Politecnico di Milano

Facoltà di Ingegneria dei Sistemi
Corso di Laurea in Ingegneria Matematica

"A framework to implement a multifrontal scheme on GPU architectures with OpenCL"

Relatore:
prof. L. Formaggia
Politecnico di Milano

Correlatore:
prof. S. Deparis
EPFL Lausanne

Federico Edoardo Bonelli (mat. 739446)
Anno Accademico 2010/2011

Abstract

In this work we analyze an open-source multifrontal solver implementation (UMFPACK) and modify it to transfer the computation load on an OpenCL device, typically a GPU. To achieve this result the dbOpenCL library has been created, which allows a neat integration of OpenCL code into existent C or C++ code. An analysis and profiling of both the original and the modified UMFPACK implementation is given while solving an example problem built with the LifeV finite element library.

In questo lavoro analizziamo una implementazione open-source di un risolutore multifrontale (UMFPACK). Questa implementazione è in seguito modificata per trasferire le operazioni computazionalmente intensive su una scheda grafica o un qualunque dispositivo compatibile con OpenCL. Per ottenere questo risultato abbiamo realizzato la libreria dbOpenCL, che permette una facile integrazione di codice OpenCL in programmi C o C++ già esistenti. Effettuiamo inoltre il profiling e l'analisi di entrambe le versioni di UMFPACK mentre fattorizzano una matrice di esempio prodotta dalla libreria di risoluzione di problemi ad elementi finiti LifeV.

Contents

Chapter 1

Introduction

Multifrontal methods are very performant direct methods for matrix factorization. These methods are especially useful in computational fluid dynamic problems and mathematical simulation programs, but they are extensively used in many other applications. To perform these factorizations on different, potentially more powerful, hardware would allow faster resolutions, hence improving performances on many applications. For this reason the work aimed at the implementation of a modified version of UMFPACK (Unsymmetric Multifrontal Solver) which exploits graphic hardware for the full matrix operations. This is achieved through the use of the OpenCL library and language for parallel computation devices, which allows the use of GPUs (Graphics Processing Units) as well as other high performance devices, like multi core CPUs. In the process of developing such a modified version of UMFPACK, an OpenCL specific library has been developed. This library, called dbOpenCL, has the main purpose of being easy to integrate within existing C and C++ codes. This would permit to easily modify many computational intensive applications in order to use OpenCL and GPUs.

However, the use of external devices for computation purposes poses several issues and overhead costs that could slow down the resolution. These have to be checked case by case to ensure that they do not overwhelm the eventual advantages. In this thesis I analyze a test case realized with the LifeV library for distributed finite element problems with a domain decomposition approach. The role of UMFPACK in such a computation is explained and the profiling of a test execution is detailed. Profiling has been made for both the original and the modified version of UMFPACK, to understand the consequences of the integration of OpenCL.

In chapter 2 we briefly explain the theoretic mechanism lying behind our finite element test case and how this problem is solved using a cluster of computers through a domain decomposition approach. Our test case becomes, with this method, a parallel problem: therefore an appropriate introduction to parallel computing is given. In the same chapter we give a short explanation of the Gaussian elimination method, since its comprehension is necessary to understand the multifrontal method.

In chapter 3 we will give a description of the frontal and multifrontal methods, as well as an explanation of why they give better performances then the usual Gaussian elimination. In the same chapter we focus our analysis on the profiling

of UMFPACK. This was an important preliminary step of the work, because it was useful to decide which parts of UMFPACK to translate in OpenCL.

In chapter 4 we explain how the graphic hardware can be used for a general purpose computation. This poses different problems and opportunities: both are detailed in the chapter. Different languages permit the use of GPUs for computing problems, but in this work we focus on OpenCL, mainly because it is supported by a vast quantity of devices. A description of the OpenCL library and language is given, as well as an explanation of the main concepts necessary to handle OpenCL programs.

In chapter 5 a description of the work on the dbOpenCL library is given. The objectives of the library are stated and a design is given which respect the prefixed goals. We will detail some internal mechanics of the library and some of the design patterns used. Finally we will show in detail how the dbOpenCL library is used within the UMFPACK to achieve a smooth integration of OpenCL code into the existing plain C code.

Finally in chapter 6 we show in details how and in which parts the original UMFPACK implementation has been modified. A profiling of the modified UMFPACK factorization is done in the same chapter, allowing us to give the conclusive remarks.

Chapter 2

Large scale fluid dynamics problems

Solving fluid dynamics problems is a computationally demanding and important task for many engineering applications. There are countless fields where a simulation is theoretically feasible but hardly computed because of many factors, and the main one is the need of sufficient computing power available for computing the solution in an acceptable time. Hardware sure have improved in the last decades, but so have the problems we try to solve: as a result we are always in the need of new ways to improve our computational power.

The kind of problem we are interested in is the solution of partial differential equations (PDE), since such are the computational fluid dynamics simulations, which can be solved with different methods. The finite element method (FEM) is a widely known technique, but many others are possible: the finite volumes method, finite difference method, or the spectral element method, which is in final analysis a high order FEM which gives interesting results in some applications. This thesis has focused on the finite element method, therefore we will be speaking particularly of this method, but many considerations will be easy to adapt to the others we mentioned.

2.1 Finite Element Methods for elliptical problems

2.1.1 The homogeneous Poisson problem: strong and weak formulations

We will now briefly introduce a typical PDE and how it is solved with the Finite Element method. As an example we take the homogeneous Poisson problem defined on a domain $\Omega \subset \mathbb{R}^2$ with a boundary called $\partial\Omega$: the problem consists in finding u so that:

$$\begin{cases} -div(\mu \nabla u) = f & \text{in } \Omega \\ u = 0 & \text{on } \partial\Omega \end{cases} \tag{2.1}$$

where $\mu \in L^\infty(\Omega)$ and $\mu \geq \mu_0 > 0$.

If we try to solve the Poisson problem as it is we ought to look for a solution $u \in C^2(\Omega)$, because u must be derived twice. Real applications show that this is not always necessary and, on the contrary there are many cases in which $u \in C^2(\Omega)$ is much too restrictive. We therefore call this kind of problem the strong formulation of the Poisson problem, whose solution must be found in $C^2(\Omega)$, and we admit a much less restrictive problem to be called weak.

The weak formulation of the Poisson problem is an integral problem, and it can be derived from the strong formulation:

$$-\int_\Omega \Delta u v \, d\Omega = \int_\Omega f v \, d\Omega$$

we multiplied both side by an arbitrary test function v, and we subsequently integrated on the Ω domain. If the test function v is sufficiently regular we can now use the integration by parts to find Green's identity for the laplacian operator. As a result we obtain a more symmetric formulation:

$$\int_\Omega \nabla u \cdot \nabla v \, d\Omega - \int_{\partial\Omega} \frac{\partial u}{\partial n} v \, d\gamma = \int_\Omega f v \, d\Omega$$

If we choose v wisely as being $v|_{\partial\Omega} = 0$, then the line integral on the boundary is null. This gives us the weak formulation for the homogeneous Poisson problem:

$$\text{find } u \in H_0^1(\Omega) \ : \qquad \int_\Omega \nabla u \cdot \nabla v \, d\Omega = \int_\Omega f v \, d\Omega \quad \forall v \in H_0^1(\Omega) \tag{2.2}$$

where $H_0^1(\Omega)$ is the space of functions over Ω which are square integrable and whose first derivative is square integrable. This is a far bigger space than $C^2(\Omega)$, thus it poses weaker constraints on the solution. To simplify notations we will henceforth refer to the spaces $H_0^1(\Omega)$ or $H^1(\Omega)$ as space V, since many results are valid no matter in which space the solution lies.

This weak problem is much more convenient than the strong one in applications that permit f to be a generalized function, such as in deformation problems, where we might want to study a concentrated load applied on a single point. In that case $f = \delta_{\text{Dirac}}$ and the equation (2.1) could not possibly be satisfied by any $u \in C^2(\Omega)$.

We will often refer to the weak formulation using the associated bilinear form a and functional F, defined in this case as follows:

$$a \ : \ V \times V \to \mathbb{R}, \ a(u,v) = \int_\Omega \nabla u \cdot \nabla v \, d\Omega$$

$$F \ : \ V \to \mathbb{R}, \qquad F(v) = \int_\Omega f v \, d\Omega$$

Now that the definitions are made, we can rewrite the weak formulation problem (2.2) as:

$$\text{find } u \in V \ : \qquad a(u,v) = F(v) \qquad \forall v \in V \tag{2.3}$$

2.1.2 The approximate problem

Until now we only spoke of the exact problem, in strong and weak formulation. In the simulation we are forced to use the approximate problem: this happens for at least two reasons. The first one is that computers cannot handle true real numbers, but only their finite approximation, so it would be impossible for a computing device to find a solution u as simple as a constant irrational $u|_\Omega = \pi$. The second reason is that computers are provided with finite memory, therefore it is impossible for them to save a function's values on all its continuous domain points.

Since these problems are shared by all of the branches and methods of numerical analysis, there are theorems that show conditions under which a numerical method could prove worthy. Let us focus on the most important of these theorems, starting with some other definitions.

We define an exact problem $\mathcal{P}$ with some exact known parameters g and an exact solution u. In the same way we have the approximate problem $\mathcal{P}_N$ with g_N and u_N which are respectively the approximate known parameters and solution. We can, without loss of generality, say that these problems are solved when solutions u and u_N are found so that:

$$\begin{aligned} \mathcal{P}(u, g) &= 0 \\ \mathcal{P}_N(u_N, g_N) &= 0 \end{aligned}$$

The integer number N represents the dimension of the approximate problem, and it is a limited number mainly because of the real device memory limitation.

Definition. A numerical method is convergent if $\|u - u_N\| \to 0$ when $N \to \infty$, using an appropriate norm.

The convergence of a method is the final goal: it guarantees us that if we had enough memory and enough computational power we could achieve an approximate solution as close as desired to the exact solution.

Definition. A numerical method is consistent if $\mathcal{P}_N(u, g) \to 0$ for $N \to \infty$. The latter is said to be strongly consistent if $\mathcal{P}_N(u, g) = 0 \; \forall N$.

The consistency is a feature of the method itself, as it does not depend on the approximate solution or input data. We obviously need to ensure that the approximation error of the initial data does not influence the approximate result too heavily.

Definition. A numerical method is stable if small perturbations of the data g_N give small variations of the solution. More precisely:

$$\forall \epsilon > 0 \; \exists \delta = \delta(\epsilon) > 0 \; : \; \forall \delta g_N \; : \; \|\delta g_N\| < \delta \Rightarrow \|\delta u_N\| \leq \epsilon \; \forall N$$

where $u_N + \delta u_N$ is the solution of the problem with perturbed data:

$$\mathcal{P}_N(u_N + \delta u_N, g_N + \delta g_N) = 0$$

Finally we arrive at this very important statement, the equivalence theorem, which lets us prove the convergence of a method through its consistency and stability, that are often far easier to prove:

Theorem. *A consistent method is convergent if and only if it is stable.*

2.1.3 The Galerkin method

Let a be the bilinear form associated to an elliptical problem, which may resemble our previous Poisson problem. This can be written on the domain $\Omega \in \mathbb{R}^d$ as follows:

$$\text{find } u \in V : \qquad a(u,v) = F(v) \qquad \forall v \in V \tag{2.4}$$

where $V \subset H^1(\Omega)$, a is a bilinear and coercive form, F is a limited functional. Under these conditions both the existence and the uniqueness of the solution are granted. We now want to approximate this problem using a different space for the solution: let $V_h \subset V$ be a family of subspaces dependent on the parameter $h > 0$, each of them of finite dimensions $dim(V_h) = N_h < \infty$.

The Galerkin problem, defined as follows, is our approximate problem:

$$\text{find } u_h \in V_h : \qquad a(u_h, v_h) = F(v_h) \qquad \forall v_h \in V_h \tag{2.5}$$

Once again we stress the fact that each V_h must be of finite dimension, otherwise it will not be possible in any way for a computer to represent a function u or v in V_h. Since V_h is of finite dimension, we can enumerate a base $\{\varphi_j, j = 1, 2, \ldots, N_h\}$. Having that both a and F are linear in v_h, the equation (2.5) is verified if and only if it holds for each base function φ_j, since every possible $v_h \in V_h$ will be a linear combination of the latter. In the exact same way we use the fact that a is linear in its first argument $u_h \in V_h$. Indeed, since

$$u_h(x) = \sum_{j=1}^{N_h} u_j \varphi_j(x)$$

with $u_j \in \mathbb{R}$, we can write the following system as an equivalent formulation of the equation (2.5):

$$\sum_{j=1}^{N_h} u_j a(\varphi_j, \varphi_i) = F(\varphi_i), \qquad i = 1, 2, \ldots, N_h \tag{2.6}$$

Among all the interesting aspects of this formulation, the most important of all surely is that it is a linear system: let us define the matrix A, called the stiffness matrix, and the vector $\mathbf{f}$

$$\begin{aligned} a_{ij} &= a(\varphi_i, \varphi_j) \\ f_i &= F(\varphi_i) \end{aligned}$$

If we define the vector $\mathbf{u}$ using the coefficients u_j we can express the equation (2.6) in this familiar way:

$$A\mathbf{u} = \mathbf{f}$$

where $\mathbf{u}$ is the unknown vector, which can be found simply by solving the linear system. The Galerkin method we exposed is stable, strongly consistent and its solution exists and is unique, under the hypotheses we assumed for the problem (2.4).

2.1.4 The finite element method: performance considerations

The finite element method is nothing but a wise choice of subspaces V_h for the Galerkin method, hence it inherits all its good properties. To describe the subspaces of the finite element method we should start with the concept of mesh: let our PDE be defined on the domain $\Omega \in \mathbb{R}^d$, where usually $d = 1, 2, 3$.

For the finite element method we must partition the domain Ω into a triangulation (because for $d = 2$ these are in fact triangles) $\mathcal{T}_h$. This is composed of sub-regions $K \in \mathcal{T}_h$, so that

$$\Omega_h = int(\cup_{K \in \mathcal{T}_h} K)$$

where $int(\cdot)$ represents the inner part of the region defined by $\mathcal{T}_h$, as we do not bother now with the borders, we will be adding them in a few moments. The triangulation should be such that $\Omega_h \to \Omega$ if $h \to 0$, with h being a positive real parameter linked to the dimension of the mesh step, somehow playing the role of $\frac{1}{N}$ with the convention of the section 2.1.2.

Once the triangulation $\mathcal{T}_h$ is defined we introduce the finite element method space

$$V_h = X_h^r = \left\{ v_h \in C^0(\overline{\Omega}_h) : v_h|_K \in \mathbb{P}_r, \forall K \in \mathcal{T}_h \right\}$$

which is the space of continuous functions over the closure of the approximated domain Ω_h and, at the same time, polynomials of degree r on each element K of the triangulation. Without much of a change we also define the space

$$\overset{\circ}{X}{}_h^r = \{ v_h \in X_h^r : v_h|_{\partial\Omega_h} = 0 \}$$

These spaces are respectively sub-spaces of $H^1(\Omega_h)$ and $H_0^1(\Omega_h)$, and as the degree r of the polynomials grows they tend to fill their parent space. Quite a few possible bases exist for the spaces X_h^r and $\overset{\circ}{X}{}_h^r$, but one class of them is particularly well fit for computing purposes. This is a class of bases with functions having a very small support. When such a basis is used, the integral of the generic $\int_{\Omega_h} D\varphi_i D\varphi_j \, d\Omega_h$ gives a null result on most combinations of i and j, which leads to a 0 in the corresponding element of the stiffness matrix. Since this causes the stiffness matrix to be very sparse and often divided into easily identifiable blocks, we are hence allowed to use particular time and memory saving algorithms.

Let us now focus on the computing-expensive parts of these methods. When assembling the stiffness matrix we must calculate its values, which means computing the integrals defining the weak formulation of our PDE problem. These integrals have to be found with quadrature rules, because the expression of the the functions involved, like μ, may not allow exact integration. In some problems this can be an expensive part of the whole computation. In any case this is an easy task to parallelize, because of the fact that most of the computing time is spent calculating the integrals on each single element.

However, the most expensive part of the finite element method usually is solving the linear system defined by the stiffness matrix.

There is also another big issue when solving such problems: the stiffness matrix can be of considerable size. In fact it is of size n by n, where n is the number of unknowns of the problem, and this is roughly proportional to the number of

elements of our simulation domain. This means, in full matrix notation, n^2 real (or sometime complex) numbers to stock in memory: even for small problems this matrix can be huge. The amount of data to be saved into memory for the stiffness matrix is an important problem. Fortunately it has been solved very efficiently with algorithms and matrix stocking formats that permit us to just save the useful information (non zeros entries) of the stiffness matrix. Since the stiffness matrix is often a sparse matrix, this lowers the number of entries to save to $\mathcal{O}(n)$, which is considerably better. This aspect will be analyzed in detail further on, when talking about the frontal and multifrontal methods. These methods considerably increase the size of matrices computers can keep in memory, considering the fact that the stiffness matrix will be very sparse if the basis for the finite element method has been chosen wisely.

2.2 How to tackle computationally intensive problems

The easiest way one could think of in order to achieve better computational power is to accelerate the speed of a single computing unit, without changing neither algorithm nor computational architecture. This is a very good solution, but it has become economically expensive: higher frequencies mean more heat to dissipate, and to achieve a better dissipation it is necessary to miniaturize the components further.

We call miniaturization the technological process that brings through time to higher densities of transistors on microchips of the same size. Since 1958, year when the first integrated circuits were produced, the miniaturization process brought to an exponential growth of this density. This fact was firstly observed by Gordon E. Moore, co-founder of Intel, in 1965.

Moore's empiric law has been respected flawlessly until now, both because he was right and because this law has been used for decades to plan the research effort of major microelectronics companies. However there are physical limits to this technological process: miniaturization cannot continue forever, the atomic scale is a sure limit for this exponential trend.

On a much larger scale there are problems in developing transistors smaller than approximately 16 nanometers, because of quantum tunneling effects. Whatever the ultimate limit for electronic miniaturization will be, it exists and technology is rapidly approaching it, hence different solutions will be mandatory when this will happen.

An alternate solution is to add computing units into what we can call a cluster. Indeed, if the algorithm permits to split the problem into different parts and then to merge them again to form the solution, then it is possible to use different computing devices simultaneously. In an ideal case, and with a perfectly parallel algorithm, the n different computing units will work concurrently and the solution will be available in just an nth of the time necessary for the same solution to be computed on a single CPU.

However, this is an extremely rare situation, as almost none of the existing algorithms is perfectly parallel, and a lot of research is needed to achieve a good degree of concurrency in solving even some of the easiest problems. On the other hand any effort in this direction is well worth it, because of the low cost of modern

computational units: in 2010 an easily affordable personal computer can have two or four cores, and servers have 16 cores as a standard. With a reasonably small investment any laboratory could afford a cluster with dozens of cores, which could deliver an impressive amount of computational power at a very limited cost, without even considering the possibility of using vector processors. In this thesis we will take a further step and discuss their use in the chapter on the General Purpose GPU.

Because of the low performances achieved with classic algorithms on a cluster of computers, a lot of effort is put into the study of new algorithms and methods that are capable to reach a high level of parallelization. Upon studying such methods one must keep in mind some specific problems that normally do not arise when considering classic algorithms.

2.2.1 Overview of parallel computing paradigms

We might want to expose some example of parallel computing before to list some of the problems that arise when it is performed. We say a computation to be parallel if it is carried out concurrently on different computing devices.

Many forms of parallel computing are done without even being noticed. For example when a CPU performs an integer sum it is performing the same operation over multiple bits, simultaneously. This is called bit level parallelism. The more bits a CPU can manage in a single operation, the more parallel it is.

Another more complex form of parallelism is called instruction level parallelism. This is the capability of a CPU to perform different operations at the same time.

Many complex operations need a sequence of different low level operations to be performed. Different transistors of a CPU are dedicated to each low level operation. Using them simultaneously permits to carry out simultaneously different low level operations. A complex operation is achieved once a complete sequence of low level operations is finished. This must be a sequence, hence low level operations cannot be done concurrently. Therefore the concurrent low level operations are taken by different complex operations, with the result that multiple complex operations are running simultaneously.

Similarly there are super scalar computing units, which have multiple areas dedicated to the same low level operation. These can run multiple low level operations at the exact same time. Using a super-scalar instruction-parallel computing unit would result in even more complex operations executed concurrently.

Super scalar computation needs unordered schedulers to decide which instruction can be parallelized without affecting the outcome of the computation. In most of the algorithms, in fact, it is important to maintain the operations in the right order. To stir them randomly can result in undesirable effects.

All of the previous kind of parallel computation paradigms are performed at hardware level, without much possibility for controlling them from within a program.

A different kind of parallel paradigm is the data parallelism. This is a possibility for algorithms that perform the same operation on multiple data. Usually such an algorithm uses loops. If instead we were to distribute different chunks of data to different computers, we could decrease the total computation time to a fraction. This is our first example of cluster. Data parallelism can be performed efficiently by vector computers, or by GPUs, which can perform the same operation on large

quantities of variables.

The last kind of parallel paradigm is the task parallelism. If an algorithm needs different computations to be finished, and those computations are not to be done in a specific order, we can distribute them on different computing units.

Once the right kind of parallelism is found for a certain algorithm we must consider other factors. For example it can happen that an algorithm is only partially parallelizable.

Gene Amdahl firstly pointed out that if an algorithm is made by sequential and parallel sections, the time spent for the sequential sections is a lower bound for the total computation time. This is called Amdahl's law. For example we could have an algorithm that solves a problem on a cluster and computes the post-processing on a single workstation.

If the post-processing alone takes 1 hour, it is impossible for the whole algorithm to run in less than 1 hour, no matter how many nodes we have in the cluster.

