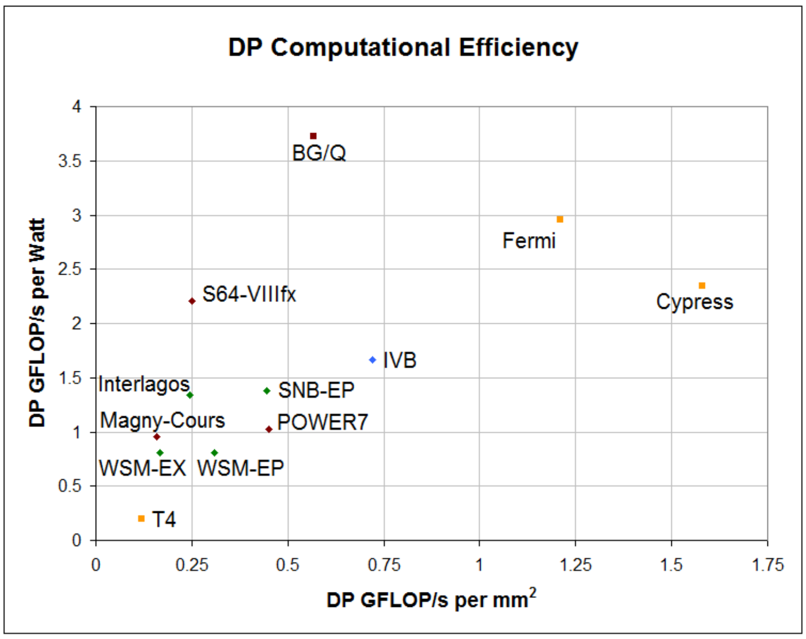

Figure 1.3: 2012 performance per watt and performance/mm2 of silicon for various CPUs and throughput devices. Conventional microprocessors are shown with diamonds, while squares denote throughput processors. The color indicates the process technology, with blue representing 22nm, yellow for 40nm and brown indicating 45nm.

1.2.2 Architecture & Programming Model.

In November 2006, NVIDIA introduced CUDA, a general purpose parallel computing architecture – with a new parallel programming model and instruction set architecture. That hastened the parallel computation within NVIDIA GPUs to solve many complex computational problems in a more efficient way than CPUs. This advantage comes from GPUs to be able to reach the Amhdal's law limit in highly parallelizable problems. As GPU are nowadays manufactured with an incredibly large number of cores. When developing the parallel code is important to make use of the largest possible number of cores because GPUs single cores are much slower than CPUs cores. Thus the performance lies in the number of GPU cores utilised.

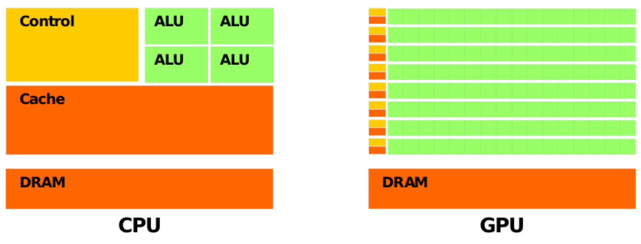

Figure 1.4: Basic structure of a typical CPU (left) and GPU (right).

CUDA programming model is based on C code and despite the differences it will be relatively familiar to C developers. CUDA C extends C by allowing the programmer to define C functions, called kernels. Kernels when called are executed N times in parallel by N different CUDA threads, as opposed to only once like regular C functions, figure 1.6 it is shown how the main program executes sequentially and when CUDA kernel it is called the program executes inside GPU device in parallel.

As it is possible to see kernels when executed create a grid of blocks. Which at the same time are a grid of threads. This is an important concept because a *thread is the minimum piece of code that is paralellizable, and it can not be divided.* Threads are closely related to Amhdal's law, and they dictate program's performance. The smaller the threads are, the better to GPU performance. As it has been said GPUs has an incredible large number of cores, this implies an incredibly larger number of threads that can be implemented at a time.

Blocks are grouped threads that execute in a single core inside the gpu. Threads inside a block are executed in warps of 32 threads at a time and can share data via Block's cache. Anyway it has to taken into account that in case of exceed cache memory the performance will be seriously affected.

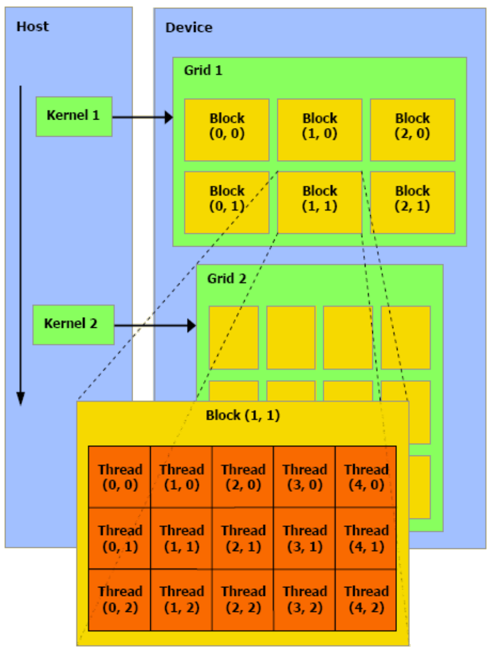

Figure 1.5: Host and device execution of GPU code.

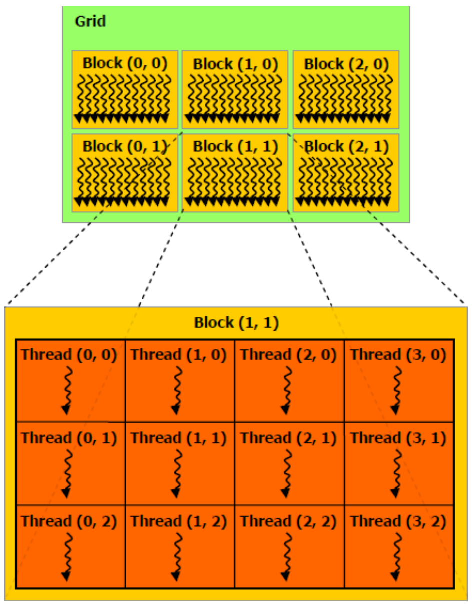

Figure 1.6: Execution inside a GPU device. Grid is divided in Blocks which are build of Threads that are executed in Warps of 32 Threads every clock cycle.

Coalesced Memory Access.

When the kernels are been executed Blocks and Threads have access to device global memory (figure 1.4). Memory access it has always being important both in CPU and in GPU to ensure efficient code. In GPU programming case this is a particular problem, to be handled with care.

A coalesced memory transaction is one in which all of the threads in a half-warp access global memory at the same time. This is oversimple, but the correct way to do it is just have consecutive threads access consecutive memory addresses.

So, if threads 0, 1, 2, and 3 read global memory 0x0, 0x4, 0x8, and 0xc, it should be a coalesced read.

In a matrix example, keep in mind that you want your matrix to reside linearly in memory. You can do this however you want, and your memory access should reflect how your matrix is laid out. So, the 3x4 matrix below.

