CUDA implementation of the solution of a system of linear equations arising in an hp-Finite Element code.

Author: Javier Osés Villanueva [1+],
Directors: David Pardo [2*], and Ricardo Celorrio[3+]

[*]Departamento de Matemática Aplicada y Estadística e I.O.,
University of the Basque Country UPV/EHU, and Ikerbasque.
[+]Departamento de Matemática Aplicada, University of Zaragoza.

February 1, 2013

[1]javieroses85@gmail.com
[2]dzubiaur@gmail.com
[3]celorrio@unizar.es

Abstract

The FEM has proven to be one of the most efficient methods for solving differential equations. Designed to run on different computer architectures, technological improvements have led over the years to the fast solution of larger and larger problems. Among these technological improvements, we emphasize the development of GPU (Graphic Processor Unit).

Scientific programming in graphics cards was extremely difficult until 2006 the company NVIDIA developed CUDA (Compute Unified Device Architecture). It is a programming language designed for generic computing which does not require knowledge of traditional graphics programming.

GPUs are capable of performing a large number of operations simultaneously. This capability makes them very attractive for use in FEM. One of the parts of the FEM which requires large computational capacity is the solution of systems of linear equations.

In this work, an algorithm for solving systems of linear equations in CUDA has been implemented. It will be applied as a part of a hp-FEM code that tries to solve Laplace equation. The aim of this study is to compare the performance of an an implementation of a solver in CUDA vs. a C implementation and check if CUDA has advantages over traditional programming.

For that purpose, we select an algorithm suitable for GPU programming. The iterative algorithms have properties that fits to CUDA programming architecture. However, the use of these algorithms require from double precision arithmetic to minimize round-off effects. Nowadays, only high performance GPUs are able to work in double precision.

FEM matrices are sparse and the use of compression format for the system matrix is needed. Exist multiple compression formats and we select one which better fits to the matrix structure that FEM generates in our problem.

The implementation in CUDA introduces improvements in execution times compared to traditional programming in C. Recent works has proved that it can be obtained programs that works until 80 times faster. But, this result can not be generalized because the improvements depends on differential equation, boundary conditions, mesh generation, FEM, model of GPU, version of CUDA(now 5.0), and of course implementation.

Contents

Chapter 1

Introduction

The finite element method FEM is a numerical technique for finding approximate solutions of partial differential equations associated with physical problems on different types of geometry. Since its inception in 1950 until today, the use of FEM has spread of continuum mechanics to many fields such as heat transfer, fluid mechanics, and even the study of biological systems.

The FEM converts a problem defined in terms of linear second order partial differential equations in a linear system of equations. For a basic introduction on FEM, see [53] [28]. Currently exist different types of FEM, including:

AEM(Applied Element Method)

GFEM(Generalized Finite Element Method)

hp-FEM

hpk-FEM

XFEM(Extended finite element method)

S-FEM(Smoothed finite element method)

Spectral methods

Meshfree methods

Discontinuous Galerkin methods

Finite element limit analysis

Stretched grid method.

In this master thesis, we chose hp-FEM [12]. Although its computational implementation is somewhat challenging, it is currently one of the most efficient methods due to its adaptability and the fast convergence of their solutions.

From a computational point of view, the problem with FEM solution consists of the following tasks:

1.- Pre-processing:

1.-Definition of geometry.

2.-Mesh generation.

3.-Boundary conditions.

2.- Calculation:

1.-Generation of the basis functions.

2.-Numerical integration.

3.-Solving the system of linear equations.

3.- Post-processing:

1.-Determination of approximation errors.

The steps that have greater computational cost are the numerical integration and the solution of the system of linear equations. In this master thesis, we will focus on the efficient solution of systems of linear equations. There are two types of solvers: direct and iterative. Direct solvers have explicit expressions for the solution. These methods provide an exact solution (up to round-off errors). In contrast to this, iterative methods calculate an approximate solution in every step and the accuracy of the solution increases with the number of the iterations, achieving a superior computational speed than direct solvers. An advantage of the iterative solvers is that can provide sufficiently accurate solutions in a reduced number of iterations. A disadvantage of iterative methods that the former only over direct methods is to calculate approximations to the solution.

The hp-FEM method applied to Laplace equations generates symmetric positive definite matrices with a specific pattern of sparsity. In practical applications the dimension of the stiffness system can be quite large. Systems of dimension 1000000 are commonly solved by commercial software. Larger ones, up to tens or hundreds of millions, are being solved on supercomputers.

To solve large systems it is necessary to have a high computational capacity and a code to run in parallel. Initially, this capability was only available

to large computer centers and parallel programming was extremely complex. Today, this has changed due to technological advances in GPUs. Initially designed as graphics accelerators were able to perform very specific tasks [49]. Now, however are programmable devices with capacity to perform many operations simultaneously.

In 2006, NVIDIA, one of the leading companies in the development of graphics cards created CUDA. This new language makes it easier to parallel program in parallel using GPUs because it was created as a extension of C language. For more information see [33, 35–37, 39].

The main objective of this master thesis is to implement a solver of systems of linear equations in CUDA in a hp-FEM method applied to Laplace equation.

A second objective is to determine whether the implementation of this program in CUDA provides significant advantages versus its implementation in sequential CPU.

In the last years have appeared many works related with GPUs [5], CUDA, FEM, and linear equation solvers. But commonly each publication focuses in very specific themes [13, 15, 22]. However, recently works have been published in which CUDA it is commonly used in FEM methods [22, 34, 42, 43]. In contrast, this master thesis focuses in the CUDA implementation of an iterative solver in a hp-FEM method, which is undeveloped.

In the second chapter we are going to expose the basic ideas about CUDA. In the third chapter we will explain de model problem and the FEM formulation applied to Laplace equation. In the forth chapter we will see the most common solvers. In the fifth chapter it is explain how have been implemented the algorithm. The sixth chapter shows the results obtained. In the last chapter we will describe the conclusions.

Chapter 2

Parallel programming in CUDA

The first section of this chapter discusses basic ideas on parallel computing. The second section explains what is CUDA and its most important characteristics.

2.1 Parallel computing

The most basic unit of operation is a transistor, which is able to perform operations in binary. Its main characteristics are size, operation frequency, power consumption, and heat dissipation. From the beginning of computers, the objective has been to increase the number of operations that an integrated circuit can perform.

Normally, the two ways to increase the number of operations per unit time are to increase the number of transistors or to increase the operating frequency. As technology has evolved, the transistors have been reduced in size and the operating frequencies are getting higher. According to Moore's law [47] [29], we can expect that the number of transistors on an integrated circuit is doubles every 18 months. Current technology allows to work with frequencies from 1.5 to 4 GHz. A common size of one transistor is measured in nanometers and an integrated circuit can easily have hundreds of millions of transistors.

This tendency cannot be sustained indefinitely because the current silicon technology has physical limits on the scale and frequency of operation. It also happens that grouping large amounts of transistors increases heat generation. Therefore, it is necessary to have cooling systems in order to ensure correct functioning of the circuits, which also increases the power consumption of the device. The hardware limitations have forced developers to find another way to increase performance.

The solution is easy, use many computers working together and simulta-

neously. But to perform this task requires the development of different kind of software. Traditionally, software has been created for serial computing. To solve a problem, we construct an algorithm that it is implemented in a serial instruction stream. These instructions are executed in the central processing unit of a computer (CPU). When an instruction is completed, the following one is executed.

Parallel computing (see [17] [44]) is a programming technique in which many instructions are executed simultaneously. A great way to deal with problems is to divide it into smaller problems that can be solved simultaneously.

There are different types of parallel computing, according to the instructions to be parallelized: bit-level operations, instructions, data or tasks. Parallel computers can be classified according to their hardware into two groups.

The first group is composed of multicore and multiprocessing computers with multiple processing elements in a single machine, see([32] [9] [21] [8]). The second group are the Clusters, the MPP(Massively Parallel Processing) and Grids that use multiple computers to work on the same task. The GPUs are in the first group (see [41], [19], [50], [54]).

2.1.1 Amdahl Law

As explained above, we can assume that the ability to parallelize can be used to improve processing speed indefinitely. But it is not. To increase speed by parallelization of a code, one needs to know which parts of this can be parallelized. Amdahl's Law gives us a way to calculate the maximum improvement of a code when a part of this is improved. [24]

$$A = \frac{1}{(1 - F_m) + \frac{F_m}{A_m}} \tag{2.1.1}$$

A is the acceleration or velocity gain achieved in the entire programming code due to the improvement of one of its sub-codes.

A_m is the acceleration or velocity gain achieved into the sub-codes improved.

F_m is the fraction between the time of execution of the sub-code improved and the time of execution of the complete code.

