UNIVERSITAT POLITÈCNICA DE VALÈNCIA

Department of Mechanical and Materials Engineering

Master Thesis

Optimization of a finite element code implemented in MATLAB®. On the use of GPUs for High Performance Computing.

Presented by: D. José Manuel Navarro Jiménez

Supervised by: Dr. D. Juan José Ródenas García

Valencia, July, 2014

Master Thesis

Optimization of a finite element code implemented in MATLAB®. On the use of GPUs for High Performance Computing.

for the degree of

Master of Science in Mechanical and Materials Engineering

presented by

D. José Manuel Navarro Jiménez

at the

Department of Mechanical and Materials Engineering

at Universitat Politècnica de València

Supervised by

Dr. D. Juan José Ródenas García

Valencia, July, 2014

Abstract

The Department of Mechanical and Materials Engineering has developed a 2D Finite Element code based on geometry independent Cartesian grids (cgFEM) capable of solving shape optimization problems as well as making patient-specific analyses using medical images. A similar code in 3D (FEAVox) is currently under development. Both codes are implemented in MATLAB®, a simple and intuitive programming language but with a higher computational cost than compiled languages such as C++ or FORTRAN.

The objective of this Thesis is to develop programming procedures to improve the performance of the existing and the currently under development software. Among other optimization techniques this Thesis will focus on the use of Graphics Processing Units (GPU) for high performance computing.

The use of these techniques has led to a software that, despite being implemented with MATLAB®, improves the computational efficiency of commercial software which is developed using compiled programming languages.

Resumen

El Departamento de Ingeniería Mecánica y de Materiales ha desarrollado un código de Elementos Finitos 2D basado en mallados Cartesianos independientes de la geometría (cgFEM) capaz de resolver problemas de optimización topológica y de realizar análisis específicos de paciente a partir de imágenes médicas. Se está desarrollando actualmente un código similar 3D (FEAVox). Ambos códigos están implementados en MATLAB®, un lenguaje de programación sencillo e intuitivo pero menos eficiente computacionalmente que otros lenguajes compilados como C++ o FORTRAN.

El objetivo de este Trabajo Fin de Máster es desarrollar procedimientos de programación que permitan aumentar el rendimiento computacional del software que ha sido o está siendo desarrollado en el Departamento. De entre las técnicas de optimización disponibles, se hará hincapié en el uso de tarjetas gráficas (GPU) como medio de computación de alto rendimiento.

La utilización de estas técnicas ha permitido obtener un software de EF que, pese a estar implementado en MATLAB®, mejora el rendimiento computacional de software comercial desarrollado con lenguajes de programación compilados.

Resum

El Departament d'Enginyeria Mecànica i de Materials ha desenvolupat un codi d'Elements Finits 2D basat en mallats Cartesians independents de la geometria (cgFEM) capaç de resoldre problemes d'optimització topològica i de realitzar anàlisis específics de pacient a partir d'imatges mèdiques. Actualment s'està treballant en un codi similar 3D (FEAVox). Ambdós codis estan implementats en MATLAB®, un llenguatge de programació senzill i intuitiu però menys eficient computacionalment que altres llenguatges compil·lats com C++ o FORTRAN.

Aquest Treball Fi de Màster té com a objectiu desenvolupar procediments de programació que permeten millorar el rendiment computacional del software que ha sigut o està sent desenvolupat al Departament. De les tècniques d'optimització disponibles, aquest Treball es centrarà en l'utilització de targetes gràfiques (GPU) com a mitjà de computació d'alt rendiment.

L'ús d'aquestes tècniques ha permés obtindre un software d'EF que, a pesar d'estar implementat en MATLAB®, millora el rendiment computacional del software comercial elaborat amb llenguatges de programació compil·lats.

Agradecimientos

En primer lugar quisiera dar las gracias a mi director, Juanjo, por las horas que ha dedicación que le brinda no solamente a mi, sino a todos los estudiantes que están bajo su dirección, que no son pocos. Quisiera extender este agradecimiento al profesor Manuel Tur y al resto de profesores del Departamento por los consejos que me han proporcionado durante este tiempo.

Es un placer trabajar en la sala de becarios del DIMM. Hay personas que se van temporalmente (¡Eva y Camila, volved pronto!), otras que se van de manera casi definitiva (¡te echamos de menos Enrique!) y nuevos compañeros, pero el buen ambiente permanece siempre en el grupo de trabajo. Requieren una especial mención *el chache* Onofre (un *blowminder* de la mecánica computacional) y mi compañero de fatigas del Máster, Santi.

Por último, pero no menos importante, a mi familia por ayudarme siempre. A mis padres, a María, a Isabel. Porque se visten de secretarios, de traductora, de editora... todo con tal de hacerme el trabajo más llevadero.

A todos vosotros, muchas gracias.

Contents

Chapter 1

Introduction

The Finite Element Method (FEM) has become an important tool for the industry. Nowadays less prototypes have to be manufactured while launching a new product because multiple simulation tools are available for designers. For example, with the combination between optimization algorithms with FEM it is possible to achieve an optimal shape of a component under given boundary conditions. This means reducing the amount of material employed in manufacturing the component, and the corresponding product cost reduction.

FEM is also starting to be applied in scientific fields that are not related to the classical manufacturing industries, like medical science. For example, surgical teaching paradigm is undergoing a major change with the appearance of surgery simulators. While years ago surgeon students learned how operate on patients by the "*watch and learn*" paradigm, nowadays they can try to operate on a simulator without compromising a real patient's health. FEM analysis can be performed in these simulators in order to give a physical feedback through a joystick.

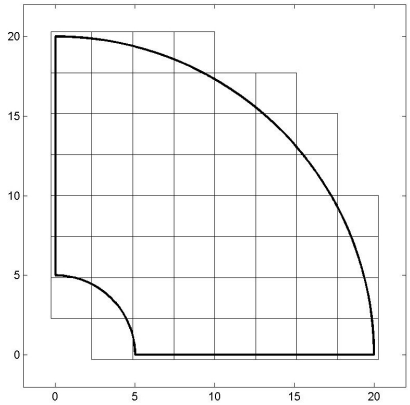
However, the main difficulty to overcome in all these cases is the high computational cost of the numerical analysis. Even with High Performance Computing (HPC) workstations some analyses with a huge amount of degrees of freedom (DOFs) can take hours to be solved. In other cases, like real time simulations, the available time to solve each problem is not enough to reach a sufficiently accurate solution.

1.1 cgFEM

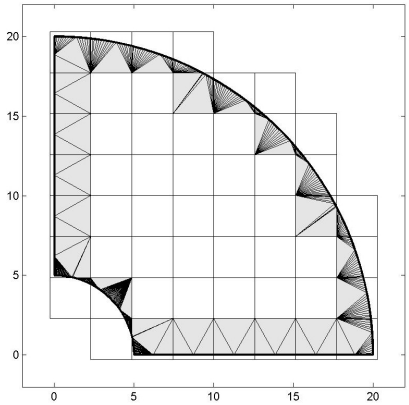
As FEM provides the industry with several benefits, a lot of resources have been invested in order to improve all aspects of the FEM, and specially those regarding computational cost. Following this research line, the approach made by the Department of Mechanical and Material Engineering of the Universitat Politécnica de Valencia has been exploring the possibilities and limits of geometry independent Cartesian grids, developing a 2D FEM code called cgFEM [9], [7].

The FE mesh is usually constructed in 2D by the division of the real domain into a group of linear or curved, triangular and/or quadrilateral subdomains that have no overlapping between them. cgFEM instead uses two different meshes (Fig. 1.1). The mesh used for the FEM approximation is called the approximation mesh. The only requirement of this mesh is that it has to cover all the problem's domain (Fig. 1.1a). The second one is called the integration mesh. All the numerical integrations are performed on this mesh. The integration mesh is created by dividing the elements of the approximation mesh that are intersected by the domain into integration subdomains. A Delaunay triangulation is performed using the internal nodes and some intersection points of the boundary (Fig. 1.1b and 1.1c). Then, cgFEM implicitly takes

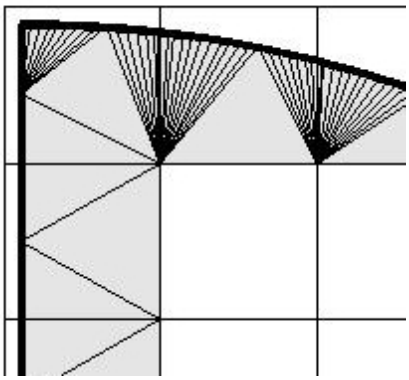
into account the geometry of the problem's domain through the numerical integration process.



(a) Approximation mesh.



(b) Integration mesh.



(c) Detail of the integration mesh.

Figure 1.1: Different meshes used in cgFEM.

Internal elements, those fully located in the interior of the domain, are treated as standard FEM elements, whose integrals are evaluated using a Gauss quadra-

ture for quadrilaterals common for all elements. Boundary elements are integrated using a triangle Gauss quadrature in all of the subdomains created with the Delaunay triangulation, as shown in Figure 1.2.

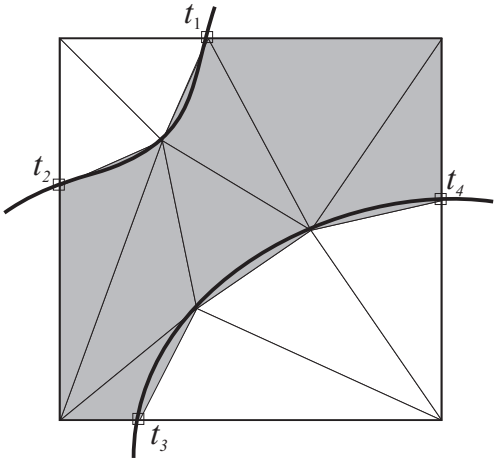
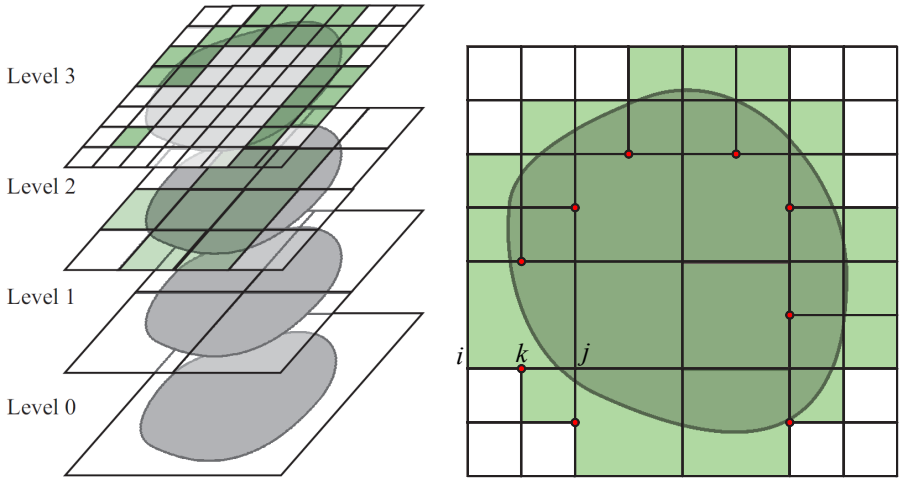


Figure 1.2: Integration subdomain generation.

cgFEM uses a set of Cartesian grids hierarchically structured that allows for a great computational cost reduction. Each mesh is composed by regular quadrilaterals. The 0-level mesh includes one single element that covers the whole calculation domain. Next levels are created by splitting the previous level elements into 4 new elements. Therefore, level 1 will have 4 elements, level 2 will have 16 elements, and so forth. Generally, the n-level mesh will have 2^{2n} elements. Due to the hierarchical structure of the Cartesian grid, it is trivial to calculate geometrical properties of each element like coordinates, topology or size. Then, it is not mandatory to store all this information, thus allowing for a considerable reduction of RAM memory usage.

Figure 1.3 shows an example of the hierarchical mesh structure in cgFEM. The calculation domain is initially superimposed on the Cartesian grid pile (Fig. 1.3a). Then the approximation mesh that cgFEM will use for solving



(a) Cartesian grid pile.

(b) Analysis mesh. ∘ nodes with multi-point constrais for C^0 continuity.

Figure 1.3: Difference between the Cartesian grids pile and the analysis mesh.

the problem is a combination of elements from each of the Cartesian grid levels so that the final mesh (also known as calculation mesh) includes the whole calculation domain without overlapping elements (Fig. 1.3b).

All analyses start with a uniform mesh. There are three different refinement options available in cgFEM. The first option is a geometrical refinement that adapts the mesh depending on the domain's curvature. There is also a *h*-adaptive refinement based on the numerical solution's precision in each part of the domain evaluated by means of error estimation techniques. The last option is onlyfor the analysis of a medical image. The refinement is then performed based on statistical values related to the gray-level distribution of the image's pixels within each element [2].

It is worth to highlight that when different Cartesian grids are combined, there are some nodes belonging to the smaller elements that are located on the side

of other bigger element (see Fig. 1.3b). These nodes are called *hanging nodes*. In order to ensure C^0 continuity along these sides, a relation between hanging nodes' displacements and displacements of the nodes located on the same side of the adjacent element has to be added to the equation system. These constraining equations are called *Multi-Point Constrains* (MPC), thus, this hanging nodes are also called MPC nodes.

cgFEM has been used by the research group in the Department as the basic FEM code for developing new features like a displacement-based error estimator (SPR-CD, [7]), an error estimator based on quantities of interest [8] or a new efficient nested solver [11]. cgFEM is also able to directly create FEM models using medical images and mix them with geometrical entities that simulate surgical implants [2], [12].

According to [1], a study at Sandia National Laboratories (USA) revealed that the generation of the finite element numerical model, this is, creating a geometry suitable for a FEM analysis and generating a proper calculation mesh, uses 80% of the total time spent on the analysis, whereas only 20% is devoted to the numerical analysis. The Cartesian elements together with the hierarchical structure make cgFEM a highly efficient methodology in terms of computational cost, as generating h-adapted meshes even for complex geometries is a simple task in this case.