Scalability of the problem is another problem. Some problems, architectures or algorithms are said to scale well if the amount of calculation does not vary with the number of computing units involved. If T is the time needed to compute a solution on a single computing unit, a distributed computer with N computing units can find the solution of a perfectly scalable problem in $T_s = \frac{T}{N}$ time.

Algorithms are usually non perfectly scalable. Therefore we have that the total time T_{ns} needed for computing the solution on a N-node distributed computer is $T_{ns} > \frac{T}{N}$.

If T_{ns} is too much bigger of $\frac{T}{N}$ we say that the algorithm is badly scaled. These kind of algorithms are obviously not fitted to parallel computation.

2.2.2 Level of parallelism and kind of parallel computers

For a programmer the most interesting forms of parallelism are the data and task parallelism. This is because one can directly control how the computation is split. These paradigms can both be implemented using clusters or multi-core CPUs.

It is a difficult task to achieve good performances with such architectures, because of the need of synchronization and communication between computing units. There are algorithms that need different cluster nodes to communicate often, which is undesirable for the cluster performances. Instead other algorithms do not need any communication at all between nodes during the computation.

Depending on the quantity of communication and synchronization needed we classify the algorithms in fine-grained or coarse-grained synchronized, or embarrassingly parallel. Embarrassingly parallel algorithms are best computed with clusters because of their low need of communication.

The low need of communication allows cluster nodes to be "far" from one another. That means that the nodes can be distributed over slow networks, like the Internet, without impacting the cluster performance too much.

The more an algorithm is fine-grained, the more performant the communication layer of the parallel computer must be. Depending on the characteristic of the communication layer and computing units composing a parallel computer we may call it differently.

Parallel computers may be multi-core architectures: these are microchips integrating multiple cores. This kind of computer shares the same memory between

cores. A different possibility is a Symmetric MultiProcessor system, which is like a multicore computer with the difference that the different cores resides on different microchips. This allows many more computing units to use the same memory, but it poses different communication problems.

If the computing units are placed on different computers communicating through a network we call it a distributed computer. There are different kind of distributed computers depending on the speed of the network and the similarities between nodes.

Notably we call cluster a distributed computer with a high speed network. This is usually a dedicated network. Clusters are said to be homogeneous if the nodes are identical or at least similar in computing capabilities. A cluster is heterogeneous in the other case. Load balancing in heterogeneous clusters is a delicate matter, because one slow node could be a bottleneck for the whole cluster performance.

Opposite to the cluster, a grid is a highly distributed computer on very slow network, usually the Internet. This is often an heterogeneous distributed computer. Grids can be used only when an embarrassingly parallel algorithm is used, which often is given by a data parallelism.

2.2.3 The data exchange problem

An algorithm working in parallel on different computation units must spread the initial data before starting the calculation, and finally it must gather the information produced by each single unit, further on called a "node". In architectures like single multi-core computers this could at first thought be seen as a simple task, since everything is located in the same memory, and each core will be accessing the portions of data that it needs. Anyhow consistency and synchronization problems arise, and they are sometimes difficult to tackle efficiently.

In a cluster the situation can become worse: an effective algorithm must take into consideration that the transfer of data between different nodes is much slower than the computation itself, even by some orders of magnitude. It is not rare that after having successfully divided the problem and the necessary data into many chunks, more time is spent transferring each chunk to the appropriate computer than would be spent simply computing the solution all at once.

These are cases where the computation to be performed on each piece of data is not computationally intensive enough, even if the algorithm is perfectly parallel. To bring this into light, let us say that the algorithm has to work on d units of data, and that for each unit of data it has approximately w seconds of computing work to perform (we assume homogeneous nodes). Likewise, we can imagine an average of t seconds spent to transfer a single data unit. If our cluster is composed of N workers, and the data is originally stocked in one of them, the total time T in seconds to perform the computation is

$$T = \frac{d}{N-1}t + \frac{d}{N}w$$

If $T > dw$ we could simply have left the whole computation task to the worker that already had the data in its memory, since in that case the computation time would have been lower. The condition in which the parallelization proves to be worthy is whenever

$$T = \frac{d}{N-1}t + \frac{d}{N}w < dw$$

which after basic assumptions on the variables becomes

$$\frac{t}{w} < \frac{(N-1)^2}{N}$$

The latter holds in a very particular and favorable case:

- each computing unit in the cluster only needs its own part of input data, which is of size $\frac{d}{N}$.
- the output resulting from the work of each node is of negligible size.
- each computing unit does not need to wait for any other in order to finish its job. This means that the algorithm does not need any communication or synchronization between different nodes.

2.2.4 Synchronization between nodes of a cluster

If the problem is a little more complicated than the one we considered before, it may happen that a perfectly parallel algorithm does not exist. Actually, a very favorable case in real circumstances is to work with a problem which can be parallelized in many places, but in which the algorithm now and then has to gather the output of previously distributed jobs.

As an example, let us consider the behavior of an algorithm which has to perform different time steps, where the previous result decides which kind of operation is to be computed subsequently. A similar algorithm must ensure that all of the nodes have finished their own job before giving the command of starting the new time step. This can leas to a sub-optimal behavior when the cluster is heterogeneous, or when the time w needed to compute the necessary operations on a data unit can vary significantly between different chunks of data. In these cases the whole cluster could be forced to wait for one slow node to finish its job, wasting considerable computing power.

On a cluster with known node-performance, which only has to use one specific algorithm, statistical or probability studies can be performed in order to find out the odds of having one node slowing down the others. As a consequence, the load of data to give to every node can be balanced. These analyses are usually very complex but well worth the effort whenever applied to very heterogeneous clusters, or to distributed systems with huge quantities of nodes that are spread over slow networks.

2.2.5 Synchronization between threads, race conditions

Synchronization between different cores of the same computer is a much more complex matter than synchronization on distributed clusters. In fact, these two cases differ considerably, since cores on the same computer can share a common memory. To be precise it is not necessary to have different cores to encounter such a problem. Even with a single core it is generally possible to create multiple threads executing different parts of a program. Threads are autonomous executions which could run concurrently depending on the possibilities of the architecture.

If just one core is available, different threads will interleave their executions. This poses the exact same problems as if different threads had been scheduled on different cores, because one cannot control how they interleave.

However, for computing purposes it is a good thumb rule not to use more threads than the number of available cores. In fact, using too many threads would increase the time spent saving and restoring a thread state when it must let another one execute on the same core.

For this reason we will talk about cores executing concurrently, but we must remind that the same reasoning would still be true if we were talking about threads. Both different cores and different threads can access the same shared memory concurrently, and this poses problems that are difficult to manage.

In a straightforward way, an algorithm could force all of the threads to work as if they were different computing units on different nodes of a cluster (thus separating the memory areas on which they work). If this took place without sharing portions of memory between them, the synchronization needed would very much be the same as the one described before. As a drawback, such an algorithm has to take care of transferring data between the separate portions of memory, in a way similar to that used in the cluster, but with increased communication speed.

A slightly more efficient way to use the fact that the different threads share the same memory is to let each of them access the data freely as soon as it needs it. In such a system, however, we could lose memory consistency: if two or more threads are accessing the same bit of memory they could modify it in an unforeseen way, causing both of the threads to fail.

Let us highlight the problem through this classic example, in which we consider two threads that increase the same variable a by 1. If we start with the variable $a = 0$ we could have them to run one after the other:

```
(Thread A) read a from memory to register (a = 0)
(Thread A) increment register
(Thread A) write a from register to memory (a = 1)
(Thread B) read a from memory to register (a = 1)
(Thread B) increment register
(Thread B) write a from register to memory (a = 2)
```

If the two threads are running these same instructions simultaneously, the results can be unpredictable:

```
(Thread A) read a from memory to register (a = 0)
(Thread B) read a from memory to register (a = 0)
(Thread A) increment register
(Thread B) increment register
(Thread A) write a from register to memory (a = 1)
(Thread B) write a from register to memory (a = 1)
```

Since this kind of behavior can arise, a program running multiple threads must synchronize the access to shared resources. This means that some of the threads will be waiting idle for one of the others to finish its job and unlock the resource. This is normally not much of a computing time issue, but a really impelling coding problem, since debugging such unpredictable behaviors can be a very hard task.

More precisely the problem arises when the algorithm requires access to a resource in write mode. In a multi-threaded environment we must ensure that the

different threads will write into the same memory area avoiding behaviors like the one we have just shown.

When a resource is accessed in write mode concurrently, like in the example, we may encounter what is called a race condition. Great care must be taken to ensure that the resource is accessed in the correct way.

The problem is that race conditions can lead to unpredictable results, since the outcome of the computation depends on the order by which the different threads access the resource. This is not established beforehand, unless special directives are given explicitly by the programmer.

Those special directives guarantee what is called the synchronization between threads. Explicitly synchronized threads can avoid race conditions entirely.

2.2.6 Communication between cluster nodes

The situations we have considered so far consist of almost isolated nodes, with very simple communication needs. What would happen if instead we had to solve a problem where nodes have to communicate intensively in rather unusual ways? As an example we can take a distributed algorithm which simulates the flow of vehicles through traffic lights in a probabilistic way.

Let us set each node of the cluster to simulate a traffic light. Each traffic light lets cars flow towards the neighboring lights following a given probability law, depending on how many cars were waiting in line. Every node starts with a given amount of cars in line, and must draw 4 values from its probability law. The drawn values show the number of vehicles that went from that crossing to each one of the four roads departing from it. Once each node has computed these values, it must communicate with the others to tell how many cars have passed under each neighboring node's "jurisdiction". At this point a critical aspect of the problem becomes clear: the kind of communication that is needed can lead to various timing problems.

It could happen that some nodes that have probability laws from which it is particularly hard to generate keep building up delay with respect to other nodes. Since the work of one node is necessary as an input for other nodes to proceed with the computation, as a consequence we could have a whole region of the cluster being slower then the rest of it, with nodes which are unnecessarily idle. In addition there would be messages toward that slow region which have been sent from other faster regions, but which have not been elaborated yet. These would keep piling up, generating a memory issue.

If our algorithm wanted to track the path of each single car, it would have to tell its neighbor not only the number of vehicles, but also an identification number for each of them, let us say the license plate. In this case the amount of data to transfer at each time step depends on the output of the problem, and the whole communication system of the cluster could easily be the bottleneck, hence returning to a situation where a single node working alone could be faster than the whole cluster.

These example want to show how problems that are trivial in a scalar computing environment become very complicated when one wants to optimize them in a distributed memory parallel computer.

2.3 Domain Decomposition approach

We have already illustrated the importance of having an appropriate algorithm when trying to use clusters to quicken a simulation. For this reason mathematical models and problem formulations which are specifically fit for this purpose exist. Although their study began in the past for other reasons, they are now very well adapted for the implementation of distributed algorithms, allowing us to rely on convergence and stability theorems when using clusters.

The most naive way to spread a PDE problem into different chunks is to let the cluster work on a partitioned domain, with each node located in one of the sub-regions. Luckily this kind of algorithms work very well, even though the mathematical reasoning needed to understand them, and to make them perform so well, is all but naive.

These methods are, as a further advantage, apt for the resolution of heterogeneous problems, where different sub-regions are characterized by different PDEs. Since this is the way we actually use to build distributed simulation algorithms, it is well worth an appropriate introduction. A complete description of this kind of methods is available in the book [(4)] from A. Quarteroni.

We need the problem to be defined on the domain $\Omega \in \mathbb{R}^d$ with $d = 2, 3$, and the borders $\partial\Omega$ must be lipschitzian. Let L be a generic elliptical differential operator, so that the PDE problem is:

$$\begin{cases} Lu = f & \text{in } \Omega \\ u = \varphi & \text{on } \Gamma_D \\ \frac{\partial u}{\partial n} = \psi & \text{on } \Gamma_N \end{cases} \tag{2.7}$$

where Γ_D and Γ_N are the portions of the boundary with respectively Dirichlet and Neumann border conditions, such that $\Gamma_D \cup \Gamma_N = \partial\Omega$ and $\Gamma_D \cap \Gamma_N = \emptyset$, and φ and ψ are functions defined on Γ_D and Γ_N. There are two main ways to split this problem: one of them needs the different sub-regions of Ω to be overlapping, the other does not have this requirement.

Further on in this section we will be using an homogeneous pure Dirichlet problem as example, which simplifies (2.7) into:

$$\begin{cases} Lu = f & \text{in } \Omega \\ u = 0 & \text{on } \partial\Omega \end{cases} \tag{2.8}$$

The weak formulation of the latter is:

$$\text{find } u \in V = H_0^1(\Omega) \ : \qquad a(u, v) = F(v) \qquad \forall v \in V$$

and F is the functional defined by f, while a is the bilinear form associated to the operator L.

2.3.1 The Schwarz method

This method requires overlapping regions. If one splits the domain Ω into two sub-regions Ω_1 and Ω_2 we must assure that $\Omega_1 \cap \Omega_2 \neq \emptyset$, and obviously that $\overline{\Omega}_1 \cup \overline{\Omega}_2 = \overline{\Omega}$. The intersection between Ω_1 and Ω_2 will be called Γ_{12}, while the internal border of the region Ω_i will be called $\Gamma_i = \partial\Omega_i \backslash (\partial\Omega \cap \partial\Omega_i)$.

With a guess $u_2^{(0)}$ over the border Γ_1 the Schwarz method is written as follows:

$$\begin{cases} Lu_1^{(k)} = f & \text{in } \Omega_1 \\ u_1^{(k)} = u_2^{(k-1)} & \text{on } \Gamma_1 \\ u_1^{(k)} = 0 & \text{on } \partial\Omega_1 \backslash \Gamma_1 \end{cases} \tag{2.9}$$

$$\begin{cases} Lu_2^{(k)} = f & \text{in } \Omega_2 \\ u_2^{(k)} = \begin{cases} u_1^{(k)} \\ u_1^{(k-1)} \end{cases} & \text{on } \Gamma_2 \\ u_2^{(k)} = 0 & \text{on } \partial\Omega_2 \backslash \Gamma_2 \end{cases} \tag{2.10}$$

This defines two possible iterative methods, each of which must be solved for all steps k until convergence. If we choose to use $u_1^{(k)}$ as the border condition for the problem over Ω_2 we can compute a solution without any guess of u_1. Such an approach, called the multiplicative Schwarz method, will not permit the computations to be performed concurrently, which in our case is not desirable.

The other case, called the additive Schwarz method, is a good choice as a parallel algorithm. It is worth to observe, however, that these are iterative methods: in both of them we need to perform multiple resolutions of multiple smaller problems, instead of a single resolution of the same problem on a bigger domain.

This is not always a bad result: on the contrary both methods can prove useful when the problem on the original domain is too big. In fact if the unsplit problem generates a stiffness matrix of dimension N we now have the possibility to solve matrices which are approximately of size $\frac{N}{n}$ instead, where n is the number of sub-domains in which we split Ω. For this reason these method can be used even on a single computing unit with great performance improvements.

In order to make these methods work we need to ensure that

$$lim_{k\to\infty} u_i^{(k)} = u|_{\Omega_i} \qquad \forall i$$

or in other words we desire the solutions found on the sub-domains to converge to the real solution on the whole domain.

A theorem states that the Schwarz method on the problem (2.8) has a linear convergence to the real solution, depending on the size of the overlapping region Γ_{12}.

2.3.2 The Schur method, or Dirichlet-Neumann method

This is a method without overlapping sub-regions. Taking this into account, for our example we have to consider two regions Ω_1 and Ω_2 so that $\overline{\Omega}_1 \cup \overline{\Omega}_2 = \overline{\Omega}$ and $\Omega_1 \cap \Omega_2 = \emptyset$, but with $\overline{\Omega}_1 \cap \overline{\Omega}_2 = \Gamma$, as to say that their closures share a frontier. This method is granted to work by the equivalence theorem:

Theorem 1 (Equivalence theorem)**.** *Let u be the solution to the problem* (2.8)*. If we define $u_i = u|_{\Omega_i}$ for $i = 1, 2$,* then we have that each u_i may also be found as the solution of the following problem:

$$\begin{cases} Lu_1 = f & in\ \Omega_1 \\ u_1 = 0 & on\ \partial\Omega_1 \backslash \Gamma \\ \\ Lu_2 = f & in\ \Omega_2 \\ u_2 = 0 & on\ \partial\Omega_2 \backslash \Gamma \\ \\ u_1 = u_2 & on\ \Gamma \\ \frac{\partial u_1}{\partial \mathbf{n}_L} = \frac{\partial u_2}{\partial \mathbf{n}_L} & on\ \Gamma \end{cases}$$

where $\frac{\partial \cdot}{\partial \mathbf{n}_L}$ *is the conormal derivative defined as:*

$$\frac{\partial u}{\partial \mathbf{n}_L} = \sum_{i,j=1}^{2} a_{ij} \frac{\partial u}{\partial x_j} \mathbf{n}_i$$

with $\mathbf{n}_i$ *as the ith components of the* $\mathbf{n}$ *vector which is the normal vector exiting* Ω_1 *on its border.*

If we split the problem in two parts, to let different computing units solve them, we come to this iterative formulation, to be solved for $k > 0$:

$$\begin{cases} Lu_1^{(k)} = f & \text{in } \Omega_1 \\ u_1^{(k)} = 0 & \text{on } \partial\Omega_1 \backslash \Gamma \\ u_1^{(k)} = u_2^{(k-1)} & \text{on } \Gamma \end{cases}$$

and

$$\begin{cases} Lu_2^{(k)} = f & \text{in } \Omega_2 \\ u_2^{(k)} = 0 & \text{on } \partial\Omega_2 \backslash \Gamma \\ \frac{\partial u_2^{(k)}}{\partial \mathbf{n}} = \frac{\partial u_1^{(k)}}{\partial \mathbf{n}} & \text{on } \Gamma \end{cases}$$

Both these problems are well defined, the whole borders of both Ω_1 and Ω_2 are covered with boundary conditions, but we observe that the second is a mixed problem with Dirichlet and Neumann conditions. This method is consistent, thanks to the theorem of equivalence, but it is not always convergent. To ensure convergence it is necessary to have Ω_1, where the pure Dirichlet problem is solved, bigger in size than Ω_2. This odd condition is not always easy to satisfy when splitting the problem.

This problem can be fixed using a different Dirichlet condition on Γ for the first problem:

$$u_1^{(k)} = u_2^{(k-1)}$$

becomes

$$u_1^{(k)} = \theta u_2^{(k-1)} + (1-\theta) u_1^{(k-1)}$$

with $0 \leq \theta < 1$, hence relaxing the Dirichlet condition. The good news is that it is possible to determine a value θ_{max} for the problem such that with $\theta < \theta_{max}$ the method surely converges.

Another issue arises, this time concerning concurrency. The problem solved on Ω_2 at step k needs information about the solution u_1 of the other problem at the

same step k. Exactly like before with the Schwarz method we could use a slightly out-dated information about u_1 using its value at time $k-1$.

In both the Schwarz and the Schur methods we have the need of a complex and heavy communication between the different cluster nodes, especially those who are computing neighbor sub-regions, exactly like in section 2.2.6.

2.3.3 Domain Decomposition methods as preconditioner

Through algebraic manipulations we can prove that the Shwarz method is equivalent to a preconditioned Richardson method. The preconditioner P_{as} is on outcome of the local finite element problems $A_i\mathbf{u}_i = \mathbf{f}_i$, and it is calculated inverting each stiffness matrix A_i.

A preconditioner is said to be optimal if the contioning number of the preconditioned problem does not depend on the size of the partition.

Instead P_{as} is sub-optimal. In fact if it is applied to the complete finite element algebraic problem $A\mathbf{u} = \mathbf{f}$, the following holds:

$$K_2(P_{as}^{-1}A) \leq C\frac{1}{\delta H}$$

where H is a measure of the size of the sub-domains and δ is a measure of the superposition area between them. If we divide the same problem into more sub-domains, H will be smaller, hence the conditioning number will grow.

However the preconditioned problem has a very good property. The preconditioned matrix $Q_a = P_{as}^{-1}A$ for the Schwarz method is symmetric and positively defined after having defined an appropriate scalar product.

Therefore it is possible to use efficent iterative methods like the conjugate gradient method to solve the problem.

The resolution of each local system is a prerequisite to build such a preconditioner. As a consequence, it is important to quickly compute the inverse of the each local matrix A_i.

2.4 Direct linear system solvers

We described a couple of iterative methods apt to solve PDE problems with a Domain Decomposition approach. These methods need several resolutions of local approximate problems, and finally converge to local solutions which are equivalent to the global solution on the local sub-regions.

These methods permit to split the problem on different computing devices, but they do not describe how to solve each local problem. These local problems are smaller than the original one, and are usually solved with traditional approaches, without further concurrency: after the local stiffness matrix has been assembled, the linear system is solved.

In which way the local linear system is solved is decided by the implementation, but we could discuss a little further about the choice of a direct or iterative method for it. Iterative methods have some properties which cannot be ignored.

Indeed, they permit the highest possible precision, because one could indefinitely let the method refine the solution. They have fewer memory constraints, because even the most advanced of them seldom use more than $\mathcal{O}(n^2)$ variables to solve a matrix of dimension $n \times n$, since they only need the original matrix and

some vectors saved in the memory. This is actually the most attractive feature of these methods, because one cannot possibly use more memory than what is available. Hence these methods allow the resolution of very large matrices, that could not be done otherwise.

As a drawback, despite of these good features, we have that iterative methods need preconditioning. They are very sensible to a badly scaled problem, and a ill-conditioned matrix could slow down the resolution to unreasonable timings.