In simple terms, Amdahl's Law says that it is the algorithm the one that decides the speed improvement not the number of processors. At the end, you reach a situation in which the algorithm cannot be parallelized anymore. [1]

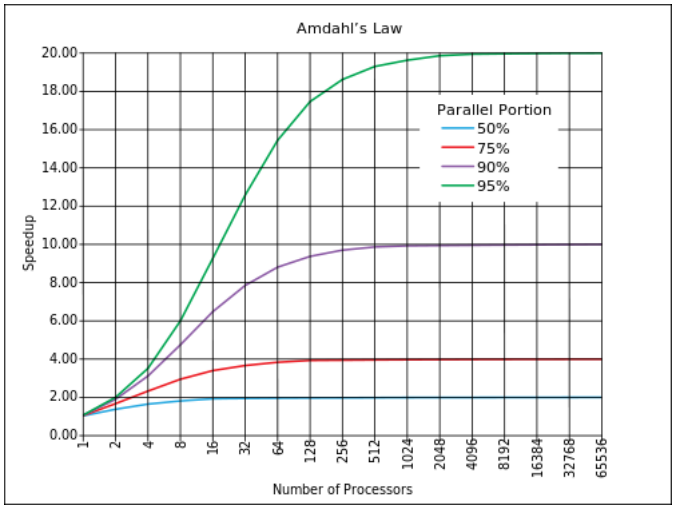


Figure 2.1: The increase of speed of a program using multiple processors in distributed computing fraction is limited by the sequential program. Picture taken from wikipedia.org

2.2 CUDA

CUDA is the computing engine in NVIDIA GPUs that is accessible to software developers through variants of industry standard programming languages. Although CUDA was designed as an extension of C, it has also adapted to other languages like Python, Perl, Fortran, Java, ruby Lua , Haskell, MATLAB,IDL and Mathematica.

CUDA gives developers access to the virtual instruction set and memory of the parallel computational elements. GPUs have a parallel throughput architecture that emphasizes executing many concurrent threads slowly, rather than executing a single thread very quickly. This approach of solving general purpose problems on GPUs is known as GPGPU [41].

2.2.1 Basic CUDA concepts

These concepts are extensively explained in [39].

Thread

Thread is a group of instructions that is the minimum GPU parallel computing unit. The execution of each instruction of the thread is in series and it is isolated from the rest of threads. [46]

Thread Block

A set of threads which share a Stream Multiprocessor, its shared memory space, and thread synchronization primitives.

Grid

Multiple Thread Blocks are organized in a two dimensional grid. Each block can be identified by a two dimensional index, it is accessible using the order **blockIdx**.

Kernel

CUDA allows to define functions, called kernels. When called, they are executed N times in parallel by N different CUDA threads. Where N is fixed by the programmer [39, p. 7].

Warp

It is a group of 32 threads that are created, managed, scheduled, and executed in a Stream Multiprocessor. It uses SIMT (Single Instruction Multiple Thread) architecture to perform all of these tasks. [20]

2.2.2 GPU architecture

The execution of multiple data at the same time requires a basic structure that is repeated. It does not mean that a GPU is composed only by a one type of basic structure. In contrast, it is organized by different groups of basic units which have a hierarchical organization. This organization allows to manage and schedule multiple threads simultaneously. The basic unit of processing is Stream Processor(SP). A group of eight SP formed a Stream Multiprocessor(SM). Two SM formed a Texture Processor Cluster(TPC). To perform all of these processes is needed the use of different types of memories [33].

Texture Processor Cluster

Each TPC is composed of:
One Stream Memory Controller.
Two Stream Multiprocessor .
One Texture Unit L1.

Stream Multiprocessor

Each SM is composed of:
Eight Stream Processor.

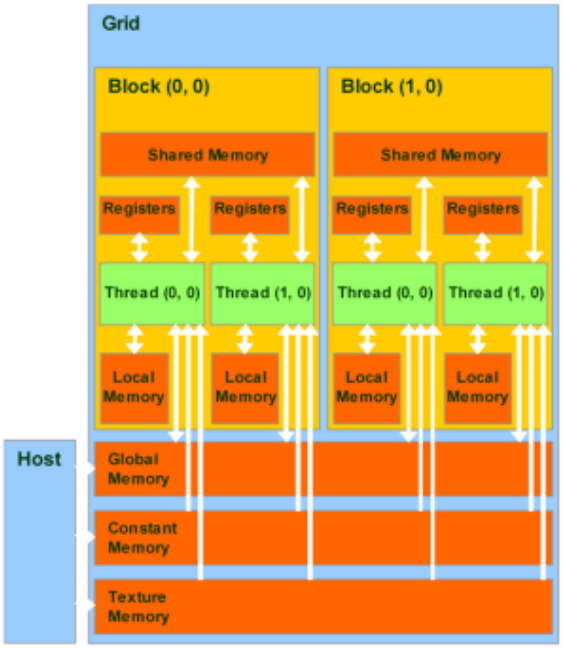


Figure 2.2: Host and device execution of GPU code. Picture taken from [39]

Two Special Function Unit.
One Shared Memory.
One I cache.
One C cache.
One Multi-threaded instruction fetch and issue unit (MTissue).

Stream Processor

Each of this contains a Multiply-Add(MAD) unit. SP is the basic unit of calculation in GPU.

Special Function Unit

SFU are used to perform transcendental functions. Each SFU contains four floating-point multipliers.

Global Memory

It is a large off-chip Dynamic Random Access Memory (DRAM). At the beginning of the execution all data is stored here. It can be directly addressable from a kernel using pointers.

Shared Memory

Because it is on-chip, the shared memory space is much faster than the local and global memory spaces. Shared memory is divided into equally-sized memory modules, called banks, which can be accessed simultaneously. The banks are organized such that contiguous 32-bit words are assigned to successive banks and each bank has a bandwidth of 32 bits per two clock cycles. For devices of compute capability 1.x, the most common (warp) size is 32 bits, and the number of banks is 16.

Local Memory

Local memory is used to describe memory owned by a single thread. This memory is often composed of on-chip registers, but can also contain off-chip memory when the on-chip space is depleted.

Constant Memory

On-chip memory of 8 KB per SM(64 KB total), with data originally residing in global memory. The cache is single ported, so simultaneous request within an SM must point to the same address or delays will occur.

Coalesced Memory Access

Coalesced (see [11, 18, 23]) memory transaction occurs when all the threads in a half-warp access global memory at the same time. The correct way to do this is to have consecutive threads accessing consecutive memory addresses. It is simpler to see this with an easy example.

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, one should take into account that if the matrix resides linearly in memory, the performance will increase. Memory access should reflect how your matrix is stored in the device memory. If we take the 3x4 matrix below:

$$\begin{matrix} 0 & 1 & 2 & 3 \\ 4 & 5 & 6 & 7 \\ 8 & 9 & a & b, \end{matrix}$$

then the memory access could be perform row by row, in a way that (r, c) maps to memory $(r \cdot 4 + c)$.

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

Assuming that every matrix element has to accessed, there are at least two ways to perform the GPU memory access.

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

Both forms of accessing the device memory read each element once. But to achieve the maximum performance, only the second way is **coalescent**. In the first option, memory access is 0, 3, 6, 9, which is neither consecutive nor coalesced. The second option is 0, 1, 2, 3, is consecutive and therefore **coalescent**.

2.2.3 Information flux in GPU

The fact that CUDA is a C extension does not imply that we can still follow traditional programming techniques. Although there are some similarities. To leverage the capabilities of GPUs, it is necessary to work with dynamic memory [23]. Indeed, GPU programming requires also the use of the CPU. This section explains the structure of a typical CUDA code.

The use of dynamic memory allows us to manage the memory through the

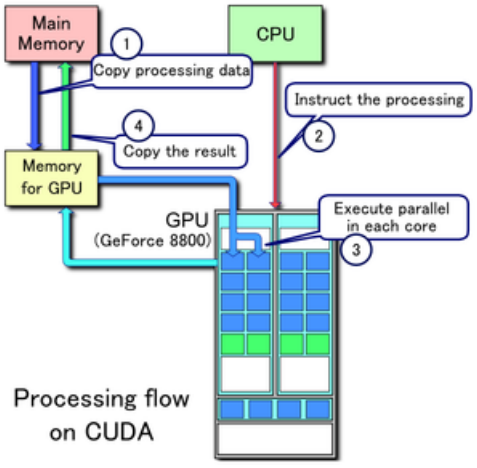


Figure 2.3: CUDA processing flow. Picture taken from [39]).

program execution. Usually, the procedure to allocate memory for a vector called Vector of type float of size 10 is as follows:

```
1  float Vector[10];
```

Defining the vector by this way implies that its size cannot change throughout the program execution.In contrast, using dynamic memory, we have:

```
1  float *Vector;
   Vector = float*malloc(10*sizeof(float));
```

The size of the array is a variable and it can change during the execution of the program.