1.2 FEAVox

A 2D FEM program is of little use for industrial applications, as almost all real problems are 3D problems. Therefore, after several years acquiring experience and knowledge developing the 2D code cgFEM, the Department has decided to take the next step by implementing a new FEM code in 3D named *FEAVox*.

The aim is to use all this know-how to implement in 3D all the features existing in cgFEM.

It is thought that the computational cost of the analysis made in cgFEM can be reduced by optimizing the code. As the chosen programming language, MATLAB®, is not a compiled language, one trivial optimization decision could be translating all the code to a compiled language such as C or C++. However, this change is not likely to happen in the short term due to two main reasons. First, MATLAB® is an excellent programming language for the fast implementation of new algorithms. There are plenty of very efficient high level tools for numerical analysis that allows the programmer to concentrate in the algorithm itself rather than in the coding formalisms of other programming languages. Second, most of the students that helped in the development of cgFEM have a good background in mechanical engineering, extremely useful for the development of the code, but a very limited experience on C, C++ or FORTRAN.

In this Thesis, MATLAB® special features for code performance improvement are studied and then applied to cgFEM code. Among all optimization techniques available it is worth to outline the usage of Graphics Processing Units (GPU) for parallel computation, as it is a relatively new technique with lots of possibilities in the field of numerical computation.

The research made in this Thesis is intended to provide the Department with the technical *know-how* on GPU computing as well as other optimization procedures for MATLAB® programming language. As the new 3D code implementation is currently in progress, the conclusions obtained in this Thesis will set the programming basis for an optimal implementation of FEAVox in terms of computational cost.

Chapter 2

Optimizing code using MATLAB®

The performance enhancement can be achieved by acting on different levels. All computer applications have two main areas in which one can make modifications, hardware and software. It is trivial that an upgrade on the workstation's components can produce a performance improvement. A faster CPU provides faster calculations and more RAM memory allows to perform bigger analysis in terms of DOF avoiding the use of the hard disk drive.

The first improvement regarding the software is the programming language. MATLAB® language needs no compiler, as it works as an interpreted language. This is, each time a code line is executed, MATLAB® *"translates"* the line into machine code and then the instruction is executed. This particular feature of interpreted languages make them slower than other compiled codes like C, C++ or FORTRAN, specially when programs grow bigger and more complex. For this reason, MATLAB® is often used to create simple programs that solve

small problems, and when it comes to High Performance Computing (HPC) compiled programming languages like C++ are used.

As said in the MathWorks® overview web page [4], *"MATLAB® is a high-level language and interactive environment for numerical computation, visualization, and programming."*. MATLAB® is able to deal with matrices and arrays and operate with them in a highly effective way (actually the name stands for ***Mat****rix* ***Lab****oratory*). This feature makes this language very suitable for numerical computation. Actually, the usage of MATLAB® is widespread among diverse engineering and science fields such as mechanical engineering, biology, signal and image processing, control systems and so on.

Despite having that handicap in comparison with complied languages, users can reach a reasonable performance of MATLAB® code by taking advantage of the special features like high efficiency in handling arrays and parallel computation. In this chapter a series of techniques and optimization procedures are given in order to improve any MATLAB®-based program's performance.

It is worth to outline that the first way to improve the efficiency of a computer program should be changing the main "*philosophy*" on which the code is based. For example, the usage of Cartesian grids in FEM represents an important philosophy change with respect to the classical FEM implementation. This different point of view led to the construction of a highly effective data structure that provides with multiple possibilities to enhance the code's performance.

After the programming language and the main "*philosophy*" are established it is important to check if the algorithms used to accomplish the function's objectives are optimal or could be improved. This will be explained in section 2.1. Once the proper algorithm is selected it is time to take advantage of the chosen programming language. In this case, the benefits of "*vectorizing*" with

MATLAB® are explained in section 2.2. The last optimizing technique that this Thesis covers is a mix between a hardware and a software improvement, this is, the usage of GPU for parallel computation. Finally, section 2.4 shows a basic example comparing all optimization techniques.

2.1 Algorithm change

Usually when working with big complex programs it is not trivial to find an algorithm that fits current function's purpose. In addition, it is very common that the first algorithm designed is inefficient, so it has to be improved in a second step. And even when the algorithm has been optimized one could still find a different algorithm that could be much more efficient than the first one.

An example of algorithm change that leads to a considerable performance improvement is shown in [10]. The first objective of that project was to improve the refinement routine of cgFEM, as it was the most important bottleneck in the program (the refinement process consumed up to 85% of the whole computation time in some large problems).

The old refinement procedure is shown in Algorithm 2.1.1. The input variable, `SubN`, contains the number of all the elements to refine and the number of times that it has to be split. `Split2D` algorithm only splits one element at a time, so the algorithm has to be run each time for each element times the number of times to be split.

Algorithm 2.1.1: SPLIT2D(*Element*)

Active ←Active elements in the calculation mesh
Neighb ←neighbors of *Element* of the same mesh level
Parent ←parent elements of *Neighb*

Step 1: Check if active neighbors of *Element* must be refined
for $iSide \leftarrow 1$ **to** 4

$$\textbf{do}\ \begin{cases} \textbf{if}\ \exists Neighb[iSide] \\ \quad \textbf{then}\ \begin{cases} iNeighb \leftarrow Neighb[iSide] \\ \textbf{if}\ (Active[iNeighb] = \textbf{false}\ ) \\ \ \textbf{and}\ (Active[Parent[iNeighb]] = \textbf{true}\ ) \\ \ \ \textbf{then}\ \text{SPLIT2D}(Parent[iNeighb]) \end{cases} \end{cases}$$

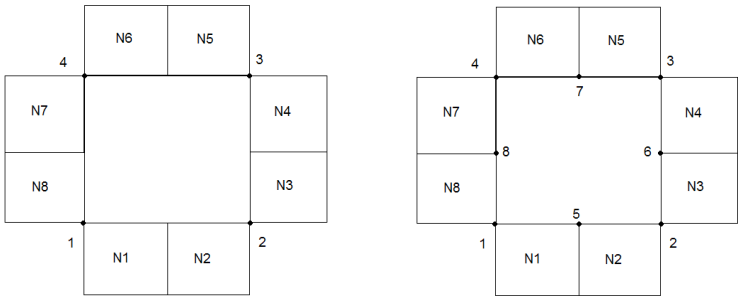
Step 2: Split current *Element* and activate new elements
Children ←Children elements of *Element*
Active[*Element*] ← **false**
for $i \leftarrow 1$ **to** 4
do *Active*[*Children*[*i*]] ← **true**

∗This algorithm is executed once for each element and refinement level

The main disadvantage of this algorithm was the recursive structure that it had to check if neighbor elements of the element to refine had to be also refined in order to obtain a suitable mesh. Also the splitting task was a sequential process for all elements, dramatically increasing the computation time for fine meshes. Moreover, cgFEM only stored the information of neighbor elements in the hierarchical structured meshes, so to find which were the neighbor elements of a given one in the calculation mesh several calls to *sparse* matrices and logical conditions were used. Although MATLAB® *sparse* matrices have

several features to enhance memory saving and accelerate some matrix operations, reading a single value of a big *sparse* matrix is likely to be slow.

The new algorithm proposed as solution for the problem consisted of two main blocks. The first block obtained the mesh neighbor elements of any set of active elements in an efficient way. This function is called `FindRealNeighbors`. Figure 2.1 shows all possible neighbors of a given element and the chosen arbitrary numeration of them.



(a) Linear element (b) Quadratic element

Figure 2.1: Relative position between an element and its neighbors in the mesh.

It can be noticed that there are up to two possible neighbors at each side of the element. This is because the level difference of two contiguous elements is forced to be not greater than one, and it will be the only requirement to refine an element. The initial mesh in cgFEM is composed by elements of uniform size. As the refinement routine will enforce this maximum level difference, all the following meshes will have it too. Also it is remarkable to say that if a neighbor element is greater than the given one, it should occupy two neighbor positions in the neighbor's vector.

cgFEM includes a function that returns all the elements attached to a node of the calculation mesh, i.e., nodal connectivity. The indexation of the elements is shown in Figure 2.2. Getting the nodal connectivity of all the element's nodes and mixing both indexations in Figures 2.1 & 2.2 we can obtain a relationship between neighbor position and value in nodal connectivity (Table 2.1), then gathering and reordering the information in nodal connectivity the neighbor elements vector is obtained.

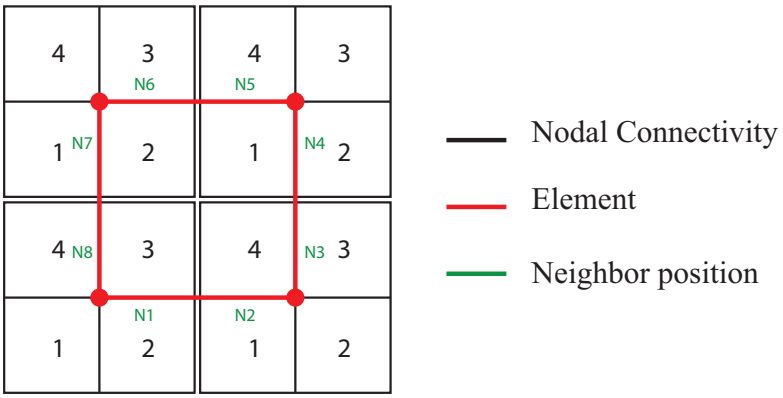
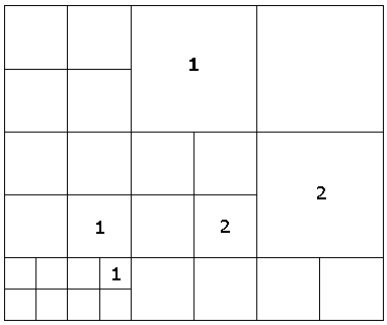


Figure 2.2: Nodal connectivity and neighbor indexation.

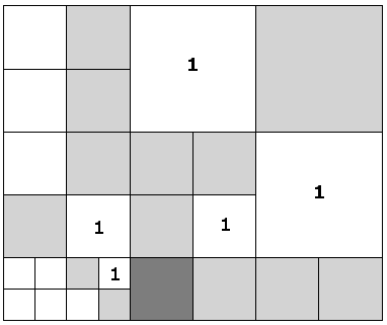
Neighbor position	1	2	3	4	5	6	7	8
Vertex node	1	2	2	3	3	4	4	1
Nodal connectivity	2	1	3	2	4	3	1	4

Table 2.1: Relationship between neighbor element and nodal connectivity index.

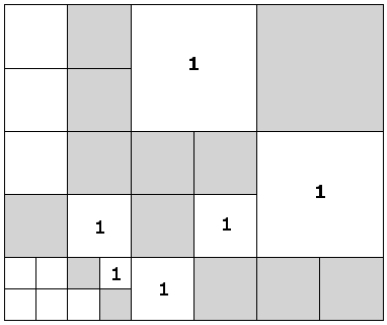
The second block of the new algorithm performs the refinement of the mesh. The input to this function is the same as the old routine, `SubN`. In order to provide a better understanding of the new refinement procedure, Figure 2.3 contains a graphical example of a random mesh refinement process.



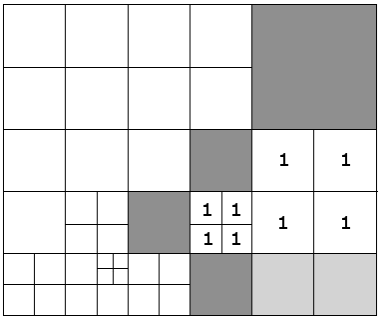
(a) Initial mesh and required refinement. Each iteration refines the mesh one level.



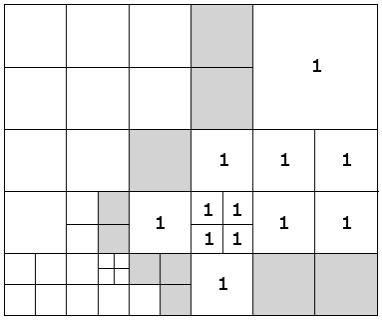
(b) First refinement, only one level. Dark gray element has to be refined as well.



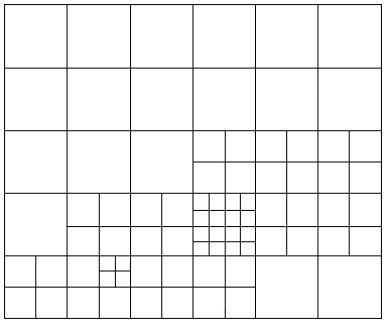
(c) The mesh requirement is fulfilled. Elements with 1 are split.



(d) Second iteration. Dark gray elements have to be refined as well.



(e) The mesh requirement is fulfilled. Elements with 1 are split.



(f) Final mesh. Only two iterations were needed.

Figure 2.3: Mesh refinement example

As each element contained in `SubN` might need a different number of refinements (Figure 2.3a), the new refinement algorithm will be performed in an iterative loop. Each iteration will refine all elements that need to be split at least once, together with all neighbor elements that must be split as well in order to fulfill the mesh level requirement (Figures 2.3b & 2.3c). After that, `SubN` vector is updated (Figure 2.3d) and the algorithm is executed again (Figure 2.3e) until there are no elements remaining in `SubN`, so the mesh is successfully refined (Figure 2.3f). The algorithm used to refine the mesh each iteration is shown in Algorithm 2.1.2.

Algorithm 2.1.2: CREATENEWMESH(*Elements*)

Active ←Active elements in the calculation mesh
Neighb ← FINDREALNEIGHBORS(*Elements*)
Finish ← **false**

Step 1: Add all additional elements that need refinement
while *Finish* = **false**
do
- *Level* ←Level of *Elements*
- *NeighbLevel* ←Level of *Neighb*
- *NewElements* ←Neighbor elements where *NeighbLevel* < *Level*
- **if** *NewElements* is not empty
  - **then**
    - *NewNeighb* ← FINDREALNEIGHBORS(*NewElements*)
    - *Elements* ← *Elements* ∪ *NewElements*
    - *Neighb* ← *Neigb* ∪ *NewNeighb*
  - **else** *Finish* ← **true**

Step 2: Split current *Elements* and activate new elements
Children ←Children elements of *Elements*
Active[*Elements*] ← **false**
Active[*Children*] ← **true**

∗This algorithm is executed only once for each refinement level

It is very important to outline that the new algorithm is only executed as many times as the maximum number of refinements of all elements in `SubN`. This number is usually between 1 and 5, depending on the refinement parameters, and is independent of the number of elements to refine. That feature makes this algorithm much more suitable for fine meshes than the older one, because the increase in the number of elements to refine only affects the size of the vectors, but the number of operations remains the same. Therefore, the computational cost increase is negligible.