On the other hand, direct methods are very memory consuming and not so precise, as they do not refine the solution: the numerical approximation of the various operations involved will cause badly scaled problems to have very rough solutions, with bad residuals. The real advantage of this kind of methods is the finite time need, and the possibility to solve the system once and for all, using the solution several times thereafter. Most of these methods calculate a useful decomposition of the original matrix instead of the inverse of the matrix.

The interesting part is that the best preconditioner one could imagine for an iterative method is, almost by definition, the inverse of the original matrix. Hence one could calculate such an inverse with a direct method, and then solve the system iteratively until the desired precision is reached.

This way an arbitrary precise solution could be achieved in an almost fixed time, as long as the system to solve is small enough. The need of a small problem is caused by the high memory consumption of direct methods. Since such an iterative method is perfectly conditioned, the time needed for the solution refinement becomes almost negligible, and the whole computation effort is represented by the direct solver.

2.4.1 Gaussian elimination method

This method is the ancestor of most of the direct methods we use nowadays, including the frontal method, and for this reason it needs an appropriate introduction.

The Gaussian elimination method can be used to both solve a linear system with a given constant term and to find an easily solvable matrix decomposition. The latter is more interesting because it permits to decompose the matrix once and for all, allowing us to solve the linear systems with different constant terms.

The goal of such a method is to find two matrices L and U such that $A = LU$. Here A is the original stiffness matrix, $A, L, U \in \mathbb{R}^{n \times n}$, L is a lower triangular matrix and U an upper one. Once we have such a decomposition we can easily solve the linear system for any given constant term $\mathbf{b}$.

$$LU\mathbf{x} = \mathbf{b}$$

is solved by forward and backward substitution in $2n^2$ floating point operations leading to the solutions of two problems in sequence:

$$\begin{aligned} L\mathbf{x}_L &= \mathbf{b} \\ U\mathbf{x} &= \mathbf{x}_L \end{aligned}$$

The Gaussian elimination method itself is performed in $\frac{2}{3}n^3$ flops, hence justifying the reuse of the same decomposition on multiple constant terms where possible.

Let us have a closer look at the method itself. In the simplest form it is nothing but a sequence of subtractions of equations in a linear system:

$$A\mathbf{x} = \mathbf{b} \qquad \begin{cases} a_{11}\mathbf{x}_1 + a_{12}\mathbf{x}_2 + \ldots + a_{1n}\mathbf{x}_n = \mathbf{b}_1 \\ a_{21}\mathbf{x}_1 + a_{22}\mathbf{x}_2 + \ldots + a_{2n}\mathbf{x}_n = \mathbf{b}_2 \\ \vdots + \vdots + \ddots + \vdots = \vdots \\ a_{n1}\mathbf{x}_1 + a_{n2}\mathbf{x}_2 + \ldots + a_{nn}\mathbf{x}_n = \mathbf{b}_n \end{cases} \tag{2.11}$$

From this system we obtain some multipliers, one for each line under the first:

$$m_{i1} = \frac{a_{i1}}{a_{11}} \quad \forall 1 < i \leq n$$

If we subtract the first line multiplied by the correspondent coefficient m_{i1} from all of the others we now have only zeros as coefficients of $\mathbf{x}_1$ in all the equations under the first

$$\begin{cases} a_{11}\mathbf{x}_1 + a_{12}\mathbf{x}_2 + \ldots + a_{1n}\mathbf{x}_n = \mathbf{b}_1 \\ 0 + (a_{22} - m_{21}a_{12})\,\mathbf{x}_2 + \ldots + (a_{2n} - m_{21}a_{1n})\,\mathbf{x}_n = \mathbf{b}_2 - m_{21}\mathbf{b}_1 \\ \vdots + \vdots + \ddots + \vdots = \vdots \\ 0 + (a_{n2} - m_{n1}a_{12})\,\mathbf{x}_2 + \ldots + (a_{nn} - m_{n1}a_{1n})\,\mathbf{x}_n = \mathbf{b}_n - m_{n1}\mathbf{b}_1 \end{cases} \tag{2.12}$$

since the first coefficient of the each line beneath the first becomes

$$a_{i1} - m_{i1}a_{11} = a_{i1} - \frac{a_{i1}}{a_{11}}a_{11} = 0$$

The linear system (2.12) is written synthetically $A^{(1)}\mathbf{x} = \mathbf{b}^{(1)}$ and represents the output of the first step of the Gaussian elimination.

At the same time the linear system we obtain is equivalent to the first one, in other words $A^{(1)}\mathbf{x} = \mathbf{b}^{(1)}$ will lead to the same solution as $A\mathbf{x} = \mathbf{b}$. The element a_{11} is called pivot element, and it obviously must not be null. If this happens we could permute the columns in order to find a suitable nonzero pivot. This operation is called pivoting, and it is often performed, even when not necessary, to allow a large number to be used as pivot, increasing stability. To complete the Gaussian elimination method we have to perform further steps over the next lines, using the multipliers m_{ij} $j = 2, \ldots, n-1$.

The linear system at the $(n-1)$th step will look like this:

$$A^{(n-1)}\mathbf{x} = \mathbf{b}^{(n-1)} \qquad \begin{cases} a_{11}\mathbf{x}_1 + a_{12}\mathbf{x}_2 + \ldots + a_{1n}\mathbf{x}_n = \mathbf{b}_1 \\ 0 + 0 + \ldots + \widetilde{a}_{2n}\mathbf{x}_n = \widetilde{\mathbf{b}}_2 \\ \vdots + \vdots + \ddots + \vdots = \vdots \\ 0 + 0 + \ldots + \widetilde{a}_{nn}\mathbf{x}_n = \widetilde{\mathbf{b}}_n \end{cases}$$

with the associated $A^{(n-1)} = U$ matrix being an upper triangular matrix. It is possible to perform similar operations in order to arrive to a lower triangular matrix L.

As we stated before, at each step the linear system we built is equivalent to the original one, and once the U matrix has been calculated, the computing time needed to achieve the result is negligible.

The method does not factorize A, it simply finds an equivalent system which is easier to solve. To achieve the desired decomposition of A into L and U we must look at the operations needed to perform each step of the Gaussian elimination, trying to reconstruct which matrices M_i have been applied to A. In other words we want to find the matrices M_i such that:

$$(\prod_{i=0}^{j-1} M_{j-i})A = A^{(j)} \qquad j = 1, \ldots, n-1$$

If we try such an analysis we find that each $A^{(j)}$ is equal to $A^{(j-1)}$ except for the fact that the jth line has been subtracted from all the equations $k > j$ beneath, after a multiplication by m_{kj}. This is easily written into matrices M_j which are in lower triangular form, like the following one:

$$M_j = \begin{bmatrix} 1 & \ldots & 0 & 0 & \ldots & 0 \\ \vdots & \ddots & \vdots & \vdots & & \vdots \\ 0 & & 1 & 0 & & 0 \\ 0 & & -m_{j+1,j} & 1 & & 0 \\ \vdots & & \vdots & & \ddots & \vdots \\ 0 & \ldots & -m_{k,j} & 0 & \ldots & 1 \end{bmatrix}$$

Since the product and the inversion of lower triangular matrices is still a lower triangular one, we have that from this

$$(\prod_{i=0}^{n-1} M_{n-i})A = A^{(n-1)} = U$$

our result follows:

$$A = \left(\prod_{i=0}^{j-1} M_{j-i}\right)^{-1} U = LU$$

2.4.2 Sparse matrices

In the section 2.1.4 we spoke about the memory problem one could encounter when solving very large finite element problems. We have also said that there are methods which are able to only stock the useful information about the stiffness matrix, hence reducing the amount of memory needed. These methods aim to save exclusively the entries which are different from zero, called nonzero entries, or simply nonzeros. The ratio between nonzeros and the number of total entries is called sparsity ratio, and matrices which have few nonzeros are said to be sparse, while those with many nonzeros are called full. More precisely, in a sparse matrix the number of nonzero entries is $\mathcal{O}(N)$, being N the dimension of the matrix. The sparsity ration of a sparse matrix, then is $\mathcal{O}(N^{-1})$.

Let us consider some figures to illustrate the importance of special ways to memorize a sparse matrix. A fairly good workstation at the time of writing may have 4GB of random access memory. The usual format for storing real numbers for scientific purposes is double precision floating-point, which uses 64 bits, 8 bytes.

Then, the maximum size of a full matrix that can be stored on such a system is 23170.

When considering stiffness matrices for a finite element problem solved on an unstructured grid of linear elements, say, the number of nonzero entries is approximately $25N$. Assuming an overhead of $2N$ for the extra housekeeping required by a sparse storage scheme, on the same memory we may store a matrix of size 1988410, which is three orders of magnitude more.

Now that we have shown the importance of working with sparse matrices, we would like to know how the Gaussian elimination method performs on such matrices. Unfortunately in its standard form it does not perform well, and let us look at an example to understand why.

Example. Let A be a matrix we want to factorize with the Gaussian elimination method

$$A = \begin{bmatrix} 1 & 2 & 3 & 4 & 5 \\ 1 & 1 & \cdot & \cdot & \cdot \\ 1 & \cdot & 1 & \cdot & \cdot \\ 1 & \cdot & \cdot & 1 & \cdot \\ 1 & \cdot & \cdot & \cdot & 1 \end{bmatrix}$$

this is commonly called an "arrow" matrix. The subsequent matrix $A^{(1)}$ will be

$$A^{(1)} = \begin{bmatrix} 1 & 2 & 3 & 4 & 5 \\ 0 & -1 & -3 & -4 & -5 \\ 0 & -2 & -2 & -4 & -5 \\ 0 & -2 & -3 & -3 & -5 \\ 0 & -2 & -3 & -4 & -4 \end{bmatrix}$$

In just one step we filled every zero entry with a nonzero, hence dramatically increasing the memory need. Therefore, if we use the standard Gaussian elimination to factorize a very large sparse matrix, we would immediately have to cope with memory problems.

For this reason the Gaussian elimination is the preferred way in problems where full matrices are to be solved, but not in our case, since the finite element method generates very large and very sparse matrices. Therefore, specific refinements of this method have to be put in place to avoid fill-in.

Chapter 3

Overview of UMFPACK

"UMFPACK is a set of routines for solving unsymmetric sparse linear systems, Ax=b, using the Unsymmetric MultiFrontal method."[1]

It is a state of the art computing library that lets one to perform direct linear system solving on sparse matrices, with excellent performance, acceptable memory consumption and keeping a good degree of sparsity on the factor matrices L and U through suitable manipulations of the matrix A. It is used by different important scientific computing programs as direct solver, Matlab[2] among others.

It is not the only choice, of course, as other libraries are available (superLU and MUMPS for instance) but it generally is a good choice. It is used by LifeV[3], the finite element library developed by the mathematics department of Politecnico di Milano, with the collaboration of École polytechnique fédérale de Lausanne (EPFL), INRIA (REO) in France and Emory University in the United States of America.

Since my thesis aimed to find new ways to improve the performances of LifeV when solving large distributed problems, we will focus our attention on how UMFPACK solve linear systems, because this is where typically the most part of the computation is spent on the single node of the cluster and overall.

3.1 Frontal and Multifrontal solvers

The whole concept of a frontal solver is to perform a matrix factorization exploiting its sparsity. This is achieved with an efficient permutation that usually gathers all the nonzero entries near the main diagonal (but other possibilities are not excluded). After the nonzeros are gathered, a special matrix called front is assembled in full notation. This is not a memory bottleneck, because of the reduced size of the frontal matrix. In fact the front matrix is usually much smaller than the original matrix. Many front matrices must be assembled to consider all the nonzeros of the original matrix.

When a frontal matrix is assembled, the method uses a high performance algo-

[1] Source: main page for UMFPACK, http://www.cise.ufl.edu/research/sparse/umfpack/, as seen on 14 feb 2011

[2] http://www.mathworks.com/products/matlab/

[3] http://www.lifev.org/

rithm to perform algebraic operations (more precisely it calls BLAS[4] routines), and achieve a local factorization. The local factorization is performed over all of the frontal matrices: this will cause each piece of the stiffness matrix to be factorized. Subsequently all of the local factorizations are assembled together to compute the global factorization, in sparse format once again. This is just a presentation of the method, that will be detailed in the following paragraphs.

The Multifrontal method is an evolution of the Frontal method firstly introduced by B. M. Irons: "A frontal solution program for finite element analysis, International Journal for Numerical Methods in Engineering, Volume 2, Issue 1, pages 5–32, January/March 1970". This method allowed the resolution of finite element problems even with the limited amount of memory available in the '70s.

3.1.1 The elimination tree

For frontal methods it is fundamental to exploit the fill-in. Zeros are easy to track between different decomposition steps, since just multiplications and additions are performed. Because of this fact we can foresee where the fill-in will occurs by simple algorithms. This is important to speed up the real numerical work afterward.

In this example we will consider the Cholesky decomposition, which applies to symmetric positively defined matrices (SPD). This method finds a decomposition $A = LL^T$ if $A \in \mathbb{R}^{n \times n}$, with L being a lower triangular matrix and L^T its transpose. This is an important example because for a large class of finite element method problems SPD matrices are produced.

Example. Let A be a SPD matrix with the following sparsity pattern

$$A = \begin{array}{c} \\ 1 \\ 2 \\ 3 \\ 4 \\ 5 \\ 6 \\ 7 \\ 8 \\ 9 \end{array} \begin{array}{c} \begin{array}{ccccccccc} 1 & 2 & 3 & 4 & 5 & 6 & 7 & 8 & 9 \end{array} \\ \left[\begin{array}{ccccccccc} a & & & & & & \bullet & \bullet & \bullet \\ & b & & \bullet & & \bullet & & & \\ & & c & & \bullet & & & \bullet & \\ & \bullet & & d & & & & \bullet & \bullet \\ & & \bullet & & e & \bullet & & \bullet & \\ & \bullet & & & \bullet & f & & & \bullet \\ \bullet & & & & & & g & \bullet & \bullet \\ \bullet & & \bullet & \bullet & \bullet & & \bullet & h & \\ \bullet & & & \bullet & & \bullet & \bullet & & i \end{array}\right] \end{array} \tag{3.1}$$

where none of the diagonal element is zero. Let us perform the usual elimination method. We obtain the following sparsity pattern for the L matrix

$$L = \begin{array}{c} \\ 1 \\ 2 \\ 3 \\ 4 \\ 5 \\ 6 \\ 7 \\ 8 \\ 9 \end{array} \begin{array}{c} \begin{array}{ccccccccc} 1 & 2 & 3 & 4 & 5 & 6 & 7 & 8 & 9 \end{array} \\ \left[\begin{array}{ccccccccc} a & & & & & & & & \\ & b & & & & & & & \\ & & c & & & & & & \\ & \bullet & & d & & & & & \\ & & \bullet & & e & & & & \\ & \bullet & & \circ & \bullet & f & & & \\ \bullet & & & & & & g & & \\ \bullet & & \bullet & \bullet & \bullet & \circ & \bullet & h & \\ \bullet & & & \bullet & & \bullet & \bullet & \circ & i \end{array}\right] \end{array}$$

[4] Basic Linear Algebra Subprograms. Many different versions exist, which are usually distributed as libraries. Most of them share the same interface, therefore it is possible for UMFPACK to use any of those.

where the white circles $\circ$ are the newly added nonzero entries.

Why such a fill-in occurred? Let us look at the element at position $(6, d)$ in the A matrix (which is zero). When performing the subtraction of line 2 from the rows beneath, the multiplier $m_{j2} = \frac{a_{j2}}{a_{22}} = \frac{a_{j2}}{b}$ is null for all zero entries of the column 2 beneath the diagonal. This means that for those lines j with $m_{j2} = 0$ the subtraction is not really performed, and the sparsity pattern is preserved. For all of the other lines we will have fill-in on those columns in which line 2 have nonzeros.

Row 2 had nonzeros on columns d and f, hence we will be adding nonzeros on those columns, just on the lines where $m_{j2} \neq 0$. The lines where $m_{j2} \neq 0$ are $\{4, 6\}$, therefore the line 2 is filling up the positions $\{(4, d), (6, d), (4, f), (6, f)\}$. The entries which were already nonzero are not to be considered, since they will not alter the sparsity pattern. Those over the main diagonal do not bother us either. The only entry filled by line 2 is then $(6, d)$.

Let us now build a graph with the following rule, which applies to symmetric matrices, and that for SPD matrices generates a tree, called elimination tree:

Definition. The elimination tree is a graph with a node for each column of the matrix L and which has the edge (j, k) if and only if $j = min(s > k : \ l_{sk} \neq 0)$. Since this graph is in fact a tree, we call j parent of k.

For non SPD matrices the graph will not generally be a tree, and the considerations that follow will only be partially valid.

To visualize the meaning of this definition we produce a matrix L_T which has only the first nonzero of each column, since we are just interested in the $min(s > k : \ l_{sk} \neq 0)$. We represent the other nonzero entries of L which became zeros in L_T with $\times$.

$$L_T = \begin{array}{c} 1 \\ 2 \\ 3 \\ 4 \\ 5 \\ 6 \\ 7 \\ 8 \\ 9 \end{array} \begin{bmatrix} a & & & & & & & & \\ & b & & & & & & & \\ & & c & & & & & & \\ & \bullet & & d & & & & & \\ & & \bullet & & e & & & & \\ & \times & & \circ & \bullet & f & & & \\ \bullet & & & & & & g & & \\ \times & & \times & \times & \times & \circ & \bullet & h & \\ \times & & & \times & & \times & \times & \circ & i \end{bmatrix}$$

We can now easily build our elimination tree, where each line j is parent of all the columns k corresponding to the entries $l_{T,jk} \neq 0$. A visual representation of such a tree is in figure (3.1).

This graph has several interesting properties. If we look at the first three columns and lines of matrix (3.1) we notice that each multiplier m_{ij} for this submatrix is zero. In other words the first line will not influence the second or the third, neither the second will be subtracted from the third. They are independent, we could compute their local factorizations concurrently. This is represented, in the tree, by the fact that these rows are leafs.

Separate branches of the tree do not influence each other. The computation of a branch must, however, be finished and applied before the root of that branch is computed. Let us be more precise: imagine to perform one Gaussian elimination step from the matrix (3.1).

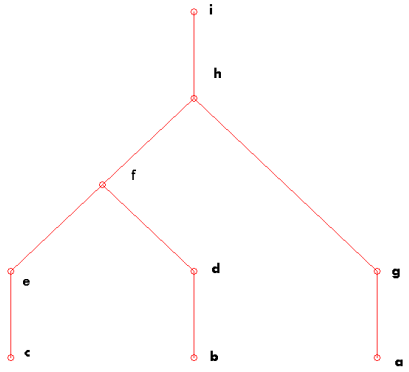

Figure 3.1: Elimination tree corresponding to matrix (3.1)

Example.

$$A = \begin{bmatrix} a & \cdot & \cdot & \cdot & \cdot & \cdot & a_{17} & a_{18} & a_{19} \\ \cdot & b & \cdot & a_{24} & \cdot & a_{26} & \cdot & \cdot & \cdot \\ \cdot & \cdot & c & \cdot & a_{35} & \cdot & \cdot & a_{38} & \cdot \\ \cdot & a_{42} & \cdot & d & \cdot & \cdot & \cdot & a_{48} & a_{49} \\ \cdot & \cdot & a_{53} & \cdot & e & a_{56} & \cdot & a_{58} & \cdot \\ \cdot & a_{62} & \cdot & \cdot & a_{65} & f & \cdot & \cdot & a_{69} \\ a_{71} & \cdot & \cdot & \cdot & \cdot & \cdot & g & a_{78} & a_{79} \\ a_{81} & \cdot & a_{83} & a_{84} & a_{85} & \cdot & a_{87} & h & \cdot \\ a_{91} & \cdot & \cdot & a_{94} & \cdot & a_{96} & a_{97} & \cdot & i \end{bmatrix} \tag{3.2}$$

We will start the elimination as usual, from the first line down, and we will mark with an $\cdot^{(n)}$ the modified entries at step n. With the $\times^{(n)}$ we indicate the entries which are set to zero by the elimination step n and are usually used to store the corresponding element of matrix L.

$$A^{(1)} = \begin{bmatrix} a & \cdot & \cdot & \cdot & \cdot & \cdot & a_{17} & a_{18} & a_{19} \\ \cdot & b & \cdot & a_{24} & \cdot & a_{26} & \cdot & \cdot & \cdot \\ \cdot & \cdot & c & \cdot & a_{35} & \cdot & \cdot & a_{38} & \cdot \\ \cdot & a_{42} & \cdot & d & \cdot & \cdot & \cdot & a_{48} & a_{49} \\ \cdot & \cdot & a_{53} & \cdot & e & a_{56} & \cdot & a_{58} & \cdot \\ \cdot & a_{62} & \cdot & \cdot & a_{65} & f & \cdot & \cdot & a_{69} \\ \times^{(1)} & \cdot & \cdot & \cdot & \cdot & \cdot & g^{(1)} & a_{78}^{(1)} & a_{79}^{(1)} \\ \times^{(1)} & \cdot & a_{83} & a_{84} & a_{85} & \cdot & a_{87}^{(1)} & h^{(1)} & a_{89}^{(1)} \\ \times^{(1)} & \cdot & \cdot & a_{94} & \cdot & a_{96} & a_{97}^{(1)} & a_{98}^{(1)} & i^{(1)} \end{bmatrix}$$

As we see, performing the elimination of the first line from all of the other resulted in changes to just the last three rows, which are indeed ancestors of a on the elimination tree, hence they are modified by the elimination performed with the descendant row. The second step will be equally interesting.

$$A^{(2)} = \begin{bmatrix} a & \cdot & \cdot & \cdot & \cdot & \cdot & a_{17} & a_{18} & a_{19} \\ \cdot & b & \cdot & a_{24} & \cdot & a_{26} & \cdot & \cdot & \cdot \\ \cdot & \cdot & c & \cdot & a_{35} & \cdot & \cdot & a_{38} & \cdot \\ \cdot & \times^{(2)} & \cdot & d^{(2)} & \cdot & a_{46}^{(2)} & \cdot & a_{48} & a_{49} \\ \cdot & \cdot & a_{53} & \cdot & e & a_{56} & \cdot & a_{58} & \cdot \\ \cdot & \times^{(2)} & \cdot & a_{64}^{(2)} & a_{65} & f^{(2)} & \cdot & \cdot & a_{69} \\ \times^{(1)} & \cdot & \cdot & \cdot & \cdot & \cdot & g^{(1)} & a_{78}^{(1)} & a_{79}^{(1)} \\ \times^{(1)} & \cdot & a_{83} & a_{84} & a_{85} & \cdot & a_{87}^{(1)} & h^{(1)} & a_{89}^{(1)} \\ \times^{(1)} & \cdot & \cdot & a_{94} & \cdot & a_{96} & a_{97}^{(1)} & a_{98}^{(1)} & i^{(1)} \end{bmatrix}$$

The entries modified are only on the lines d and f, which are ancestors of b in the tree. The elements h and i are equally ancestors, but they are not modified. This is because they will be modified eventually by the elimination performed with lines d and f, whose values depends on line b, as we have seen. We may say that the dependance of rows h and i on row b is indirect, through rows d and f.