First Step: Allocate memory in the host

All the elements necessary for the execution on GPU (arrays and structures) have to have a partner in the CPU. Normally, when naming their corresponding pointers, a prefix is added as follows: h_{*variablename*}, d_{*variablename*}.

```
   float *h_Vector;
2  float *d_Vector;
```

Second Step: Allocate memory in the device

CUDA provides a function similar to malloc() in C called cudaMalloc() which allows us to allocate memory on the GPU.

```
cudaMalloc((void**)&d_Vector, 10*sizeof(float))
```

Third Step: Initializing values in the host

The values are initiated in the CPU. The values of the vector h_A can be initialized as:

```
1 for(i=0;i<10;i++)
    h_Vector[i]=... // expresion to give values to the vector in
        the host;
```

Forth Step: Copying values from the host to device

Cuda provides another function to copy the values from the CPU memory to the GPU memory.

```
cudaMemcpy(d_Vector,h_Vector,10*sizeof(float),
    cudaMemcpyHostToDevice );
```

d_Vector :destination
h_Vector :origin
10*sizeof(float):size of memory to copy
cudaMemcpyHostToDevice :type of instruction

Fifth Step: Executing Kernel in the device

Before running the kernel, you need the number of threads and blocks that will need to perform the operation. We create two variables of type dim3(three dimensional arrays). If we are to operate with vectors of size DIM and blocks the size MAX_BLOCK_SIZE, we need at least DIM/MAX_BLOCK_SIZE+1 blocks and DIM threads.

```
dim3 blocks((DIM/MAX_BLOCK_SIZE)+1 ,1,1);
dim3 threads(MAX_BLOCK_SIZE,1,1);
```

After that, we can invoke the kernel:

```
Kernel_name<<<blocks,threads>>>(kernel_arguments);
```

Sixth Step: Copying values from the device to the host

With the same function of the forth step we can copy the result of the kernel from the device to the host.

```
cudaMemcpy(h_Vector,d_Vector,10*sizeof(float),
    cudaMemcpyDeviceToHost);
```

h_Vector :destination
d_Vector :origin
10*sizeof(float): size of memory to copy
cudaMemcpyDeviceToHost: type of instruction

Seventh Step: Free memory from the device

After finishing the kernel execution, it is necessary to free the GPU memory. This is done with the instruction cudaFree(pointer). It is also a good practice to free the CPU memory.

```
cudaFree(d_Vector);
free(h_Vector);
```

2.2.4 Programming features in CUDA

In summary, in order to make a program in CUDA, we want algorithms to exhibit the following features:

Parallelizable:

All algorithms should follow a sequence of steps and in general, not all the steps can be parallelized. Typically, algorithms with many conditionals and perform operations when the difference between the number of inputs and outputs are very large. One example is the dot product of two vectors. This operation has 2N inputs (for dimension N) and only one output value.

Simplicity in the operations:

We have seen in the previous sections that one SM contains 8 SP an 2 SFU. In one clock cycle the SM can perform eight operations simultaneously (additions and multiplications). On the other hand, SFU can only perform two operations simultaneously, which are powers, logarithms, and, trigonometric functions. It means that the basic operations performed by an algorithm implemented in CUDA should be additions and multiplications.

Minimizing the number of accesses between CPU and GPU:

To perform operations in the GPU, it is necessary copy the information from the CPU host to the GPU device. But done this requires time. For large amount of data, the copy process generates a bottleneck. This effect can reduce considerably any gain of speed. For this reason, it is necessary to minimize the transfer of information between the CPU and the GPU.

Minimizing dispersion in the execution of threads:

To maximize the capabilities of the GPU, the number of operations assigned to the threads should be as similar as possible. A CUDA Kernel is executed as many times as threads are, and it cannot finish until each of all threads are completed. It means, that the execution time of a kernel will increase with the number of calculations of the largest thread.

See the following example: We can perform matrix vector multiplication with a kernel. We will compute it using one thread for each row.

$$\begin{pmatrix} 1 & 1 & 0 & 0 \\ 1 & 1 & 0 & 0 \\ 1 & 1 & 0 & 0 \\ 1 & 1 & 1 & 1 \end{pmatrix} \begin{pmatrix} 1 \\ 1 \\ 1 \\ 1 \end{pmatrix} \tag{2.2.1}$$

With this choice, the first three threads have to wait until the last one ends. The execution time of the kernel doubles for only one thread.

Reduce the use of structures:

The current version of CUDA supports the use of structures, but it is not fully optimized. For this reason, the use of structures should be limited as much as possible. If structures are used, they should be programmed in a simple way. In contrast with other programming languages, the use of structures of structures is highly inefficient in CUDA.

Avoid recursion:

In the traditional programming is very common see expression like $a[i] = a[i] + b[i]i = 0...N$. It means that the value of the sum $a[i] + b[i]$ will be

stored in $a[i]$ once the addition has completed. By difference, in CUDA you have to wait until all threads finish a instruction to perform another one. Recursion expressions cannot be used in CUDA.

Chapter 3

Model problem and Finite Element Formulation

The objective is to implement in CUDA an algorithm to solve a system of linear equations arising from a hp-FEM method, and test it in the Laplace equation. Once done, we will compare the execution times of the program implemented in C in parallel with the implementation in CUDA.

3.1 FEM

The finite element method has its origins in the theory of structures. 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 algebraic equations. The limitation on the number of equations that could be solved is one of the major constraints of the method. The introduction of the digital computer has made possible the solution of large systems of equations.

Nowadays, Finite Element methods are used to solve differential equations problem in many areas of science and technology, including mechanics, electromagnetism, fluid mechanics and many others. 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 analytical solutions. Moreover, most of the systems that involve differential equations do not have a known exact solution form. When it is not possible to find an explicit solution, it may be numerically approximated using computational techniques.

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 difficulties, introduced by irregular geometry or other discontinuities, render makes the problems intractable analytically.

To obtain a solution, simplifying assumptions must be considered, reducing the problem to one that can be numerically approximated.

Numerical methods provide approximate values of the unknown quantity in the region of interest (computational domain). In the FEM, this region of interest is divided into numerous connected subregions or elements within which approximate functions which are usually polynomials which are used to represent the unknown quantity, see [53] [28].

3.1.1 Variational Formulation of Laplace Equation.

To solve the problem, we will first consider the Laplace equation with Dirichlet boundary conditions.

$$(SP)\begin{cases} -\Delta u = f, & \text{in } \Omega \subset \mathbb{R}^3, \\ u|_\Gamma = 0, & \text{on } \Gamma = \partial\Omega, \end{cases} \tag{3.1.1}$$

Where Ω is a bounded open domain in the space $\mathbb{R}^3 = \{x = (x_1, x_2, x_3) : x_i \in \mathbb{R}\}$ with boundary Γ. Notice that:

$$\Delta u = \frac{\partial^2 u}{\partial x_1^2} + \frac{\partial^2 u}{\partial x_2^2} + \frac{\partial^2 u}{\partial x_3^2} \tag{3.1.2}$$

Defining $\widehat{n} = (n_1, n_2, n_3)$ as the normal(outward)to Γ. $d\overrightarrow{x}$ denotes the element of volume in $\mathbb{R}^3$, and ds the surface element along Γ.
In general, we find that,

$$\int_\Omega \frac{\partial v}{\partial x_i} w d\overrightarrow{x} + \int_\Omega v \frac{\partial w}{\partial x_i} d\overrightarrow{x} = \int_\Gamma v w \widehat{n}_i ds, \qquad i = 1, 2, 3.$$

Integrating by parts in the three coordinates:

$$\int_\Omega \nabla v \cdot \nabla w d\overrightarrow{x} = \int_\Gamma v \frac{\partial w}{\partial n} ds - \int_\Omega v \Delta w dx,$$

where the normal derivative in the outward normal direction to the boundary Γ, is:

$$\frac{\partial w}{\partial n} = \frac{\partial w}{\partial x_1} n_1 + \frac{\partial w}{\partial x_2} n_2 + \frac{\partial w}{\partial x_3} n_3.$$

u satisfies 3.1.1 and the solution to the variational problem is $u \in V$. Since we only consider Dirichlet boundary conditions, the term associated with Neumann boundary conditions vanishes. Thus, we have:

$$a(u, v) = (f, v) \qquad \forall v \in V,$$

where

$$a(u, v) = \int_\Omega \nabla u \cdot \nabla v d\overrightarrow{x},$$

$$(f, v) = \int_\Omega f v d\overrightarrow{x}.$$

Ω can be subdivided into a set $T_h = C_1, \ldots, C_m$ of non-overlapping cubes C_i,

$$\Omega = \bigcup_{i=1}^{m} C_i = C_1 \bigcup C_2 \bigcup \ldots \bigcup C_m,$$

We can now define V_h as follows,

$$V_h = \{\text{v: v is continous on } \Omega, v|_k \text{ is linear in K } \in T_h, v|_\Gamma = 0\},$$