A model of a hollow cylinder under internal pressure shown in Figure 4.1 was used as a toy problem in order to compare refinement performance by means of computation time:

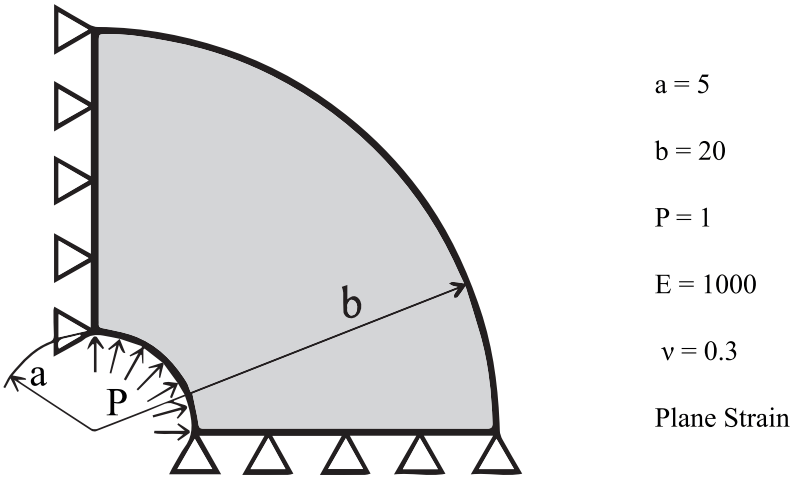


Figure 2.4: Hollow cylinder under internal pressure model.

An h-adaptive procedure was performed with both linear and quadratic elements in order to reach fine meshes. Finest meshes reached 3.75 and 3.45 million of degrees of freedom respectively for linear and quadratic cases. The results are shown in Figure 2.5. It is clear that the computational cost has been drastically reduced, specially in the linear case.

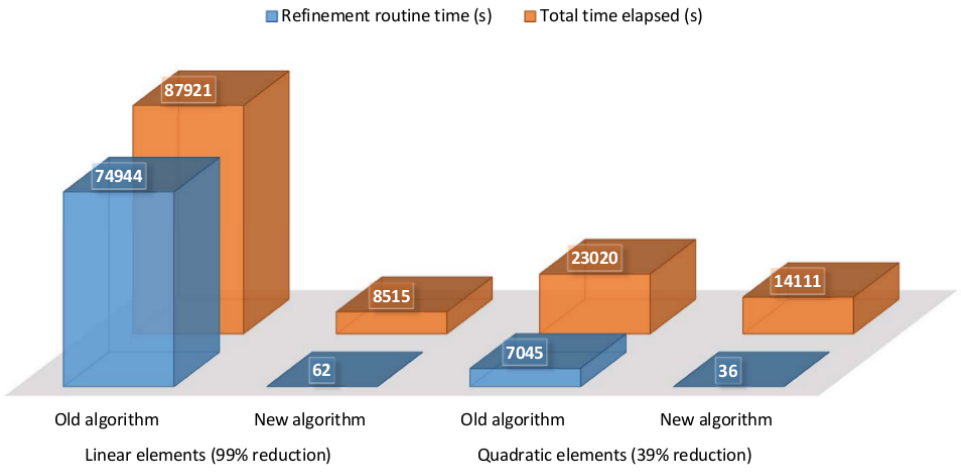


Figure 2.5: Performance test of the new refinement routine.

2.2 Vectorization

As it has been said before, MATLAB® is able to deal with matrices and arrays and operate with them in a highly effective way. This statement involves not only typical mathematical operations like additions, multiplications or evaluation of trigonometric functions, but also the reading and writing of elements inside an array or a matrix.

For example, let **A** be a random $n \times m$ matrix. The standard procedure to read an entire column of that matrix in C should be a `for` loop through all the rows of the desired column. However, MATLAB® has the colon operator `(:)` so you can read an entire column of a given matrix by just typing `A[:,Column]`. Also if you want to read several columns you can substitute the number in the `Column` variable for an array with all the columns wanted. This particular way of accessing elements in an array is called *Matrix Indexing*. Further information about different types of matrix indexing can be found in MATLAB® Documentation Center [4].

By changing the code above to read matrix data we are *vectorizing* the code. Quoting [4]: *"The process of revising loop-based, scalar-oriented code to use MATLAB® matrix and vector operations is called vectorization"*. Some of the main benefits of vectorizing are the code becoming easier to understand, as it is shorter, as well as the fact that vectorized code in MATLAB® language is frequently faster than a code with loops.

Vectorizing is not only an improvement technique but a whole programming philosophy while coding in MATLAB® language. This means that it is needed a change of mind for a programmer that starts using this language in order to reach high code performance, but after that learning process vectorizing should be like one more rule to follow while programming, like the language's syntax rules. The following sections explain different ways of vectorization with general examples in order to have a better understanding of this technique.

2.2.1 Modify some values of an array.

Taking a look again at the second step of algorithms 2.1.1 & 2.1.2 the only difference is that there is no `for` loop in the new algorithm. This is because

the operation of writing the value 1 on multiple positions of the vector `Active` can be done in one single step by using matrix indexation. Also remember that the old algorithm worked for a unique element and one split at a time, while the new algorithm splits once all elements that require refinement. Thus, the variable `Elements` is not an scalar but an array, so the same matrix indexation is performed.

2.2.2 Transform scalar operations' loop into array operations.

Here some general examples are presented in order to give a full sight of vectorization possibilities regarding scalar operations. Chapter 3 will show more practical examples, taking into account both Finite Element Method and cgFEM code distinctive features.

Parallel dot product of arrays The first and easiest way of vectorizing scalar operations can be found in the dot product of an array. Let $\mathbf{a}, \mathbf{b}$ be random vectors with the same number of components. The resulting dot product of these two vectors (vector $\mathbf{c}$) fulfills $\mathbf{c}_i = \mathbf{a}_i \times \mathbf{b}_i$. This can be easily performed by using MATLAB®'s *array operations*, this is, introducing a period (.) before the operators $*$, / or $\wedge$. In this example, the code should be something like `c=a.*b`. The only requirement to perform these array operation is that both arrays must have the same dimensions. This means that $\mathbf{a}$ and $\mathbf{b}$ can be $N-$dimension matrices. The resulting variable will be of the dimension and size of $\mathbf{a}$ and $\mathbf{b}$.

Scalar operations between arrays with different dimensions This is another common situation when performing numerical computations. For

example, let $\{\mathbf{x_1}, \mathbf{x_2}...\mathbf{x_n}\}$ be column vectors with the same number of components, and $\{\lambda_1, \lambda_2...\lambda_n\}$ different scalars. Finally, let $\{\mathbf{y_1}, \mathbf{y_2}...\mathbf{y_n}\}$ be vectors resulting from the following operation:

$$\mathbf{y_i} = \mathbf{x_i}\lambda_\mathbf{i} \qquad i = 1, 2, ...n \tag{2.1}$$

This operation should be performed with a loop through all vectors, but there is a faster way to achieve the same results, specially if the number of elements grows higher. The first step is to concatenate all vectors and scalars in order to apply vectorization:

$$\mathbf{Y} = [\mathbf{y_1}, \mathbf{y_2}...\mathbf{y_n}] \qquad \mathbf{X} = [\mathbf{x_1}, \mathbf{x_2}...\mathbf{x_n}] \qquad \lambda = [\lambda_1, \lambda_2...\lambda_n] \tag{2.2}$$

In this case an array operation would not work since both arrays $\mathbf{X}$ and λ have different number of dimensions, but there is a MATLAB® built-in function that can solve this problem, *bsxfun*. This function performs array operations between different size matrices provided that only one dimension has a different size. The syntax to perform this operation should be `Y=bsxfun(@times,X,`λ`)`.

It can be noticed that when using *bsxfun* the desired element-wise operation must be introduced as a function handle, that has to be a MATLAB® built-in function. A list of the available functions is shown in Table 2.2. To perform more complex, element-wise operations between arrays we can use *arrayfun*. This function acts like *bsxfun* but it works with any function handle that contains a set of scalar operations (like a series of sums and multiplications).

`@plus`	`@atan2d`
`@minus`	`@hypot`
`@times`	`@eq`
`@rdivide`	`@ne`
`@ldivide`	`@lt`
`@power`	`@le`
`@max`	`@gt`
`@min`	`@ge`
`@rem`	`@and`
`@mod`	`@or`
`@atan2`	`@xor`

Table 2.2: MATLAB® built-in functions available for `bsxfun`, [4]

Matrix-vector multiplications The last vectorization case concerns multiplications between a common matrix and different vectors. This situation occurs often in FEM as it will be explained in Chapter 3. Take for example the relationship between stress and strain by means of the elasticity theory in the linear elasticity case:

$$\boldsymbol{\sigma} = \mathbf{D}\left(\boldsymbol{\epsilon} - \boldsymbol{\epsilon}_0\right) + \boldsymbol{\sigma}_0 \tag{2.3}$$

where:

$\boldsymbol{\epsilon} = \{\epsilon_x\ \epsilon_y\ \epsilon_z\ \gamma_{xy}\ \gamma_{yz}\ \gamma_{zx}\}^T$**:** strain field.

D: material properties matrix.

$\boldsymbol{\epsilon}_0, \boldsymbol{\sigma}_0$**:** initial strain and stress.

if initial strain and stress are not considered, assuming that the strain is already calculated at all points, the stress calculation at each point p becomes a simple multiplication:

$$\boldsymbol{\sigma_p} = \mathbf{D_p}\boldsymbol{\epsilon_p} \tag{2.4}$$

This operation would be normally worked out with a loop through points performing that multiplication. However, one can code a `for` loop for the different materials (different mechanical properties, means different **D** matrices). The number of materials is always much lower than the number of points and the `for` loop will result in a much better performance in terms of computation time.

As in the previous example, the first step is to concatenate arrays in order to perform the vectorization. The stress of all points with the common material i is calculated as follows:

$$\begin{aligned} &\sigma_{\mathbf{i}} = [\boldsymbol{\sigma}_{\boldsymbol{i}}^{\boldsymbol{p_1}}, ..., \boldsymbol{\sigma}_{\boldsymbol{i}}^{\boldsymbol{p_n}}] \qquad \epsilon_{\mathbf{i}} = [\boldsymbol{\epsilon}_{\boldsymbol{i}}^{\boldsymbol{p_1}}, ..., \boldsymbol{\epsilon}_{\boldsymbol{i}}^{\boldsymbol{p_n}}] \\ &\sigma_{\mathbf{i}} = \mathbf{D_i}\epsilon_{\mathbf{i}} \qquad i = 1, ..., N_{Materials} \end{aligned} \tag{2.5}$$

where $\sigma_{\mathbf{i}}$ contains the stress evaluated at all points that have material i properties.

Resolution of multiple systems of equations The previous procedure can be applied when it comes to solving linear systems of equations that have a common coefficient matrix. This occurs when solving the FE equation system with multiple load cases and some other cases explained in Chapter 3. The vectorized FE equation system considering all load cases has exactly the same shape of the original, this is:

$$\mathbf{KU} = \mathbf{F} \tag{2.6}$$

However, in this case **F** is not a vector but a matrix that contains each different load case in each column. MATLAB®'s solving command (\) has a better performance solving this "vectorized equation system" than if all different equation systems were solved in a `for` loop where each iteration had a different load case. It is easy to understand why it is more efficient vectorizing equation systems: when the system is vectorized the coefficient matrix has to be factorized only once, while an inverse substitution is performed for each load case. It is widely known that factorizing the coefficient matrix is the hardest step while solving an equation system in terms of computational cost, specially when equation systems have a high amount of variables, so reducing the number of times the matrix has to be factorized becomes in a reduction of computing time.

2.3 Parallel computation through GPU

2.3.1 General-Purpose Graphics Processing Unit

At present, Graphics Processing Units (GPUs) are used not only for graphical rendering, but also as a co-processing device for massive computations in different fields such as mechanical computation, medical imaging or financial analysis. This new generation of devices are called General-purpose GPU, or GPGPU.

Almost all personal computers are provided with a GPU. These devices were originally created for 3D rendering, specially for video-games. As the gaming industry grew up, graphic rendering became more complex and there was a huge development in GPUs. In order to achieve more complex visual effects,

the first application programming interfaces (APIs) for GPUs, like OpenGL or Direct3D (inside DirectX), were deployed. However, these APIs were still graphics-oriented, and it was very hard for programmers to develop software for general-purpose computations.

NVIDIA® was the first company that introduced a specific technology for General-purpose computing with GPUs, it is called CUDA (compute unified device architecture). CUDA was originally designed to work in C language, but currently there are other programming languages that allow GPU computing through NVIDIA® CUDA, such as C++ or MATLAB®. The main competitor in GPU sector, AMD (former ATI), also developed a lower level API for GPUPU called Stream SDK. This API was not successful at all and finally AMD opted joining the open source project OpenCL. NVIDIA® GPUs can also work with OpenCL, but a comparison made in paper [5] shows that CUDA is slightly more effective than OpenCL when using NVIDIA® hardware. Despite being a registered technology that only works with the same company's GPU devices, CUDA has become a standard when using GPUPU applied to scientific computation.

A program developed with CUDA is usually divided into two parts. The CPU handles with the data management, memory transfers between different devices and sets the GPU execution settings, whereas the GPU acts as a co-processing device performing massive parallel computations through special functions called *kernels*.

The main idea behind GPGPU is that instead of having a handful of powerful processing units, like modern CPUs which have between 4-8 cores, GPU devices are provided with a huge amount of small processors (between one and two thousand in nowadays' devices) that are only able to execute the same instructions all at a time. This special feature makes GPU devices very suit-

able for all kinds of parallel computations, like adding a constant number to all the components of a vector.