We could perform the elimination steps using the branches $a \to g$ and $b \to d$ concurrently, because they are independent, but we could never mix the order of calculations within the path $b \to d \to f \to h \to i$, otherwise wrong values could be used.

Another useful proposition about the elimination tree: since each descendant k influence all of its ancestors j, adding sometimes nonzeros, we have that

Proposition. *If j is an ancestor of k, then the nonzero pattern of the vector $(l_{jk}, \ldots, l_{nk})$ is contained in the pattern of $(l_{jj}, \ldots, l_{nj})$*

This statement is useful when talking about the front matrices.

3.1.2 Front and update matrices

It is now time to introduce the concept of front, and frontal matrix. All we did until now was to understand in which order to compute the decomposition operations. Incidentally we understood that not every step of a factorization modifies each entry of the matrix.

Because of the sparsity of the original matrix we have many multipliers $m_{jk} = 0$, meaning that just a few entries are modified at each step. If we could regroup all of these entries in a full matrix, we would be able to perform all the required operations with fully optimized BLAS routines, which would be a great advantage.

In matrix (3.2) we saw that the first line influences just the down-right most 3×3 block, that is the block composed of its ancestors: (g, h, i). The same step obviously modifies each nonzero entry in the first column too, since it will erase them to zero, leaving space for the corresponding elements of L.

Let us define the front matrix F_1 for this first column/row:

$$F_1 = \begin{bmatrix} a = a_{11} & a_{17} & a_{18} & a_{19} \\ a_{71} & 0 & \dots & 0 \\ a_{81} & \vdots & \ddots & \vdots \\ a_{91} & 0 & \dots & 0 \end{bmatrix}$$

We notice that if we would have to perform the first step of factorization on F_1 we would obtain

$$F_1^{(1)} = \begin{bmatrix} a = a_{11} & a_{17} & a_{18} & a_{19} \\ \times^{(1)} & & & \\ \times^{(1)} & & U_1 & \\ \times^{(1)} & & & \end{bmatrix}$$

with $U_1 \in \mathbb{R}^{3\times 3}$. This matrix U_1 is called the update matrix from column 1, because it contains the nonzero entries one should sum to the corresponding entries of A to obtain the updated matrix $A^{(1)}$. We saw with this example how a decomposition step on a sparse matrix of size 9×9 can be performed on a full 4×4 matrix.

Since g is parent of node a in the elimination tree, the corresponding front matrix, F_7 can be built from the original matrix A recalling at the same time that the decomposition performed using line a has changed some nonzero values:

$$F_7 = \begin{bmatrix} g = a_{77} & a_{78} & a_{79} \\ a_{87} & & \\ a_{97} & & \mathbf{0} \end{bmatrix} + \overline{U}_7$$

where $\overline{U}_7$ is the contribution matrix from the sub-tree underlying the node g, associated to the 7th column. Such a matrix is called sub-matrix update tree, and must not be confused with U_7 or U_1, which are the update matrices result of a frontal factorization. However, since a was a leaf and g is its parent, and since g does not have any descendant other than a, in this particular case $U_1 = \overline{U}_7$.

We introduced with an example the meaning of the front matrix F_j, the sub-tree update matrix $\overline{U}_j$, which contains all of the descendants contributions to F_j, and the update matrix U_j, suited of the factorization performed with the first column of F_j. It is worth noticing how U_j depends on just the nonzero values of the jth row and column.

We now give more precise definitions.

Definition. Let $A \in \mathbb{R}^{n\times n}$ be a symmetric positive defined matrix and L the output of its Cholesky factorization. Consider the $r+1$ indexes of the nonzero entries in column j of matrix L, $i_0 = j, i_1, \dots, i_r$, which means that column j has r nonzero entries below the diagonal. Let $T[j]$ be the sub-tree of the elimination tree rooted in the node j, so that $T[j]$ contains j and all of its descendants.

The frontal matrix for column j is defined:

$$F_j = \begin{bmatrix} a_{jj} & a_{j,i_1} & \dots & a_{j,i_r} \\ a_{i_1,j} & 0 & \dots & 0 \\ \vdots & \vdots & \ddots & \vdots \\ a_{i_r,j} & 0 & \dots & 0 \end{bmatrix} + \overline{U}_j$$

where the sub-tree update matrix $\overline{U}_j$ collects all the contributions from the descendant nodes factorization:

$$\overline{U}_j = - \sum_{k \in T[j]/\{j\}} \begin{pmatrix} l_{jk} \\ l_{i_1,k} \\ \vdots \\ l_{i_r,k} \end{pmatrix} \begin{pmatrix} l_{jk} & l_{i_1,k} & \dots & l_{i_r,k} \end{pmatrix}$$

Although $\overline{U}_j$ is defined by unknown elements of L, we will shortly see how it is calculated.

One could verify that if $k < j$ and $l_{jk} \neq 0$, then the node k is a descendant of j in the elimination tree. Therefore we have that the first column of matrix $\overline{U}_j$ is:

$$(\overline{U}_j)_{\cdot,1} = - \sum_{k<j} l_{jk} \begin{pmatrix} l_{jk} \\ l_{i_1,k} \\ \vdots \\ l_{i_r,k} \end{pmatrix}$$

which we call the complete update column for j.

When performing one step of factorization on the matrix F_j the operation gives as result this decomposition, which is in lower and upper triangular block-matrices:

$$F_j = \begin{bmatrix} l_{jj} & 0 & \dots & 0 \\ l_{i_1,j} & & & \\ \vdots & & I & \\ l_{i_r,j} & & & \end{bmatrix} \begin{bmatrix} l_{jj} & l_{i_1,j} & \dots & l_{i_r,j} \\ 0 & & & \\ \vdots & & U_j & \\ 0 & & & \end{bmatrix}$$

with U_j equal to:

$$U_j = - \sum_{k \in T[j]} \begin{pmatrix} l_{i_1,k} \\ \vdots \\ l_{i_r,k} \end{pmatrix} \begin{pmatrix} l_{i_1,k} & \dots & l_{i_r,k} \end{pmatrix}$$

It is worth noticing that since we build the front matrices F_j in the order they are in the elimination tree, from the leaves to the root, we are assured that the nonzero structure of each descendant is a subset of the ancestor's one. This is important because in F_j we only have nonzero entries, those taken by the nonzero structure of node j. When we add $\overline{U}_j$ and F_j we are in fact updating the matrix F_j with the nonzero contributions from the descendant nodes. Since the nonzero structure of the descendant nodes is a subset of that of j, every contribution will have its proper place. In fact we have that if c is a child of j in the elimination tree, each nonzero entry of the update matrix U_c finds it right place in its parent sub-tree update matrix $\overline{U}_j$.

This was not much of an issue in our example, because we explored just a simple part of it. It is in fact obvious when the parent j has only one child: the parent front will be one column/row smaller than the child's one, and no manipulation is needed to mutate U_c into $\overline{U}_j$.

When more than a child exists, however, one must modify the update matrices U_{i_k} wisely, in order to move every entry to its right position. We call extended-addition $\boxplus$ such an operation.

Each entry of every matrix F_j, $\overline{U}_j$ or U_j represents a precise position in the original matrix A. The extended-add operation uses the information about the original entry position when adding different matrices.

Example. Let us define two matrices R and S from the original matrix A:

$$R = \begin{bmatrix} a_{33} & a_{35} \\ a_{53} & a_{55} \end{bmatrix} \qquad S = \begin{bmatrix} a_{11} & ba_{15} \\ ca_{51} & da_{55} \end{bmatrix}$$

Then the result of the extended-addition between R and S will be:

$$R \boxplus S = \begin{bmatrix} a_{11} & 0 & (1+b)a_{15} \\ 0 & a_{33} & a_{35} \\ (1+c)a_{51} & a_{53} & (1+d)a_{55} \end{bmatrix}$$

Now that all of the tools have been described, we can finally write the main theorem:

Theorem. *If c_k for $k = 1, 2, \ldots$ are children of j in the elimination tree, then*

$$F_j = \begin{bmatrix} a_{jj} & a_{j,i_1} & \ldots & a_{j,i_r} \\ a_{i_1,j} & 0 & \ldots & 0 \\ \vdots & \vdots & \ddots & \vdots \\ a_{i_r,j} & 0 & \ldots & 0 \end{bmatrix} \boxplus U_{c_1} \boxplus U_{c_2} \boxplus \ldots$$

Since for the leaves there are no children, to assemble their frontal matrices there is no need to calculate any update matrix. The aforementioned theorem, joint with this consideration, permits us to easily build a recursive algorithm to calculate a Cholesky factorization.

3.1.3 Differences between unifrontal and multifrontal methods

What we did in this section was to give an idea of how a uni/multifrontal method achieve the decomposition of large sparse matrices using optimized operations on smaller matrices in full notation.

The main difference between a unifrontal method and a multifrontal one is the order in which the different frontal matrices are formed, factorized and applied. We stated that different branches in an elimination tree are independent, that means that one could, for example, compute the leaves in no matter which order. One could even calculate the frontal decompositions for the leaves concurrently, all at the same time.

In a single front method we choose an order and once the frontal decomposition is found we apply the update matrices immediately. We use the memory of just one front at each time: after the front is factorized in full notation the modifications are applied, and a new front will be assembled, overwriting the old one.

A multifrontal method instead will generally retain the update matrices for the computed fronts, as it will apply them only when needed, and always directly into the parent frontal matrix.

This technique provides some benefit and one main drawback: higher memory consumption. If our elimination tree had one node with more than one child, then

the memory needed to compute the frontal matrix for that node would be, at least, of the size of all the update matrices of the children put together.

We might actually want to perform concurrently the operations on different branches, even if it is not necessary to call this a multifrontal method. In that case we would need additional memory, since there are even more active fronts at the same time.

3.1.4 The decomposition fill-in

The method presented so far has enabled us to reduce the factorization of a sparse matrix to a succession of operations on smaller, full, matrices and to foresee where the fill-ins occur, we are not yet preventing them in any way. One of our purposes was to reduce fill-in, so to decrease the memory needed for the factorization.

This is done with permutations, of both rows and columns, considering the constraints with respect to special cases like the symmetric one. The problem of finding the best possible permutation to reduce fill-in is an NP-complete problem, hence it is largely unfeasible: it would take longer to calculate the permutation matrix than the factorization itself. Because of this fact some heuristic algorithms are used instead to achieve good permutations.

One largely diffused heuristic is the minimum degree algorithm, that permutes the matrix at each factorization step, using as pivot the one that holds fewer off-diagonal nonzeros on the column. Such a pivot is bound to generate the fewest possible nonzero entries, with few exceptions. Problems arise when this pivot has accidentally become zero or in is too small. In these cases the elimination step is unfeasible or leads to an unstable factorization, respectively. Thus any frontal method must check for some numerical stability conditions when using each pivot, in order to guarantee a good quality of the solution. If these conditions are not satisfied, another permutation must be found for the sub-matrix that is still to decompose, and the heuristic method must be ran again.

When calculating the permutation one can use the graph theory, since graphs can be built in order to represent the fill-in and the entries that will be used to assembly each frontal matrix. This is the state of the art method to look at these problems. Graph theory is used to manipulate such a graph, it is used by the best algorithms to find the permutation matrices and the elimination trees (which are called assembly graphs in multifrontal methods). For a detailed description of these techniques we recommend the article describing the mechanism used by UMFPACK, from P. Amestoy, T. A. Davis and I. S. Duff, "An approximate minimum degree ordering algorithm, SIAM Journal on Matrix Analysis and Applications, vol 17, no. 4, pp. 886-905, Dec. 1996".

3.2 Benchmark of UMFPACK

We already explained the reasons why UMFPACK is a crucial library for many scientific applications. Adapting important parts of UMFPACK for running on different hardwares would mean to have more possibilities to use it efficiently. Adapting it for running concurrently on a distributed computer is a sure advantage.

For these reasons my work aimed to develop an alternative implementation of UMFPACK which runs the dense matrix operations on an OpenCL device. This does not only mean to have UMFPACK running on GPUs, but on every possible

OpenCL capable device. These are, at the moment of writing, GPUs, multi-core CPUs and embedded devices of various nature, mainly for portable devices.

However we must be sure about which parts of UMFPACK could take advantage from such different hardwares. This is done by profiling a real case UMFPACK execution in test conditions. A finite element stiffness matrix has been used for this purpose. More precisely, we have used a matrix produced by one of the LifeV test cases.

What LifeV already does is to distribute the computation on clusters of computation units with a domain decomposition method. If each, or some, computer in the cluster had access to a GPU controlled vie the OpenCL language, it could use it to calculate the solution of the local problem. This way we could have two levels of paralleism, effectively increasing the number of computation units available. Having an OpenCL version of UMFPACK could allow this and many other parallel possibilites for achieving low cost high performance computing.

The problem used for the profiling session is defined on a 3D mesh with 1728 (12^3) linear elements. The first significant result is that only the 2.91% of the matrix factorization time was used for the symbolic factorization. This is the stage where UMFPACK decides the column permutation in order to reduce the fill-in. In this part, UMFPACK uses only information about the nonzero structure of the matrix. This symbolic factorization consists mainly of logical computations, with few to none floating point calculations, hence there is no purpose in transposing it on a GPU.

Fortunately the part that uses floating point operations is also the most demanding one: 97.09% of the total time is spent in the numeric factorization, where the actual calculations are done. We can now split the numeric factorization in its different operations, each with its time percentage.

The numeric factorization computation time is furthermore split into:

- 85.70% front factorization with Goto2 BLAS (very fast BLAS implementation)
- 3.33% local search for pivot rows, to increase numerical stability
- 3.23% storage of frontal LU matrices
- 2.91% frontal matrices assembly
- 1.78% frontal matrices growth
- 1.46% columns rescale
- 0.95% creation of new storage units for more contribution matrices
- 0.53% initialization and rescaling of the matrix
- 0.05% clean up
- 0.04% new frontal matrices initialization
- 0.02% other operations

As we might have expected, most of the computation time is spent in the BLAS functions. This is a very good fact, because it means that accessory algorithm parts are working efficently. If the BLAS computation is parallelizable (and it is), we would be in a favourable case.

The BLAS computation is divided itself in different BLAS operations, which we detail now. These are the percentage of time spent for each BLAS operation in the UMFPACK function UMFPACK_blas3_update, which performs the front factorization.

- 98.25% DGEMM (General Matrix Multiply, $C \leftarrow \alpha AB + \beta C$)
- 1.69% DTRSM (Triangular Matrix-Matrix Solve, which finds X so that $AX = \alpha B$, or $XA = \alpha B$, with A transposed or not)
- 0.06% DGER (Rank 1 operation, $A \leftarrow \alpha xy^T + A$)

We could be tempted of just translating in OpenCL the DGEMM operation, since it is responsible for most of the computation time. Anyway this would lead to very bad performances.

In fact, computing part of the operations on the GPU and part on the CPU would require a continue data transfer between host and device. Such a communication need would be unbearable, causing extremely low performance, surely much lower than if we used just the CPU. For this reason we must compute every operation that uses the full matrices on the GPU. This way we can have those matrices stored in the GPU memory without any necessity of data transfer.

Chapter 4

General Purpose GPU and OpenCL

In the '80s personal computers were already a common luxury facility. Since then there were computer games: some of them involved the resolution of puzzles, others were interesting adversary to some popular games, like chess. Many of them were action games, like the classic tennis. This late class of games was structurally of a different kind, a completely new way to use a personal computer.

They required real time interaction, and that meant that both the input was to be collected quickly, and a visual output was to be produced instantly. While the input system was already a complete keyboard, and the output system was a modern CRT monitor or sometimes a television set, what those computers were lacking was computational power to compute complex output images in real time.

In the last two decades, between 1990 and 2010, three dimensional computer games became a standard, replacing almost completely the usual 2D computer graphic. As we said, a good computational power is needed to produce real time three dimensional video images. In modern personal computers this computational power is not given by the central processing unit (CPU) which has already a hard time to manage the increasingly sophisticate logic behaviors hidden into a game.

In modern computers the CPU uses lots of coprocessor to achieve better speed when doing specialized job. Many specialized coprocessors exist, but the one we are interested in is called graphic processing unit (GPU).

GPUs are nowadays specialized processors with huge computing capability. Their role is mainly to stock big and complex sets of triangles in an ideal three dimensional environment, to assemble several of them as the CPU commands, to apply specific colors to them and, finally, to calculate how these sets would look through a virtual window that represents the computer monitor.

All of these operations are simple, but they are to be performed fast enough to guarantee a frame rate of approximately 30 frames per second to maintain scene fluidity. The quantity of triangles needed to achieve an acceptable degree of realism in the represented shapes is huge. Therefore GPUs are needed, because the CPUs is not sufficient to achieve the necessary computational power.

The key words about GPUs are "huge quantity of data" and "real time computation", and these are the two reason why GPUs are so interesting for mathematical simulations. However they were not so usable for that purpose until a few years

ago, as we will see.

4.1 What is GPGPU?

As we stated, the role of a GPU is to move triangles, to color them and to represent them. All of this is achieved with simple operations on 3×3 or 4×4 matrices, needed to perform the geometric transformations on the triangles edges. To color them, instead, interpolations are used, to squeeze image pixels to their right position in the final scene representation.

These operations are interesting but insufficient for almost any numerical simulation, therefore they are not enough to justify the effort of changing an algorithm in a fashion that uses the GPU computational power.

However in the last years GPUs grew capable of doing the most amazing graphic effects, like blur for foggy environment, transparencies for glasses, windows and liquids, reflections as well as partial reflections, for water effects.

Instead of producing different hardware for each effect, the graphic units manufacturers started conceiving programmable graphic units, which could run arbitrary user defined programs in order to manipulate images in some of the key steps of the graphic rendering.

With the possibility to use the GPU to perform any kind of arithmetic operation on specific images, some pioneers started exploiting GPUs to perform other types of computations, on more general data: that is the concept of general purpose graphic processing units, or GPGPU. This evolution also required the development of specialized languages to program GPU computations.

4.1.1 Situation before GPGPU specific languages

It is interesting to know how it has been possible to extend the capability of a GPU to perform calculation on generic data instead of images. There are three main step needed to perform a calculation on a GPU, and those are the same with or without a specific GPGPU language.

Firstly one have to transfer the data on the GPU memory. Graphic units do not share CPU memory for many reasons, therefore any computation on the GPU must be performed on data loaded on the GPU. After data has been uploaded the computation has to be done. Finally one must transfer the results back to the CPU memory, to use them.

When GPGPU was done using non-specific languages, one had to unload, compute and download data with sets of instructions made for graphic purposes. Firstly the data was to be arranged in an image, typically a single 2D squared texture. This texture, if represented, was naturally nothing that a human could perceive, it simply stores the data information in the 4-channel colored pixels, where each pixel was represented by four variables.

This texture was uploaded onto the device with the usual graphic functions and, later, applied to a square of defined size. With a well studied positioning, one can assure that the drawn scene is filled completely and solely by this square, with the texture applied on it. If one was to draw the scene on the monitor at this point, he would see the texture as it is. Otherwise one could write a program to be applied pixel by pixel on the drawed scene. This is exactly where our GPGPU

algorithm needs to work: it is in this step that one can use an arbitrary program on the image.

Finally one should transfer the result back to the CPU memory. Unfortunately it has been drawn on the monitor, and saved nowhere. To achieve GPGPU another step is needed before the rendering is done. One can specify to the GPU that the result of a rendering must be drawn on a texture. This was conceived to allow texture mesh-ups, and permits us to save the computed scene. After the rendering is done, the custom program has been applied to every pixel and the image has been saved into a texture, this is downloaded.

Many constraints have to be taken into consideration: these are constraints about the size of the textures used, the kind of custom program one can write, the way to access the pixels. Although these are not really restrictive when using real images, like one is supposed to do with a GPU, these can become serious problems for the use of a GPU for numerical purposes.

For example there are speed issues with algorithms that use too many variables, or with read-write access to pixels. These are operations that in image processing usually are not done.

4.2 Overview of hardware and languages for GPGPU

Even with heavy restrictions and with all the difficulties of using a GPU beyond its purposes, there were applications where this kind of architecture proved to be dozens of time faster than a CPU.

Some of the first successful experiments were on Monte-Carlo simulations, because the nature of the problem itself is already heavily parallel. In these kind of statistical computations we can be hundreds of times faster on a GPU than on a CPU.

These are applications where one does not need high precision, nor thread interaction between different computation units. They also have a good quantity of computation to perform on each data chunk, therefore the data transfer is not an issue.