0	1	2	3
4	5	6	7
8	9	a	b

could be done row after row, like this, so that (r, c) maps to memory $(r \cdot 4 + c)$.

0	1	2	3	4	5	6	7	8	9	a	b

Suppose you need to access element once, and say you have four threads. Which threads will be used for which element? Probably either.

thread 0:	0, 1, 2		thread 0:	0, 4, 8
thread 1:	3, 4, 5	or	thread 1:	1, 5, 9
thread 2:	6, 7, 8		thread 2:	2, 6, a
thread 3:	9, a, b		thread 3:	3, 7, b

Which is better? Which will result in coalesced reads, and which will not?

Either way, each thread makes three accesses. Let's look at the first access and see if the threads access memory consecutively. In the first option, the first access is 0, 3, 6, 9. Not consecutive, not coalesced. The second option, it's 0, 1, 2, 3. Thus is consecutive and therefore **coalesced**.

1.3 Finite Element Problem.

The physical concept on which the finite element method is based has its origins in the theory of structures. The idea of building up a structure by fitting together a number of structural elements was used in the early truss and framework analysis approaches employed in the design of bridges and buildings in the early 1900s. By knowing the characteristics of individual structural elements and combining them, the governing equations for the entire structure could be obtained. This process produces a set of simultaneous algebraic equations. The limitation on the number of equations that could be solved posed a severe restriction on the analysis. The introduction of the digital computer has made possible the solution of the large-order systems of equations.

Finite Element methods are based on differential equations to solve problems. Differential equations arise in many areas of science and technology, areas so disperse as classical mechanics, electromagnetism, fluid mechanics and basically any science and technology area. Differential equations strength rely on their capability to link together a physical variable and its variation. This simple concept makes this mathematical tool one of the most essential mathematical knowledge to every scientist and engineer.

Differential equations are mathematically studied from several different perspectives, mostly concerned to the set of functions that satisfy the equation. Only the simplest differential equations have explicit solution formulas. Moreover most of the systems that involve differential equations study do not have an exact solution form. When it is not possible to find an explicit solution it may be numerically approximated using computing techniques. The theory of dynamical systems puts emphasis on qualitative analysis of systems described by differential equations, while many numerical methods have been developed to determine solutions with a given degree of accuracy.

This makes the coupling of differential equations and high performance computing an incredible tool to approximate solutions in engineering problems. While the governing equations and boundary conditions can usually be written to these problems but difficulties introduced by irregular geometry or other discontinuities render the problems intractable analytically. To obtain a solution simplifying assumptions must be performed, reducing the problem to one that can be solved, or a numerically approximated. Numerical methods provide approximate values of the unknown quantity only at discrete points in the region. In the finite element method, the region of interest is divided up into numerous connected subregions or elements within which approximate functions which are usually polynomials and are used to represent the unknown quantity.

1.3.1 Poisson's Equation Variational Formulation.

$$(WP)\begin{cases} -\Delta u = f, & x \in \Omega\mathbb{R}^2 \\ u|_\Gamma = 0 \end{cases} \tag{1.3.1}$$

To solve the problem written in weak formulation to Poisson's equation it is necessary to find a function which Laplacian equals,

$$\Delta u = \frac{\partial^2 u}{\partial x_1^2} + \frac{\partial^2 u}{\partial x_1^2} \tag{1.3.2}$$

Using Green's formula.

$$\int_\Omega \Delta u \cdot v dx = -\int_\Omega \nabla u \cdot \nabla v dx + \int_\Gamma v \frac{\partial u}{\partial n} d\gamma \tag{1.3.3}$$

Where the gradient is equal

$$\begin{aligned} \Delta u &= \frac{\partial^2 u}{\partial x_1^2} + \frac{\partial^2 u}{\partial x_2^2} = u_{x_1 x_1} + u_{x_2 x_2} \\ \nabla u &= \left(\frac{\partial^2 u}{\partial x_1^2}, \frac{\partial^2 u}{\partial x_2^2} \right) = (u_{x_1}, u_{x_2}) \end{aligned} \tag{1.3.4}$$

The product of the gradients

$$\nabla u \cdot \nabla n = (u_{x_1}, u_{x_2})(v_{x_1}, v_{x_2}) = u_{x_1}, v_{x_1} + u_{x_2}, v_{x_2} \tag{1.3.5}$$

And the derivative respect to $\overrightarrow{n}$.

$$\frac{\partial u}{\partial n} = \nabla u \cdot n = u_{x_1} \cdot n_1 + u_{x_2} \cdot n_2 \tag{1.3.6}$$

to find a variational formulation, first a trial space is needed.

$$(VP) \begin{cases} v \in \Omega \mathbb{R} \\ \quad l^0(\Omega) \end{cases} \tag{1.3.7}$$

If we take a function and apply scalar product

$$\begin{gathered} -\int_\Omega (\Delta u) \cdot v dx = \int_\Omega f \cdot v dx; \forall v \in V \\ \text{Green's function} \\ \int_\Omega \nabla u \nabla v dx - \int_\Gamma v \frac{\partial u}{\partial n} d\gamma = \\ \int_\Omega \nabla u \nabla v dx - 0 = \end{gathered} \tag{1.3.8}$$

This trial space must satisfy continuity and boundary conditions $v_\Gamma = 0$. It is known that if a solution exist is unique.

Making the problem discontinuous, is equal as it is done in one dimension. A trial space V_h is needed, and this space has to satisfy.

$$V_h \subset V \;\; s.t. \;\; dim(V_h) < \infty \tag{1.3.9}$$

This space will have a base $\varphi_1, \varphi_2, \ldots, \varphi_M$ in V_h. As proved before the system could be proposed in two formulations, and is easier to solve in variational formulation. The steps needed to solve the problem, are.

1. Find a suitable base in the trial space.

$$u_n = \sum_{j=1}^{M} u_j \varphi_j(x) \tag{1.3.10}$$

2. Find the **stiffness matrix** and **load tensor**.

$$(VP_h)\begin{cases} u_n \in \mathbb{R} \\ a\left(\sum_{j=1}^{M} u_j \nabla\varphi_j(x), \nabla\varphi_i(x)\right) = L\left(\varphi_i(x)\right) i = 1, \dots, M \end{cases} \tag{1.3.11}$$

And operations needed to perform this task are.

$$\begin{aligned} \mathbf{A} &= (\varphi_j(x), \varphi_i(x)) & \int_\Omega \nabla\varphi_j \cdot \nabla\varphi_i \\ \mathbf{b} &= L(\varphi_i(x)) & \int_\Omega f \cdot \varphi_i dx \end{aligned} \tag{1.3.12}$$

1.3.2 hp-FEM.

hp-FEM is a general version of the finite element method FEM. This numerical method is based on polynomials approximations that makes use of elements of variable size h and polynomial degree p. The origins of hp-FEM date back to the pioneering work of Ivo Babuska who discovered that the finite element method converges exponentially fast when the mesh is refined using a suitable combination of h-refinements which is make by dividing elements into smaller ones and p-refinements increasing polynomial order in shape functions. This exponential convergence makes the method one of the best possible choice when implementing a numerical simulation.

hp-FEm efficiency relies on the capability of approximate functions with larger polynomial order or smaller piecewise-linear elements. This capability is also extended to all the elements inside the grid. And what is more important different elements may have different size h or different polynomial order approximation p and that is known as hp-adaptivity. hp-adaptivity as a combination of h-adaptivity splitting elements in space while keeping their polynomial degree fixed and p-adaptivity increasing their polynomial degree.