Figure 2.6 shows the hardware structure of a CUDA GPU device. Each processing unit inside a GPU has some memory slots exclusively associated to it (*registers*). A processing unit together with its corresponding registers is called a *thread*. Threads can also be organized in *blocks*. When any code is executed on a GPU device, a copy of this code is sent to each block, and all threads run all the instructions contained in the code at the same time. This means that if there are any conditional statements in the code (like *if* or *while*), threads that do not fulfill the statement will not continue executing the next instructions until all other threads have finished running the instructions given inside the conditional statement.

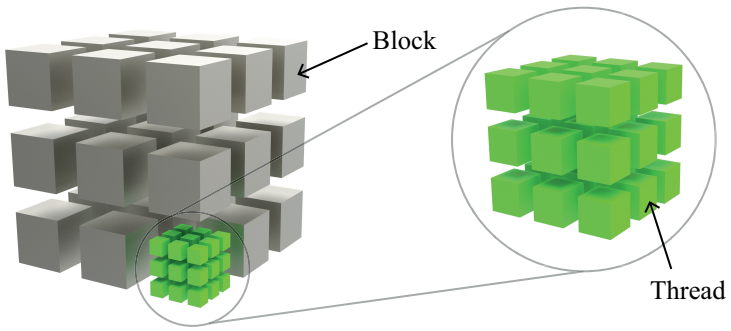


Figure 2.6: GPU grid configuration.

The configuration of threads and blocks define the computation *grid*. These grid has to be configured for each kernel run, and the performance of the program can be improved by choosing a near-optimal setting for each kernel [6].

CPUs have a hierarchical memory structure composed by registers, cache memory, RAM memory and hard drives. Similarly to the CPU memory structure, GPUs have the following hierarchical memory structure:

Registers: Each thread can only access its own registers. It is a small but fast memory.

Shared memory: All the threads in the same block have a shared memory so that they can interact between them.

Constant memory: Read-only, fast memory that all threads of the GPU can access.

Texture memory: Special memory related to graphics rendering structure.

Global memory: All threads of the GPU can access this slots, but the read-write speed is rather slow compared to the previous memory types.

As the read and write speed of each type of memory is different, having an optimal memory management between all types can be critical in terms of computing time.

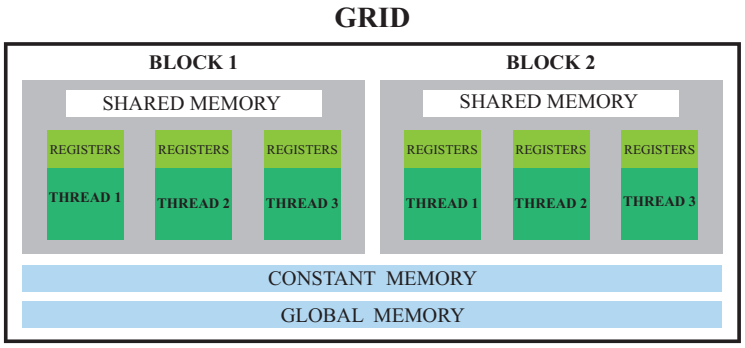


Figure 2.7: Hierarchical memory structure of a GPU.

2.3.2 Using GPU in MATLAB®

MATLAB®'s Parallel Computing Toolbox lets users take advantage of GPU computing features to improve programming efficiency specially when processing large amounts of data. However, MATLAB® does not allow the programmer to manage the GPU grid configuration, or handling with all different types of memory explained before. The parameters available in MATLAB® are total device memory and free device memory, so GPU will act as a black box that performs the desired operations with an unknown configuration of the grid. This makes programming more simple but less effective, as the calculation grid will usually be non-optimal. If the GPU kernel is programmed in C, it can be introduced in MATLAB® as a `mex` file. In this case it is possible to set the grid configuration, but this option will not be considered in this thesis, as it involves programming in a different language.

GPU functions in MATLAB® MATLAB®'s function `gpuDeviceCount` returns the number of devices that are available. Typing `gpuDevice(i)` MATLAB® returns all the information of GPU device number `i`, like the name of the device, available memory, free memory, CUDA version, maximum number of threads and maximum number of blocks.

Any MATLAB® statement whose variables are stored in a GPU device will be performed in the GPU (Figure 2.8 shows the available built-in functions for GPU computing). To transfer data from the CPU to the GPU the command `gpuArray` is used. For example, to transfer variable `a` to the GPU memory, just type `a = gpuArray(a)` (as the name of the variable remains the same, the original `a` variable in CPU memory will be erased). Also the typical commands used to create special arrays like `rand, zeros, ones, eye` have their GPU version just by adding "'`gpuArray.`"' before the standard command, so that

typing `gpuArray.zeros(5)` will create a 5 by 5 random matrix in the GPU memory.

abs	colon	fft2	iscolumn	mrdivide	sech
acos	complex	fftn	isempty	mtimes	shiftdim
acosh	cond	fftshift	isequal	NaN	sign
acot	conj	filter	isequaln	ndgrid	sin
acoth	conv	filter2	isfinite	ndims	single
acsc	conv2	find	isfloat	ne	sinh
acsch	convn	fix	isinf	nnz	size
all	cos	flip	isinteger	norm	sort
and	cosh	fliplr	islogical	normest	sprintf
angle	cot	flipud	ismatrix	not	sqrt
any	coth	floor	ismember	num2str	squeeze
arrayfun	cov	fprintf	isnan	numel	std
asec	cross	full	isnumeric	ones	sub2ind
asech	csc	gamma	isreal	or	subsasgn
asin	csch	gammaln	isrow	pagefun	subsindex
asinh	ctranspose	gather	issorted	perms	subsref
atan	cumprod	ge	issparse	permute	sum
atan2	cumsum	gt	isvector	plot (and	svd
atanh	det	horzcat	kron	related)	tan
besselj	diag	hypot	ldivide	plus	tanh
bessely	diff	ifft	le	pow2	times
beta	disp	ifft2	length	power	trace
betaln	display	ifftn	log	prod	transpose
bitand	dot	ifftshift	log10	qr	tril
bitcmp	double	imag	log1p	rand	triu
bitget	eig	ind2sub	log2	randi	true
bitor	eps	Inf	logical	randn	uint16
bitset	eq	int16	lt	rank	uint32
bitshift	erf	int2str	lu	rdivide	uint64
bitxor	erfc	int32	mat2str	real	uint8
blkdiag	erfcinv	int64	max	reallog	uminus
bsxfun	erfcx	int8	mean	realpow	uplus
cast	erfinv	interp1	meshgrid	realsqrt	var
cat	exp	interp2	min	rem	vertcat
ceil	expm1	interp3	minus	repmat	xor
chol	eye	interpn	mldivide	reshape	zeros
circshift	false	inv	mod	rot90	
classUnderlying	fft	ipermute	mpower	round	
				sec	

Figure 2.8: MATLAB® built-in functions available for GPU computing, [4].

After the calculations are performed the data has to be usually transferred back to the CPU to be stored, as the GPU has less memory than the available in the RAM and hard disk drives. The command used to transfer a variable from GPU to CPU memory is `gather`. It is very important to use as less as

possible both `gpuArray` and `gather`, because the overheads between CPU and GPU can become the bottleneck of the program.

The special functions `bsxfun` and `arrayfun` presented in section 2.2 also work with GPU variables providing a boost in vectorized operations, as the calculations in the GPU are performed in parallel. There is also another special function only available for GPU computing, `pagefun`. This function is similar to `bsxfun`. The inputs are 3D matrices, and `pagefun` performs a MATLAB® built-in function to all the pages of these matrix. For example let $\{\mathbf{A_1}, \mathbf{A_2}...\mathbf{A_k}\}$ be $m \times n$ arrays, and let $\{\mathbf{B_1}, \mathbf{B_2}...\mathbf{B_k}\}$ be $n \times p$ arrays. We group all A_i and B_i matrices in 3D matrices so that the size of A is $m \times n \times k$ and the size of B is $n \times p \times k$. Then we use `pagefun` with the `mtimes` function (this performs a matrix multiplication) by typing `pagefun(@mtimes,A,B)`. The result is a matrix (C) with a size of $m \times p \times k$ where $C_i = A_i \times B_i$. This feature allows to vectorize not only the dot product between arrays (shown in section 2.2), but also matrix multiplications between arrays if the user has a GPU device.

Memory management GPU cards usually have less available memory than a CPU. It is common to have between 8 to 32 gigabytes of RAM available for the CPU, and modern GPUs have between 1 and 3 gigabytes of memory. In addition, if the CPU runs out of RAM memory it may temporarily use part of the hard drive space. If the GPU runs out of memory it will not use the hard drive space and an error will occur in the program, so it is strongly advisable to check if the GPU memory is enough to perform all calculations before executing the operations. If there is not enough memory available in the GPU the data will have to be split into smaller pieces so that each piece of data can be processed by the GPU. Therefore, in these cases, the whole data will have to be processed using a `for` loop.

Using multiple GPUs When calculations become large enough, increasing the number of GPU devices can be a remarkable option for upgrading the calculation workstation with a relatively low cost. MATLAB® needs that every GPU is controlled by a different processing unit. As modern CPUs usually have between 4 and 8 core processors and the maximum number of GPUs that fit in a motherboard are 4, this should not be a problem for any machine. To assign each GPU to a different core a MATLAB® parallel pool is created with the number of workers (meaning parallel processors) equal to the number of GPUs available, then each GPU is assigned to a worker.

NVIDIA® has also developed the *SLI* technology, which essentially is a connecting hardware for multiple GPUs, so that the computer only detects one device, but it has the memory capacity and calculation power of all GPUs connected. Working with SLI connected GPUs in MATLAB® is exactly the same as working with only one GPU, because the device's available parameters are still total memory and free memory. In the case of SLI connected GPUs the user cannot distribute the data between the GPUs, so again the grid configurations might be non-optimal. However, this is still a powerful and very simple way to enhance the efficiency of the workstation with no changes in the code.

This thesis has been focused on exploring the possibilities of GPU computing in MATLAB® with only one device. Nevertheless, researching the capabilities and performance improvement of using multiple GPUs in MATLAB®, with or without SLI technology, is one of the future investigation objectives in the researching group.

2.4 Code optimization example

In order to clarify the use of the optimization techniques and to give an idea about how can a code be optimized step by step, a complete example of a code optimization process is given here. It is remarkable to say that this example should only be used as a guide about what can be done while programming in MATLAB® and to give some figures to the performance improvements mentioned in previous sections of this Chapter. Nevertheless, there is no need to go through this code optimization process while programming new code, as all vectorization and GPU techniques can be implemented at the very first time of the programming phase.

The function created in this example performs operations that are very similar to some FEM typical calculations, like the stress calculation at integration points and the evaluation of the strain energy for each element. We will assume that all the elements have some common properties as the same number of integration points located at the same local coordinates, but each element will have random material properties and random displacements. These assumptions are only made in order to have a simple piece of code that takes into account the features of FEM operations.

To calculate stress we start from equation 2.4, which is repeated below for convenience:

$$\boldsymbol{\sigma_p} = \mathbf{D_p}\boldsymbol{\epsilon_p}$$

where mechanical strains are produced by the displacements at nodes which is the solution of the FE problem. Applying the FE interpolation, strain can be calculated as follows

$$\boldsymbol{\epsilon} = \mathbf{Lu} = \mathbf{LNu^e} \qquad \rightarrow \qquad \boldsymbol{\epsilon} = \mathbf{Bu^e} \qquad \mathbf{B} = \mathbf{LN} \tag{2.7}$$

where:

L: Matrix that contains the derivative operators.

N: Shape functions used to interpolate the displacements inside the element from the nodal displacement values.

$\mathbf{u^e}$: Values of the displacements at the nodes of the elements. Solution of the FE system of equations.

then FE stress can be calculated at any point using the displacement solution at nodes with this expression:

$$\boldsymbol{\sigma} = \mathbf{DBu^e} \tag{2.8}$$

where, in these code example, assuming 2D case:

D: $3 \times 3 \times Number\ of\ elements$. 2D Material properties matrix. Each element will have its random properties.

B: $3 \times 8 \times Number\ of\ integration\ points$. Derivatives of the shape functions at the integration points. This data will be common for all elements in order to simplify the example.

$\mathbf{u^e}$: $8 \times Number\ of\ elements$. Displacements at nodes of each element.

Regarding strain energy, a more detailed explanation about its calculation will be given in Chapter 3, and here we will perform a simple version with algebraic operations similar to those used in these kind of calculations. To sum up, strain energy at elements is calculated as a numerical integration of the strain energy inside the element. Here we will disregard the numerical integration features (weights and jacobian matrix) because including them should be only a matter of repeating the same kind of operations. Instead, we will consider this equation to calculate the strain energy-like value at each element that will be called the *accumulative value* E^e:

$$E^e = \sum_{i}^{nIP} \boldsymbol{\sigma}_i^{\boldsymbol{e}} \boldsymbol{\epsilon}_i^{\boldsymbol{e}} \tag{2.9}$$

where, in these code example:

- E^e**:** $1 \times Number\ of\ elements$. Accumulative value E at each element.
- nIP**:** Number of integration points per element. Constant in this example.
- $\boldsymbol{\sigma^e}$**:** $Number\ of\ elements \times (Number\ of\ integration\ points \times 3)$. Each row has the stress at all integration points of each element.
- $\boldsymbol{\epsilon^e}$**:** $Number\ of\ elements \times (Number\ of\ integration\ points \times 3)$. Each row has the strain at all integration points of each element.

The example is divided into three parts: the first one contains which is supposed to be the original piece of code to improve, then in the second part the code is improved by means of vectorization, and finally in the third part GPU computation is brought in to go a step further in the optimization process. After those three parts there are some results showing the performance comparison between different versions.

2.4.1 Original Code

This is the initial function that performs the established calculations. The code is shown in Listing 2.1. The structure is very similar to a C-code function. Initial variables simulate the available data in the program and are created as random variables. Then both output variables are pre-allocated in memory, and finally calculations are performed in a loop through elements and integration points inside elements.