A graphic unit is bound to be faster than a CPU in this kind of problems, exactly because it can manage and produce huge quantities of data computation very fast.

The way a GPU achieve such a result is with hundreds of cores: each of them computes a chunk of data (formerly a pixel). One could think that this is obviously way better than any CPU, with one, two, four or even 16 cores.

The answer is obviously negative. As we saw in section 2.2, problems occur when many cores are performing operations on a shared portion of memory. This is a limitation of GPU computing: such an architecture would have to use just one core at a time with the vast majority of known algorithms. In situations like this a CPU is much faster.

The biggest limitation of such an architecture is also the reason of its incredible computational power: each core is bound to perform the same low level operation as all of the other at the same time. Cores perform all the arithmetic, but they share the same logic unit, hence they must act synchronously performing all the same operation. Exactly because of this feature we can nowadays install hundreds of GPU cores on a single chip at a reasonable cost. We can see in the figure (4.1)

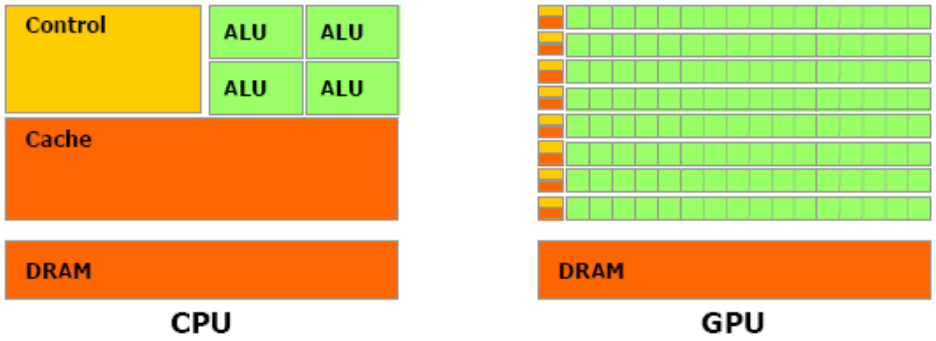

Figure 4.1: GPUs can process much more data concurrently, because of a higher number of arithmetic units.

the difference between a modern CPU and GPU, with the GPU having many more arithmetic units for each control unit.

We will be using the term "to mirror" to refer to this feature: a core is mirroring another if they are performing the same operation at the same clock time.

A further limitation is the need of data transfer between the GPU and CPU memory. Normally, large scale computations manipulate large sets of data, which must be transferred to and from the GPU memory.

This is achieved with high speed bus connections, which can reach enormous speed[1] for this time being. Transferring a huge quantity of data, however, is a time consuming operation which causes an overhead with respect to a computation performed directly by the CPU.

The data transfer problem is crucial even in the most modern architectures. Imagine that a suitable algorithm is found, one that permits to avoid synchronization problems between cores. Imagine then that the same algorithm is mirroring perfectly among the different cores, using them all. The data transfer problem is something you cannot avoid, because at least the results need to be sent back on the CPU memory, to permit the further use of them, or simply to save them on a persistent memory.

4.2.1 The OpenCL language and library

Because of these limitations, the same algorithm just transposed on GPU usually performs very badly, usually much worse than on the CPU. To code efficient programs on a multi-core environment it is necessary not just to adapt algorithms, but to change the way of thinking them. In the basic classes of computer science one is taught the basic logic blocks, like the conditional block and the loops.

Nowadays, in my point of view, this is not enough. As naturally as someone can think of an algorithm section that is repeated over and over in a loop, it is necessary to teach how some kind of computations can be solved with loops, but they are faster with a parallel execution. In many cases this is as simple as it looks, therefore it is a pity not to teach it the right way from the first time, well before students can get much harder and unnatural concepts like pointers.

[1] PCI Express 2.0 allows a theoretical limit of 16GB/s.

4.2.2 OpenCL, a language for parallel computing devices

However it is necessary to understand some non natural concepts to implement a fast algorithm on GPUs. Not many years ago the situation was even worse: because of the fast development of this new way of computing, GPU vendors started releasing new programming languages, just for GPGPU. The main problem was that GPUs are all different, especially those from different vendors, therefore every language was specifically designed for one particular architecture.

The two biggest vendors were Nvidia, with the CUDA interface and language, and ATI, with its Close to The Metal interface which permitted the development of the Brook language. These languages were thought for a single architecture, therefore they were reflecting and exposing that single architecture mechanisms.

Fortunately in the past three years graphic card vendors did an effort trying to unify and standardize some of the concepts shared between all of their architectures, and they finally realized a new language specification. This new language is called OpenCL and it is bound to work on all architectures that supports it, which are now the vast majority of the new graphic cards. OpenCL stands for Open Computing Language, and it is a standard like OpenGL (Open Graphic Language).

The OpenCL language has started thank to an effort of the major graphic card vendors as well as other big companies involved in information technology. It was designed to work out of the box on GPUs, but it is not limited to them. It is both a language and a C library which permits to write programs to be executed on a parallel processor, and to run them from a standard environment supporting C libraries. It could be use on fast digital processing systems, and implementations have already been made to use the CPUs multi-core structure itself as a parallel device.

An OpenCL code looks like a C program, in fact it is a C99 extension (and sometimes restriction). Not every aspect of the C99 specification could be adapted to a parallel environment.

4.2.3 The SIMT/SIMD architecture

As we stated, to design fast algorithms with the OpenCL language we need to know how the parallel computing is performed and which are the restrictions that limit us.

A graphic processing units implements what is called the SIMT (Single Instruction Multiple Thread) architecture, a slight modification of the more famous SIMD (Single Instruction Multiple Data) that is the architecture implemented by an ideal vector processor. A computing unit implementing SIMD must be able to fetch a set of data and elaborate all of the single data units at the same time, using the very same instruction. As we said before, this poses a very restrictive constraint: it is impossible to choose, based on a condition, not to process a specific unit of data. It is impossible, in other words, to have a conditional block based on a non constant value, which is exactly the point of normal conditional blocks.

The SIMT structure, however, allows such a conditional behavior. All of the thread must perform mirrored operations, but they can choose to deactivate themselves. When a conditional block is reached each thread evaluates the condition, each one on its own data. All of the threads that must execute the first possible block of conditional instructions start the execution, while all of the others wait,

Algorithm 4.1 Pseudo code for a CPU performing an outer product

```
allocate vector a, 8 entries
allocate vector b, 8 entries
allocate matrix C, 64 entries
fill a with values
fill b with values
loop (i from 1 to 8)
        if a[i] != 0
                loop (j from 1 to 8)
                        C(i,j) = a[i]*b[j]
                end loop
        else
                loop (j from 1 to 8)
                        C(i,j) = 0
                end loop
        end if
end loop
```

deactivated. When the end of the first block is reached, all of the other threads activate and whose that were running deactivate, until the end of the other block is reached. This means that if a code uses lots of conditional blocks, it is bound to perform badly on a GPU, which is not at all the case with normal programs written for CPUs.

Let us think at a useful example: we are performing an outer product between two vectors of size 8, in order to build an update matrix for a multifrontal solver. On a CPU we could avoid computing a zero line issued of a zero element of the first vector, simply by testing if that entry is zero.

On a CPU with an algorithm like the pseudo code (4.1) we would save 8 multiplications at the cost of just one conditional test. Let us look instead at the pseudo code (4.2), which is the same algorithm if it was written for a GPU with 8 concurrent threads.

This naive algorithm is ran on all of the 8 possible threads, and each thread knows its unique identifier, which could be an integer between 1 and 8. When the conditional block is reached, every thread with a nonzero entry in the first vector starts the first loop, while every other core is waiting.

When the first set of threads finishes the loop, the else statement is reached. The first set of threads is deactivated and the set with a zero entry in the first vector is activated again, computing its own loop.

In this example we had all of the threads waiting idle for a loop to finish, bringing bad performances, since two loops are ran instead of one. The algorithm (4.3) is instead a non optimized outer product on GPU, but as we can easily see it only runs one loop, hence being faster than the latter.

4.2.4 Synchronization between workers in OpenCL

A graphic processing unit can run many simultaneus threads, but they still are a finite quantity. OpenCL, like other previous languages for GPGPU, permits to run

Algorithm 4.2 Pseudo code for a GPU performing a naively optimized outer product

```
allocate vector a, 8 entries
allocate vector b, 8 entries
allocate matrix C, 64 entries
fill a with values
fill b with values
allocate integer i
i = my core identifier (from 1 to 8)
if a[i] != 0
        loop (j from 1 to 8)
                C(i,j) = a[i]*b[j]
        end loop
else
        loop (j from 1 to 8)
                C(i,j) = 0
        end loop
end if
```

Algorithm 4.3 Pseudo code for a GPU performing a standard outer product

```
allocate vector a, 8 entries
allocate vector b, 8 entries
allocate matrix C, 64 entries
fill a with values
fill b with values
allocate integer i
i = my core identifier (from 1 to 8)
loop (j from 1 to 8)
        C(i,j) = a[i]*b[j]
end loop
```

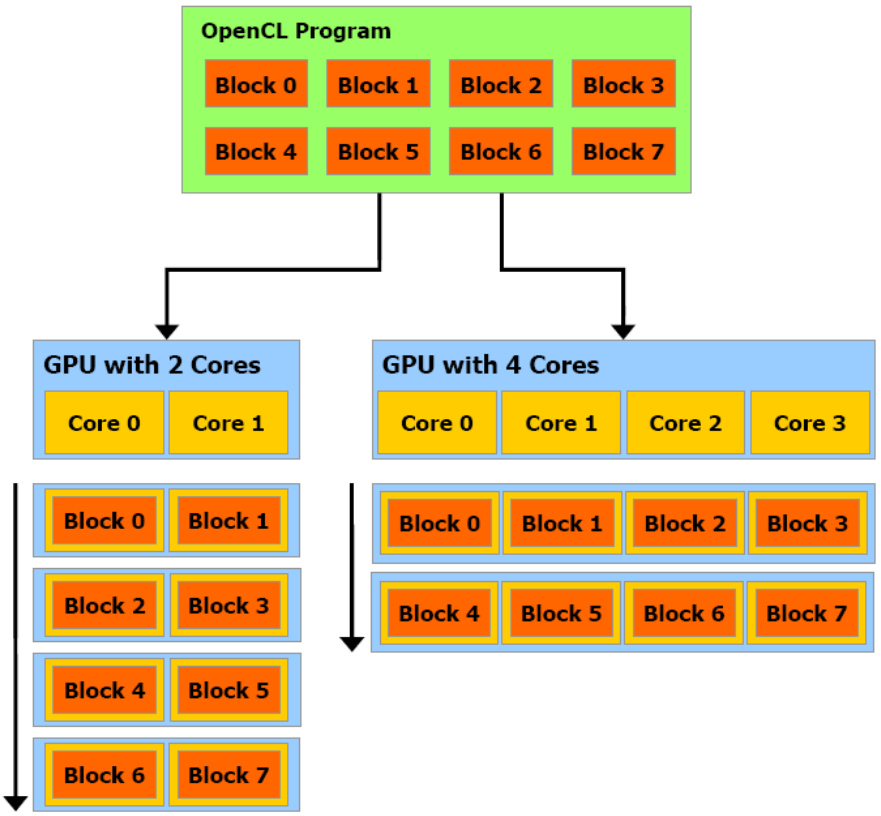

Figure 4.2: The workers-threads-blocks abstraction is needed to use the same program on different GPUs

an algorithm with more workers than those that can physically run concurrently, being a worker a concept similar to a thread, as the OpenCL specification states.

We will be using the term worker as a scheduled OpenCL program execution that must compute a single job, while the term thread will be used to refer to concurrent threads. The term block (of threads) will be used to define a set of threads that can run concurrently on a single GPU core. The size of a block depends, among other factors, on how many ALUs each GPU core possesses.

Different workers must be ran within threads, hence some of the workers will be running concurrently, while the others will be ran as soon as a thread is available. The same OpenCL program can run on different GPUs, which can run different quantities of threads, depending on how many cores they have: the block/worker abstraction layer was made to let reuse the same program on many different devices, as shown in figure 4.2.

Since workers are executed sometimes concurrently, and sometimes sequentially in unknown order, one could ask if OpenCL gives any way to achieve synchronization between workers. The answer is obviously yes, but just between a restricted number of workers, that we will call a local group. In fact, workers are organized

in local groups, while all of the local groups form the global group. Workers within a local group are given the possibility to synchronize their execution with a set of barrier instructions.

A barrier is an instruction that must be reached by all of the workers in the local group before any of them is allowed to go further. Different barriers are made to ensure that specific operations are finished by all of the workers in a local group: for example it is possible to synchronize just those workers that must write on the memory.

The number of workers in a local group is limited by the features of the device and the number of variables used by the OpenCL program. This happens because it can be that some of the workers have reached a barrier, but the number of workers is higher than the number of possible threads. Those executing workers which reached the barrier must save their execution registries and variables and let other workers use their thread to reach the barrier.

If the program uses too many variables, then the number of workers that can save their state decreases, hence the maximum allowed local group size decreases too. Since there is no way to synchronize operations between workers of different local groups, the maximum allowed global group size is not bound.

As we saw, synchronization between workers in a local group involves data operations, since a thread state must be saved before another worker is assigned to the same thread. In the same way there are data operations when a worker must resume its previous state to go on with the execution. These operations could become significant bottlenecks: for this reason one should avoid putting barriers whenever it is possible. If a synchronization is mandatory, one should verify in each case if the program running on the GPU is faster than the same running on CPU, because it might not be the case.

Because of the division in local and global groups, each OpenCL worker may define its own variables in different scopes. A variable defined in the private scope will be usable only by the same worker that defined it. Since the code is shared among all workers, each worker will have a private copy of that variable. Local scope variables will be shared among all of the workers in the local group.

Global scope variables will be shared among all of the workers. The data transferred from the CPU memory to the GPU memory can only resides in the global scope, hence syncronization might be necessary when accessing this data.

4.2.5 Data scattering and gathering

As we explained in section 2.2.5, problems arise when different cores access the same memory location in an unsynchronized way. Race conditions may be hard to find, hard to debug, and they can very well cause a code to behave apparently randomly. In SIMT structures we cannot say that the operations are unsynchronized, because they are perfectly mirrored between threads. Still race conditions may happen, exactly because of the concurrent execution.

A parallel program will usually have to compute huge quantities of data, normally larger than the number of threads allowed by the hardware. Let us think at all this data as an array, where each entry is a single data unit.

Algorithms operating on sparse matrices access the memory mainly in two ways (but other situations may arise as well). These two different behaviours are somehow opposite. We talk about data gathering if every thread access in read

Algorithm 4.4 Pseudo code for a GPU performing an outer product without any loop

```
allocate global vector a, 8 entries
allocate global vector b, 8 entries
allocate global matrix C, 64 entries
fill a with values
fill b with values
allocate private integer myID
allocate private integer i
allocate private integer j
myID = my core identifier (from 1 to 64)
i = floor( myID / 8 )
j = myID - 8 * i
C(i,j) = a[i]*b[j]
```

mode to a data entry. This behaviour does not pose any race condition, in fact it is an example of perfectly parallel algorithm, where different threads do not interact in any way.

Problems may arise when data scattering is happening. That is when two or more threads are accessing in write mode to the memory. If those threads write on the same memory portion, they could cause unpredictable behaviours. While data gathering is rarely a problem, data scattering is a more complex matter, both to implement and to use in a program.

We can look at what happens in the example given in Algorithm 4.4, which is another way to perform the outer product.

We can see in the pseudo code that we had 64 workers available, one for each entry of the result matrix C. With so many workers we do not need any loop. At the same time we are using more of the cores concurrently (64 instead of 8), which may lead to an important speed increase. With an integer division we can have each worker to calculate its own row and column. Naturally, this comes at the cost of a subtraction, a multiplication and the integer division.

Each worker will access in write mode to its own entry of the matrix C: this is a scattering operation that brings no problems at all, because it is carried out on different memory entries. The gathering operation is performed while accessing read only to the vectors entries. There will be 8 workers with the same row number i and 8 with the same column number b. Being that these operations are read only, this is a perfectly usable algorithm on modern GPUs.

As we saw, scattering operations may be used safely in certain cases, but more complex behaviors are allowed. Let us say that we want to check the symmetry of a full matrix A. In the example presented with Algorithm 4.5 we will put the information about the symmetry in a variable $symm$, where at the end of the program $symm = 1$ will mean that the matrix was symmetric.

This example is not optimized, because a double check is made for each entry. However, we will not present an optimized version of it, because it is not necessary for our intent.

When the algorithm starts different variables are allocated. One important variable is $symm$, which has been defined in the global scope. It was necessary to

Algorithm 4.5 Pseudo code for a GPU performing a symmetry check

```
allocate global matrix A, 64 entries
fill a with values
fill b with values
allocate private integer myID
allocate private integer i
allocate private integer j
allocate global integer symm
symm = 1
myID = my core identifier (from 1 to 64)
i = floor( myID / 8 )
j = myID - 8 * i
if A(i,j) != A(j,i)
        symm = 0
end if
```

allocate it in global scope because otherwise it would not be possible to transfer it back to the CPU memory at the end of the check.

No problems occour when the worker writes on the $myID$, i and j variables, because they are in the private scope of each core. Something different happens when writing on the *symm* variable, because it is in the global scope, i.e. it is common to all cores. Each worker assigns 1 to the global variable just after the allocation. This is a scattering write operation, where every worker writes on the same variable.

All of the workers which encounter a non symmetric entry will execute the write on *symm*, hence this is a scattering operation using a common memory entry. As we can see, each single worker that encounters a non symmetric entry can tag the whole matrix as non symmetric. No race condition can happen, since this is a plain assignment and the *symm* variable is not modified anywhere else.

In the Algorithm 4.6 we show an example of bad programming, where the scattering operation brings to unpredictable behaviours. Instead of simply checking if the matrix is symmetric we want to know how many entries are unsymmetric. We initialize *unsymm* to 0 and we let each core to add 1 to *unsymm* if its entry is non symmetric. Having more workers on the same increment instruction leads to race conditions, exactly as we saw in section 2.2.5.

4.2.6 Asynchronous devices

Graphic processing units posses their own clock, therefore they usually compute operations with a frequency different from that of the CPU. Current GPUs have even adaptable clock frequencies, to save energy and heat when low processing power is needed.

The fact that usually the GPU and the CPU do not share the clock is another reminder that they are two different processors, and they can (and should) run separately, executing different operations. We say that GPUs are *asynchronous devices*, because they can run operations even when the CPU is doing something else.

Algorithm 4.6 Erroneous pseudo code for a GPU performing a non-symmetric entries count

```
allocate global matrix A, 64 entries
fill a with values
fill b with values
allocate private integer myID
allocate private integer i
allocate private integer j
allocate global integer unsymm
unsymm = 0
myID = my core identifier (from 1 to 64)
i = floor( myID / 8 )
j = myID - 8 * i
if A(i,j) != A(j,i)
        unsymm = unsymm + 1
end if
```

This is a very useful observation, because in most of the applications where GPGPU is involved performances are a very high concern. Because of the asynchronous feature of the GPUs we may let the CPU perform other operations (e.g. input/output operations) while the GPU is computing.

The OpenCL specification lets the programmer decide whether to use synchronous or asynchronous operations. To use the synchronous operations means to lose computing power, because the CPU will wait idle until the GPU finishes the operation.

To manage asynchronous operations in ways that do not result in possible crashes is not trivial. One must ensure that a computation is finished before to download its result on the CPU memory, or to delete its input parameters. This is not as simple as it seems, especially with non object oriented languages or when the computation is launched by one portion of the code and the results are downloaded by another one.

To manage asynchronous operations efficiently is all but easy. There are OpenCL functions that let the CPU wait until a specific asynchronous operation is finished, but if they are used without the real need of it we risk to go back to a synchronous behavior, with the CPU waiting idle most of the time.

4.2.7 Operations scheduler

Since the GPU is an asynchronous device the CPU is allowed to continue the computation before the GPU has finished to work. If the CPU launch another OpenCL program while the GPU is working on the previous, the latter will end in a queue.

Once the previous operation is finished, the GPU scheduler will decide which program to execute choosing them among those in the queue. The OpenCL specification permits two kinds of schedulers: the ordered and the unordered one.

The ordered scheduler will execute the operations in the queue one by one, using the queue as a FIFO (first in first out) list. This is the simpler choice,

because one can avoid warring about the synchronization of the GPU commands themselves. Since there is already a lot of synchronization to do, this could be a good choice for a moderately performing code.

The most performing choice is the unordered scheduler. When the GPU finishes the execution of a command, the scheduler will pick another job from the queue in the most performing order.

How can such a behavior be more performing? There might be situations when a scheduled work does not use the whole power of the GPU. If another program in the queue has similar operations, or if it is exactly the same program launched a second time, the scheduler could decide to run them at the same time.

Such an advanced scheduler lets one use the full computational power of a device even if the OpenCL programs are not fully optimized. A mere ordered scheduler could let a fully optimized OpenCL program work on too little data at each time, hence letting part of the GPU power unused. The unordered scheduler is a better choice if the scheduler itself is well coded, which might not be the case if the GPU vendor did a poor job on it. If instead we have a good quality unordered scheduler, there still could be problems with operations which have to be executed in order.

For this reason when enqueueing an OpenCL program with an unordered scheduler it is important to specify which already scheduled operations must be finished before that program can be ran. For once this is not an hard task, since the needed order of sequential operations is usually a well known fact of an algorithm.

For the whole section we spoke of operations instead of programs. This is because in OpenCL upload and download operations share the same queue of the launched programs, and they can be synchronous or asynchronous as well.