Related to work's problem, the code implemented must have this capability and be able to properly simulate different situations where h and p can vary as the user likes. Up to the first version polynomial approximation is fully implemented up to $9th$ order polynomials. And h implementation forces h to be equal to the three directions in space, making this way an homogeneous grid.

Chapter 2

Problem and Proposed Solutions.

The main goal of this article is to measure the ratio of execution times of two calculus algorithms for the same problem. One of them is a **C** code with **CPU** parallelelized implementation. The second code makes the same calculations inside a **General Purpose Graphics Processor Unit (GPGPU)**. Both codes runs under the same execution, as will be explained later in this work.

The code simulates a time static and homogeneous three dimensional grid. Distance between elements h and polynomial order approximation p is fixed at compilation time as well as the number of elements per side which will give the total number of elements. This allows to simulate different grid sizes and polynomial approximations to compare executions times.

Solutions to the main difficulties arisen when creating the algorithms will be now briefly discussed.

2.1 Transformed Space.

As seen in section 1.3.1. It is easier to solve the problem inside a transformed space. The space chosen is an homogeneous three dimensional cube in coordinates $[-1:1], [-1:1], [-1:1]$. This space has the advantage to perfectly fit the selected integration method which is **Gauss Quadrature** defined between $[-1:1]$. Gauss quadrature can be utilised in $[a:b]$ spaces, transforming that space. So the advantage to directly transform,

$$[x_a : x_b][y_a : y_b][z_a : z_b] \to [-1:1][-1:1][-1:1] \tag{2.1.1}$$

is to make only one space transformation. To transform space is easy in this particular case where the grid is cubic and homogeneous. Let x_c, y_c, z_c be the coordinates in the center of the element inside the weak formulation space.

Then the coordinates in variational formulation space to a point x, y, z, will be.

$$\xi = \frac{x - x_c}{h} \qquad \eta = \frac{y - y_c}{h} \qquad \zeta = \frac{z - z_c}{h} \tag{2.1.2}$$

Where ξ, η, ζ define the coordinates inside transformed space.

2.2 Shape Functions, Lagrange's Polynomials.

Shape functions choice it is not trivial. Shape functions must have value $\varphi_{ij} = 1$ when $i = j$ and have value $\varphi_{ij} = 0$ in any other case. Furthermore it is mandatory to take into account the calculus succession necessary to obtain a correct result.

The program implements Lagrange's polynomials as shape functions. Given a set of points x_i, y_i Lagrange polynomial is the polynomial of the least degree that at each point x_i assumes value y_i.

Given

$$(x_0, y_0), \ldots, (x_j, y_j), \ldots, (x_k, y_k) \tag{2.2.1}$$

where no two x_i are the same, the interpolation polynomial in the Lagrange form is a linear combination.

$$\begin{aligned} \ell_j(x) &:= \prod_{\substack{0 \le m \le k \\ m \ne j}} \frac{x - x_m}{x_j - x_m} \\ &= \frac{(x - x_0)}{(x_j - x_0)} \cdots \frac{(x - x_{j-1})}{(x_j - x_{j-1})} \frac{(x - x_{j+1})}{(x_j - x_{j+1})} \cdots \frac{(x - x_k)}{(x_j - x_k)} \end{aligned} \tag{2.2.2}$$

Given the initial assumption that no x_i are equals, every $x_i - x_j \ne 0$, so the expression it always properly defined. As requested in equation **??**, Lagrange's polynomials satisfy that particular condition.

For all $i \ne j$, $\ell_j(x)$ includes the term $(x - xi)$ so the numerator will be zero at $x = x_i$

$$\begin{aligned} \ell_{j \ne i}(x_i) &= \prod_{m \ne j} \frac{x_i - x_m}{x_j - x_m} \\ &= \frac{(x_i - x_0)}{(x_j - x_0)} \cdots \frac{(x_i - x_i)}{(x_j - x_i)} \cdots \frac{(x_i - x_k)}{(x_j - x_k)} = 0 \end{aligned} \tag{2.2.3}$$

On the other and, if $i = j$.

$$\ell_i(x_i) := \prod_{m \ne i} \frac{x_i - x_m}{x_i - x_m} = 1 \tag{2.2.4}$$

There is one more main reason to use Lagrange's polynomials. It is related to high performance computing. Shape functions must be hierarchical. When to obtain the $n - esime$ value of a polynomial is $f_n(x) = f_{n-1}(x) \cdot x$.

As an example let assume that we need to evaluate function $f(x)$. Then $f(x)$ is a hierarchical function if can be evaluated like.

$$\begin{aligned} f_1(x) &= 1 \\ f_2(x) &= x \\ f_3(x) &= x^2 \\ f_4(x) &= x^3 \end{aligned} \tag{2.2.5}$$

And continue evaluating until the polynomial ends. As can be seen any polynomial like, $f(x) = a + bx + cx^2 + \ldots + nx^{n+1}$ can be hierarchically computed. And there is a huge mistake on trying that approach in high performance computing, and at any computing problem in general. The fact is when approaching a polynomial result with that method x it is very likely to reach extreme values, zero or infinite. Inside our particular case. The program uses a transformed space between $[-1 : 1]$, and up to $9th$ order polynomials so succession x^n will reach zero value returning a wrong result. By using Lagrange's polynomials this issue does not appear because there is no more potency to be evaluated. Instead subtractions are evaluated and multiplied and x value is no longer modified. On the other and Lagrange's polynomials introduce a difficulty when its gradient is calculated. Because it is necessary to introduce an if condition which is not advisable at all, but it is a minor issue when compared to the evaluation point reaching to zero.

This is an easy and systematic method of generating shape functions of any order now can be achieved by simple products of Lagrange polynomials in the two or more coordinates. Thus, in three dimensions, if we label the node by its column and row number, I, J and K we have.

$$N_a \equiv N_{IJK} = l_I^n(\xi) l_J^m(\eta) l_K^p(\zeta) \tag{2.2.6}$$

where n, m and p stand for the number of subdivisions in each direction.