Listing 2.1: Optimization example. Original code.

```
% Initial data
DOFperNode = 2;
Comp = 3;
IntegrPts = 4;
NodesPerElem = 4;
NumElements = 1000; % This is the parameter to sweep
D = rand(Comp,Comp,NumElements);
B = rand(Comp,NodesPerElem*DOFperNode,IntegrPts);
u = rand(NodesPerElem*DOFperNode,NumElements);
epsilon = rand(NodesPerElem*(DOFperNode+1),NumElements);

% Output variables
sigma1 = zeros(NumElements,IntegrPts*Comp);
E_def = zeros(NumElements,1);

% Calculation. Element loop
for iElement = 1:NumElements
    uElem = u(:,iElement);
    DElem = D(:,:,iElement);
    epsilonElem = epsilon(:,iElement);
    for iPoint = 1:IntegrPts
        % Stress
        stress=DElem*B(:,:,iPoint)*uElem;
        sigma1(iElement,(iPoint-1)*Comp+1:iPoint*Comp) =
            stress;
        % Accumulative value E
        E_def(iElement) = E_def(iElement)+...
            sum(stress'*epsilonElem((iPoint-1)*Comp+1:iPoint*
                Comp));
    end
end
```

2.4.2 CPU optimized code

This optimized version is focused on vectorizing all possible operations to get rid of as most single operations inside loops as possible. The vectorized code is shown in Listing 2.2. The most clear example is with the accumulative value (line 28). The accumulative value at each point is calculated with a single scalar product, so performing a dot product with stress and strain data matrices we vectorize this operation. The next step is to accumulate the values at each element, therefore we perform an addition of the point's values matrix through the first dimension, so we get a vector that has the total accumulative value at each element.

Stress calculation cannot be fully vectorized because it involves matrix-vector multiplications without a common matrix (**D** matrices were set as random for each element with this purpose), but there is still some work that can be done. The stress is a result of the multiplication of two matrices and one vector. If we use the whole matrix **u** instead of the corresponding column for each element we will perform all operations for all elements at a time. Moreover we said that **B** matrices were the same for each local integration point, so we can make a previous loop (lines 17-19) where we compute the product **Bu** for all points, and then the only operation performed in the element's loop would be a matrix multiplication. The product matrices **Bu** are stored as a 3D matrix where, after a permutation (line 20), each page has the matrices corresponding to each element, just like matrix **D**.

Listing 2.2: Optimization example. CPU optimized code.

```
% Initial data. Same as in the original code
DOFperNode = 2;
Comp = 3;
IntegrPts = 4;
NodesPerElem = 4;
NumElements = 1000; % This is the parameter to sweep
D = rand(Comp,Comp,NumElements);
B = rand(Comp,NodesPerElem*DOFperNode,IntegrPts);
u = rand(NodesPerElem*DOFperNode,NumElements);
epsilon = rand(NodesPerElem*(DOFperNode+1),NumElements);

% Output variables. E_def2 memory pre-allocation is not
    needed anymore.
sigma2 = zeros(NumElements,IntegrPts*Comp);

% Stress, step 1. Loop through different local integration
    points.
B_u = zeros(Comp,NumElements,IntegrPts);
for iPoint = 1:IntegrPts
    B_u(:,:,iPoint) = B(:,:,iPoint)*u;
end
B_u = permute(B_u,[1 3 2]);
% Stress, step 2. Loop through elements.
for iElement = 1:NumElements
    DBu = D(:,:,iElement)*B_u(:,:,iElement);
    sigma2(iElement,:) = DBu(:);
end

% Accumulative value E. Vectorized operation
E_def2 = sum(sigma2'.*epsilon,1)';
```

2.4.3 GPU optimized code

The last optimization step includes GPU computation in the code. As it has been said, it is very easy to introduce GPU in MATLAB® language, since there are only a few changes to make. First of all, the variables are now initialized directly in the GPU device, by adding `gpuArray` to the functions `rand` and `zeros`. In case the variables already exist in the CPU the transfer would be as easy as typing `GPUVariable=gpuArray(CPUVariable)`. The same thing happens when the calculations are finished, this is, we need to transfer the data back into the CPU using the command `gather()`. Listing 2.3 shows this last optimized code.

Just by performing the same calculations on the GPU we could have some gain. For example, the calculation of the accumulative value (l. 24) is exactly the same line as in listing 2.2, and its computing time is lower. But we can also add some important features in order to gain efficiency, like multiplying matrices in parallel. Using `pagefun` we multiply all matrices $\mathbf{B}$ and $\mathbf{u}$, and then all element multiplication of $\mathbf{D} \times \mathbf{Bu}$, so we just get rid of both loops that were still in the CPU-optimized code.

Listing 2.3: Optimization example. GPU optimized code.

```
% Initial data. Variables are now created in the GPU memory
DOFperNode = 2;
Comp = 3;
IntegrPts = 4;
NodesPerElem = 4;
NumElements = 1000; % This is the parameter to sweep
B = gpuArray.rand(Comp,NodesPerElem*DOFperNode,IntegrPts);
u = gpuArray.rand(NodesPerElem*DOFperNode,NumElements);
epsilon = gpuArray.rand(NodesPerElem*(DOFperNode+1),
    NumElements);
D = gpuArray.rand(Comp,Comp,NumElements);

% Output variables
sigma3 = gpuArray.zeros(IntegrPts*Comp,NumElements);

% Stress, step 1. Loop substituted by pagefun.
B_u = pagefun(@mtimes,B,u);
B_u = permute(B_u,[1 3 2]);
% Stress, step 2. Loop substituted by pagefun.
DBu = pagefun(@mtimes,D,B_u);
sigma3(:) = DBu;
% Transfer stress data from GPU to CPU memory
cpu_sigma = gather(sigma3');

% Accumulative value E at each element and data transfer to
    CPU
E_def3 = gather(sum(sigma3.*epsilon,1)');
```

2.4.4 Performance test

The performance comparison test consisted in executing all three different codes with an increasing number of elements. Computing time of each code version is measured using MATLAB®'s `tic` and `toc` functions. In the GPU case the function `wait(gpuDevice)` was used in order to avoid stopping the CPU timer while the GPU device is still performing operations. In addition, GPU code was executed and measured three times for each parameter value, and the mean value between those three times is used to calculate the speed-up ratio. CPU and GPU speed-up ratio over the original routine are evaluated as follows:

$$Ratio_{CPU} = \frac{Time_{Original}}{Time_{CPU}} \qquad Ratio_{GPU} = \frac{Time_{Original}}{Time_{GPU}} \tag{2.10}$$

The test can be divided into two parts: the first one involves *low* number of elements (from 10^3 to 10^6), and the second part goes from 10^6 to 2×10^7 elements. During the first part of the test the GPU device's memory is not saturated and all the data can be processed at once. As we reach 5 million elements the GPU's memory is not big enough to perform all the calculations. Therefore, as said in section 2.3, some memory management work has to be done in order to split all data into smaller pieces, so that each data block can be processed by the device.

Figures 2.9 and 2.10 show the results of this test. The first highlighting result is that vectorized code has an almost constant speed-up ratio, between 7.66x and 8x. The independence of the speed-up with the number of elements in the arrays shows that this computing improvement has been achieved because MATLAB® is capable of dealing with array operations in a more efficient way than performing `for` loops with scalar operations, as said before.

Regarding the GPU, in the first part of the test (Figure 2.9) the speed-up rapidly escalates with the number of elements, and starting from around 3×10^5 the speed-up ratio varies between 90x and 100x. However in the GPU case there are some considerable oscillations in the speed-up ratio as the number of elements increases. These variations can be produced due to a non-optimal setting of the GPU grid of blocks and threads. As these settings cannot be modified or read when using parallel computing features included in MATLAB® there is no way neither to find a near-optimal setting of the grid nor to demonstrate this hypothesis. Nevertheless, this should be the most probable cause of the oscillation because as it is said in [6] "gridification strongly impacts the performance of algorithms".

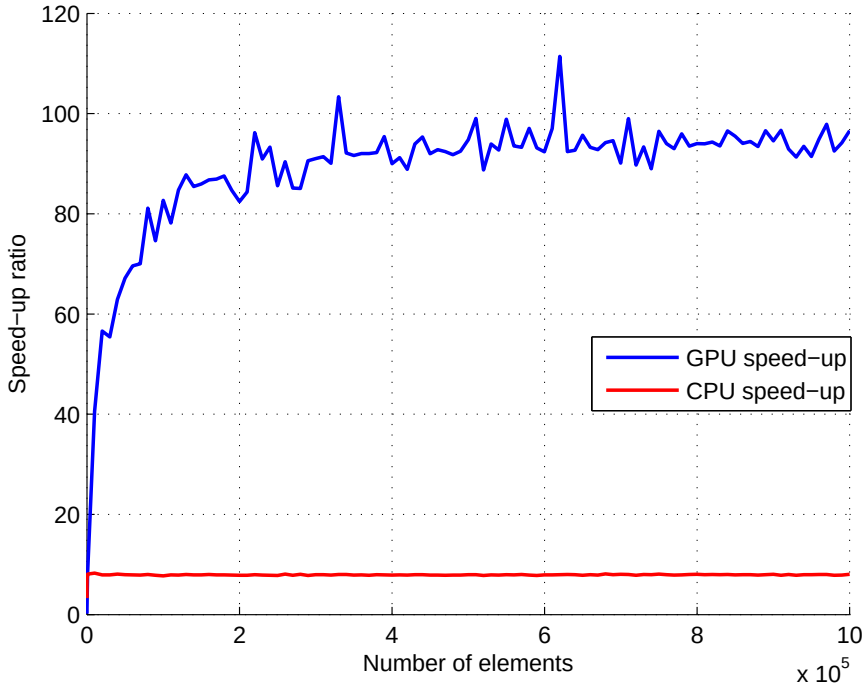


Figure 2.9: Performance test, part 1. From 10^3 to 10^6 elements

At the second part of the test (2.10) there are two zones where the GPU speed-up ratio drops to 60x. The minimum values of speed-up ratio occur at 1.45×10^7 and 1.95×10^7. As it was said before, the GPU device could calculate all data in one step up to 5 million elements. From 5 to 10 million elements two iterations were made to process all data. Then, with 10 million elements, three iterations were performed. With 15 million elements the number of iterations increased to four, and the last run with 20 million elements needed five iterations. It can be noticed that the number of elements where the speed-up dropped were the last ones where the calculations were performed in 3 and 4 iterations respectively, therefore when the device's memory was about to be fully consumed.

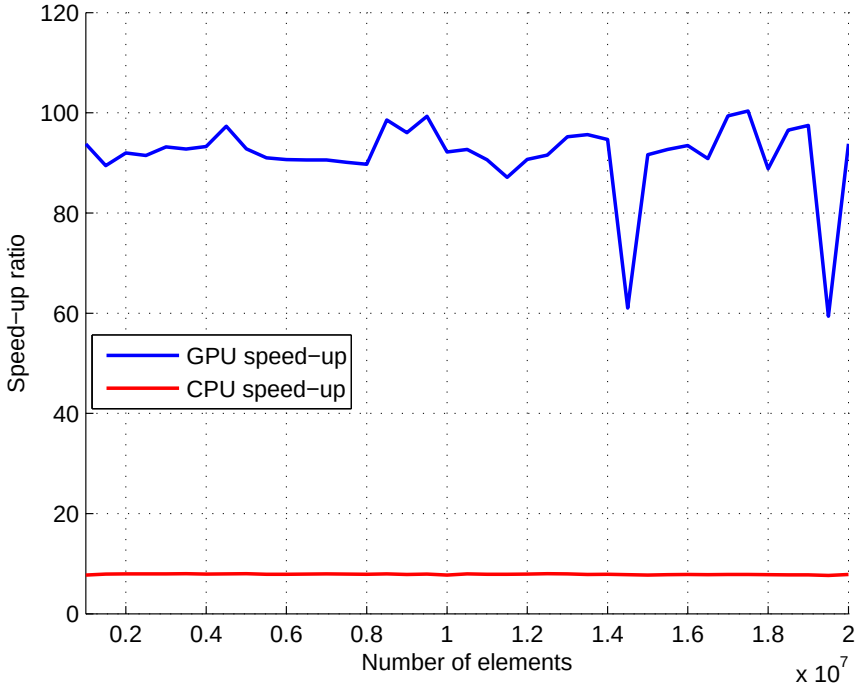


Figure 2.10: Performance test, part 2. From 10^6 to 2×10^7 elements

The results shown here clearly demonstrate that vectorizing and using GPU can dramatically increase the efficiency in terms of computational cost when using MATLAB® for numerical computations. Despite having some limitations regarding the device *gridification* that cause oscillation in the speed-up ratio, the usage of GPU computing in MATLAB® still provides an important boost when performing *single instruction multiple data* operations.

Chapter 3

Finite Element Method optimization

Any standard Finite Element Method program can be divided into three main blocks: pre-processing, resolution and post-processing. The pre-processing block has the biggest amount of user interaction, as the geometry of the domain, the material properties and the settings and generation of the mesh have to be defined. The FEM system of equations' formulation and resolution is included in the second block. Finally, in the post-processing block other results are calculated, such as stress and strain energy, and the error estimation procedure is executed.

All the optimization techniques described in Chapter 2 are now going to be applied to a FEM program. The special features of Cartesian grids and medical image-based analysis will also be taken into account.

3.1 Pre-processing

The computational time spent in the operations that require user intervention is negligible. Therefore, all the code regarding the definition of the geometry and material properties and the set up of the analysis will be left apart in this Thesis.

3.1.1 Intersection procedure

As cgFEM works with geometry independent Cartesian grids, it is trivial to obtain the topology and location of the elements in the mesh. However, the intersection of the problem's domain with the set of Cartesian grids has a high computational cost when the number of intersected elements rises.

The intersection procedure is as follows. The border of the domain is composed by curves. The algorithm selects one point on the contour to start with the boundary intersection process. This point corresponds to the 1st point of the 1st curve used to define the boundary. The intersection routine starts locating that point in the Cartesian grid pile. After that, the routine follows the contour counterclockwise from the current element to the next one, until it reaches again the starting point. At this stage, the whole contour has been intersected.