Before the scheduler run any program we always want to be sure that the needed data has been uploaded from the CPU memory to the GPU memory. Hence with an unordered scheduler one must specify that the upload operation must be finished before the program is run.

4.2.8 OpenCL glossary

As we saw synchronization is the most troubling problem when writing a parallel code. When an OpenCL program has to be ran we need to synchronize the access to the GPU memory, the order of GPU command and the timing between CPU and GPU operations.

All of the concepts that we introduced in this chapter are needed to use a parallel computing device in non trivial cases. In fortunate situations where an algorithm is very apt to be executed in a parallel fashion one can forget almost everything we have said, and still have good results.

In a previous work we showed how synchronized GPU operations with an ordered scheduler and a huge need of data transfer could accelerate the execution of a sparse matrix product with a full vector. In that case we had the GPU to perform 30% faster than the CPU, with the data transfer as bottleneck. However if the best performances have to be reached, and GPGPU is all about performances, one cannot ignore all the concepts we stated.

The OpenCL library specification defines names for most of the entities that we used in the previous descriptions. We want now to enumerate them to use them further as technical words.

OpenCL: it is both the name of the language for parallel devices (an expansion of a subset of the C99 specification), and the C library which permits to compile run and manage programs written in the OpenCL language. Since it is a C library it must be linked within a standard CPU program: OpenCL programs are launched by CPU programs, they cannot be launched on their own.

Device: it is the word which indicates an OpenCL capable processor and its memory. It might be a CPU or a GPU or anything else which can run OpenCL programs. Since just the GPU has been used for our work, device and GPU can be used as synonymous in most cases.

Host: it is the system which runs the program that will schedule the OpenCL operations. It is usually the computer which mounts the graphic card used as device. It must contain a CPU and a memory, at least. Every host can have one or more devices.

Context: a context is the virtual environment set by the OpenCL capable device driver. This environment links one or more devices with their host, the devices themselves with their accessible memory, it permits host-device synchronization, it maintains one or more command queues with their schedulers and it permits data transfer between the host memory and the devices memory. It is the environment in which OpenCL programs run.

Memory Object: it is an allocated portion of device accessible memory which can be used as data by an OpenCL program. Different kind of memory objects exist, with different features, but we will be using almost exclusively the Buffer, which is a multi/mono-dimensional array of primitive data entries. Memory objects must be allocated and deallocated by the host, and the upload and download of data to and from a memory object is an OpenCL command.

Kernel: it is a function written in the OpenCL language that can be compiled as a program and launched by the host with an OpenCL command. A kernel may call further OpenCL functions, or even other kernels since they are OpenCL functions. A kernel will run on each worker partially concurrently and partially sequentially based on the possibilities of the device.

Program: it is a kernel which has been compiled and is ready for the execution. Programs may use parameters, which can be memory objects or primitive data. A program does not return values, therefore every computation result must be written on a memory object that can be retrieved by the host after the end of the execution.

Command-queue: it is the queue where the host schedules the OpenCL command to be executed. A command-queue can posses an ordered or an unordered scheduler. That command-queue will be called respectively in-order or out-of-order queue.

Command: it is one out of a set of possible asynchronous operations. All the data transfer between host memory and memory objects are commands, as well as the launch of a kernel is a command. Each command is executed by enqueuing it in a command-queue.

Work-item: it is the OpenCL term for what we called worker. It represents one of the parallel execution of a kernel. Synchronization is permitted between work items under certain conditions.

Work-group: it is the OpenCL term for what we called local group. It is a set of work-items that shares variables defined in the local memory. Work-items in the same work-group may be synchronized with the use of barrier functions. When launching an OpenCL program the host may force the work-groups to be of a certain size or shape: n-dimensional work-groups are allowed depending on the device capabilities. If the indicated work-group size or dimensionality exceeds the maximum allowed, the enqueued execution will fail.

Global-memory: it is the scope of the memory-objects. Variables in global memory may be used by all work-items in every work-group.

Local-memory: it is the scope of variables that are shared between work-items of the same work-group. Local variables cannot be initialized on definition: they must be defined uninitialized. It is possible for the host to allocate local variables, but this feature is seldom used since the host cannot read or write on local memory.

Private-memory: it is the scope of variables private to a single work-item. Private variables are treated like standard C99 variables. The host cannot interact with private variables in any way.

Local-ID: it is an unique integer identifier starting from 0 included which identifies a work-item in its work-group. If the work-group has more than 1 dimension it is a vector which identifies the local work-item position within the array representing the work-group.

Global-ID: it is an unique integer identifier starting from 0 included which identifies a work-item among those scheduled for the execution of an OpenCL program (what we called a global-group). If the global group has more than 1 dimension it is a vector which identifies the global work-item position within the array representing the global group.

Event: it is an abstract object that permits to synchronize the CPU execution against an asynchronous operation that was previously started. When a kernel is launched, an event object is returned. Waiting on that event will cause the CPU to go idle until the kernel has finished its execution. Events are returned on request by every asynchronous operation.

Chapter 5

The dbOpenCL library

When performing computationally expensive simulations the domain decomposition approach leads us to the resolution of smaller local problems. Scattering those problems among a cluster of computing units may bring to shorter resolution times, which are usually preferable.

The speed of the algorithm used by all of the computing units will impact heavily on the overall speed of the resolution problem. Since GPGPU is usually a faster way to solve computationally intensive problems, with this project we wanted to transfer the computationally expensive parts of a local problem to GPU.

Based on the fact that often most of the resolution time for the local problem resides in solving the linear system suited of a local Galerkin problem, we decided to rewrite using GPGPU some parts of the linear solver.

We chose to rewrite parts of the linear solver package UMFPACK because of the very good performance of the latter and the vast diffusion of this library. In fact a faster UMFPACK would have a great impact on several computing programs.

Since UMFPACK is a constantly developed project, I thought that it would have been better if the GPGPU code was not integrated directly into the UMFPACK implementation. For this reason I designed a GPGPU library in OpenCL called dbOpenCL.

5.1 Goals of the library

The library is written in C++ for better ease of use and better design: it makes extensive use of the object oriented paradigm. Furthermore, a C++ header exists that gives the complete C++ binding to the OpenCL library, that is instead a plain C library.

This header gives access to all of the functionalities of the OpenCL library, and it manages the OpenCL resources by itself within the constructor and destructor methods of the various classes.

Because of its greater ease of use I preferred writing my own library using the C++ bindings. The UMFPACK library is written using the C language, therefore I could not have used the C++ bindings within UMFPACK without writing a library.

5.1.1 Goals of the library API design

I personally believe that each library must hide the most of the implementation details. In this case it would mean to give to the user the idea that to integrate GPGPU computing into a generic calculus project is easy and error-proof.

Obviously any user of the library must have some idea about GPGPU, to avoid simple errors. The needed concepts are those we introduced in section 4.2.

If the goal of hiding most of the implementation details is reached, the result will be that the integration within UMFPACK will need a minimum amount of code. Reducing the quantity of new lines into the original UMFPACK code brings some advantages.

Fewer code lines into the UMFPACK source files bring fewer bugs and errors: there is the actual possibility to write an immediately working code, without the need of long and undesirable debugging phases onto the whole UMFPACK code. Obviously the debugging has been necessary, but most of it was performed on the library itself, which is a smaller portion of code.

If too many code lines were needed into UMFPACK there was the possibility to hide the actual numeric work and expose too much of the accessory OpenCL code. This is undesirable because a complex numeric library like UMFPACK is hard to read as it is.

Keeping the needed lines to do GPGPU at a minimum will permit to more people to handle them, hence it will improve the possible diffusion of the library.

In fact another goal of the library is to give a calculus specific binding for OpenCL, excluding all of those possible choices that are useless to a calculus program, while keeping it generic and not bound to UMFPACK.

The library want to be a mean to easily integrate GPGPU into any calculus code in C or C++: such a library works fine with UMFPACK but could be used in many other calculus packages and programs.

5.1.2 Inside mechanics goals

The dbOpenCL library is studied to bring better out-of-the-box performances to a classic CPU only code. As we saw in section 4.2 there are some performance possibilities when using OpenCL.

To give better performances the library is completely asynchronous. This lets the GPU work on the numeric calculation while the CPU continues with the logic details of the multi-frontal method. To let the GPU work asynchronously is an immediate source of performance, because no CPU time is wasted.

The command-queues have a in-order scheduler, mainly because when I was writing the library the Nvidia OpenCL implementation I was using was in a beta phase which frequently gave me undocumented problems with synchronization. I did not want to add problems managing several events at a time.

While the Nvidia OpenCL driver was not stable, the main problem was a memory leak within the C++ OpenCL binding. This binding was not releasing the generated events, hence causing further problems in the event management.

Thus, although the goal was to realize a performant out-of-order execution, the library uses an in-order scheduler. When both the C++ bindings and the OpenCL implementation will be stable and definitive, it will be possible to use an out-of-order scheduler, for which the whole library is already prepared.

During the final phase of the debugging I reported the C++ binding bug and found a work around, which avoids memory leaks in the implementation, but the Nvidia driver bugs remained. When the project was already concluded, Nvidia released a new version of the driver, which corrected most of the unwanted behaviours. This means that in a next future this same library will use an out-of-order scheduler.

Since this is a calculus library, the central object is the matrix, which is implemented with the purpose of being easy to handle. A matrix in the dbOpenCL library is a memory object on the device, a Buffer in our case. This device memory object may or may not be linked with a matrix in full notation on the host memory.

If it is linked with a host matrix, the operations of uploading and downloading the matrix have to be immediate, of the greater possible ease of use.

Synchronization must be automatic. Each memory object registers the last occurred event. If the memory object is deleted, the destructor must take care of waiting for the last event to be finished.

In the same way every operation which can fail because of bad synchronization is automatically synchronized.

5.2 Design of the library

The set of classes I implemented was designed to satisfy the wanted objectives, especially the need of an easy to use interface. Furthermore, since I wanted the package to be reusable and easy to develop, I built a structure that permits to write as few code lines as possible for each new GPGPU kernel.

The result of such a design is in the following classes, which are detailed further in this section. Every class is defined within a namespace called "cl", which is the same namespace for the classes of the original C++ OpenCL wrapper.

Configuration is a class conceived to be a singleton. It must provide to the user a complete and working OpenCL environement, selecting a suitable device, mantaining a valid context for the device. It also provides and mantains the command queue for that device. Hence it encapsulates every useful singleton object we will be using afterwards. It is important to notice that this class must work without any hint on which device to choose, or how to manage it, to achieve the goal of minium user knowledge about the underlying architecture. A more sophisticated version of this class could, however, let the user specify his preferences, if any. This class has most of its methods redefined as extern functions for being used directly by the C code.

FullMatrix is the class that represents full matrix or vector, like those used by UMFPACK in the BLAS operations, with the difference that these are stored on the device memory, with a buffer. It is a template class that allows different primitive types to be used as entries of the matrix. It saves the needed information to synchronize the access and the destruction of the device memory object. If a host memory pointer is given to the constructor, it can manage the memory transfer from host to device without any further information. It supports both column-major and row-major order storage formats, which is important because the local LU factorization in

the UMFPACK implementation keeps U in row-major order while all of the other matrices are in column-major order. This class has most of its methods redefined as extern functions for being used directly by the C code. Since this is a template class, the C wrapper functions are given for the double precision floating point primitive type and the integer primitive type. The integer types considered are for both the 32 and 64 bit architectures. This is the main point where the library becomes architecture dependant. This class has most of its methods redefined as extern functions for being used directly by the C code.

KernelHelper is a class that provides an easy interface for a kernel OpenCL source code compilation. With a given source file path and the kernel name it produces and mantains the compiled Kernel object and the built Program object, which are necessary to the kernel execution on the device. It has methods to calculate the maximum local group size, which depends on the kernel and the device. It provides the methods necessary to set the kernel arguments and to launch an asynchronous execution with a given work-group size. If such a work-group size is not given, a default is used.

BlasOperatorFullMatrices is almost an interface. It is meant to be the standard structure of each class representing a specific kernel. In fact most of the following classes will be defined as child class of BlasOperatorFullMatrices class. It is not completely an interface class because it encapsulates a KernelHelper, which will be responsible for compiling the kernel and executing it. The children classes have hardcoded file paths and kernel names to the specific kernel source code. In a further development I would like not to have hardcoded file paths.

BLAS_add_value is a child class of BlasOperatorFullMatrix, with public inheritance. It works on a given sub matrix of the FullMatrix object. It adds to that sub matrix a given value. The sub matrix can be specified with an offset from the first entry of the FullMatrix, the lenght of each column (or row if the FullMatrix is in row-major order) and the number entries to modify.

BLAS_copy_raw is a child class of BlasOperatorFullMatrix, with public inheritance. It works on two FullMatrix objects, which may be the same. This kernel copies a given sub matrix of the first FullMatrix object into a second sub matrix of the other FullMatrix. The sub matrices are specified by their offset and the leading dimensions. The number of entries copied is a parameter. If the leading dimension of both the sub matrices is 1, a special buffer operation can be used instead of a kernel. The situation is automatically exploited.

BLAS_scale_raw_device_value is a child class of BlasOperatorFullMatrix, with public inheritance. It works on a FullMatrix object, dividing each entry of a specified sub matrix by a given value. The submatrix is given by an offset, the FullMatrix leading dimension for each column (or row if in row-major order), the size of the sub matrix in columns and rows (or rows and columns if in row-major order). This is an important operation when

computing the update matrices for the multi frontal method. A control is performed to avoid divisions by zero, like in the original UMFPACK code.

BLAS_set_to_value is a child class of BlasOperatorFullMatrix, with public inheritance. It sets to a given value every entry of a given sub matrix of the FullMatrix object. The sub matrix is given by an offset, the FullMatrix leading dimension for each column (or row if in row-major order), the size of the sub matrix in columns and rows (or rows and columns if in row-major order).

BLAS_*_umfpack is a set of classes, children of BlasOperatorFullMatrix with public inheritance. They mantain kernels specifically written as UMFPACK replacements. They replicate specific full matrix operations which are interesting only if used within UMFPACK. Many of them uses index vectors too, to manage column and row ordering directly on the GPU. Some of them instead contains BLAS2 and BLAS3 operations with some fixed paramter, hence they are specifically designed for UMFPACK but they could be reused.

BLAS_gemm_umfpack is a child class of BlasOperatorFullMatrix, with public inheritance. It performs a particular form of DGEMM operation. It works with three FullMatrix objects A, B and C, performing the following operation: $C = C - AB^T$, with A being an m-by-k matrix with leading dimension ldac given, B is a k-by-n matrix with leading dimension ldb given and C is m-by-n with leading dimension ldac. The original GEMM operation would instead perform the following operation: $C = \beta C + \alpha AB$ with A and/or B transposed at will.

BLAS_gemv_umfpack is a child class of BlasOperatorFullMatrix, with public inheritance. It performs a particular form of DGEMV operation. It works on a FullMatrix A and two vectors x and y which are still represented as FullMatrix objects. This kernel performs the following operation: $y = y - Ax$ where A is an m-by-n matrix with given leading dimension. The vectors x and y have appropriate dimensions. The original GEMV operation would instead perform the following operation: $y = \beta y + \alpha Ax$ with A transposed at will.

BLAS_ger_umfpack is a child class of BlasOperatorFullMatrix, with public inheritance. It performs a particular form of DGER operation. It works on a FullMatrix A and two vectors x and y which are still represented as FullMatrix objects. This kernel performs the following operation: $A = A - xy^T$ where A is an m-by-n matrix with given leading dimension, while the vectors x and y have appropriate dimensions. The original GER operation performs instead: $A = A + \alpha xy^T$ with an arbitrary α but without any A transposition. It is a rank 1 operation which can be considerably fastened by GPGPU.

BLAS_trsm_umfpack is a child class of BlasOperatorFullMatrix, with public inheritance. It performs a particular form of DTRSM operation, working on two FullMatrices A and B. It saves into the FullMatrix B the result X of the resolution of $XA' = B$, knowing that A is in lower triangular form. The original TRSM operation can solve two problems. One is to save into B the

result X of $AX = \alpha B$ with A in upper or lower triangular form, transposed at will. The second problem is to save into B the result X of $XA = \alpha B$ with A in upper or lower triangular form, transposed at will.

BLAS_trsv_umfpack is a child class of BlasOperatorFullMatrix, with public inheritance. It performs a particular form of DTRSV operation. It works on a single FullMatrix A which is in lower triangular form. It solves the problem $Ax = b$ where b is a vector taken from the same matrix A with a given offset. The result x is overwritten on the very same matrix A. The original TRSV operation would have solved the system $Ax = b$ with A being a lower or upper triangular matrix transposed at will. In the original TRSV b can be a distinct vector instead of being part of A, and the result x is overwritten on b.

BLASFullMatrices is an encapsuler for all of the BLAS_* classes. It is responsible for the singleton behavior for each child of BlasOperatorFullMatrix, in order to avoid double run-time compilations of the kernels. It keeps references to those objects and it provides the user with a standardized interface for calling the kernels. It performs the complete deallocation of all of the used kernels when the BLASFullMatrices object is destroyed. This class has most of its methods redefined as extern functions for being used directly by the C code.

5.2.1 Class interactions in C++

If the library was used within a C++ application, the classes of the "cl" namespace needed to achieve GPGPU would be 3: Configuration, FullMatrix and BLASFullMatrices.

It could have been possible to reduce the number of class needed, using just FullMatrix and BLASFullMatrices. To achieve that I should have used the Configuration singleton directly into the other two classes. However this design would have been too much restrictive.

In a future implementation which considers multiple devices on the same host we would be forced to have the Configuration object as a non singleton. For each device a Configuration object should be instantiated, so that each device will have its own OpenCL context, command-queue, memory-objects, kernels, and so on.

The way to perform GPGPU whith the dbOpenCL library can be resumed with the following C++ code.

```
using namespace cl;

// Prepare the OpenCL environement with standard parameters.
Configuration * clConf = Configuration::getConfiguration();

// Prepare the container for the kernels.
BLASFullMatrices * clBLAS = new BLASFullMatrices(clConf);

/**
 * Prepare the host matrices and vectors,
 * just one in this case.
**/
```

```
double * A = new double[N*N]

// Fill A with a suitable content.
...

/**
 * Prepare the device matrices and vectors,
 * just one in this case.
 * A has type double*: clA has type FullMatrix<double> *.
**/
FullMatrix * clA = new FullMatrix(
               A, // host memory to use
               N, // column size of the matrix
               N, // row size of the matrix
               clConf, // configuration object
               TRUE // column-major order
               );

/**
 * Start an asynchronous upload from host to device.
 * When this operation will be finished the entries
 * of clA will be equals to those of A.
**/
clA->upload();

/**
 * Perform an asynchronous operation on the device,
 * sum "value" to all of the entries in clA
 * in this case.
 * This is the first time the add_value kernel is
 * used: it will be compiled automatically.
**/
int offset = 0;
clBLAS->add_value( clA, // FullMatrix object to use
               value, // value to sum
               offset, // starting entry (by reference)
               N, // leading dimension
               N*N ); // number of entries to modify

// Start an asynchronous download from device to host
clA->download(false); // false = asynchronous

/**
 * Destroy clA.
 * This waits automatically for pending operations
 * to complete.
**/
delete clA;
```

```
// Clean up the kernels and the environement
delete clBLAS;
delete clConf;

// Clean up the memory
delete[] A;
```

In this example we used some of the basic functionalities of the library. Notably we used the automatic synchronization of pending operations when deleting the FullMatrix clA. This is the only automatically synchronized class, therefore it is useful to delete it as first thing when cleaning up at the end of the execution. This will ensure that all of the command-queue operations have finished their executions.

We took advantage of the just-in-time compilation for the kernels encapsulated by the BLASFullMatrices class. This class contains pointers for 15 different kernels. To compile them all is an useless effort if just some is used.

Therefore the only kernel used in this example code is compiled just the first (and only) time it is used. When clBLAS is deleted so are all of the compiled kernels, and their resources on the device are freed.

Notice how most of the GPGPU concepts are hidden by the design, exposing to the user just the asynchronous operations.

5.2.2 Functions interaction in C

If the library is used through its C wrapper we will be using only functions and datatypes starting by "cl_". This is a common way to recognize elements from the same package, working somehow like a name space. Even if the C language is not an object oriented language, we are just using it to access to a C++ library, henceforth the concept of object will be present any way.

We will use as example the very same algorithm as before, to look immediately at the differences and the similarities.

```
// Prepare the OpenCL environement with standard parameters.
struct cl_Configuration * clConf =
            cl_Configuration_getConfiguration();

// Prepare the container for the kernels.
struct cl_BLASFullMatrices * clBLAS =
            cl_BLASFullMatrices_init(clConf);

/**
 * Prepare the host matrices and vectors,
 * just one in this case.
**/
double * A = (double*) malloc(sizeof(double)*N*N);

// Fill A with a suitable content.
...

/**
```

```
 * Prepare the device matrices and vectors,
 * just one in this case.
**/
struct cl_FullMatrix * clA = cl_FullMatrix_init(
              A, // host memory to use
              N, // column size of the matrix
              N, // row size of the matrix
              clConf, // configuration object
              TRUE // column-major order
              );

/**
 * Start an asynchronous upload from host to device.
 * When this operation will be finished the entries
 * of clA will be equals to those of A.
**/
cl_FullMatrix_upload(clA); // object as first parameter

/**
 * Perform an asynchronous operation on the device,
 * sum "value" to all of the entries in clA
 * in this case.
 * This is the first time the add_value kernel is
 * used: it will be compiled automatically.
**/
int offset = 0;
cl_BLASFullMatrices_add_value(
              clBLAS, // object as first parameter
              clA, // FullMatrix object to use
              value, // value to sum
              offset, // starting entry (by reference)
              N, // leading dimension
              N*N ); // number of entries to modify

// Start an asynchronous download from device to host
cl_FullMatrix_download(clA, // object as first parameter
              FALSE); // false = asynchronous

/**
 * Destroy clA.
 * This waits automatically for pending operations
 * to complete.
**/
cl_FullMatrix_destroy(clA);

// Clean up the kernels and the environement
cl_BLASFullMatrices_destroy(clBLAS);
cl_Configuration_destroy(clConf);
```

```
// Clean up the memory
free(A);
```

We can see from this C version of the same code that nothing has changed, except for the way we use to call the methods.