2.3 Gaussian Quadrature, integration method.

A quadrature rule is an approximation to a defined integral. To perform the integration task the program uses Gaussian Quadrature method. This method evaluates a weighted sum of the function evaluated in certain points. More specifically if the evaluating function is a polynomial, Gaussian quadrature reach an exact solution within $N = 5$ evaluation points. Gaussian quadrature to one dimension.

$$\int_{-1}^{1} f(x)dx \approx \sum_{i=1}^{N} w_i f(x_i) \tag{2.3.1}$$

As the program runs inside a three dimensional space, quadrature has to approximate a space in three dimensions.

$$\int_{-1}^{1} f(x)dx \int_{-1}^{1} f(y)dy \int_{-1}^{1} f(z)dz \approx \sum_{i=1}^{N}\sum_{j=1}^{N}\sum_{k=1}^{N} w_i w_j w_k f(x_i) f(y_j) f(z_k) \tag{2.3.2}$$

This integration method offers an exact solution to a polynomial when evaluated up to five points. So the program implements a five-point Gauss quadrature function to obtain the best possible solution. It should be noted the differences when implementing the calculus of elements a_{ij} and b_i. If we look equation 1.3.12 the differences between both implementations will be noticed. To implement b_i algorithm shape function is evaluated itself. But when performing a_{ij} calculus shape function is not evaluated. It is the two shape's functions gradient product to be evaluated. And as said earlier that particularity introduces an if condition which is not desirable but inevitable.

Despite the gradient calculus inconvenient Gaussian quadrature method offers a good efficiency to high performance computing and it is remarkable that Gaussian quadrature is an exact solution when evaluating polynomials.

Chapter 3

Code Implementation.

In chapter 2 several problems were discussed. This chapter the solutions used while implementing the code. Sections are dedicated to each of the main issues that have been found while working in this program.

3.1 Lagrange's polynomials.

Lagrange's polynomial theory was exposed in 2.2. To implement Lagrange's polynomials it is not a difficult task. As the grid point is known, it is direct to transform a given point and its nearest neighbours to transformed space. If we denote as k point to have value 1, to implement Lagrange's polynomials inside a three dimensional space three more points are needed. This points will have value 0 and will be denoted as $x0$, $y0$, $z0$.

```
for $p \leftarrow 0$ to $p - order$ do
    $temp \leftarrow 1.0;$
    $step \leftarrow 2.0/(p+1);$

    $lpol_x \leftarrow x0;$
    $lpol_y \leftarrow y0;$
    $lpol_z \leftarrow z0;$

    $temp* \leftarrow k_x - lpol_x;$
    $temp* \leftarrow k_x - lpol_y;$
    $temp* \leftarrow k_x - lpol_z;$

    for $i \leftarrow 0$ to $p + 12$ do
        $lpol_x \leftarrow k_x + i * step;$
        $lpol_y \leftarrow k_y + i * step;$
        $lpol_z \leftarrow k_z + i * step;$

        $temp* \leftarrow k_x - lpol_x;$
        $temp* \leftarrow k_y - lpol_y;$
        $temp* \leftarrow k_z - lpol_z;$
    end
    $lpol_{coef} \leftarrow temp;$
end
```

Algorithm 1: Lagrange's polynomials implementation in a three dimensional grid.

3.2 Integration.

To obtain the solution there are two instants where the code need to make an integration. This integrations are calculated with Gaussian quadrature because as commented this method gives an exact result with only 5 points of evaluation. And it is important to say that as the code has been developed, Gaussian quadrature can not be paralellized. This is a main issue to GPU but on the other hand if paralellized gauss quadrature uses a large amount of memory which is a limited resource in GPU computing.

3.2.1 b Integration.

Equation 1.3.12 show that **b** vector is obtained by integrating the known solution with the shape functions in each point. As the shape function is composed by one Lagrange's polynomial to each polynomial order. So when integrate to obtain **b** vector with a p polynomial approximation. Then p points will be obtained.

$$b = \begin{pmatrix} n_0 \begin{cases} p_0 \\ p_1 \\ \dots \\ p_{p-order} \end{cases} \\ n_1 \begin{cases} p_0 \\ p_1 \\ \dots \\ p_{p-order} \end{cases} \\ \dots \\ n_M \begin{cases} p_0 \\ p_1 \\ \dots \\ p_{p-order} \end{cases} \end{pmatrix}$$

So first loop need to go over the polynomial approximation order (*p-order*). And to each point the integration.

$$\mathbf{b} = \int_{\Omega} f \cdot \varphi_i dx \tag{3.2.1}$$

Known solution is expressed as a polynomial. Earlier in this article was said that using polynomials to approximate shapes functions was not a good choice, because the value could get extreme values very fast. But as this implementation it is only to explore the benefit of GPU computation. Known solution has been chosen to avoid extreme values despite of been a polynomial.

```
for $p \leftarrow 0$ to $p - order$ do
    for $i \leftarrow 0$ to $5$ do
        $temp \leftarrow 1.0$ ;
        $temp_x \leftarrow 0.0$;
        for $j \leftarrow 0$ to $KnownXsize$ do
            $temp_x + \leftarrow temp \cdot knownSolX[j]$;
            $temp* \leftarrow gauss_{point}[i]$;
        end
        $temp \leftarrow 1.0$ ;
        $temp_y \leftarrow 0.0$;
        for $j \leftarrow 0$ to $KnownYsize$ do
            $temp_y + \leftarrow temp \cdot knownSolY[j]$;
            $temp* \leftarrow gauss_{point}[i]$;
        end
        $temp \leftarrow 1.0$ ;
        $temp_z \leftarrow 0.0$;
        for $j \leftarrow 0$ to $KnownZsize$ do
            $temp_z + \leftarrow temp \cdot knownSolZ[j]$;
            $temp* \leftarrow gauss_{point}[i]$;
        end
        for $j \leftarrow 0$ to $p$ do
            $temp_x \leftarrow gauss_{point}[i] - lpol_x[j]$;
            $temp_y \leftarrow gauss_{point}[i] - lpol_y[j]$;
            $temp_z \leftarrow gauss_{point}[i] - lpol_z[j]$;
        end
        $temp_x* \leftarrow gauss_{coef}[i]$;
        $temp_y* \leftarrow gauss_{coef}[i]$;
        $temp_z* \leftarrow gauss_{coef}[i]$;
        $result_x + \leftarrow temp_x$;
        $result_y + \leftarrow temp_y$;
        $result_z + \leftarrow temp_z$;
    end
    $b_n(p) \leftarrow \frac{result_x \cdot result_y \cdot result_z}{lpol_{denominator}}$
end
```

Algorithm 2: Calcule **b** vector single element.