As it happened in the old refinement routine (section 2.1), the task of finding the active neighbors of an element in the calculation mesh implied several accesses to *sparse* matrices, which are likely to be slow, specially if the matrices have a considerable size.

The new algorithm proposed in section 2.1, `FindRealNeighbors`, has also been used in the intersection routine. Before starting the intersection of the first point of the boundary with the mesh, `FindRealNeighbors` is executed with all active elements as input. Therefore, the connectivity of all elements in the mesh is available in a *full* matrix (not *sparse*), and no logical statements have to be executed to determine which neighbor element in the Cartesian grid pile is the active one.

The comparison tests shown in Chapter 4 report up to a 90% improvement of the intersection routine. This improvement ratio is achieved in tests with linear elements and a high amount of degrees of freedom, as it occurred with the improvement of the refinement routine.

In the new 3D code, FEAVox, the intersection of the geometry with the Cartesian grid pile is completely different from the cgFEM procedure. The boundary entities are now surfaces instead of lines, so a sequential procedure starting from a point and following the contour cannot be performed anymore. In FEAVox a ray-tracing procedure analogous to the ones used in rendering applications is performed. Ray-tracing is the usual task that a GPU performs while executing a video-game, so this intersection procedure is likely to be computed in parallel with GPU devices, which is one of the future works of this Thesis. Nevertheless, the intersection routine in 3D has already been implemented taking into account the vectorizing techniques.

3.1.2 *h*-adaptive refinement

The *h*-adaptive refinement improvement shown in section 2.1 should also be included in this block. This procedure is executed after the recovery techniques

are applied and the error of the solution is estimated in the post-processing block. With the information of estimated error at each element, the refinement routine generates a new calculation mesh in order to solve a new problem, so the block sequence starts again from the intersection of the geometry with the new mesh.

FEAVox, the 3D version of cgFEM is also capable of creating h-adapted meshes from CAT scans such as cgFEM does. Figure 3.1 shows a 3D mesh obtained from a dental computed tomography (CT) scan. The whole mesh contains near 3 million elements. Thanks to the new refinement algorithm that has been adapted to the 3D code this procedure took about two minutes.

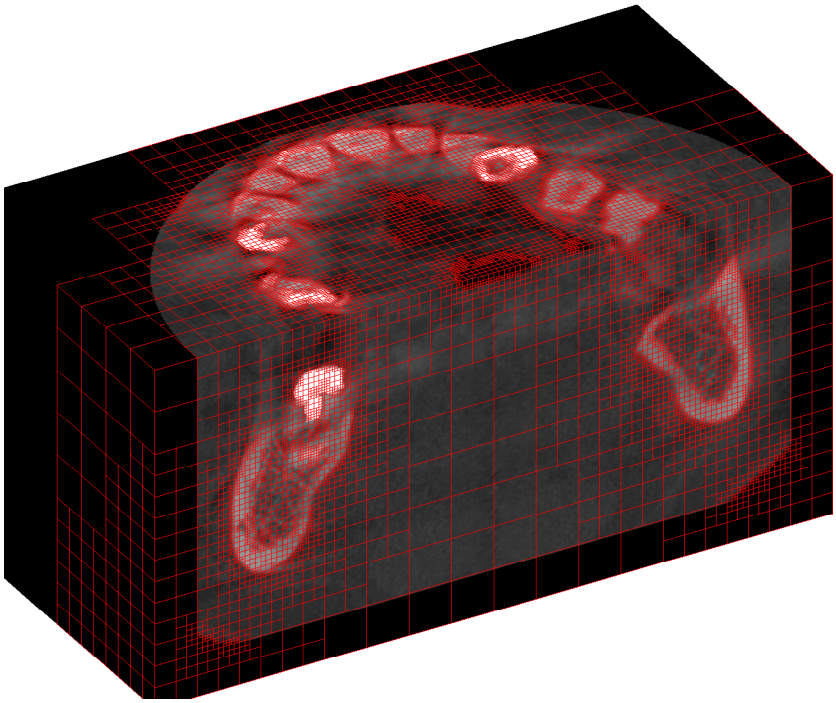


Figure 3.1: Cross section of a 3D mesh obtained from a dental CT scan.

3.2 FEM problem resolution

Once the calculation mesh is created it is time to formulate and solve the FEM system of equations, which was already presented in equation 2.6:

$$\mathbf{KU} = \mathbf{F} \tag{3.1}$$

This global system of equations is created from the assembly of element matrices $\mathbf{k}^e$, that depend on material property and the geometry of the domain, and element vectors $\mathbf{f}^e$ which depend on the boundary conditions. The optimization of this block has been focused on the creation of the element stiffness matrices, although some minor improvements were made in the integration of the boundary conditions to create vectors $\mathbf{f}^e$.

Although minor changes regarding vectorization have been made in the routine that creates element vectors $\mathbf{f}^e$, this section will focus on the improvement of the element stiffness matrices integration.

3.2.1 Stiffness matrix integration.

The element stiffness matrices is defined as:

$$\mathbf{k}^e = \int_{A^e} \mathbf{B}^T \mathbf{D} \mathbf{B} dA \tag{3.2}$$

The 0-level mesh element in cgFEM is called the *reference element*. All elements in the Cartesian grid pile of meshes are geometrically similar, they are all square elements, whose only difference is their size. Thus, it is possible to set some relation between the *reference element's* characteristics, such as the element stiffness matrix, Jacobian matrix, etc. Ródenas *et al.* defined in [13] 5 hierarchical properties that relate the element matrices of geometrically similar elements. The parameter relating the elements is the relative elements size, ς.

In the cgFEM framework $\varsigma = 2^{-L}$, being L the level of the element. Then, we could easily relate all the following properties with those of the *reference element*, indicated with the sub-index 0.

- *Jacobian matrix*: $\mathbf{J} = \varsigma \mathbf{J}_0$.
- *Inverse Jacobian*: $\mathbf{J}^{-1} = \frac{1}{\varsigma}\mathbf{J}_0^{-1}$.
- *Jacobian*: $|\mathbf{J}| = \varsigma^D |\mathbf{J}|_0$, where D is the problem dimension (2 for 2D).
- *Shape function derivatives matrix*: $\mathbf{B} = \frac{1}{\varsigma}\mathbf{B}_0$.
- *Stiffness matrix*: $\mathbf{k} = \varsigma^{D-2}\mathbf{k}_0$, for $\mathbf{D}$ being constant.

Therefore, in cgFEM, the element stiffness matrix is equal for elements fully located into the domain that have the same material. Thus, only one stiffness matrix is calculated for all those elements.

However, there are multiple materials within the elements of a medical image-based analysis, so all the element stiffness matrices have to be calculated one

by one. The numerical integration that takes place to calculate this matrix is the following one:

$$\mathbf{k}^e = \sum_{i=1}^{NIP} \mathbf{B}^T(\xi_i, \eta_i)\mathbf{D}(\xi_i, \eta_i)\mathbf{B}(\xi_i, \eta_i)|\mathbf{J}(\xi_i, \eta_i)|\omega_i \tag{3.3}$$

where:

NIP**:** Number of integration points. In this case the pixels of the image will be considered integration points as well.

(ξ_i, η_i)**:** Local coordinates of the integration points expressed in the local reference system of the element.

$\mathbf{D}(\xi_i, \eta_i)$**:** Material properties associated to each integration point. As the one integration point will be associated to each pixel in the image, the material properties of each integration point will correspond to those of the gray level of the corresponding pixel.

$|\mathbf{J}(\xi_i, \eta_i)|$**:** The determinant of the jacobian matrix evaluated at (ξ_i, η_i).

ω_i**:** Weights associated to the integration points, depending on the integration quadrature used. A Riemann integration will be used in this case, so all weights will be the same within an element.

The bounding box of the initial mesh of a medical image analysis is calculated so that all mesh levels have an integer number of pixels inside, until the mesh

level that has one pixel per element. This is made by adding "black" pixels (meaning pixels with null material properties) until the size of the image contains $N \times N$ pixels, being N a power of two. The size of the pixels is constant for all the image, so it is known that $NIP = \frac{N \times N}{2^{2p}}$, being p the mesh level of the element.

If elements of the same level contain the same amount of integration points, located at the same local coordinates, then $\mathbf{B}(\xi_i, \eta_i)$, $|\mathbf{J}(\xi_i, \eta_i)|$ and ω_i will be common for all elements of the same level. Therefore, it seems that an efficient way of calculating the stiffness matrices in a medical image analysis would be to use a loop through different mesh levels and a second loop inside through local integration points. These two loops do not imply an increase in the computational cost as the number of elements rises, because the number of iterations is smaller (the more elements in the mesh, the less integration points/pixels within each element).

The only way to avoid a third loop through elements is using GPUs, because the operations for the stiffness matrix calculation include two matrix multiplications. The evaluation of the material properties matrices ($\mathbf{D}(\xi_i, \eta_i)$) for each integration point is also performed in parallel with GPU. If the workstation does not have a GPU device, a third loop will be run calculating the matrix for each element, and $\mathbf{D}$ matrices will be gathered through a vectorized procedure.

Section 4.3 shows the results of both changes. The improvement in the integration of the stiffness matrices for medical image analyses has allowed to rapidly implement and test different integration methods shown in paper [3].

Integration data structure In cgFEM, all the information of the stiffness matrix calculation (stiffness matrix, local coordinates of integration points, **B** matrices of each integration point, material associated to each integration point) is stored in a *struct* type variable called `MatK`. Table 3.1 shows how the integration data is stored for each element in cgFEM.

Field	Type	Description
Ke	Float	Element stiffness matrix
IntegrType	Int	Flag that indicates if it is an internal (0) or boundary element (1)
OrigLev	Int	Level of the element where the integration of $\mathbf{k}^e$ has been performed
BPtG	Float	Stores matrix $\mathbf{B} = \mathbf{LN}$ evaluated at each integration point
PsiEtaG	Float	Local coordinates of the integration points in the element
AreaGauss	Float	Area associated to each integration point
Triangs	Float	Global coordinates of the triangulation vertices. Only for boundary elements
PsiEtaVert	Float	Local coordinates of the triangulation vertices. Only for boundary elements
MaterialTr	Float	Material associated to each integration point

Table 3.1: Information stored for each element.

The structure shown in table 3.1 is repeated for each element stiffness matrix that is calculated during the analysis. Therefore, `MatK` is not only a *struct* type variable, but a vector of *struct* variables. The information related to the *reference element* of each different material existing in the model is stored in the first positions of this structure. Therefore, all internal elements will point to the corresponding *reference element* information, saving a high amount of repeated data. However, each boundary element has its own data stored in `MatK`. As the vector grows higher (when the number of boundary elements increases) the time spent accessing and writing values of `MatK` becomes the most time-consuming task in the stiffness matrix creation.

The improvement of the stiffness matrix integration when analyzing medical images stores the calculated data in an auxiliary variable and then transfers

the data of multiple elements to the `MatK` variable at the same time using vectorization techniques, thus reducing the number of accesses to the *struct* variable.

A vector of *struct* type variables is difficult to manipulate. Gathering all the values of a given field and storing new data are complex, time-consuming tasks. In addition, some of the stored data in `MatK` (mainly the local coordinates, the material ID and the **B** matrices) are necessary to make some calculations that could be easily vectorized if the data were stored in another type of variable.

Taking this features into account, the data structure of the integration has been changed for the FEAVox code. `MatK` variable has been split into two different matrices: `KMatrix` and `IntegrationPoints`. The first matrix has the element stiffness matrix for all elements stored as columns. This matrix is only used for the assembly of the global stiffness matrix **K**, so the storage of the element matrices as columns may also help with the vectorization of the assembly task.

The second variable, `IntegrationPoints`, is a standard matrix that contains in each row all the information of all the integration points, such as the element that the point belongs to, global and local coordinates, associated material, integration sub-domain in case of boundary elements, and results like stress, displacements and energy norms evaluated at each point.

This modification completely changes the calculating procedure for any magnitude. Instead of having an iterative loop through elements and calculating the desired magnitude, the operation can be performed in a vectorized way (and furthermore, using GPU computation) for all the integration points in the calculation mesh, regardless of the element that they belong to.

3.2.2 System of equations' resolution

The resolution of the system of equations is usually one of the most expensive procedures in terms of computational cost, specially when it comes to 3D problems. Although MATLAB®'s command "\" can be computed with GPU variables, MATLAB® still cannot handle with sparse variables in the GPU, so at the moment there is no chance to solve the system of equations with the GPU device, as the global stiffness matrix **K** is a sparse matrix. Preliminary tests in FEAVox show that the resolution of the 3D system of equations will have a high computational cost when the number of degrees of freedom increase, so implementing iterative solving procedures using GPU parallel computation in FEAVox is one of the future research lines.

3.3 FEM solution post-processing

As the FEM formulation used in cgFEM and FEAVox is based on displacements, the solution obtained after solving the system of equations is the displacement field of the domain. However, this results are usually less interesting for the industry than the stress field. It is also important to evaluate the error of the current solution, as the FEM solution is always an approximation of the real solution. All these tasks are suitable for vectorization and for the introduction of GPU computation, as there are simple operations performed on large amounts of data.

3.3.1 Stress calculation

Earlier in section 2.4 an optimization example of a code that calculated stresses and an *accumulative value* was given. Equation 2.8 , which is repeated here for convenience showed how stresses could be calculated at integration points:

$$\boldsymbol{\sigma} = \mathbf{DBu^e}$$

It has been said before that in cgFEM there is a relationship between the *reference element's* characteristics and the internal element's ones. Regarding the stress calculations, the relationship between the shape function derivatives matrix is $\mathbf{B} = \frac{1}{\varsigma}\mathbf{B}_0$, where ς is a value related to the element's level. Then, the stress value for the integration points of an internal element is:

$$\boldsymbol{\sigma}^e = \mathbf{DB}\frac{\mathbf{1}}{\varsigma^{\mathbf{e}}}\mathbf{u^e} \tag{3.4}$$

The displacements and scaling factor of all internal elements can be multiplied using `bsxfun` as it was shown in section 2.2.2. Then, **D** and **B** matrices are common for all elements, so the stress at all integration points of all internal elements is calculated by vectorizing the matrix-vector multiplication, as it was also shown in section 2.2.2.