The space V_h consists of all continuous functions that are linear on each cube and vanish on Γ. Defining the basis of V_h as follows.

$$\varphi_j(N_i) = \delta_{ij} \equiv \begin{cases} 1 \text{ if } i = j \\ 0 \text{ if } i \neq j \end{cases} \quad i, j = 1, \ldots, M. \tag{3.1.3}$$

Thus, the support of φ_j consist of the cubes with the common node n_j. The function $v_h \in V_h$ has now the expression,

$$v_h(x) = \sum_{j=1}^{M} \eta_j \varphi_j(x), \quad \eta_j = v(n_j), \quad \text{for } x \in \overline{\Omega} \tag{3.1.4}$$

It is possible to formulate the finite element method for 3.1.1 starting from the variational formulation 3.1.3.

$$a(u_h, v_h) = (f, v_h) \quad \forall v_h \in V_h.$$

Where the *stiffness matrix* is a MxM matrix, whose elements are defined as:

$$a_{ij} = a(\varphi_i, \varphi_j),$$

and $b = (b_i)$ is a size M vector which elements are defined as

$$b_i = (f, \varphi_i).$$

Stiffness matrix $\mathbf{A}$ is usually computed by summing the contributions of the different cubes:

$$a(\varphi_i, \varphi_j) = \sum_{C \in T_h} a_C(\varphi_i, \varphi_j),$$

where,

$$a_C(\varphi_i, \varphi_j) = \int_C \nabla\varphi_i \cdot \nabla\varphi_j d\overrightarrow{x},$$

Then, stiffness matrix and load vector are defined in their discrete version as:

$$\begin{cases} A = \sum_{C \in T_h} \int_C \nabla\varphi_i \cdot \nabla\varphi_j d\overrightarrow{x}, \\ b = \int_\Omega f\varphi_i d\overrightarrow{x}, \end{cases} \tag{3.1.5}$$

Where A matrix, is a sparse matrix. Each diagonal entry of the matrix represents the interaction of each basis function with itself, while off-diagonal entries correspond to interactions between different basis functions. Basis functions that do not share support result in a zero contribution to the stiffness matrix. Thus, the resulting matrix is highly populated with zeros.

3.1.2 hp-FEM

The hp-FEM (see [25], [12], [7])is a general version of the FEM. This numerical method is based on polynomials approximations that makes use of elements of variable size h and polynomial degree p. This method converges exponentially fast, when the mesh is refined using a suitable combination of h-refinements and p-refinements. The h-refinements are performed by dividing elements into smaller ones. The p-refinements are obtained by increasing the polynomial order in shape functions (see [53]). This exponential convergence (see [10]) makes the method one of the best possible choice when implementing a numerical simulation.

The 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 more importantly, different elements may have different size h and and/or different polynomial order approximation p, which is known as hp-adaptivity.

The implemented code have this capability and be able to properly simulate different situations where h and p can vary as requested by the user.

Chapter 4

Linear equation solvers

In this chapter we are going to study the basic characteristics of the most common linear equation solvers. After that we will justify the selection one of them to be implemented in CUDA. To learn more about GPU implementation of linear algebra, see [31].
The problem is expressed mathematically by this way. We want to solve the following linear system of n equations with n unknowns $x_1, x_2, \ldots, x_n$:

$$\begin{cases} a_{11}x_1 + a_{12}x_2 + \ldots + a_{1n}x_n = b_1 \\ a_{21}x_1 + a_{22}x_2 + \ldots + a_{2n}x_n = b_2 \\ \vdots \\ a_{n1}x_1 + a_{n2}x_2 + \ldots + a_{nn}x_n = b_n \end{cases} \tag{4.0.1}$$

In matrix form, we have:

$$Ax = b \tag{4.0.2}$$

$$\begin{pmatrix} a_{11} & a_{12} & ... & a_{1n} \\ a_{21} & a_{22} & ... & a_{2n} \\ \vdots & \vdots & \ddots & \vdots \\ a_{n1} & a_{n2} & ... & a_{nn} \end{pmatrix} \begin{pmatrix} x_1 \\ x_2 \\ \vdots \\ x_n \end{pmatrix} = \begin{pmatrix} b_1 \\ b_2 \\ \vdots \\ b_n \end{pmatrix} \tag{4.0.3}$$

For the existence of solution $det(A) \neq 0$.

4.1 Gaussian elimination

This linear algebra procedure first perform a forward elimination. Gaussian elimination reduces a given system to ones triangular. Second, it performs a backward elimination to solve the linear system [4].

First step : eliminate x_1 in the rows:2,...,n by linear combination of rows.
Second step : eliminate x_2 in the rows:3,...,n by linear combination of rows.

Iteratively after n-1 steps

$$\begin{pmatrix} u_{11} & u_{12} & ... & u_{1n} \\ 0 & u_{22} & ... & u_{2n} \\ \vdots & \vdots & \ddots & \vdots \\ 0 & 0 & ... & u_{nn} \end{pmatrix} \begin{pmatrix} x_1 \\ x_2 \\ \vdots \\ x_n \end{pmatrix} = \begin{pmatrix} k_1 \\ k_2 \\ \vdots \\ k_n \end{pmatrix} \tag{4.1.1}$$

Gaussian Elimination algorithm: forward elimination and triangular form

$$k = 1, \ldots, n-1 \begin{cases} a_{ij}^{(k+1)} = a_{ij}^{(k)} & i = 1, \ldots, k \quad j = 1, \ldots, n \\ a_{ij}^{(k+1)} = 0 & i = k+1, \ldots, n \quad j = 1, \ldots, k \\ a_{ij}^{(k+1)} = a_{ij}^{(k)} - \dfrac{a_{ik}^{(k)} a_{kj}^{(k)}}{a_{kk}^{(k)}} & i = k+1, \ldots, n \quad j = k+1, \ldots, n \\ b_i^{(k+1)} = b_i^{(k)} & i = 1, \ldots, k \\ b_i^{(k+1)} = b_i^{(k)} - \dfrac{a_{ik}^{(k)} b_k^{(k)}}{a_{kk}^{(k)}} & i = k+1, \ldots, n \end{cases} \tag{4.1.2}$$

By backward elimination, it means that we star by eliminating x_n in the rows:1,...,n-1 by linear combination of rows. By iteration of this process the matrix only have ones in the main diagonal, so we get the solution x.

4.2 LU decomposition

We will study a direct method for solving linear systems: the LU decomposition. Given a matrix A, the aim is to build a lower triangular matrix L and an upper triangular matrix which has the following property: diagonal elements of L are unity and A=LU. [6]
For the resolution of linear system : Ax=b, the system becomes

$$LUx = b \Leftrightarrow \begin{cases} Ly = b & (1), \\ Ux = y & (2). \end{cases} \tag{4.2.1}$$

$$L = \begin{pmatrix} 1 & & & \\ l_{21} & 1 & & \\ \vdots & \vdots & \ddots & \\ l_{n1} & l_{n2} & \cdots & l_{nn} \end{pmatrix} U = \begin{pmatrix} u_{11} & u_{12} & \cdots & u_{1n} \\ & u_{22} & \cdots & u_{2n} \\ & & \ddots & \vdots \\ & & & u_{nn} \end{pmatrix} \tag{4.2.2}$$

We solve the system (1) to find the vector y, then the system (2) to find the vector x. The resolution is facilitated by the triangular shape of the matrices.

$$Ly = b \Leftrightarrow \begin{cases} y_1 = b_1/l_{11} \\ y_i = \dfrac{1}{l_{ii}}(b_i - \displaystyle\sum_{j=1}^{i-1} l_{ij}y_j) \quad \forall i = 2, 3, \ldots, n. \end{cases} \tag{4.2.3}$$

$$Ux = y \Leftrightarrow \begin{cases} x_n = y_n/u_{nn} \\ x_i = \dfrac{1}{u_{ii}}(y_i - \displaystyle\sum_{j=i+1}^{n} u_{ij}x_j) \quad \forall i = n-1, n-2, \ldots, 1. \end{cases} \tag{4.2.4}$$

Before starting the system solution L and U must be found, which typically are solved by Gaussian elimination.