3.2.2 A Integration.

As in previous section, equation 1.3.12 which is the necessary calculus to obtain each matrix element a_{ij}. **A** is a diagonal matrix made up by boxes. This boxes size is equal to polynomial approximation order $(p-orderxp-order)$. This way the interaction between one point and its nearest neighbours is calculated to every polynomial combination.

$$a_{ij} = \int_{\Omega} \nabla\varphi_j \cdot \nabla\varphi_i \tag{3.2.2}$$

$$A = \left(\begin{array}{ccccc}
 & n_0 & n_1 & \dots & n_M \\
n_0 & \left\{\begin{array}{lcccc} & p_0 & p_1 & \dots & p_{p-order} \\ p_0 & a^{00}_{00} & a^{00}_{01} & & a^{00}_{0p} \\ p_1 & a^{00}_{01} & a^{00}_{11} & & a^{00}_{1p} \\ \dots & & & & \\ p_{p-order} & a^{00}_{0p} & a^{00}_{1p} & & a^{00}_{pp} \end{array}\right. & \dots & \dots & \dots \\
n_1 & \left\{\begin{array}{lcccc} & p_0 & p_1 & \dots & p_{p-order} \\ p_0 & a^{01}_{00} & a^{01}_{01} & & a^{01}_{0p} \\ p_1 & a^{01}_{01} & a^{01}_{11} & & a^{01}_{1p} \\ \dots & & & & \\ p_{p-order} & a^{01}_{0p} & a^{01}_{1p} & & a^{01}_{pp} \end{array}\right. & \dots & \dots & \dots \\
\dots & \dots & \dots & \dots & \dots \\
n_M & \left\{\begin{array}{lcccc} & p_0 & p_1 & \dots & p_{p-order} \\ p_0 & a^{0M}_{00} & a^{0M}_{01} & & a^{0M}_{0p} \\ p_1 & a^{0M}_{01} & a^{0M}_{11} & & a^{0M}_{1p} \\ \dots & & & & \\ p_{p-order} & a^{0M}_{0p} & a^{0M}_{1p} & & a^{0M}_{pp} \end{array}\right. & \dots & \dots & \dots
\end{array}\right)$$

Before get into integration calculus it is advisable to perform an algorithm who obtain the gradient of a given Lagrange's polynomial. Lets assume that given polynomial has order p, then the algorithm will be. Let assume that p equals 3 in this case.

$$\begin{aligned}
\varphi &= \frac{1}{C}(x - x_1)(x - x_2)(x - x_3) \\
&\quad (y - y_1)(y - y_2)(y - y_3) \\
&\quad (z - z_1)(z - z_2)(z - z_3) \\
\nabla\varphi &= \frac{1}{C}\left\{(x - x_2)(x - x_3) + (x - x_1)(x - x_3) + (x - x_1)(x - x_{M-1})\right\} \cdot l_y l_z \\
&\quad \left\{(y - y_2)(y - y_3) + (y - y_1)(y - y_3) + (y - y_1)(y - y_{M-1})\right\} \cdot l_x l_z \\
&\quad \left\{(z - z_2)(z - z_3) + (z - z_1)(z - z_3) + (z - z_1)(z - z_{M-1})\right\} \cdot l_x l_y
\end{aligned} \tag{3.2.3}$$

```
$grad_x \leftarrow 1.0$;
$grad_y \leftarrow 1.0$;
$grad_z \leftarrow 1.0$;
for $i \leftarrow 0$ to $p+1$ do
    for $j \leftarrow 0$ to $p+1$ do
        if $i \neq j$ then
            $grad_x[i]* \leftarrow gauss_x[point] - lpol_x[j]$;
            $grad_y[i]* \leftarrow gauss_x[point] - lpol_y[j]$;
            $grad_z[i]* \leftarrow gauss_x[point] - lpol_z[j]$;
        end
    end
end
```

Algorithm 3: Calculate the gradient of a Lagrange polynomial *lpol* at a given integration point inside gauss quadrature *point*.

Now it is time to calculate the gauss quadrature. When two polynomials $\varphi_i^n \varphi_j^m$ interact, their interaction goes to a point in the matrix **A** this point is inside a box which represents the interaction between element $N_i N_j$. The box is a square matrix of side equal to the polynomial approximation order $p - order$. And element a_{ij}^{nm} represents the interaction between element i

and j, when evaluating shape functions n and m.

```
for $p - order1 \leftarrow 0$ to $p - order$ do
    for $p - order2 \leftarrow 0$ to $p - order$ do
        for $i \leftarrow 0$ to 125 do
            $derivate_x \leftarrow 0;$
            $derivate_y \leftarrow 0;$
            $derivate_z \leftarrow 0;$

            $grad_1 \leftarrow calculategradientoflpol_1;$
            $grad_2 \leftarrow calculategradientoflpol_2;$

            for $p1 \leftarrow 0$ to $p - order1$ do
                for $p2 \leftarrow 0$ to $p - order2$ do
                    $derivate_x + \leftarrow grad_x[p1] - grad_x[p2];$
                    $derivate_y + \leftarrow grad_y[p1] - grad_y[p2];$
                    $derivate_z + \leftarrow grad_z[p1] - grad_z[p2];$
                end
            end
            for $p1 \leftarrow 0$ to $p - order1$ do
                $derivate_x * \leftarrow gauss_x[i] - lpol1_y[p1];$
                $derivate_x * \leftarrow gauss_x[i] - lpol1_z[p1];$

                $derivate_y * \leftarrow gauss_x[i] - lpol1_x[p1];$
                $derivate_y * \leftarrow gauss_x[i] - lpol1_z[p1];$

                $derivate_z * \leftarrow gauss_x[i] - lpol1_x[p1];$
                $derivate_z * \leftarrow gauss_x[i] - lpol1_y[p1];$
            end
            for $p2 \leftarrow 0$ to $p - order2$ do
                $derivate_x * \leftarrow gauss_x[i] - lpol2_y[p2];$
                $derivate_x * \leftarrow gauss_x[i] - lpol2_z[p2];$

                $derivate_y * \leftarrow gauss_x[i] - lpol2_x[p2];$
                $derivate_y * \leftarrow gauss_x[i] - lpol2_z[p2];$

                $derivate_z * \leftarrow gauss_x[i] - lpol2_x[p2];$
                $derivate_z * \leftarrow gauss_x[i] - lpol2_y[p2];$
            end

            $derivate_x * \leftarrow gauss_{point}[i];$
            $derivate_y * \leftarrow gauss_{point}[i];$
            $derivate_z * \leftarrow gauss_{point}[i];$
            $result \leftarrow \frac{erivate_x \cdot derivate_y \cdot derivate_z}{lpolCoef_1 \cdot lpolCoef_2};$
        end
        $a_{p-order1} p - order2 \leftarrow result;$
        $a_{p-order2} p - order1 \leftarrow result;$
    end
end
```

Algorithm 4: Gaussian quadrature for two Lagrange's polynomias gradient in a 3D grid.

3.3 System integration.

Once we have build the different algorithms needed to solve the system it is time to develop a piece of code that uses those algorithms to solve the problem proposed in equation 1.3.12. It is important to remark that **A** matrix is made by boxes. Each row is formed by several boxes which represents the interaction between two element. Inside a row we will find the box that represents the interaction with the element itself, and the boxes who represents the element interaction with its nearest neighbours.

```
for $idx \leftarrow 0$ to $Number\ of\ Elements$ do
    $tSpace \leftarrow Get\ Transformed\ space\ associated\ to\ idx$;
    for $li \leftarrow 0$ to $4$ do
        $lpol \leftarrow Get\ Lagrange's\ polynomials$;
    end
    $b \leftarrow Integrate\ tSpace_0\ with\ lpol_0$;
    $A_0 \leftarrow Integrate\ tSpace_0\ with\ lpol_0$;
    for $li \leftarrow 1$ to $4$ do
        $A_{li} \leftarrow Integrate\ tSpace_{li}\ lpol_{li}$;
    end
end
```

Algorithm 5: Calculate **A** matrix and **b** vector. **A** matrix is made of boxes