When it comes to medical image analyses **B** matrices are common but each integration point has a different **D** matrix. In this case, the stress at all integration points can be calculated at once through GPU parallel computing. If the GPU is not available in the workstation, a loop through all integration

points has to be performed, just like in the calculation of element stiffness matrices.

Stress calculation is already vectorized in FEAVox for internal elements. However, boundary elements have different number of integration points so a `for` loop is needed for those elements. GPU computation has been applied to compute all boundary elements at once using `pagefun`. A test problem with up to 150000 boundary integration points showed a speed-up of 18x in this task.

3.3.2 Error estimation

After the FE solution of the first analysis mesh has been obtained, new meshes are created following a h-adaptive refinement procedure that aims to minimize the error in energy norm of the solution. The exact error in energy norm of the solution is given by:

$$|||\mathbf{e}|||_\Omega^2 := \int_\Omega (\boldsymbol{\sigma} - \boldsymbol{\sigma}^h)^T \mathbf{D}^{-1} (\boldsymbol{\sigma} - \boldsymbol{\sigma}^h) \, \mathrm{d}\Omega \tag{3.5}$$

where $\boldsymbol{\sigma}^h$ is the FE stress field and $\boldsymbol{\sigma}$ is the exact stress field, which is usually unknown. In order to estimate the error in energy norm cgFEM uses the Zienkiewicz & Zhu (ZZ) error estimator (3.6), presented in [14], where $\boldsymbol{\sigma}^*$ is an improved stress field, more accurate than $\boldsymbol{\sigma}^h$.

$$|||\mathbf{e}|||_\Omega^2 \approx \mathscr{E}_{ZZ}^2 := \int_\Omega (\boldsymbol{\sigma}^* - \boldsymbol{\sigma}^h)^T \mathbf{D}^{-1} (\boldsymbol{\sigma}^* - \boldsymbol{\sigma}^h) \, \mathrm{d}\Omega \tag{3.6}$$

Particularizing (3.6) at each element domain the estimation of the error in energy norm at element level can be obtained. This information will determine if an element has to be refined or not. The numerical integration required to calculate the error in energy norm at each element is as follows:

$$|||\mathbf{e}|||_e^2 \approx \sum_{i=1}^{NIP} (\boldsymbol{\sigma}_i^* - \boldsymbol{\sigma}_i^h)^T \mathbf{D}^{-1} (\boldsymbol{\sigma}_i^* - \boldsymbol{\sigma}_i^h) |\mathbf{J}(\xi_i, \eta_i)| \omega_i \tag{3.7}$$

cgFEM calculates, at the same routine, not only the error in energy norm, but also the energy norm of both the FE solution and the recovered solution, given by:

$$\begin{aligned} \left|\left|\left|\mathbf{u}^h\right|\right|\right|_\Omega^2 &= \int_\Omega (\boldsymbol{\sigma}^h)^T \mathbf{D}^{-1} \boldsymbol{\sigma}^h \, \mathrm{d}\Omega \\ \left|\left|\left|\mathbf{u}^*\right|\right|\right|_\Omega^2 &= \int_\Omega (\boldsymbol{\sigma}^*)^T \mathbf{D}^{-1} \boldsymbol{\sigma}^* \, \mathrm{d}\Omega \end{aligned} \tag{3.8}$$

The calculation of all these magnitudes has been optimized using both vectorization techniques and GPU computing reporting up to a 90% computation time reduction in this specific routine in cases with a high number of degrees of freedom (around 2-3 million).

SPR technique The ZZ error estimator requires obtaining an improved stress field $\boldsymbol{\sigma}^*$. This improved field is calculated using the Super-convergent Patch Recovery (SPR) technique, presented also by Zienkiewicz & Zhu [15]. This technique, first defines a patch of elements $\mathcal{P}^i$, that is a set of elements

sharing a vertex node i, this node is also called the patch assembly node, see Figure 3.2.

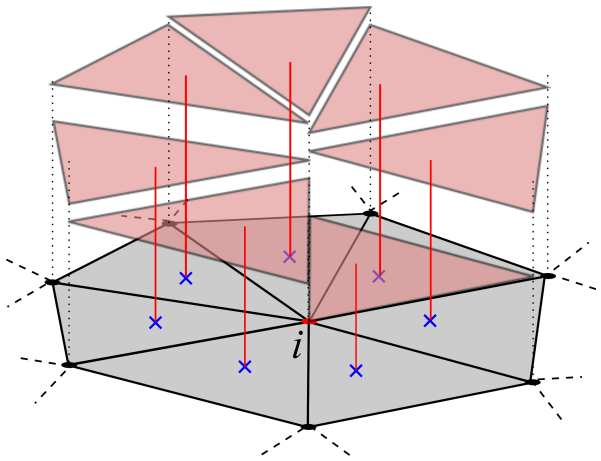


Figure 3.2: Representation of a patch of linear triangular elements. The black points indicates the nodes of the mesh and the red node is the patch assembly node. The transparent surfaces indicate the FE stress field $\boldsymbol{\sigma}^h$. The super-convergent points are indicated by blue crosses.

As shown in Figure 3.3, a polynomial surface per component (3.9) (of the same degree as the FE interpolation) is fitted to the FE stress values at the super-convergent points of the patch by using a least square approach:

$$\hat{\sigma}_k^*(\mathbf{x}) = \mathbf{p}(\mathbf{x})\mathbf{a}_k \quad k = xx, yy, xy \tag{3.9}$$

where $\mathbf{p}(\mathbf{x}) = \{1, x, y\}$ for the linear case, $\mathbf{p}(\mathbf{x}) = \{1, x, y, x^2, xy, y^2\}$ for the quadratic case, and $\mathbf{a}_k$ are the corresponding coefficients for each stress component. In this case, each component k of the stress field could be recovered

independently by minimizing the following functional:

$$\Phi_{SPR} = \sum_{gp}^{NGP} (\mathbf{p}(\mathbf{x}_{gp})\mathbf{a}_k - \boldsymbol{\sigma}_k^h(\mathbf{x}_{gp}))^2 \tag{3.10}$$

yielding a linear system of equations per component $\mathbf{M}\mathbf{a}_k = \mathbf{H}_k$, where NGP indicates the number of integration (sample) points and:

$$\begin{aligned} \mathbf{M} &= \sum_{l}^{NGP} \mathbf{p}^T(\mathbf{x}_l)\mathbf{p}(\mathbf{x}_l) \\ \mathbf{H}_k &= \sum_{l}^{NGP} \mathbf{p}^T(\mathbf{x}_l)\boldsymbol{\sigma}_k^h(\mathbf{x}_l) \end{aligned} \tag{3.11}$$

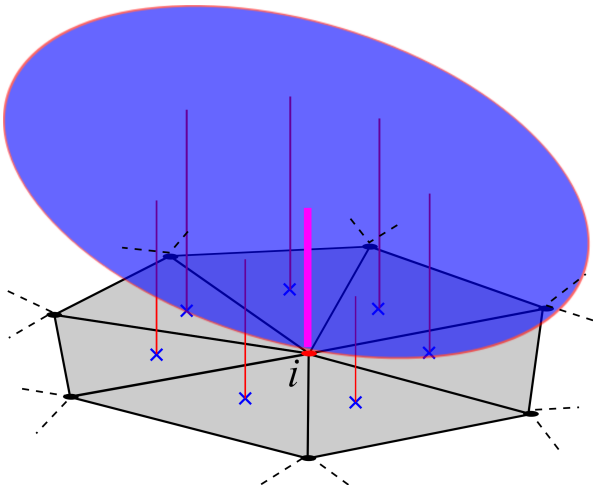


Figure 3.3: Representation of the least squares fitted polynomial surface. The pink line represents the stress value at the assembly node, the only one that is retained in the stantard SPR.

The recovered stresses $\hat{\boldsymbol{\sigma}}_i^*$ at each node are obtained particularizing these surfaces at the assembly node. Finally, following the same process at each assembly node of the mesh we end up with a nodal stress representation.

It can be noticed that matrix **M** contains coordinates of the elements within the patch. Therefore, this matrix will be equal for all patches with a common location of the integration points. In the cgFEM framework, due to the topological features of Cartesian grids and the maximum level difference between adjacent elements, there are only 19 possible configurations of patches composed by internal elements. Figure 3.4 represents all these possible configurations of internal patches.

It was shown in section 2.2.2 that multiple systems of equations can be vectorized if the coefficient matrix is common for all of them. This improvement has been performed in the SPR routine, vectorizing the construction of the vectors $\mathbf{H}_k$ corresponding to all patches with the same shape and resolving all these patches at once. Performance tests have reported up to a 40% improvement in the SPR technique function.

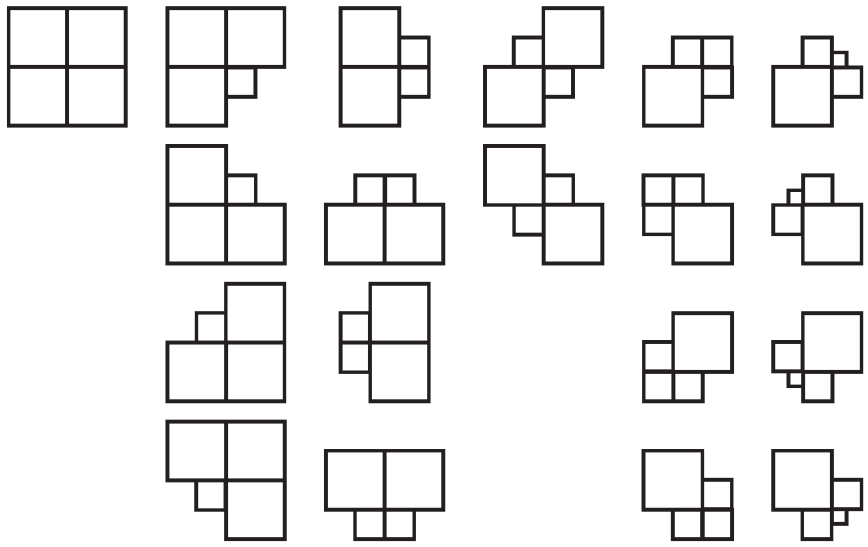


Figure 3.4: Possible configurations of internal patches in 2D.

The cgFEM code implements the SPR-CD technique [7] developed at the CIMM that provides locally equilibrated stress fields by taking into account the equilibrium and compatibility equations using constrain equations during the process. The subroutines of this more advanced recovery technique have been also implemented using vectorization techniques.

Chapter 4

Overall performance tests

The last part of this Thesis consists of several tests to measure the overall computational cost improvement in cgFEM associated to the code optimization process.

The optimization improvements have been grouped into two main groups, CPU improvements (meaning vectorization) and GPU improvements. Thus, the structure of the performance tests is similar to the one used in section 2.4.4. A first execution of the program without code improvements is run to set the reference time. After that, the CPU vectorization changes are applied in a second run. Finally, GPU computing is activated in the last run of the code.

The changes in the h-adaptive routine are taken into account in the first run that sets the reference time, so the performance improvement due to this new refinement is not considered in these results.

The workstation used to measure all computing times in this Thesis had a Intel Xeon E5-2609 CPU with 16GB RAM memory. The GPU device used for parallel computing was a ASUS GeForce GTX780 with 3GB RAM memory. Although the performance improvement results in this Thesis are expressed as speed-up ratios, it is worth to outline that the CPU and GPU device are decisive parameters that affect the absolute computational cost. In fact, some of the examples shown here have been tested in a different workstation with a Intel i7-4770K CPU and the same RAM memory and GPU, reporting computation times twice as fast as the times shown in this chapter in all cases.

The benchmark problems that are going to be used to test the performance of the cgFEM code are a hollow cylinder under internal pressure, a flywheel under external torque and a medical image-based analysis. At the end of the Chapter a qualitative comparison is made between cgFEM and a commercial FE code, ANSYS® .

4.1 Hollow cylinder under internal pressure

The model of this problem was already used to test the new refinement routine in section 2.1, and is shown again in Figure 4.1 for convenience.

Two different tests have been made using this model, one with linear elements and other with quadratic elements. A h-adaptive analysis was executed for both tests, with the aim to reach non-uniform meshes with a high number of degrees of freedom (dofs). The last iterations of both tests had 760×10^3 dofs for the linear element case and over 1.2×10^6 dofs in the case of quadratic elements. Tables 4.1 and 4.2 show the number of dofs and the estimated error in energy norm of each mesh in the h-adaptive process.

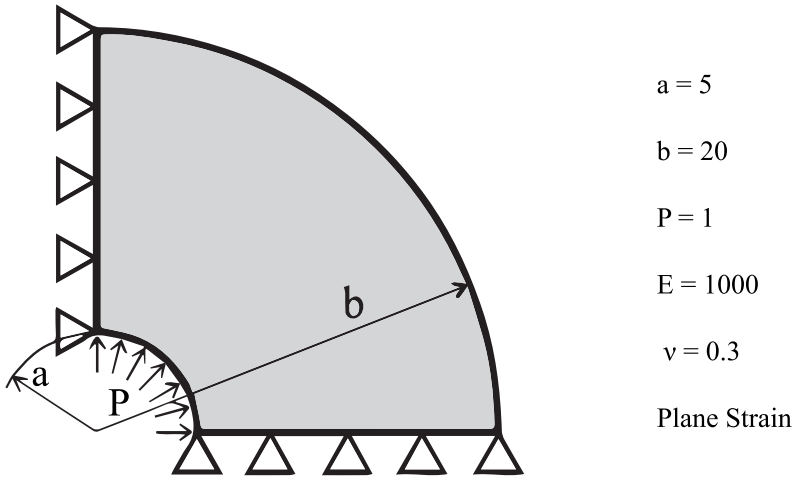


Figure 4.1: Hollow cylinder under internal pressure model.