The number of lines needed to perform each operation is the same, and a standardized naming convention have been used to ease the coding. We will investigate the naming convention later, but we can already see that to use the C wrapper does not increase the complexity nor the lenght of the program.

5.3 Interesting implementation details

Now that the goals and the design for my library have been defined, we start to look at the way I used to realize the library. Notably we can find some naming convention, the details of how some goals have been reached, and an overview of some interesting paradigms I used.

5.3.1 The double C and C++ implementation

The final implementation of the dbOpenCL library is a collection of classes and methods in C++, but they can be used as plain C structures and plain C functions. Almost every method, including the constructor and destructor, have its C function counterpart.

The library headers have preprocessor directives that let one include them as both C++ and C headers, like showed in the following example code. Notice how the example class is usable by both C and C++ code.

```
#ifdef __cplusplus

namespace ex {
    class Example {
        public:
        Example(primitive_type par1);
        virtual ~Example();
        int action(primitive_type par1);
    };
}

extern "C"
    ex::Example * ex_Example_init(primitive_type par1);
extern "C"
    void ex_Example_destroy(ex::Example * obj);
extern "C"
    primitive_type ex_Example_action(
                ex::Example * obj,
                primitive_type par1);

#else

struct ex_Example;
```

```
struct ex_Example * ex_Example_init(primitive_type par1);
void ex_Example_destroy(struct ex_Example * obj);
primitive_type ex_Example_action(
               struct ex_Example * obj,
               primitive_type par1);

#endif /* __cplusplus */
```

The Example class is defined within the namespace "ex". Every single method of the class Example is defined two times, once in the usual way for the C++ implementation and once for the C implementation. The C++ functions are defined with the keyword "extern", which indicates to the C++ compiler that the function must be compiled and the symbol must be produced in a way that permits to a C linker to link against it.

The C functions are defined with a standardized naming convention: firstly the namespace followed by an underscore, then the name of the class with its underscore, and finally the method name.

For each class we define in the same way a structure named with the previous convention. Since the C language is not an object oriented language, the easier way I could find to simulate a constructor behavior was to define a standard "init" function for each class, which takes the parameters of the class constructor and returns a pointer to a newly allocated object.

The destructor is in the same way defined as a destroy function, which accepts an object pointer.

Class methods are adapted: the first parameter of each adapter function like ex_Example_action() is a pointer to the object, and the real method parameters follow.

This is a C++ library, thus only the C++ implementation is given. The C definitions are just made in order to let the library functions be called by a C source code, like those of UMFPACK.

Let us look the source file corresponding to the previous example header, which we imagine to be saved into the "Example.h" file.

```
#include "Example.h"
namespace ex {

    Example::Example(primitive_type par1){
        // Example class code
    }
    Example::~Example() {
        // Example class code
    }
    primitive_type Example::action(primitive_type par1){
        // Example class code
    }
}

ex::Example * ex_Example_init(primitive_type par1){
    return new ex::Example(par1);
}
```

```
void ex_Example_destroy(ex::Example * obj){
   delete obj;
}
primitive_type ex_Example_action(
                  ex::Example * obj,
                  primitive_type par1){
   return obj->action(par1);
}
```

The adapter functions are just calling the C++ methods, using the passed object pointer. This is in fact a mere standardized C wrapper for a C++ class, with the peculiar fact that both the wrapper and the actual implementation are on the same files.

I would like to point the fact that such a standardized C wrapper for C++ classes could be automatized, in order to generate automatic wrappers.

We notice that it is not possible for a C code using this library to handle the actual object: just a pointers are given. This is important because it avoids completely any possible problem when passing the object to a function.

When we pass an object to a C++ function, the copy constructor is used to create the local variable in the function scope. In absence of a copy constructor the standard one is used.

The standard copy constructor would copy the object fields, mostly like the C implementation would do when passing a structure to a function.

If instead a non standard copy constructor was to use, the C implementation would not have the possibility to call it.

The use of objects instead of pointers would have other bad consequences. To let the C code pass or copy an object like if it was a structure, I would have to add in the C header at least the structure definition with its fields. This would give redundant code, increasing the possibility of errors and the maintenance cost.

5.3.2 Singletons

The singleton paradigm is used when it is undesirable to produce more than one instance of a class. Some of the entities which are implemented as classes in the library maintain their own meaning only if just one instance is created. An example of this is the Configuration object, since it represent the parameters we desire to use when running OpenCL code.

As a matter of performance the BLAS kernels are singleton too, because their compilation takes some time, therefore it is undesirable to perform it more than once for each kernel.

When using the C++ language many options are possible to create a singleton, but most of them are subtly wrong or unperformant. There are singletons that are initialized at the very beginning of the program like the following.

```
// Singleton1.h
class Singleton1 {
private:
        static Singleton1 instance;
        Singleton1();
public:
```

```
        static Singleton1 & getInstance();
}

// Singleton1.cpp
Singleton1::instance();

Singleton1::Singleton1(){};

Singleton1 & Singleton1::getInstance(){
        return Singleton1::instance;
}
```

We notice that the only constructor has been declared private, hence there is no possibility to build a Singleton1 object outside the class itself. The only way to get a valid instance of the Singleton1 class is to call the getInstance() method, which is static.

Static methods can be called on non initialized objects, because they are not really object methods: they are more class methods. The static method getInstance() will return the "instance" field, which has been defined static itself.

Static fields are shared among every instance of the class because they belong to the class itself, and not to the objects instances of that class. The problem with static fields is exactly that they belong to the class, hence they are allocated at the very beginning of a program execution, when the class is defined. This would behave undesirably when some singleton is compiled but not used.

Another possibility, more elegant, is the following.

```
// Singleton2.h
class Singleton2 {
private:
        Singleton2();
public:
        static Singleton2 & getInstance();
}

// Singleton2.cpp
Singleton2::Singleton2(){};

Singleton2 & Singleton2::getInstance(){
        static Singleton2 instance;
        return instance;
}
```

We meet another use of the keyword static. This kind of use is a little more obscure of the first two, probably because it is not so frequent.

When the static keyword is used within a method or a function it must be used in front of a variable definition. This definition must be initialized immediately.

In this case we are defining the variable "instance" inside the method getInstance(), and we are returning it immediately after. Without the static keyword this code would allocate a new variable "instance" at each time it was called. The variable would be returned and the local scope copy would be destroyed.

The static keyword instead defines a permanent local copy which will survive the exit of the method, and will be there again the next time we call it. Since at the next calls of the method getInstance() the variable "instance" has been created already, the line with the static keyword in front of it will not be executed twice.

This method, however, would once again use memory from the beginning of the program execution, because "instance" is a statically defined variable of a method. The only advantage would be that in the case Singleton2 was a class with a time consuming constructor, the constructor would be called just on the first getInstance() call, and not before. This could be the case of the Kernel operations, which have a time consuming compilation in the constructor.

These two implementations are interesting but useless in my case, because I can only return pointers to objects since this is the only way to use the C wrapper.

A much simpler singleton model is used in this case, a model which presents problems anyway.

```
// Singleton3.h
class Singleton3 {
private:
        static Singleton3 instancePointer;
        Singleton3();
public:
        static Singleton3 * getInstance();
}

// Singleton3.cpp
Singleton3::instancePointer = NULL;

Singleton3::Singleton3(){};

Singleton3 * Singleton3::getInstance(){
        if( ! instancePointer ){
                instancePointer = new Singleton3();
        }
        return instancePointer;
}
```

Once again we use a static field, but this time it is a pointer to a Singleton3 object. Since it is a pointer it is usually much smaller than the object itself, and it is initialized at a NULL value for no cost.

The first time the getInstance() method is called the pointer is tested. If it is NULL we allocate a new Singleton3 object and assign its reference to the instance pointer. The pointer itself will be returned.

This code obviously instantiates only one Singleton3 object, but it present a memory leak, which can be dangerous. In fact the newly allocated Singleton3 object would reside on the program heap until the end of the execution. This is not such a problem, because being it a singleton it would have last until the end of the program execution in any case.

Furthermore, when the execution ends the operative system claims the heap memory of the program, including the outlasting Singleton3 object.

This method can be dangerous not for the memory leak itself, but because

the destructor method was never called. Not calling the destructor method means that if the object has some resource reserved outside the heap, that resource would never be released.

In the case of OpenCL kernels this would mean that the compiled OpenCL programs would never be erased from the device memory, with a slow increase of that memory occupied space.

There is an easy way to avoid such a dangerous behavior, which is in fact the way I used to implement the kernel operations: we can use a factory/pool system.

5.3.3 Factory and pool paradigms

There are cases when an object cannot be easily instantiated. This is when factories are used: a factory is an object that is responsible for allocating new objects of other classes with standard parameters or automatically discovered tunings. A factory can be used for automatic allocations.

At the same time we use pools when a lot of new objects are needed, or when we would have lots of allocations and deallocations of the same kind of object, maybe even with the same constructor parameters.

A pool can keep track of the newly allocated objects and call the destructors when they are not needed anymore. Simple pools can avoid repeated destructions and allocations of the same object by letting the user recycle the objects.

If a factory is used by the pool we can have an automatic allocation system for a pool. Let us consider a really sophisticate factory-pool system as example of the capabilities of such a system.

Imagine to have a message driven program: it must perform different operations based on which message it receives. In a simple example we could have just one kind of message, A.

When A is received, the program performs a complex operation which needs a variable quantity of about thousands of objects of the same kind, they could be buffers or arrays, or node-objects for a graph structure, their meaning is not important. The important fact is that they are costly to allocate, or that their constructor is time consuming.

Without a factory-pool system, at each message A received the computer would have to allocate that variable but huge quantity of objects, spending some time in the allocation process. After the computation required by A is done, those objects are deallocated, and the system would be idle until the next message A is received.

With a simple pool system, when A is received for the first time all the allocation work is done, and after the work is finished the objects are put into the pool, without deallocating them. On the next A messages the program would take the already allocated objects directly from the pool, saving time and computation, hence performing faster.

A pool must always check if there is a reasonable amount of free memory for other operations: in the case there was not such a reasonable amount, the pool must deallocate some of the objects within the pool itself to free some memory.

In situations where a program works with a constantly varying amount of free memory, the pool would often be shrunk to just a few objects, hence when the message A was received we would have to allocate all of the objects all over again.

If a factory is associated to a pool, the pool itself could decide of both deallocate and allocate objects, to keep the amount of objects to a reasonable maximum. Such

a maximum is the number of objects that we need to be immediately available when the message A is received.

When the pool detects that there is not enough memory, it will deallocate some objects. When it detects that the memory is available again and that the CPU is idle it can allocate more objects to reach the reasonable maximum.

Similar factory-pools systems are used in applications where a fast time response is necessary, but much simpler versions can be used to accomplish different objectives.

In my case I had the kernel objects which have time consuming constructors. They have to be used several times in different portions of the UMFPACK code, therefore the simpler solution would have been to allocate and deallocate them each time.

This is not a desirable solution, so I opted for a singleton, which had the problem of not releasing the resources after the program execution ends.

I built a specialized pool which collects all of the kernel objects with a pointer for each kind of kernel. Such a pool is only used for recycling the same kernel object over and over, and when the kernels are not needed anymore, they are destroyed by the pool itself.

This is a factory-pool, because it is responsible for instantiating the kernels too, ensuring that just one kernel of each kind is allocated. Since the kernel objects are allocated just once, and used every time they are needed, they are used as singletons.

```
// Pool.h
class Pool {
        private:
                Kernel1 * kernel1;
                Kernel2 * kernel2;
        public:
                Pool();
                ~Pool();
                Kernel1 * getKernel1();
                Kernel2 * getKernel2();
}

// Pool.cpp
Pool::Pool() : kernel1(NULL), kernel2(NULL) {};

Pool::~Pool() {
        delete kernel1;
        delete kernel2;
}

Kernel1 * Pool::getKernel1(){
        if( ! kernel1 ){
                kernel1 = new Kernel1;
        }
        return kernel1;
}
```

```
Kernel2 * Pool::getKernel2(){
        if( ! kernel2 ){
                kernel2 = new Kernel2;
        }
        return kernel2;
}
```

We can see how the singleton-like behavior is generated by the getKernel1() and getKernel2() methods, without the need of special coding on the classes Kernel1 and Kernel2. This is because of the design of this special Pool class.

When the pool is deleted, all of the kernel objects are regularly deallocated calling their own destructor, hence freeing every related resource.

5.3.4 Automatic memory synchronization

We know that the memory objects must be synchronized carefully with some operations, like their destruction.

Since the kernels executions as well as the buffer transfers are asynchronous operations, we must be sure that they are finished before erasing the buffers involved. A buffer delete operation is synchronous, and can be done at any moment, causing the kernel that was using it to fail.

We can see the implementation details.

```
namespace cl {
   template<class T>
   class FullMatrix {
      protected:
      Event lastUse;
      bool used;
      Event lastUpload;
      bool uploaded;

      public:
      FullMatrix( /* constructor parameters */ );
      virtual ~FullMatrix();
      void upload();
      void download(bool sync);
      Event getLastUse();
      Event setLastUse(const Event &event);
      void waitUpload();
   };
}

namespace cl {
   template<class T>
   FullMatrix<T>::FullMatrix(
            /* constructor parameters */
            ) :
            lastUse(), used(false),
            lastUpload(), uploaded(false) {}
```

```
template<class T>
FullMatrix<T>::~FullMatrix() {
    if (used) {
        lastUse.wait();
    }
    if (uploaded) {
        lastUpload.wait();
    }
}

template<class T>
Event FullMatrix<T>::getLastUse() {
    return lastUse;
}

template<class T>
Event FullMatrix<T>::setLastUse(const Event &event) {
    lastUse = event;
    used = true;
    return lastUse;
}

template<class T>
void cl::FullMatrix<T>::upload() {
    if ( /* matrix contains 0 entries */ ) {
        return;
    }
    Event newevent;
    /* enqueue the upload */
    ...
    /* save the upload event into newevent */
    ...
    lastUpload = newevent;
    uploaded = true;
}

template<class T>
void cl::FullMatrix<T>::download(bool sync) {
    if ( /* matrix contains 0 entries */ ) {
        return;
    }
    Event newevent;
    /* enqueue the download */
    ...
    /* save the download event into newevent */
    ...
    lastUse = newevent;
    used = true;
```

```
    }

    template<class T>
    void FullMatrix<T>::waitUpload() {
        if (uploaded) {
            lastUpload.wait();
        }
    }
};
```

From the implementation we kept just the necessary details. As we can see from the code for the full matrix, I have a field for each kernel usage operation event: it is rewritten from every kernel operation which writes or reads on the buffer. It is overwritten at every buffer download too. Since these operations are inserted in an ordered command-queue, the last event written will be the last to be finished, therefore it is enough to save the last one to know when every event will be finished.

This is an example kernel execution code, that shows how each full matrix last usage event is overwritten with the kernel execution event. This example is a simplified part of object I use to call the kernel responsible for copying entries from a FullMatrix to another.

```
void BLAS_copy_raw::operator()(
        FullMatrix<cl_double> * A,
        cl_uint & offset_a,
        cl_uint & lda,
        FullMatrix<cl_double> * B,
        cl_uint & offset_b,
        cl_uint & ldb,
        cl_uint & len ) {

        operation_kernel.setArg(0,
                        A->getEntries_device());
        operation_kernel.setArg(1,
                        sizeof(cl_uint), &offset_a);
        operation_kernel.setArg(2,
                        sizeof(cl_uint), &lda);
        operation_kernel.setArg(3,
                        B->getEntries_device());
        operation_kernel.setArg(4,
                        sizeof(cl_uint), &offset_b);
        operation_kernel.setArg(5,
                        sizeof(cl_uint), &ldb);
        operation_kernel.setArg(6,
                        sizeof(cl_uint), &len);
        Event ev = operation_kernel.enqueueAsync(
                                                localSize,
                                                localSize);
        A->setLastUse(ev);
        B->setLastUse(ev);
```

}

We see that among the other arguments of the kernel we use both the FullMatrix A and B. The lastUse event of each of them is overwritten with the event issued of the kernel asynchronous invocation.

Notice that I used the same mechanism to save the last upload event. It is important to save the upload events as well as the kernel usage events, because if an upload operation is reading from the host memory we must ensure that until the operation is finished the host memory is not modified. Otherwise the upload will fail. Therefore we must call the waitUpload() method on the interested FullMatrix object before modifying any entry of the host matrix, to ensure that the previous upload operation is finished.

There is an observation that we can make about the lastUse event field. When the code will integrate an out-of-order scheduler, it will be necessary to save separately all of the kernel read only events in a list, because we cannot be sure about which will be finished first. The kernel writing events, instead, are forcely to be executed in order to ensure that the original algorithm is not changed, therefore just one write event will be necessarely saved.

In an out-of-order command-queue we must ensure that each input buffer has been updated before it is used. This is possible because of the design of the OpenCL library, that permits to specify prerequisites for each operation inserted in the command-queue.

The last write event of each buffer will be used as prerequisite for successive kernels that use that buffer as input. In the same way we must ensure that the buffer is not overwritten before all of the kernels that uses it as input have finished their execution.

For this reason we must use all of the saved read events as prerequisites for each kernel that will write on the buffer.

As we said, the code has been prepared for an out-of-order execution, therefore these improvements can be applied in a second time without changing the design of the library.

5.4 Examples of use in the UMFPACK code

One of the purposes of the dbOpenCL library is to allow a simple and neat integration with exitent code. Now we want to show some extract from our modified UMFPACK implementation, to demonstrate how simple the integration can be.

Our version of UMFPACK can double compile into a standard UMFPACK or an OpenCL capable one, depending on the value of the preprocessor define OPENCL. This has been useful during the code development, to quickly switch from one behaviour to another.

Therefore if one compiles this modified version of UMFPACK with a -DOPENCL option it will use the OpenCL support, otherwise it will have the normal behaviour. One must ensure that a proper OpenCL device is present and working on the system before compiling UMFPACK with the OPENCL directive.

Since the most computationally expensive part of the whole factorization is the BLAS kernel for the frontal factorization, we start showing how dbOpenCL integrates there.

```
// from the "umf_blas3_update.c" file
```

```
#ifdef OPENCL
cl_BLASFullMatrices_ger_umfpack(Work->clBLAS,
                    m, n,
                    Work->L_device,
                    Work->U_device,
                    Work->C_device,
                    d);
#else
BLAS_GER (m, n, L, U, C, d) ;
#endif /* OPENCL */
```

The precompiler directives ensure that just one of the operations is performed. To ease the work I wrote OpenCL kernels specifically fit for the UMFPACK operations, to use the exact same parameters.

Obviously all of the array pointers L, U, C representing the front are sobstituted with their respective device memory-objects. Those are L_device, U_device and C_device, which are pointers to FullMatrix objects. Since this is a plain C code, they are defined as structure pointers here.

This is a case where all of the necessary information was already on the device. Let us look at another situation.

```
// from the "umf_assemble.c" file
#ifdef OPENCL
struct cl_FullMatrix * Cols_device = NULL;
Cols_device = cl_FullMatrix_INT_vector_init (
                    Cols, // host memory pointer
                    ncols, // size will be ncols * sizeof(INT)
                    Work->clConf ); // Configuration object
cl_FullMatrix_INT_vector_upload (Cols_device);

struct cl_FullMatrix * S_device = NULL;
S_device = cl_FullMatrix_init(S, // host memory pointer
                nrows, ncols, // matrix will be nrows x ncols
                Work->clConf, // Configuration object
                CL_TRUE); // matrix in column-major order
cl_FullMatrix_upload(S_device);
#endif
```

In this file we need to upload on the device memory some information, specifically an integer array and a double precision full matrix. Both of the objects are defined as cl_FullMatrix structure pointers, because they are in fact instances of the same FullMatrix class. Only the template parameter is different in the two cases.

Vectors are supposed having a row dimension of 1. Both of the memory-objects are created specifying a host memory pointer that addresses the data which have to be mirrored. When the asynchronous upload functions/methods are called, the FullMatrix object will transfer the data pointed by that host memory pointer.

We notice also that the integer operations are named generically as INT, while we said that different operations were made for operating on 32 and 64bits integers. The standard UMFPACK implementation checks at compile time if it is working on a 32 or 64 bits architecture, and compiles differently.

I implemented the same mechanism for the OpenCL implementation, therefore if a 32bit architecture is found, all of the INT_vector operations will compile into int32_vector operations. The opposite happens otherwise. I must however say that I did not test this system on a 32bit operative system.

After the Cols_device and S_device objects have been used, we must erase them. The following code is automatically synchronized to avoid unwanted behaviors.

```
// from the "umf_assemble.c" file
#ifdef OPENCL
cl_FullMatrix_destroy(S_device);
S_device = NULL;

cl_FullMatrix_destroy(Cols_device);
Cols_device = NULL;
#endif
```

There might be situations where we have to modify host memory. If this memory location was being asynchronously copied on the device, and we were to modify it, the upload would fail. Therefore we show an example of how such a behavior is synchronized.

```
// from the "umf_assemble.c" file
#ifdef OPENCL
cl_FullMatrix_INT_vector_waitUpload (Cols_device);
#endif

// code that modifies Cols host memory
```

The waitUpload() method has to be called before any modification to the host memory is performed. This instruction waits for any Cols_device's unfinished upload to terminate. This obviously has a negative impact on the performance. Anyway it is a necessary operation to ensure predictable behaviors while allowing most of the transfer to be asynchronous.