3.4 Matrix Assembly.

Before facing system solving, it is necessary to assembly **A** matrix. Already we have the values of the integrations but they are not properly distributed. As mentioned before **A** is formed by boxes, this step turns this box-formed matrix into a Compressed Storage Row (CRS) matrix. The main problem to perform this task is to handle memory directions properly. There are several ways to do it and in this work I have chosen a solution that implies look for every value twice and write it once. This solutions has been taken because reading memory is much faster than writing.

```
for $idx \leftarrow 0$ to $Number\ of\ Elements$ do
    $Matrix\ is\ copied\ row\ by\ row$ for $p \leftarrow 0$ to $p - order$ do
        for $i \leftarrow 0$ to $Boxes\ Behind\ the\ diagonal$ do
            for $j \leftarrow 0$ to $p - order$ do
                $CRS[cont] \leftarrow A[(idx - (i + 1)) * p - order^2 * 4) + (i * p - order^2) + (p * p - order) + j;$
                $cont + +;$
            end
        end
        for $j \leftarrow 0$ to $p - order$ do
            $CRS[cont] \leftarrow A[idx * p - order^2 * 4) + (p * p - order) + j;$
            $cont + +;$
        end
        for $li \leftarrow 1$ to $Boxes\ after\ the\ diagonal$ do
            for $j \leftarrow 0$ to $p - order$ do
                $CRS[cont] \leftarrow A[idx*p-order^2*4)+(i*p-order^2)+(p*p-order)+j;$
                $cont + +;$
            end
        end
    end
end
```
Algorithm 6: **A** matrix assembly into Compressed Row Storage format.

Chapter 4

Results.

Before expose and discuss obtained results is important to describe the hardware equipment that is going to be used. The program will run in a home computer with a CPU *AMD Phenom(tm) II X6 1090T Processor* which has a top frequency of *3.3Mhz*. Mother board has 8GB as RAM memory. And finally the GPU which is an *GeForce GTX 550 Ti* Nvidia graphics card, all of this running under *Linux Ubuntu 12.04*. This equipment presents two direct problem, given the fact that CPU is much better than GPU. And the fact that this GPU can only operate in *float* which is single precision with no have value in engineering world. Anyway this work is about compare the rate between CPU and GPU executions.

Previous chapters have widely talked about GPU its structure and its programming model. So the results are presented directly. Figure 4 represents the time required to execute integration process. Figure 4 represents the time taken to transform **A** from boxess to CRS format. Lastly Figure 4 represents the ration between the executions time. Every execution has been made with $h = 5$ and it is the number of elements and the shape function polynomial approximation the variables that are handled. This may not appear a good solution, but after many executions parameter h was almost irrelevant when measuring time execution.

Polynomial order approximation has chosen to be 3 and 6. Higher polynomials order make the program to fully occupy device and cause problems to the OS graphics environment which is running inside the same GPU so no data could be acquired to higher polynomial approximation order in this precise equipment.

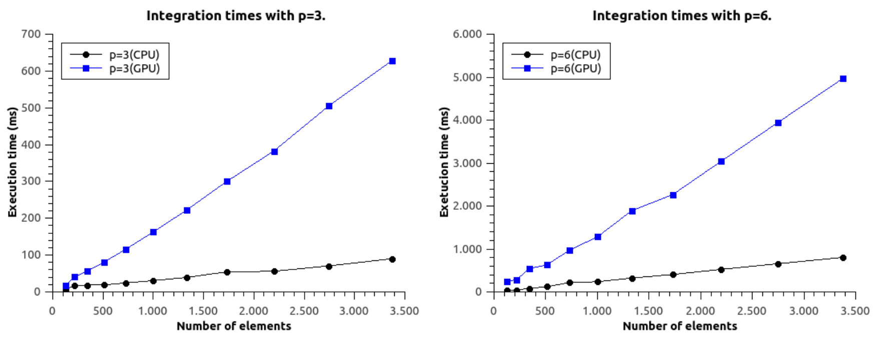

Figure 4.1: This chart represents executions times to different grid sizes. As we can see the GPU implementation it is not as god as it should. And it runs much slower that parallel CPU.

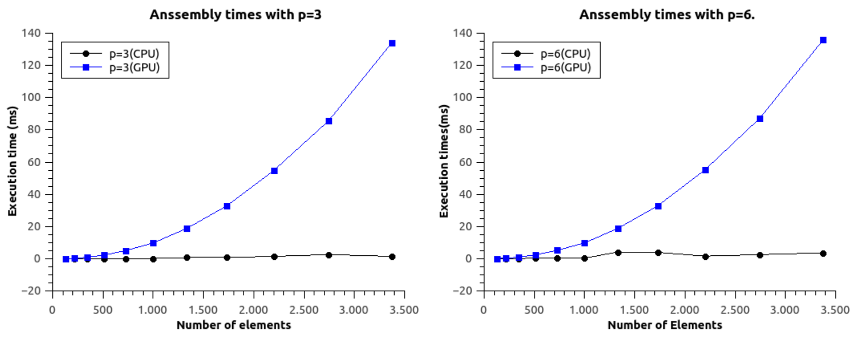

Figure 4.2: Once more CPU parallel performance outcomes when task involves reading and writing. In this particular case the incredible difficulty of making **A** vectors coalesced severely penalizes the GPU.

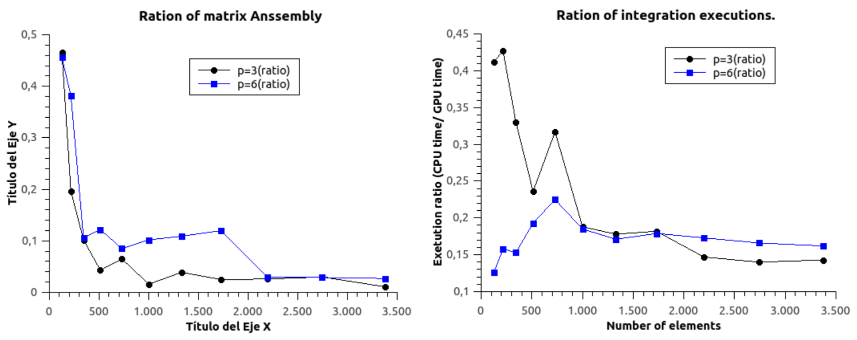

Figure 4.3: This particular chart express the time ratios between the different executions that have been made.

Chapter 5

Conclusions And Future Work.

This work represents first step towards the implementation of a fully operational hp-FEM solver. It is true that in this work results does not support the idea of a functional hp-FEM solver but despite the poor results exist are a number of factors that worth to analyse. CUDA is a new programming language and a new programming structure this translates into a huge lack of support. There are few specialized books and even few of those books are oriented to high performance computation. Most of those CUDA specialized books address issues related to graphic user interface and image processing improvement not engineering problems. In conjunction with this first problem author's CUDA inexperience surely has derived into a poorly optimised implementation. As it is easy to see in figure 4 vectors are clearly not coalescence which is explained in section 1.2.2. When it is referenced to integration problem it is Amhdal's law who has tricked the results. Implemented algorithm is good enough to have parallelism in 6 cores as CPU has. Problem appear when the number of cores grown but the velocity of themselves descends. And with that number of cores a completely new algorithm is needed.