4.3 Cholesky decomposition

Given a symmetric positive definite matrix A, the aim is to build a lower triangular matrix L which has the following property: the product of L and its transpose is equal to A. [2]

$$A = LL^T \tag{4.3.1}$$

$$\begin{pmatrix} a_{11} & a_{21} & a_{31} \\ a_{21} & a_{22} & a_{32} \\ a_{31} & a_{32} & a_{33} \end{pmatrix} = \begin{pmatrix} l_{11} & 0 & 0 \\ l_{21} & l_{22} & 0 \\ l_{31} & l_{32} & l_{33} \end{pmatrix} \begin{pmatrix} l_{11} & l_{21} & l_{31} \\ 0 & l_{22} & l_{32} \\ 0 & 0 & l_{33} \end{pmatrix} \tag{4.3.2}$$

$$\begin{pmatrix} a_{11} & a_{21} & a_{31} \\ a_{21} & a_{22} & a_{32} \\ a_{31} & a_{32} & a_{33} \end{pmatrix} = \begin{pmatrix} l_{11}^2 & l_{21}l_{11} & l_{31}l_{11} \\ l_{21}l_{11} & l_{21}^2 + l_{22}^2 & l_{31}l_{21} + l_{32}l_{22} \\ l_{31}l_{11} & l_{31}l_{21} + l_{32}l_{22} & l_{31}^2 + l_{32}^2 + l_{33}^2 \end{pmatrix} \tag{4.3.3}$$

For the diagonal elements (l_{kk}) of L there is a calculation pattern:

$$\begin{aligned} l_{11} &= \sqrt{a_{11}} \\ l_{22} &= \sqrt{a_{22} - l_{21}^2} \\ &\vdots \\ l_{kk} &= \sqrt{a_{kk} - \sum_{j=1}^{k-1} l_{kj}^2} \end{aligned} \tag{4.3.4}$$

For the elements below the diagonal (l_ik, where $i > k$) there is also a

calculation pattern:

$$\begin{aligned} l_{21} &= \frac{1}{l_{11}} a_{21} \\ l_{31} &= \frac{1}{l_{11}} a_{31} \\ l_{32} &= \frac{1}{l_{22}} (a_{32} - l_{31} l_{21}) \\ &\vdots \\ l_{ik} &= \frac{1}{l_{kk}} \left(a_{ik} - \sum_{j=1}^{k-1} l_{ij} l_{kj} \right) \end{aligned} \tag{4.3.5}$$

For the resolution of linear system : Ax=b, the system becomes

$$LL^T x = b \Leftrightarrow \left\{ \begin{array}{ll} Ly = b & (1), \\ L^T x = y & (2). \end{array} \right. \tag{4.3.6}$$

We solve the system (1) to find the vector y, then the system (2) to find the vector x. The resolution is facilitated by the triangular shape of the matrices.

4.4 Conjugate Gradient

This numerical method allows you to solve linear systems whose matrix is symmetric and positive definite. The search for successive directions makes possible to reach the exact solution of the linear system. [45] [3]
A is a $n \times n$ symmetric and positive definite matrix ($A^T = A, x^T A x > 0, \forall x \in \mathbf{R}^n$).
We can define the following scalar product on $\mathbf{R}^n$:

$$\langle u, v \rangle_A = u^T A v \tag{4.4.1}$$

Two elements$u, v \in \mathbf{R}^n$ are A-conjuguate if:

$$u^T A v = 0 \tag{4.4.2}$$

Conjugate Gradient Method consists in building a vectorial sequence (p_k) of n A-conjugate vectors . Consequently, the sequence $p_1, p_2, \ldots, p_n$ form a basis of $\mathbf{R}^n$. The exact solution $x_\star$ can be expanded like follows:

$$x_\star = \alpha_1 p_1 + \cdots + \alpha_n p_n \tag{4.4.3}$$

where

$$\alpha_k = \frac{p_k^\top b}{p_k^\top A p_k}, k = 1, \ldots, n. \tag{4.4.4}$$

The exact solution $x_\star$ is also the unique one minimizer of the functional $J(x) = \frac{1}{2}x^\top Ax - b^\top x, \quad x \in \mathbf{R}^n$.We can compute$\nabla J(x) = Ax - b, \quad$. It means that :

$$\nabla J(x_\star) = 0 \tag{4.4.5}$$

We define the residual vector of the linear system:

$$r_k = b - Ax_k = -\nabla J(x_k) \tag{4.4.6}$$

r_k is the direction of the gradient of the functional J in x_k.
The new direction of descent p_{k+1} is the same as its A-conjugation with p_k, we have then:

$$p_{k+1} = r_k - \frac{p_k^\top Ar_k}{p_k^\top Ap_k} p_k \tag{4.4.7}$$

It is the choice of the coefficient $\frac{p_k^\top Ar_k}{p_k^\top Ap_k}$ which allows the A-conjugation of the directions p_k.
If we compute$(Ap_{k+1}, p_k) = 0 \forall k$,because p_{k+1}, p_k are A-conjugation vectors.

4.5 Preconditioned Conjugate Gradient

Solving linear systems resulting from the finite elements method shows the limits of the conjugate gradient. Preconditioners are often used to increase the convergence rate . The convergence rate of the iterative methods depends on the spectral condition number of the preconditioned system.A good convergence is obtained when the spectral condition number is bounded and close to one.The technique of Preconditioned Conjugate Gradient Method consists in introducing a matrix C subsidiary. [16]

It happens sometimes that the spectral condition number $\kappa(A)$ is too high (eigenvalues are not well distributed). Preconditionnement consists in introducing regular matrix $C \in \mathcal{M}_n(\mathbb{R})$ and solving the system:

$$C^{-1}(Ax) = C^{-1}b \Leftrightarrow Ax = b \tag{4.5.1}$$

When choosing a preconditioner must take into account the computational cost of inverting the matrix C, and the memory size required for storage. The following subsections present two preconditioners.

4.5.1 Jacobi Preconditioner

Jacobi Preconditioner consists in taking the diagonal of A for the matrix C, i.e.

$$C_{ij} = \begin{cases} A_{ii} & \text{if} i = j, \\ 0 & \text{if} i \neq j. \end{cases} \tag{4.5.2}$$

Advantages of such preconditioner are the facility of its implementation and the low memory it needs. But we can find other preconditioners such that resolution of the linear system is fastest.

4.5.2 SSOR Preconditioner

SSOR Preconditioner(Symmetric Successive Over Relaxation) We decompose the symmetric matrix A like follows:

$$A = L + D + L^{\top} \tag{4.5.3}$$

where L is the strictly lower part of A and D is the diagonal of A. SSOR Preconditioner consists in taking:

$$C = (\frac{D}{\omega} + L)\frac{\omega}{2-\omega}D^{-1}(\frac{D}{\omega} + L^{\top}) \tag{4.5.4}$$

where ω is a relaxation parameter. A necessary and sufficient condition of the Preconditioned Gradient Method algorithm is to fix the parameter ω in the interval $[0, 2]$.

4.6 The most suitable algorithm

In this section, we will review the main features of the algorithms a procedures to discuss which is the best choice for programming in CUDA. According to the ideas expose in section 2.2.4.

Gaussian elimination

First: Presents disparity in resource use as initially need to make threads n zeros in the first column, n-1 threads for zeros in the second and so on until only need one. In breach of the criterion of *parallelizable*.
Second: Presents recursion element by element. In breach of the criterion of *programming in CUDA*.

LU decomposition

First: It presents a big problem when parallelize the equations 4.2.3,4.2.4. To determine $y_i and x_i$ it is necessary to find them and know the x_i, y_i earlier. In breach of the criterion of *parallelizable*.
Second: Apart inherits parallelization problems Gaussian elimination.

Cholesky decomposition

First:In 4.3.4 we see how they are using to calculate l_{ii}square roots, in breach of the criterion of *simplicity in the basic operations*.
Second: In 4.3.4, 4.3.5 equations are summations of different sizes, thus leading to different times of computation in breach of the criterion of *minimizing dispersion in the execution of threads*.

Conjugate Gradient

In equation 4.4.7 presents recursion, also in the evolution of the solution x, in breach of the criterion of *avoid recursion.*

Preconditioned Conjugate Gradient

The recursion problem is still present but with suitable preconditioner the number of iterations required may be reduced considerably.
All algorithms have problems when we try to select one to implement in CUDA. However, the problem of recursion can be solved using auxiliary vectors to store the results of the operation. Another option is use libraries such as CUBLAS that can perform recursion with specific functions [38, p. 25].
With all this, the most appropriate algorithm to parallelize on CUDA are the Conjugate Gradient and Preconditioned Conjugate Gradient.

Chapter 5

Implementation

In this chapter we explain how CUDA CG and PCG solvers are implemented. The first section focuses on the algorithm. In the second section we will explain the need to implement a matrix compression method and will see some examples. The third section focuses on the basic operations needed to perform the algorithm. The last section explain the structure of the code.