As we see we expose a very simple interface to the final user. This will also permict an easy integration with subsequent versions of UMFPACK, when they will be released.

Chapter 6

Results

The dbOpenCL library has been developed for the integration with UMFPACK, but one of my objectives in the design phase was to ensure the reusability.

A code which can be used in different situations of the same sort is usually preferable to a specific one, because of the possibility to have a larger community involved in the debugging. Small useful libraries that are published under an Open Source license could eventually be used in different projects, hence augmenting the possibility of people interested in the smooth execution of the library.

My particular experience suggested me another immediate application for this library. The LifeV library for the resolution of finite element method problems presents some highly parallel portions of code, especially in the stiffness matrix assemble phase. In that phase the involved functions are numerically integrated and their gradient is calculated.

This is a very local but mandatory work, and it could be computed with GPGPU, with the possibility of giving better performances.

As a result of my work, I realized this reusable library, as it was in my intentions, and I used it to transfer all of the full matrix operations of the UMFPACK implementation on the device.

This does not give an out-of-the-box increase of performance, because the BLAS operations implemented in my library are not optimized. However, this is a marginal problem, because the library still was successful in its main goal: it is now easy and efficient to transfer full matrix operations on the GPU.

It is easy to implement new operations, and immediate to add the library in whatever C or C++ program. To integrate the dbOpenCL into a program it is sufficient to include the needed header files and to link against the library.

The library was compiled and conceived for a 64bit system, but I discerned some parts which are sensible to change in a 32bit environment, so that an eventual adaptation is still possible.

In the same way the library is conceived for double precision floating point operations, therefore the objects and the C wrappers are defined for just double and integers. However the objects themselves are templates, which let the user decide which primitive type to use. Other primitive types were not tested.

It could be interesting to test a single precision implementation, because some GPUs which support OpenCL does not support the double precision, but just the single one. However these GPUs are bound to disappear, as every new graphic

card supports double precision computation.

6.1 Changes on UMFPACK

In this section we want to specify exactly which parts of UMFPACK were modified to permit GPGPU. This is an important list because it permits to know when synchronization and data transfer occur. With these information we will be able to explain some of the benchmark results on the modified UMFPACK version.

The UMFPACK modified files are:

umf_version.h This file originally defines some useful preprocessor macros. It is responsible for defining some functions name, depending on the version being compiled (for complex or real numbers) and the wanted or allowed integer size (32 or 64 bit). In this file I added a control that disable the OpenCL implementation for all of the complex versions. Furthermore I added the definition for the INT_vector functions depending on the architecture. It seemed appropriate to add in this file the includes of the necessary dbOpenCL header files.

umf_internal.h This file is responsible for defining most of the structures used by UMFPACK. Pointers to the Configuration and BLASFullMatrices objects have been added in the WorkType structure, which is used in UMFPACK in a unique copy and passed as a variable through most of the functions. This variable is usually named "Work". Originally in Work there are also pointers to the host memory location representing the front, hence pointers to their device counterparts is added.

umfpack_numeric.c In this file we find "main" function for the numeric factorization, called UMFPACK_numeric(). This is responsible for setting up the Work structure and calling the underlying solve function, called UMF_kernel(). After UMF_kernel() finished it is responsible for cleaning up the Work structure. The changes to this files reflect those I made to the WorkType structure. In this file I acquire the Configuration and BLASFullMatrices singletons. Here I allocate and destroy most of the FullMatrix objects.

umf_kernel_init.c This file initializes the memory for UMF_kernel(). It also performs a matrix scale when needed. Minor changes have been applied to this file. These interest the case where the matrix structure have changed between the symbolic and numeric factorization. When this happens the initialization function must clean up the memory reserved for the matrices and return an error. This code was modified to destroy the device matrices too.

umf_kernel.c This file exposes the true algorithm of UMFPACK. Here we loop over different fronts and call the underlying functions that perform the needed calculation for each phase. This file have been modified for profiling purposes, to measure and understand which specific operation need more time. No operational modification were made to this file.

umf_start_front.c This file contains the UMF_start_front() function, which is called by the UMF_kernel() function. It estimates for each front matrix chain an appropriate size for the front matrices and allocate them. This function has been modified to allocate the same FullMatrix objects on the device.

umf_local_search.c This file contains the UMF_local_search() function, which is called by the UMF_kernel() function. It searches for a suitable row permutation grating a high quality pivot. It subsequently build an appropriate symbolic representation for this row permutation. It also moves the values from outside to inside the front, if necessary. It has been modified because the original version uses some BLAS operations (DTRSV, DGEMV) and many copy operations on front matrices. The same operations have been changed to operate on the device. In this function two integer vectors are used to represent the column and row permutations. These vectors are often modified during the function. Synchronization with waitUpload() functions was needed to ensure correctness of the execution. This is a bottleneck section.

umf_blas3_update.c This file contains the UMF_blas3_update() function, which is called by the UMF_kernel() function. It is responsible for most of the double precision computation of the UMFPACK algorithm. It works on already built front matrices and operates BLAS functions (DGER, DTRSM, DGEMM). This file was modified to use the OpenCL BLAS implementation of the same functions. A considerable speed up could be obtained by optimizing the OpenCL implementation of these BLAS operations. At the moment just an example implementation is given, which is correct but not performing.

umf_store_lu.c This file contains the UMF_store_lu() and UMF_store_lu_drop() functions, which are called by the UMF_kernel() function. They are responsible for saving the front matrix after a partial local factorization is performed. These functions rely on the UMFPACK's complex memory management system. In our modified version the front matrices must be downloaded from the device to be stored in the host memory. This operation include both a download and a synchronization. This is a bottleneck section.

umf_extend_front.c This file contains the UMF_extend_front() function, which is called by the UMF_kernel() function. As we saw in section 3.1.2 we often need to extend a front to include new rows and columns. This function operates on the full front matrices, hence each operation have been modified to include its OpenCL counterpart on the device. In this function we perform exclusively copy and set operations on the device matrices. For this purpose the number of parameters of one utility function inside this file changes automatically when the OPENCL define is set. Fortunately in this function all the data is transferred from device memory objects to other device memory objects, hence the transfer is very fast and does not represent a bottleneck.

umf_create_element.c This file contains the UMF_create_element() function, which is called by the UMF_kernel() function. When a front factorization is finished we stock the contribution matrix using UMFPACK's

memory management system. To achieve this we modified the function to download the full contribution matrix on the host memory. This is done with just one synchronized operation, which means that both a data transfer and a synchronization are performed. This is a bottleneck section.

umf_init_front.c This file contains the UMF_init_front() function, which is called by the UMF_kernel() function. Each time we finished the factorization for a front we can start working on another front. This operation recycle the old front matrices by copying the new data into them and putting to zero the unwanted portions. The OpenCL version performs these operations on the device and excludes some CPU intensive portions that would be unnecessarily executed. The data transfer is performed exclusively between different device memory objects, hence it is very fast and does not represent a bottleneck.

umf_assemble.c This file contains the UMF_assemble() and UMF_assemble_fixq() functions, which are called by the UMF_kernel() function. This delicate section performs the extended add operation of previously calculated contribution matrices, like described in section 3.1.2. These are very complicated functions, with both sparse and full matrix operations. Many uploads are necessary because the old contribution blocks are stocked in the UMFPACK's memory management system, but they must be assembled into the front matrices on the device. In these functions I had to allocate temporary FullMatrix objects, which are automatically synchronized before they are destroyed. Therefore these functions have many uploads and synchronizations between host and device. This is a bottleneck section.

umf_scale_column.c This file contains the UMF_scale_column() function, which is called by the UMF_kernel() function. This is a memory transfer intensive function which represents a major bottleneck in the classic UMFPACK version. It scales the pivot column and moves the correspondent column and row. The OpenCL version performs the same operations on the device FullMatrix objects. Since the data transfer is performed between device memory objects, it is very fast and does not represent a bottleneck.

umf_grow_front.c This file implements part of the UMFPACK's memory management system. It may happen that when the front is extended the allocated memory for it must be enlarged. When this happens UMFPACK tries to extend it if possible, otherwise it copies it on a larger consecutive memory portion. In the OpenCL implementation we cannot extend the memory objects, hence we simply download every FullMatrix, destroy it, create it larger, and upload it. Due to the original UMFPACK implementation, this behavior is given by the interaction of 3 files: umf_grow_front.c, umf_get_memory.c, umf_garbage_collection.c. This behavior can and should be optimized by creating the larger FullMatrix objects before destroying the original one. This way we could avoid some expensive host-device data transfer. This behavior uses higher memory peaks, hence it involves more tests. In case of failure the current implementation could be used as fall back. The file umf_grow_front.c is specifically used to enlarge the contribution full matrix. This is a bottleneck section.

umf_get_memory.c This file implements part of the UMFPACK's memory management system. In the modified version all of the front matrices are downloaded on the host memory in this file.

umf_garbage_collection.c This file implements part of the UMFPACK's memory management system. In the modified version all of the front matrices are uploaded back on the device memory in this file.

6.2 Benchmarks and results analysis

Profiling an OpenCL application is different from profiling classic programs. This is due to the fact that most of the computing is performed on the device. A normal profiler will assume that the whole work is done on CPU, using CPU time.

When profiling a GPGPU application, instead, we must consider transfer times and waiting times too, because they are a non negligible part of the whole computation time. Most of the times when a normal application waits it is for user or network input output purpose. In GPGPU instead the program waits for the GPU to finish a computation, therefore the time spent waiting is really used for computation after all.

For this reason I instrumented the umf_kernel.c file to give me precise information about how much time each UMFPACK function is using. The strategy is to measure the wall-clock time used to execute each function called by UMF_kernel(). The test was performed launching the execution on a computer while no other program was executing, except the operative system. Obviously the data collected is not precise, but it is nevertheless a good hint about what is happening in the OpenCL version of UMFPACK.

- 69.65% front factorization with OpenCL example BLAS (umf_blas3_update.c)
- 17.72% frontal matrices assembly (umf_assemble.c)
- 4.16% creation of storage units for contribution matrices (umf_create_element.c)
- 3.74% local search for pivot rows, for numerical stability (umf_local_search.c)
- 2.20% storage of frontal LU matrices (umf_store_lu.c)
- 1.79% columns rescale (umf_scale_column.c)
- 0.58% frontal matrices growth (umf_extend_front.c)
- 0.16% initialization and rescaling of the matrix (umf_init_front.c)

The actual execution time is much worse than the standard UMFPACK version, this is because the OpenCL version uses an example BLAS implementation. Instead, the standard UMFPACK runs with a fully optimized Goto2BLAS, which is often considered the fastest existing BLAS implementation.

What we want to analyze is the presence or not of the various bottleneck factors, like slow data transfer between host and device or synchronization. Most of the time spent on synchronization or data transfer is fixed and cannot be optimized. If it represents a large part of the whole computation time, there are few possibilities that a fully optimized OpenCL code could achieve good performances.

If instead most of the time is spent in the actual double precision calculations, we can hopefully reach good to great performances by implementing a better version of the BLAS operations. We will now analyze which percentage of the work is spent in pure computation and which instead is spent in transfer and synchronization. This is done by using the description of the OpenCL modifications in each file, as shown in section 6.1.

- The BLAS updates (umf_blas3_update.c) consist of three BLAS operations and nothing more. No transfer or synchronization is performed.

- The assembly section (umf_assemble.c) consists of many transfer and synchronized operations. Possibly it could be faster if the contribution matrices from previous fronts were stocked on the device memory instead that on the host memory.

- The creation of new storage units (umf_create_element.c) consists of a single matrix download, often repeated. Surely it would be faster if the contribution matrices were stocked on the device memory instead that on the host memory.

- The pivot search (umf_local_search.c) consists of many integer vector updates, transfer and synchronization. This part will hardly run much faster. A possibility for a faster execution is to pool the necessary temporary objects, because most of the time is spent in waiting the end of a kernel execution before reusing the same temporary FullMatrix again. This function could benefit from many temporary FullMatrix objects available in a pool.

About 70% of the time is spent in BLAS operations, hence an optimization of those kernels would bring considerable performance improvements. This is an important result. It means that the UMFPACK algorithm is well suited for running on OpenCL devices. The most computationally expensive parts are those that need the less communication and synchronization, which is a desirable quality for an algorithm.

As we said, the execution time of our modified UMFPACK is much worse than the classic one. We supposed that the reason for this was the unoptimized OpenCL BLAS, which is probable but not certain. To complete our analysis we must be sure that the additional execution time is not given by the GPGPU overhead.

In fact it may happen that for short executions the OpenCL version uses more CPU time that the normal version. This problem can arise mainly because of the OpenCL programs compilation time, but a small overhead exists in each other OpenCL instruction. Fortunately this problem is easy to discover simply by profiling the execution.

We are trying to find out if the OpenCL version uses more CPU resources than the classic version, hence a CPU profiler like Valgrind is all that we need. In the chart (6.1) we resumed the result. Notice that the cost scale is in number of instruction read, hence it is only a rough estimate of the real CPU time used by each function.

The "numeric" bar represents the total CPU cost, and it is approximately the sum of all of the others. The total CPU cost for the numeric factorization step performed with the OpenCL version is about a half (51.44%) of the classic one.

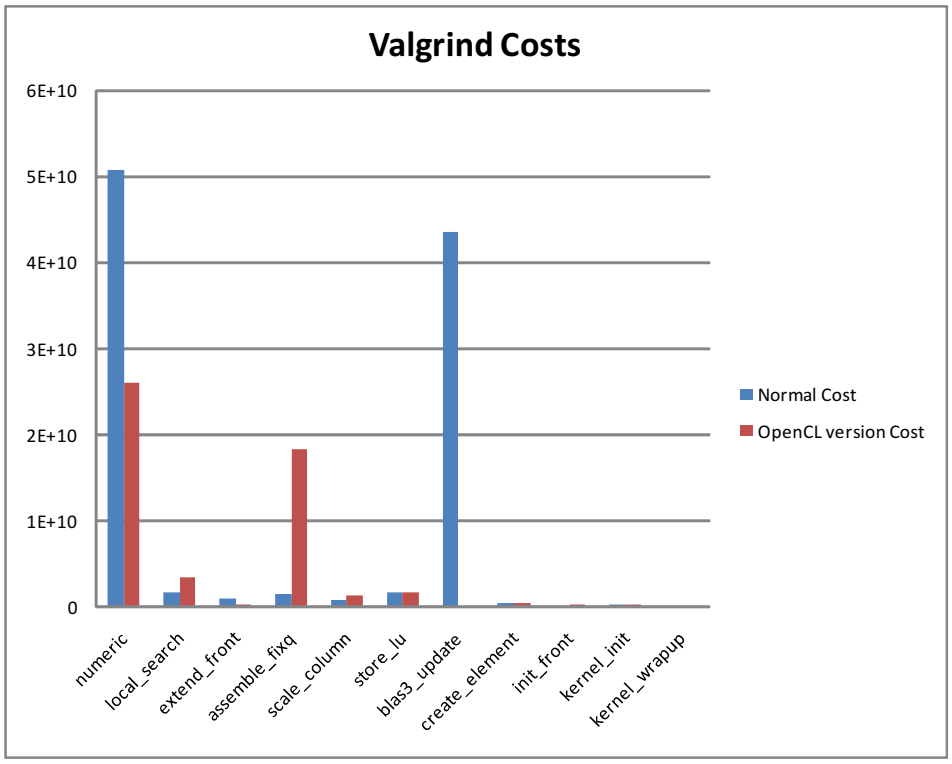

Figure 6.1: CPU cost comparison between classic UMFPACK and OpenCL version

This fact alone means that we did not introduce too much GPGPU overhead. Most of the computation has been successfully moved from CPU to GPU.

The "blas3_update" bar, which was 85.70% of the total CPU cost for the classic UMFPACK version, has become just a fraction (0.34%) in the OpenCL version. This is the most successful change to the UMFPACK behavior. Our purpose was to move the BLAS operations on GPU, hence this chart shows the best result of this project.

As we suspected before, the "assemble_fixq" column reveals to be the worse bottleneck, which is both a GPGPU and a CPU problem. The assemble() function for the OpenCL version of UMFPACK uses alone the 17.72% of the wall-clock time and 70.34% of the CPU cost. Every effort should go in improving the execution of the assemble() function, after a proper OpenCL BLAS implementation has been realized. We already suggested how a different memory management system could improve the performance of this method.

6.3 Future developments

This version of UMFPACK can be further improved in many ways. The most effective would be the implementation of fully optimized BLAS operations. Such an improvement alone would probably reduce the execution time at about a half or less, as we showed with our previous benchmark.

As we stated in section 5.1.2, the dbOpenCL library uses now an ordered scheduler because of some bugs in the OpenCL implementation. It will be a relatively easy task to adapt the actual structure to use an out-of-order scheduler for the OpenCL commands. This could bring some performance improvement, even if modest.

Rewrite the memory management system of UMFPACK would be a much harder task, but it would free a lot of host memory that now is useless. In fact the OpenCL version of UMFPACK seldom use the host memory for the front matrices. This memory is allocated and managed by UMFPACK as if the normal version was in use.

At the same time we could avoid stocking the old front contributions on the host memory. Instead we could keep them on the device memory, saving about 20% of the time. This can be done if the OpenCL device has enough memory, which is often the case when using dedicated GPGPU servers. This improvement would sum with the BLAS optimization to give a code up to 90% faster.

Another possible improvement is to let the user chose the OpenCL device. In fact if multiple devices are available the dbOpenCL library chooses a random one. A more refined version could test the features of all the available devices and use the best for our purpose. An ultimate implementation would use them all at once. However this would require much more time and human resources to achieve.

Conclusion

This work had as main purpose the one of producing an UMFPACK implementation which exploited GPUs using the OpenCL language. As a desirable side effect we could have expected an out of the box performance improvement, which we had not. It is true that GPUs have now higher FLOPS (floating point operations per second) than CPUs, but the standard UMFPACK implementation relies on a highly optimized BLAS, which uses the CPU in a very efficient way. However the profiling sessions showed that the modified implementation does not introduce excessive overheads. Hence an optimized OpenCL BLAS would probably bring the desired performance improvements.

The dbOpenCL library has been developed to allow a neater integration of OpenCL code into the existing UMFPACK source files. This library has a value of its own, and could be used again in different projects, to achieve a fast integration of OpenCL into existing applications. The library has been coded for both performances and ease of use, including different OpenCL example programs. This library could be improved with an optimized BLAS implementation and other performance modifications.

While I was developing the library I encountered many problems with the Nvidia OpenCL implementation, but most of them were corrected on later releases. This gave me the possibility to follow the improvements of the OpenCL support of one of the major GPU producers. Because of that we can say that at the moment of writing the OpenCL technology on the Nvidia architecture has entered a stable phase.

Finally we conclude that with further work the modified version of UMFPACK could use less memory and less time to achieve the same result, through the use of the graphic card memory to store the temporary matrices.

Ringraziamenti

La mia laurea è in buona parte dovuta ad Annica, che ho conosciuto in università al primo semestre del primo anno e che mi ha accompagnato lungo tutto il periodo universitario. Il nostro non è stato un percorso facile, ma la sua presenza e la sua costante dedizione sono stati aiuti insostituibili, sia dal punto di vista didattico che da quello morale e psicologico.

Il secondo rigraziamento va alla mia famiglia, non tanto per l'aiuto durante gli anni dell'università, quanto per la fiducia che mi hanno riservato. Questa indipendenza e fiducia, che tanto desideravo prima di approdare all'università, é arrivata incondizionatamente quando sono partito due anni per studiare a Lille, e da allora ho continuato a goderne.

Desidero ringraziare anche i miei amici, in special modo Luca (detto "il Monte"), che da tanti anni mi mette allegria, e col quale spero di passare ancora molto tempo a progettare attività improbabili e passatempi interessanti. Senza l'interesse da parte dei miei amici nelle mie attività informatiche probabilmente non avrei avuto l'interesse necessario per continuare, giorno dopo giorno, i miei studi paralleli di programmazione e amministratore di sistemi. Sicuramente non sarebbe iniziata la mia mania per la computazione parallela, che è direttamente sfociata in questa tesi.

Infine voglio ringraziare alcuni professori del dipartimento di Matematica, quelli che hanno saputo interessarmi costantemente. Non è un impresa facile spiegare con convinzione e passione alcune materie, specialmente le più astratte. Ringrazio quindi vivamente questi professori che nonostante insegnino da molti anni non hanno perso la passione per quello che fanno. Esempio di questi è il prof. Cercignani, del quale ho avuto l'onore di frequentare l'ultimo corso, prima che venisse a mancare. Ci vuole molto spirito per spiegare con quella passione e in quella condizione fisica a studenti dall'aria spersa e interrogativa.

A tutte queste persone, grazie.

Bibliography

[1] Joseph W. H. Liu, "The Multifrontal Method for Sparse Matrix Solution: Theory and Practice", SIAM Review, Vol. 34, No. 1 (Mar., 1992), pp. 82-109

[2] Harry M. Markowitz, "The Elimination Form of the Inverse and Its Application to Linear Programming", Management Science, Vol. 3, No. 3 (Apr., 1957), pp. 255-269

[3] Alfio Quarteroni, Riccardo Sacco, Fausto Saleri, "Matematica Numerica 2a edizione", Springer-Verlag Italia, Milano, 2000

[4] Alfio Quarteroni, "Modellistica Numerica per Problemi Differenziali 4a edizione", Springer-Verlag Italia, Milano, 2008

[5] Various authors, "OpenCL Programming Guide for the CUDA Architecture Version 3.1", Nvidia, 2010