The final conclusion about CUDA is the long road of development that still has this new programming language and the bright future is GPU computing. Although in this particular job performance was not achieved expected there countless studies have shown that the feasibility of code implentación graphics cards in order to make high performance computing.

Acknowledgment.

I want to thank and dedicate this work to all the people who have been close to me during the making of it. To my parents, to Rocio, Raúl and Javi. BIFI's people specially sysadmins that gave full support and access to their computers. I'm specially grateful to Asier Lacasta without your help it would have been impossible to do this job. Finally I want to thank David Pardo and Ricardo Celorrio by guide me with this difficult work.

Bibliography

[1] Jens Krüger and Rüdiger Westermann, *Linear Algebra Operators for GPU Implementation of Numerical Algorithms.*
Computer Graphics and Visualization Group, Technical University Munich, 10.1.1.1.3310.

[2] K. Fatahalian, J. Sugerman, and P. Hanrahan,
Understanding the Efficiency of GPU Algorithms for Matrix-Matrix Multiplication.
Graphics Hardware (2004), Stanford University
10.1.1.1.6823.

[3] Stephanie Winner *, Mike Kelley *** , Brent Pease **, Bill Rivard*, and Alex Yen ***
Hardware Accelerated Rendering Of Antialiasing Using A Modified A-buffer Algorithm.
* 3Dfx Interactive, San Jose, CA USA,
** Bungie West, San Jose, CA USA,
*** Silicon Graphics Computer Systems, Mountain View, CA USA
10.1.1.46.6965

[4] Andreas Schilling,
A New Simple and Efficient Antialiasing with Subpixel Masks.
Computer Graphics, vol. 25, no. 4, July 1991 (SIGGRAPH '91 Proceedings), pp. 133–141
10.1.1.59.3971

[5] Jeff Bolz, Ian Farmer, Eitan Grinspun, Peter Schröder
Sparse Matrix Solvers on the GPU: Conjugate Gradients and Multigrid.
Caltech
10.1.1.112.5723

[6] Tor Dokken Trond, R. Hagen Jon, M. Hjelmervik
The GPU as a high performance computational resource.
SINTEF ICT, Applied Mathematics P.O. Box 124 Blindern 0314 Oslo, Norway
10.1.1.133.5648

[7] Ian Buck, Tim Foley, Daniel Horn, Jeremy Sugerman, Kayvon Fatahalian, Pat Hanrahan
Brook for GPUs: Stream Computing on Graphics Hardware.
Graphics Hardware, Stanford University
10.1.1.135.5772

[8] Eun-Jin Im *, Katherine Yelick **, Richard Vuduc **
SPARSITY: Optimization Framework for Sparse Matrix Kernels.
* School of Computer Science, Kookmin University, Seoul, Korea
** Computer Science Division, University of California, Berkeley
10.1.1.137.5844

[9] CRAIG C. DOUGLAS * AND HYOSEOP LEE **
AN ALGEBRAIC MULTIGRID BASED PRECONDITIONER FOR THE LAPLACE TRANSFORMATION METHOD.
* School of Energy Resources and Mathematics Department, University of Wyoming, Laramie, Wyoming
** Mathematics Department, University of Wyoming, Laramie, Wyoming
10.1.1.185.724

[10] Tomáš Oberhuber 2, Atsushi Suzuki 4 , Jan Vacata 2
New Row-grouped CSR format for storing sparse matrices on GPU with implementation in CUDA.
2 Department of mathematics, Faculty of Nuclear Sciences and Physical Engineering, Czech Technical University in Prague, Trojanova 13, 120 00 Praha 2, Czech Republic
4 Laboratoire Jacques-Louis Lions, Universite Pierre et Marie Curie, Boîte courrier 187, 75252 Paris Cedex 05, France

[11] John D. Owens 1 , David Luebke 2 , Naga Govindaraju 3 , Mark Harris 2 , Jens Krüger 4 , Aaron E. Lefohn5 and Timothy J. Purcell 2
A Survey of General-Purpose Computation on Graphics Hardware.
COMPUTER GRAPHICS forum. Volume 26 (2007), number 1 pp. 80–113
1 University of California, Davis, USA
2 NVIDIA
3 Many-core Technology Incubation Group, Microsoft Corporation
4 Technische Universität München
5 Neoptica

[12] David Tarditi, Sidd Puri, Jose Oglesby
Accelerator: Using Data Parallelism to Program GPUs for General-Purpose Uses
COMPUTER GRAPHICS forum. Volume 26 (2007), number 1 pp. 80–113

Microsoft Research

[13] Erich Elsen, Vijay Pande, V. Vishal, Mike Houston, Pat Hanrahan, Eric Darve
N-Body Simulations on GPUs.
Stanford University
arXiv:0706.3060v1

[14] Anthony Danalis 1 2, Gabriel Marin 1, Collin McCurdy 1, Jeremy S. Meredith 1, Philip C. Roth 1, Kyle Spafford 1, Vinod Tipparaju 1, Jeffrey S. Vetter 1
The Scalable Heterogeneous Computing (SHOC) benchmark suite.
1 Computer Science & Mathematics Division Oak Ridge National Laboratory, Oak Ridge.
2 Department of Computer Science University of Tennessee Knoxville.
ISBN: 978-1-60558-935-0 doi>10.1145/1735688.1735702

[15] Gundolf Haase 1 , Manfred Liebmann 1 2 , Craig C. Douglas 2 , and Gernot Plank 3
A Parallel Algebraic Multigrid Solver on Graphics Processing Units.
1 Institute for Mathematics and Scientific Computing, University of Graz
2 Department of Mathematics, University of Wyoming
3 Computing Laboratory, Oxford University

[16] Zbigniew Koza 1, Maciej Matyka 1, Sebastian Szkoda 1, Lukasz Miroslaw 2 3
Compressed Multiple-Row Storage Format.
1 Faculty of Physics and Astronomy, University of Wroclaw, pl. M. Borna 9, 50-205 Wroclaw, Poland
2 Vratis Ltd., Wroclaw, Muchoborska 18, Poland
3 nstitute of Informatics, Wroclaw University of Technology, Poland

[17] Craig Couglas & David Roylance
Reactive Systems - Finite Element Analysis.
Department of Materials Science and Engineering, Massachusetts Institute of Technology, Cambridge, MA 02139

[18] Lesa Aylward, Craig Couglas and David Roylance
A transiest finite element model for pultrusion processing.
Department of Materials Science and Engineering, Massachusetts Institute of Technology, Cambridge, MA 02139

[19] William B. Langdon
Debugging CUDA.

Dept. of Computer Science, University College London Gower Street, WC1E 6BT, UK
GECCO Companion, CIGPU 2011 workshop, Simon Harding et al. Eds., p415–422.