5.1 Algorithms

This algorithm terminates when the maximum number of iterations i_{max} has been exceeded, or when $\|r_i\| \leqslant \varepsilon \|r_0\|$.
The fast recursive formula for the residual is usually used, but once every N iterations, the exact residual is recalculated to remove accumulated floating point error. N is arbitrary, it might be appropriate. If the tolerance is close to the limits of the floating point precision of the machine, a test should be added after δ is evaluated to check if $\delta \leqslant \varepsilon^2 \delta_0$ and if this test holds true, the exact residual should also be recomputed and δ re-evaluated. This prevents the procedure from terminating early due to floating point round-off error. [48] [52]

Given matrix A, vector b, starting vector x, a maximum number of iteration i_{max} and error tolerance $\varepsilon < 1$.

5.1.1 CG Algorithm

$i \leftarrow 0$
$r \leftarrow b - Ax$
$d \leftarrow r$
$\delta_{new} \leftarrow r^T r$
$\delta_o \leftarrow \delta_{new}$
while $i < i_{max}$ ***and*** $\delta_{new} > \delta_o \varepsilon^2$ **do**

end
$q \leftarrow Ad$
$\alpha \leftarrow \frac{\delta_{new}}{d^T q}$
$x \leftarrow x + \alpha d$
if *i is divisible by* i_{iter} **then**
    $r \leftarrow b - Ax$
else
    $r \leftarrow r - \alpha q$
$\delta_o \leftarrow \delta_{new}$
$\delta_{new} \leftarrow r^T r$
$\beta \leftarrow \frac{\delta_{new}}{\delta_o}$
$d \leftarrow r + \beta d$
$i \leftarrow i + 1$

Algorithm 1: CG Algorithm

5.1.2 PCG Algorithm

$i \leftarrow 0$
$r \leftarrow b - Ax$
$d \leftarrow M^{-1}r$
$\delta_{new} \leftarrow r^T d$
$\delta_o \leftarrow \delta_{new}$
while $i < i_{max}$ ***and*** $\delta_{new} > \delta_o \varepsilon^2$ **do**

end
$q \leftarrow Ad$
$\alpha \leftarrow \frac{\delta_{new}}{d^T q}$
$x \leftarrow x + \alpha d$
if *i is divisible by* i_{iter} **then**
    $r \leftarrow b - Ax$
else
    $r \leftarrow r - \alpha q$
$s \leftarrow M^{-1}r$
$\delta_o \leftarrow \delta_{new}$
$\delta_{new} \leftarrow r^T s$
$\beta \leftarrow \frac{\delta_{new}}{\delta_o}$
$d \leftarrow s + \beta d$
$i \leftarrow i + 1$

Algorithm 2: PCG Algorithm

5.2 Sparse matrix formats

This section explains the compression formats for sparse matrices. There is not a best method to compress sparse matrices because the efficiency of the method depends on the sparsity pattern.

COO

The coordinate format is a simple storage scheme. The matrix information is stored in three arrays. The element j^{th} not null is stored in j^{th} position of the array data[j]. The other arrays contain the coordinates of the element row[j],col[j].

$$\begin{pmatrix} 1 & -2 & 0 & 0 \\ 0 & 3 & 1 & 0 \\ 2 & 0 & -5 & 1 \\ 0 & 2 & 0 & 4 \end{pmatrix} \tag{5.2.1}$$

$$data[] = \begin{bmatrix} 1 & -2 & 3 & 1 & 2 & -5 & 1 & 2 & 4 \end{bmatrix} \tag{5.2.2}$$

$$row[] = \begin{bmatrix} 0 & 0 & 1 & 1 & 2 & 2 & 2 & 3 & 3 \end{bmatrix} \tag{5.2.3}$$

$$col[] = \begin{bmatrix} 0 & 1 & 1 & 2 & 0 & 2 & 3 & 1 & 3 \end{bmatrix} \tag{5.2.4}$$

It can be seen that in the row array indexes are repeated as many times as elements are not null, so we need more memory to store repeated information. However, the size of the three arrays are the same. It is an advantage in GPU computing because it produces full coalescence.

CRS

The compressive row storage format is a improvement of COO storage scheme. The matrix information is stored in three arrays of different size. Non null elements are stored sequentially in the array data[](from left to the right in rows and up to down in columns), in consequence the size of data is the number of non null elements in the matrix. The vector col[] stores the position of the element j^{th} in the row. The third is an integer vector that stores where the row starts in vector data, where the last element is the number of total non null elements.

$$\begin{pmatrix} 1 & -2 & 0 & 0 \\ 0 & 3 & 1 & 0 \\ 2 & 0 & -5 & 1 \\ 0 & 2 & 0 & 4 \end{pmatrix} \tag{5.2.5}$$

$$data[] = \begin{bmatrix} 1 & -2 & 3 & 1 & 2 & -5 & 1 & 2 & 4 \end{bmatrix} \tag{5.2.6}$$

$$col[] = \begin{bmatrix} 0 & 1 & 1 & 2 & 0 & 2 & 3 & 1 & 3 \end{bmatrix} \tag{5.2.7}$$

$$cout[] = \begin{bmatrix} 0 & 2 & 4 & 7 & 9 \end{bmatrix} \tag{5.2.8}$$

In this format cout contains the start of the row k in k^{th} position and the end in $k + 1^{th}$. This is an advantage when you have to iterate an instruction in rows. In contrast, rows have not got the same number of non null elements. It means that there will be different times of computation when we multiply a matrix by a vector.

More information about sparse matrix formats can be found in [26] [40] [30] [51]

5.3 Basic operations for CG and PCG

In this section we are going to explain all mathematical operations required to perform the CG and PCG method and its implementation in CUDA. All of they are well known.

5.3.1 Dot product

If you have two real vectors a , b of size N. We can compute its dot product as:

$$\vec{a} \cdot \vec{b} = \sum_{i=1}^{N} a_i \cdot b_i \tag{5.3.1}$$

A c code to perform dot product could be:

```
float C_DotProduct(float *a,float*b,int N)
{

  int i;
  float dot;
  dot=0.0;
  for(i=0;i<N;i++)
    dot+=(a[i]*b[i]);

  return dot;
}
```

If we think for a moment we can realise that the dot product is an operation with 2N inputs and one output. It means that at one moment the program execution must be serialized(a waste of time in CUDA). We can parallelize the pairwise sum, but the addition of the result is in serial.

```
__global__ float CUDA_DotProduct( float *a, float *b, float *
    a_dot_b,int N)
{
  int i;
  float dot;
  int row = blockIdx.x * blockDim.x + threadIdx.x;
  if( row < N && row >= 0)
    a_dot_b[row] = a[row]* b[row];

  __syncthreads();
  dot=0.0;
  for(i=0;i<N;i++)
    dot+=(a[i]*b[i]);

  return dot;
}
```

To exploit parallelism of the GPU the idea is that the addition of terms is done by pairs,and this process is repeated until the result is find. In theory it is a good idea but has some details. The dimension of the vector does not have to be a multiple of two, so there will be to keep in mind that we will have additional terms, we have not added.
Also we must remember that information can be easily shared among the threads in a Block. We can use this property to perform the addition by blocks and continue the sum adding the results of each Block.
There is another algorithm that calculates the dot product which is fully explained in this article. [27]
In the program we have used the function cublasSdot()of the CUBLAS library, which performs the dot product highly efficient. In the ([38]) its exposed how it works.

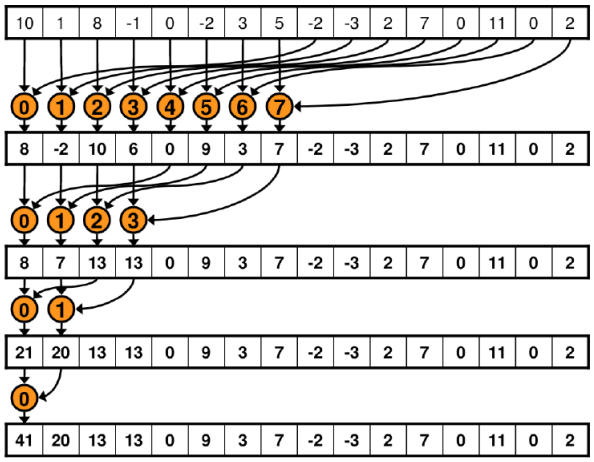


Figure 5.1: Addition of terms in dot product done by pairs. Picture taken from [27]

5.3.2 Vector addition

In some steps of the algorithm like $r = b - Ax$ we need to compute a sum of vectors of size N. This operation involves three arrays allocated in the device memory. In contrast with the dot product we have 2N input values and N output values. In this case the idea of the parallelism is very simple: Computing each element as a sum of the corresponding terms in pairs.

$$\overrightarrow{a} + \overrightarrow{b} = \overrightarrow{a+b} \tag{5.3.2}$$

$$\begin{pmatrix} a_1 \\ a_2 \\ \vdots \\ a_n \end{pmatrix} + \begin{pmatrix} b_1 \\ b_2 \\ \vdots \\ b_n \end{pmatrix} = \begin{pmatrix} a+b_1 \\ a+b_2 \\ \vdots \\ a+b_n \end{pmatrix} \tag{5.3.3}$$

We must note that despite its simplicity not all sums are performed simultaneously. So before using the result of the sum in another operation, we must be sure that all the terms have been calculated. It can be done by using the command **_syncthread()**.

```
1 void C_Vector_Addition(float *a,float *b,float *a_add_b,int N)
{
3   int i;

5   for(i=0;i<N;i++)
      a_add_b[i]=a[i]+b[i];
7 }
```

```
__global__ void CUDA_Vector_Addition(float *a,float*b,float*
    a_add_b,int N)
{
  int row = blockIdx.x * blockDim.x + threadIdx.x;
  if( row < N && row >= 0)
    a_add_b[row] = a[row]* b[row];

  __syncthreads();
}
```

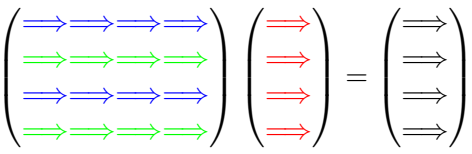
This code can't be used to solve equations like: $a = a + b$ (typically done in c)because you can't have accessed at positions to write until all operations are done. Change this way of thing is one of the most typical mistakes when programming in CUDA. [14]

5.3.3 Matrix vector multiply in CRS

Although the matrix-vector operation is well known, it should be noted that the matrix is stored in CRS format and we have to choose properly the instructions to perform for each thread of code.
For a matrix of dimension N this operation implies $N^2 + N$ input data and N output data.

$$\begin{pmatrix} a_{11} & a_{12} & \dots & a_{1N} \\ a_{21} & a_{22} & \dots & a_{2N} \\ \vdots & \vdots & \ddots & \vdots \\ a_{N1} & a_{N2} & \dots & a_{NN} \end{pmatrix} \begin{pmatrix} x_1 \\ x_2 \\ \vdots \\ x_N \end{pmatrix} = \begin{pmatrix} \sum_{i=1}^{N} a_{1i}x_i \\ \sum_{i=1}^{N} a_{2i}x_i \\ \vdots \\ \sum_{i=1}^{N} a_{Ni}x_i \end{pmatrix} \tag{5.3.4}$$

Each term of the multiply vector is a sum of products of the same row.



(5.3.5)

We can compute each element with one thread. The kernel to compute matrix vector operation with a matrix storage in a two dimensional array.

```
__global__ void Matrix_Vector_GPU(float **A,          // Matrix
                      float *vector_in,  // Input vector
                      float *vector_out, // Output vector
                      int N)             //Dimension
{
  int i; //iteration index
  int row = blockIdx.x * blockDim.x + threadIdx.x;
  if( row < N && row >= 0)
    for( j =0;j < N; j++)
      vector_out[row]+= A[row][j]*vector_in[j]
```

```
12   __syncthreads();
}
```

The kernel to compute matrix vector operation with a matrix storage in CRS format.

```
__global__ void Matrix_Vector_CRS_GPU(  float *data,    // Matrix
    in CRS format
                int *col_position,
                int *cout,
                float *vector_in,  // Input vector
                float *vector_out  // Output vector
                int N)             //Dimension
{
  int i;                                    //index
  int start,end;                            //iteration limits
  int row = blockIdx.x * blockDim.x + threadIdx.x;
  float sum;
  if( row < DIM  && row >= 0)
    {
    sum   = 0.0;
    start = cout[row];  //cout[k] tell us where the k row begins
    end   = cout[row+1];//cout[k+1] tell us where the k row ends

    for( i = start ; i < end ; i++)
      sum+= data[i] * vector_in[col_position[i]];

    vector_out[row]=sum;
    }
    __syncthreads();
}
```

5.3.4 SAXPY (Single-precision real Alpha X Plus Y)

At each step of the algorithm we have to recalculate x, r and p. All these calculations are of the form:

$$x^{(m)} \longleftarrow x^{(m-1)} + \alpha^{(m)} p^{(m-1)} \tag{5.3.6}$$

$$r^{(m)} \longleftarrow r^{(m-1)} - \alpha^{(m)} A p^{(m-1)} \tag{5.3.7}$$

$$p^{(m)} \longleftarrow r^{(m)} + \beta^{(m)} p^{(m-1)} \tag{5.3.8}$$

All of them depend on the same in the previous step. These operations are basically additions, and the kernel required to perform this operation is simple.

```
__global__ void Add_Vectors(float *a, // first input vector
                float *b, // second input vector
                float *saxpy, // addition vector
                float sign1,
```

```
                float sign2,
                int N)
{
  int row = blockIdx.x * blockDim.x + threadIdx.x;
  if( row < N && row >= 0)
    {
      saxpy[row] = sign1* a[row] + sign2* b[row];
    }
    __syncthreads();

    a[row]=saxpy[row];
}
```

Despite its simplicity is kind of operations require an intermediate vector to store the result and copy it to the original vector.The **syncthreads** command is essential because the elements of the a vector will not be overwritten until all them are calculated.
One of the premises in GPU computing is to reduce the stored memory. To take full advantage of the GPU capabilities is necessary to consider these details. Nvidia provides libraries in which these functions are optimized [38]. SAXPY (Single-precision real Alpha X Plus Y) multiplies the vector x by the scalar α and adds it to the vector y overwriting the latest vector with the result. Hence, the performed operation is $y[j] = \alpha x[j] + y[j]$. [13]

5.4 Structure of the code

The code has been structured as follows:

1.- To generate a sparse symmetric positive definite matrix A. It's the kind of matrices generated by hp-FEM. A is the stiffness matrix.

2.- To generate a solution vector $\overrightarrow{x}_{sol}$.

3.- To generate a vector $\overrightarrow{b}$ with the matrix-vector product $A\overrightarrow{x}_{sol} = \overrightarrow{b}$

4.- Using an algorithm implemented in CUDA the system of equations $A\overrightarrow{x} = \overrightarrow{b}$ is solved. The execution time t_{cuda} is measured.

5.- Using an algorithm implemented in C parallel the system of equations $A\overrightarrow{x} = \overrightarrow{b}$ is solved. The execution time $t_{parallel}$ is measured.

6.- We compare execution times $\dfrac{t_{cuda}}{t_{parallel}}$

The vector $\overrightarrow{x}_{sol}$ allows us to check the obtained solutions.

Chapter 6

Results

The results exposed in this chapter was obtained with a version of the program that works with static memory. This fact, only allows to arrive to dimension 40000 (approximately the size of the RAM memory of the CPU in double precision). This limitation is a problem to show the advantages of GPU computing. In this order of size the CUDA implementation of the solver works between 2-3 times faster in GPU than CPU. For lower size (the order of thousands) CUDA does not improves CPU implementation, the GPU programs spent the most of time copying within CPU and GPU.

For higher dimension, the implementation of the program with dynamic memory provokes segmentation faults in the execution. With dynamic memory sizes of millions can be easily reach.

Another fact relevant is that the hp-FEM in Laplace problem generates an structure of matrix that is usually solved in a reduce number of iterations (for moderate tolerances). For dimension 40000, the program finish in less than 4000 iterations.

Chapter 7

Conclusions and future work

The CUDA implementation of a iterative solver versus CPU implementation only exhibits improvements at higher dimension. At lower dimensions it is most practical CPU implementation. At this dimension, all the speed improvements in the basic operations are neglected by time of CPU to GPU copying process.

In our future work we want to solve the problem with dynamic memory implementation. The solve of this problem will provide us a very powerful to solve systems of linear equations. Also, we want to be able to implement a preconditioner. It will allow us to face higher dimension (tens of millions).

Bibliography

[1] *Amhdal's law, wikipedia.* http://en.wikipedia.org/wiki/Amdahl's_law.

[2] *Cholesky decomposition, math-linux.* http://www.math-linux.com/spip.php?article43.

[3] *Conjugate gradient method, math-linux.* http://www.math-linux.com/spip.php?article54.

[4] *Gaussian elimination, math-linux.* http://www.math-linux.com/spip.php?article53.

[5] *General-purpose computation on graphics hardware.* http://gpgpu.org/.

[6] *Lu decomposition, math-linux.* http://www.math-linux.com/spip.php?article51.

[7] M. Ainsworth and J. Oden, *A procedure for a posteriori error estimation for hp finite element methods*, Computer Methods in Applied Mechanics and Engineering, 101 (1992), pp. 73–96. Elsevier.

[8] S. Akhter and J. Roberts, *Multi-Core Programming*, vol. 33, 2006. Intel Press.

[9] G. Almasi and A. Gottlieb, *Highly parallel computing*, (1988). Menlo Park, CA (USA); Benjamin-Cummings Pub. Co.