[20] Jonathan M. Cohen *, M. Jeroen Molemaker**
A Fast Double Precision CFD Code using CUDA.
*NVIDIA Corporation, Santa Clara, CA 95050, USA
**IGPP UCLA, Los Angeles, CA 90095, USA

[21] Wilson W. L. Fung, Ivan Sham, George Yuan, Tor M. Aamodt
Dynamic Warp Formation and Scheduling for Efficient GPU Control Flow.
Department of Electrical and Computer Engineering University of British Columbia, Vancouver, BC, CANADA

[22] Graham M ARKALL
Accelerating Unstructured Mesh Computational Fluid Dynamics on the NVidia Tesla GPU Architecture .
I MPERIAL C OLLEGE L ONDON MS C . A DVANCED C OMPUTING ISO 1

[23] Ahmed Al Maashri, Guangyu Sun, Xiangyu Dong, Vijay Narayanan and Yuan Xie
3D GPU Architecture using Cache Stacking: Performance, Cost, Power and Thermal analysis.
Department of Computer Science and Engineering, Penn State University

[24] By John D. Owens, Mike Houston, David Luebke, Simon Green, John E. Stone, and James C. Phillips
GPU Computing.
Vol. 96, No. 5, May 2008 p(879-899) | Proceedings of the IEEE

[25] David Luebke 1, Greg Humphreys 2
How GPUs Work.
1 NVIDIA Research
2 University of Virginia

[26] Ali Bakhoda, George L. Yuan, Wilson W. L. Fung, Henry Wong and Tor M. Aamodt
Analyzing CUDA Workloads Using a Detailed GPU Simulator.
University of British Columbia, Vancouver, BC, Canada.

[27] Nathan Bell, Michael Garland
Implementing Sparse Matrix-Vector Multiplication on Throughput-

Oriented Processors.
NVIDIA Research.

[28] Nathan Whitehead, Alex Fit-Florea
Precision & Performance: Floating Point and IEEE 754 Compliance for NVIDIA GPUs.
NVIDIA Research. 2011

[29] Erik Lindholm, John Nickolls, Stuart Oberman, John Montrym
NVIDIA TESLA: A UNIFIED GRAPHICS AND COMPUTING ARCHITECTURE.
NVIDIA Research.0272-1732/08/ 2008 IEEE

[30] Shane Ryoo† 1, Sam S. Stone 1, Christopher I. Rodrigues 1, Sara S. Baghsorkhi 1, David B. Kirk 2, Wen-mei W. Hwu 1
Optimization Principles and Application Performance Evaluation of a Multithreaded GPU Using CUDA.
1 Center for Reliable and High-Performance Computing, University of Illinois at Urbana-Champaign
2 NVIDIA Corporation

[31] Shane Ryoo, Christopher I. Rodrigues, Sam S. Stone, Sara S. Baghsorkhi, Sain-Zee Ueng, John A. Stratton, and Wen-mei W. Hwu
Program Optimization Space Pruning for a Multithreaded GPU.
Center for Reliable and High-Performance Computing University of Illinois at Urbana-Champaign
2008 ACM 978-1-59593-978-4/08/04

[32] John Nickolls 1, Ian Buck 1, Michael Garland 1 and Kevin Skadron 2
Scalable Parallel PROGRAMMING.
1 NVIDIA
2 Universito of Virginia

[33] R.L. Taylor 1 and J.Z. Zhu 2
The Finite Element Method: Its Basis and Fundamentals.
1 Professor in the Graduate School Department of Civil and Environmental Engineering University of California at Berkeley Berkeley, California
2 Senior Scientist ESI US R & D Inc. 5850 Waterloo Road, Suite 140 Columbia, Maryland

[34] Nathan Bell and Jared Hoberock
Thrust: A Productivity-Oriented Library for CUDA.
HWU 2011 360 Ch26 2011

[35] NVIDIA
The CUDA Compiler Driver NVCC.

[36] Nathan Bell and Michael Garland
Efficient Sparse Matrix-Vector Multiplication on CUDA.
NVIDIA Technical Report NVR-2008-004, Dec. 2008.

[37] Jonathan Richard Shewchuk
An Introduction to the Conjugate Gradient Method Without the Agonizing Pain.
(1994) School of Computer Science Carnegie Mellon University Pittsburgh,

[38] Jens Breitbart
CuPP – A framework for easy CUDA integration.
Research Group Programming Languages / Methodologies Universität Kassel a Kassel, Germany
978-1-4244-3750-4/09 (2009)

[39] Verschoor, M. and Jalba, A.C.
Elastically Deformable Models based on the Finite Element Method Accelerated on Graphics Hardware using CUDA.
2012

[40] Kandasamy, Vishnukanthan
Parallel FEM Simulation Using GPUs.
2011

[41] Wozniak, M. and Olas, T. and Wyrzykowski, R.
Parallel implementation of conjugate gradient method on graphics processors.
Parallel Processing and Applied Mathematics, pages 125–135, year 2010,

[42] C.K.Filelis-PapadopoulosandG.A.GravvanisandP.I.MatskanidisandK.M.Giannoutakis
"OntheGPGPUparallelizationissuesoffiniteelementapproximateinversepreconditioning",
JournalofComputationalandAppliedMathematics (2011)
DOI:10.1016/j.cam.2011.07.016
http://sciencedirect.dogsoso.com/science/article/pii/S0377042711004110

[43] Plaszewski, Przemyslaw and Banas, Krzysztof and Maciol, Pawel
Higher order FEM numerical integration on GPUs with OpenCL.
Proceedings of the 2010 International Multiconference on Computer Science and Information Technology (IMCSIT), (2010)

[44] Oberhuber T., Suzuki A., Vacata J.
New Row-grouped CSR format for storing the sparse matrices on GPU with implementation in CUDA.
Acta Technica 56: 447-466, 2011

[45] Dziekonski, A. and Sypek, P. and Lamecki, A. and Mrozowski, M.
Finite Element Matrix Generation on a GPU.
(2012)

[46] Markus Geveler and Dirk Ribbrock and Dominik G"oddeke and Peter Zajac and Stefan Turek
Towards a complete FEM-based simulation toolkit on GPUs: Geometric Multigrid solvers.
23rd International Conference on Parallel Computational Fluid Dynamics (2011)

[47] Dziekonski, A. and Lamecki, A. and Mrozowski, M.
A Memory Efficient and Fast Sparse Matrix Vector Product on a Gpu.
Progress In Electromagnetics Research vol(116), pag(49–63) 2011

[48] Liu, K. and Wang, X. and Zhang, Y. and Liao, C.
Acceleration of time-domain finite element method (TD-FEM) using Graphics Processor Units (GPU).
, Antennas, Propagation & EM Theory, 2006. ISAPE'06. 7th International Symposium on pages(1–4) IEEE 2006

[49] Cecka, C. and Lew, A. and Darve, E.
Application of Assembly of Finite Element Methods on Graphics Processors for Real-Time Elastodynamics.
, year 2010

[50] Dominik G“oddeke and Robert Strzodka and Stefan Turek
Accelerating Double Precision FEM Simulations with GPUs.
18th Symposium Simulationstechnique (ASIM'05) (2005) pages(139-144)