[10] I. Babuška and B. Guo, *The h, p and h-p version of the finite element method: basis theory and applications, advances in engineering software.*

[11] M. Baskaran and R. Bordawekar, *Optimizing sparse matrix-vector multiplication on gpus*, IBM Research Report, (2008).

[12] C. Baumann and J. Oden, *A discontinuous hp finite element method for the euler and navier–stokes equations*, International Journal for Numerical Methods in Fluids, 31 (1999), pp. 79–95. Wiley Online Library.

[13] N. Bell and M. Garland, *Efficient sparse matrix-vector multiplication on cuda*, NVIDIA Corporation, NVIDIA Technical Report NVR-2008-004, (2008).

[14] ——, *Implementing sparse matrix-vector multiplication on throughput-oriented processors*, in Proceedings of the Conference on High Performance Computing Networking, Storage and Analysis, ACM, 2009, p. 18.

[15] J. Bolz, I. Farmer, E. Grinspun, and P. Schröder, *Sparse matrix solvers on the gpu: conjugate gradients and multigrid*, in ACM SIGGRAPH 2005 Courses, SIGGRAPH '05, New York, NY, USA, 2005. ACM.

[16] "C.K.Filelis-PapadopoulosandG.A.GravvanisandP.I.MatskanidisandK.M.Giannout *"on the gpgpu parallelization issues of finite element approximate inverse preconditioning"*, "Journal of Computational and Applied Mathematics", "InPress ("2011"), pp. "–". "".

[17] D. Culler, J. Singh, and A. Gupta, *Parallel computer architecture: a hardware/software approach*, 1999. Morgan Kaufmann.

[18] J. Davidson and S. Jinturkar, *Memory access coalescing: a technique for eliminating redundant memory accesses*, ACM SIGPLAN Notices, 29 (1994), pp. 186–195. ACM.

[19] T. Dokken, T. R. Hagen, and J. M. Hjelmervik, *The gpu as a high performance computational resource*, in Proceedings of the 21st spring conference on Computer graphics, SCCG '05, New York, NY, USA, 2005, pp. 21–26. ACM.

[20] W. Fung, I. Sham, G. Yuan, and T. Aamodt, *Dynamic warp formation and scheduling for efficient gpu control flow*, in Proceedings of the 40th Annual IEEE/ACM International Symposium on Microarchitecture, IEEE Computer Society, 2007, pp. 407–420.

[21] D. Geer, *Chip makers turn to multicore processors*, Computer, 38 (2005), pp. 11–13. IEEE.

[22] D. Göddeke, R. Strzodka, and S. Turek, *Accelerating double precision FEM simulations with GPUs*, in Proceedings of ASIM 2005 - 18th Symposium on Simulation Technique, Sep 2005.

[23] D. Grunwald and B. Zorn, *Customalloc: Efficient synthesized memory allocators*, Software: Practice and Experience, 23 (1993), pp. 851–869. Wiley Online Library.

[24] M. Hill and M. Marty, *Amdahl's law in the multicore era*, Computer, 41 (2008), pp. 33–38. IEEE.

[25] P. Houston, C. Schwab, and E. Süli, *Discontinuous hp-finite element methods for advection-diffusion-reaction problems*, SIAM Journal on Numerical Analysis, (2002), pp. 2133–2163. JSTOR.

[26] E.-J. Im, K. Yelick, and R. Vuduc, *Sparsity: Optimization framework for sparse matrix kernels*, Int. J. High Perform. Comput. Appl., 18 (2004), pp. 135–158. Sage Publications, Inc.

[27] J. JEAN and S. GRAILLAT, *A parallel algorithm for dot product over word-size finite field using floating-point arithmetic*, 12th International Symposium on Symbolic and Numeric Algorithms for Scientific Computing, (2010).

[28] C. Jhonson, *Numerical Solutions of partial differential equations by finite element methods*, 1987. Cambridge University Press.

[29] N. Kim, T. Austin, D. Baauw, T. Mudge, K. Flautner, J. Hu, M. Irwin, M. Kandemir, and V. Narayanan, *Leakage current: Moore's law meets static power*, Computer, 36 (2003), pp. 68–75. IEEE.

[30] Z. Koza, M. Matyka, S. Szkoda, and Ł. Mirosław, *Compressed multiple-row storage format*, Arxiv preprint arXiv:1203.2946, (2012).

[31] J. Krüger and R. Westermann, *Linear algebra operators for gpu implementation of numerical algorithms*, ACM Transactions on Graphics, 22 (2003), pp. 908–916.

[32] V. Kumar, A. Grama, A. Gupta, and G. Karypis, *Introduction to parallel computing*, vol. 110, 1994. Benjamin/Cummings USA.

[33] E. Lindholm, J. Nickolls, S. Oberman, and J. Montrym, *Nvidia tesla: A unified graphics and computing architecture*, Micro, IEEE, 28 (2008), pp. 39–55. IEEE.

[34] K. Liu, X. Wang, Y. Zhang, and C. Liao, *Acceleration of time-domain finite element method (td-fem) using graphics processor units (gpu)*, in Antennas, Propagation & EM Theory, 2006. ISAPE'06. 7th International Symposium on, IEEE, 2006, p. 1–4.

[35] J. Nickolls, I. Buck, M. Garland, and K. Skadron, *Scalable parallel programming with cuda*, Queue, 6 (2008), pp. 40–53. ACM.

[36] Nvidia, *The CUDA Compiler Driver NVCC.*

[37] C. NVidia, *Compute unified device architecture programming guide*, (2007).

[38] C. NVIDIA, *Cublas library*, NVIDIA Corporation, Santa Clara, California, 15 (2008).

[39] C. Nvidia, *Programming guide*, 2008.

[40] T. Oberhuber, A. Suzuki, and J. Vacata, *New row-grouped csr format for storing the sparse matrices on gpu with implementation in cuda*, Arxiv preprint arXiv:1012.2270, (2010).

[41] J. Owens, D. Luebke, N. Govindaraju, M. Harris, J. Krüger, A. Lefohn, and T. Purcell, *A survey of general-purpose computation on graphics hardware*, in Computer graphics forum, vol. 26, Wiley Online Library, 2007, pp. 80–113.

[42] C. Pagot, J. Vollrath, J. Comba, and D. Weiskopf, *A fast gpu particle system approach for isocontouring on hp-adaptive finite element meshes.*

[43] P. Plaszewski, K. Banas, and P. Maciol, *Higher order fem numerical integration on gpus with opencl*, in IMCSIT, 2010, pp. 337–342.

[44] M. J. Quinn, *Parallel computing (2nd ed.): theory and practice*, New York, NY, USA, 1994. McGraw-Hill, Inc.

[45] M. R.Hestenes and E. Stiefel, *Methods of conjugate gradients for solving linear systems*, Research of National Bureau of Standards, 49 (1952), p. 2379.

[46] S. Ryoo, C. Rodrigues, S. Stone, S. Baghsorkhi, S. Ueng, J. Stratton, and W. Hwu, *Program optimization space pruning for a multithreaded gpu*, in Proceedings of the 6th annual IEEE/ACM international symposium on Code generation and optimization, ACM, 2008, pp. 195–204.

[47] R. Schaller, *Moore's law: past, present and future*, Spectrum, IEEE, 34 (1997), pp. 52–59. IEEE.

[48] J. Shewchuk, *An introduction to the conjugate gradient method without the agonizing pain*, 1994. Carnegie Mellon University, Pittsburgh, PA.

[49] J. Stone, D. Gohara, and G. Shi, *Opencl: A parallel programming standard for heterogeneous computing systems*, Computing in science & engineering, 12 (2010), p. 66. NIH Public Access.

[50] D. Tarditi, S. Puri, and J. Oglesby, *Accelerator: using data parallelism to program gpus for general-purpose uses*, in ACM SIGARCH Computer Architecture News, vol. 34, ACM, 2006, pp. 325–335.

[51] J. V. Tomáš Oberhuber, Atsushi Suzuki, *New row-grouped csr format for storing the sparse matrices on gpu with implementation in cuda.*, (2010).

[52] M. Wozniak, T. Olas, and R. Wyrzykowski, *Parallel implementation of conjugate gradient method on graphics processors*, Parallel Processing and Applied Mathematics, (2010), pp. 125–135. Springer.

[53] O. C. Zienkiewicz, R. L. Taylor, and J. Z. Zhu, *The finite element method: its basis and fundamentals*, 2005. Butterworth-Heinemann.

[54] C. Zou, C. Xia, and G. Zhao, *Numerical parallel processing based on gpu with cuda architecture*, in Wireless Networks and Information Systems, 2009. WNIS'09. International Conference on, IEEE, 2009, pp. 93–96.