DOF	Estimated error in energy norm (%)
5542	3.0315
14734	1.0975
106094	0.4017
760206	0.1685

Table 4.1: Hollow cylinder analysis. Linear elements.

Time results for the test with linear elements are shown in Figure 4.2. The three coloured areas at the bars correspond to the three different blocks in which the FEM program was divided in Chapter 3. It can be noticed that the vectorization process has provided an important improvement in the pre-processing and post-processing blocks. The enhanced intersection procedure has led to a 60% time reduction. Moreover, the GPU implementations have an 82% total improvement of the post-processing block with respect to the original code. The overall time results report a 64% reduction in computation time with the usage of GPUs.

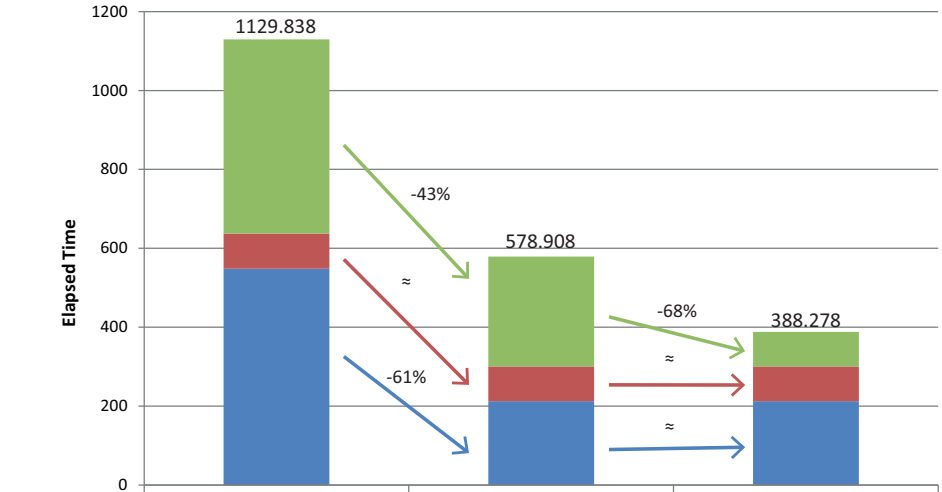
DOF	Estimated error in energy norm (%)
62572	0.0463
72066	0.0129
132098	0.0038
405240	0.0011
1239678	0.0003

Table 4.2: Hollow cylinder analysis. Quadratic elements.

The results of the test for the quadratic element's case are shown in Figure 4.3. On the one hand the overall performance improvement reaches only a 20%, which is a significantly lower reduction compared to the one achieved in the case with linear elements. But, on the other hand, the computation time of the original code with linear elements is similar to the time elapsed in the quadratic element's case, despite doubling the number of dofs.

Quadratic element meshes need less elements to obtain the same amount of dofs than meshes with linear elements. As many operations in cgFEM are performed element-wise, the computational cost in this case increases not with the number dofs but with the number of elements. This explains why the original code had a worse efficiency when using linear elements than when using quadratic ones, because all element-wise operations were performed in sequential loops. By applying vectorization techniques the dependence of the computational cost with the number of elements has decreased, and now the effectiveness of cgFEM is similar with both types of element.

The improvement of the GPU computed tasks in this case is a 38% with respect to the original code. The decrease in this speed-up is due to the same reason of the lower overall performance. GPU implementations are focused on calculations performed for internal elements, while contour elements' calculations such as energy norm or SPR stress fields are still not vectorized. Meshes with quadratic elements have also a lower ratio between internal and



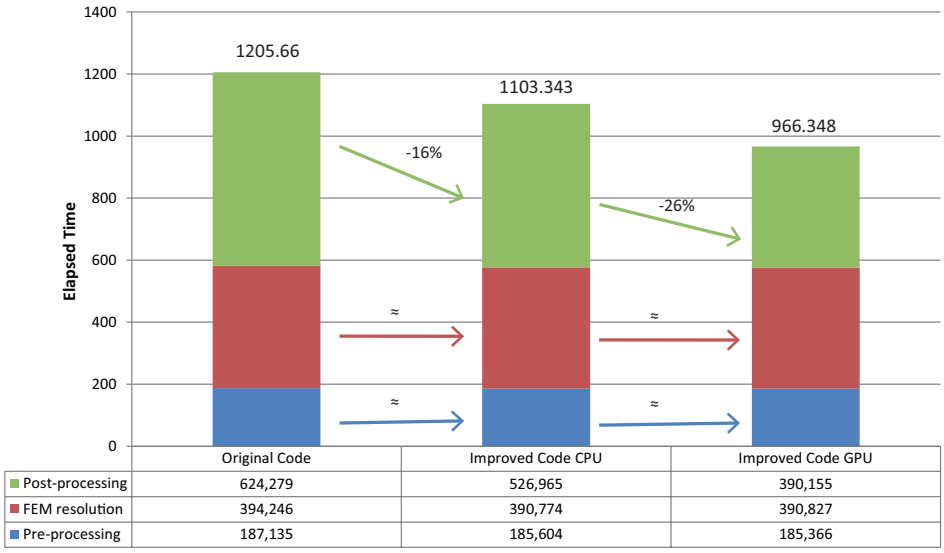
	Original Code	Improved Code CPU	Improved Code GPU
Post-processing	492,367	278,36	87,745
FEM resolution	88,977	87,881	87,871
Pre-processing	548,494	212,667	212,662

Figure 4.2: Hollow cylinder h-adaptive analysis. Time results for linear elements.

boundary elements, so the improvement achieved with GPU computation is extended to a lower number of elements.

4.2 Flywheel under external torque

The next problem used to test cgFEM is the model of a flywheel subject to an external torque. Figure 4.4 shows the model as it is introduced in the program. This model has a higher ratio of contour per surface, so all calculation meshes will have more boundary elements, and the intersection process will have an increased computational cost.



	Original Code	Improved Code CPU	Improved Code GPU
Post-processing	624,279	526,965	390,155
FEM resolution	394,246	390,774	390,827
Pre-processing	187,135	185,604	185,366

Figure 4.3: Hollow cylinder h-adaptive analysis. Time results for quadratic elements.

Again, both tests with linear and quadratic elements have been run. Tables 4.3 and 4.4 show the data of all the iterations in the analyses. It is worth to highlight that this time the refinement parameters have been changed so that the linear test had almost 3 million dofs in the last iteration. This is at the moment the biggest analysis that cgFEM has been able to solve without running out of RAM memory.

DOF	Estimated error in energy norm (%)
18488	8.711
127834	1.619
2957774	0.294

Table 4.3: Flywheel analysis. Linear elements.

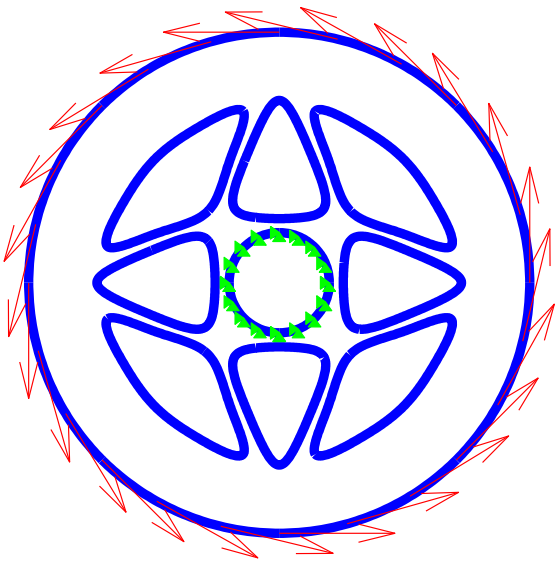


Figure 4.4: Flywheel under external torque model.

The time results shown in Figure 4.5 confirm that the old intersection routine becomes the main bottleneck in the program when the calculation mesh has a large amount of elements. The intersection procedure in this test consumes about 92% of the total analysis time with the original code. This bottleneck is drastically reduced with the proposed vectorized solution, reducing 85% of the whole computation time. GPU improvements are hidden because of the huge computational cost of the intersections in the original code. This was the main reason to include the new refinement algorithm in the original code, because the results of the refinement improvement would cover all other optimizations. Nevertheless, the new GPU calculations decrease 55% of the elapsed time of

DOF	Estimated error in energy norm (%)
15800	10.795
21414	2.610
39536	0.587
97238	0.148
296918	0.039
986956	0.012

Table 4.4: Flywheel analysis. Quadratic elements.

the post-processing block with respect to the original code, reaching an overall 86% speed-up of the program.

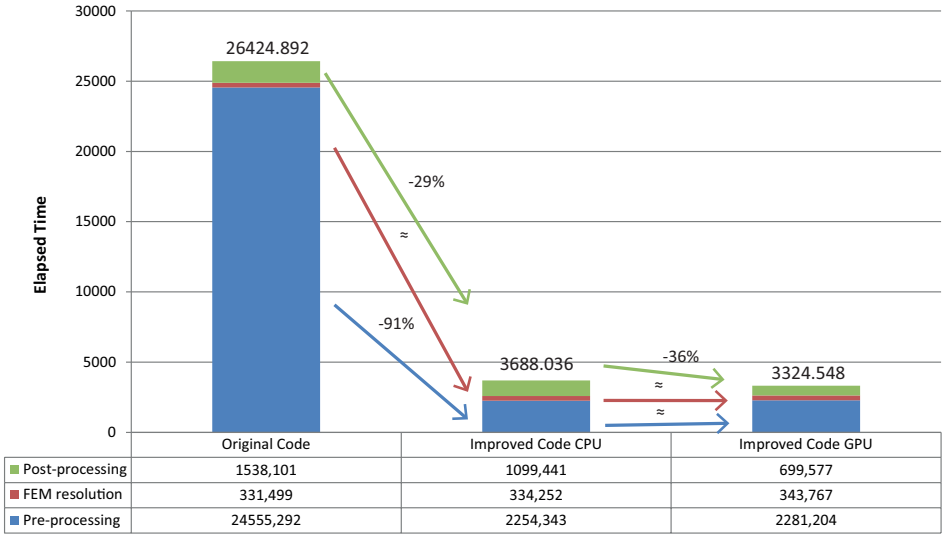
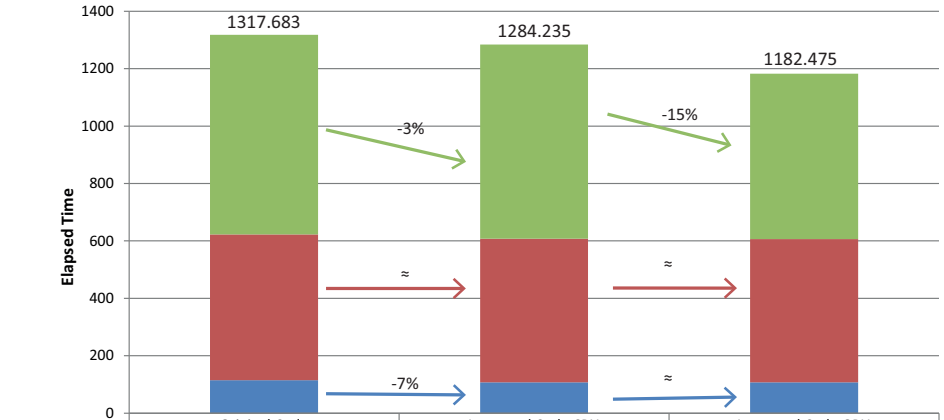


Figure 4.5: Flywheel h-adaptive analysis. Time results for linear elements.

The results of the test with quadratic elements lead to the same conclusions obtained in the cylinder test with quadratic elements. In this example the improvement is even lower (overall 10%). This is because, as it has been said, this model has a higher proportion of boundary elements in the calculation

meshes, and the vectorization of the calculations for internal elements has less impact on the total computation time. One of the future tasks is to extend some optimizations to the boundary element calculations (such as error in energy norm or stress). However, this improvements also imply some changes in the data structure, like the one exposed in section 3.2.1.



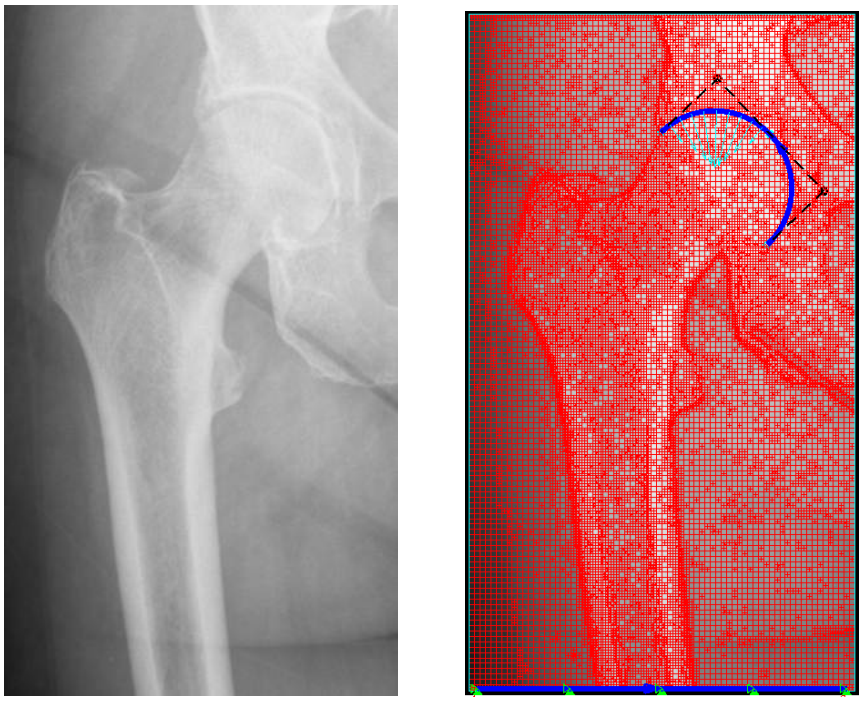
	Original Code	Improved Code CPU	Improved Code GPU
Post-processing	694,937	676,224	576,039
FEM resolution	508,083	500,823	499,268
Pre-processing	114,663	107,188	107,168

Figure 4.6: Flywheel *h*-adaptive analysis. Time results for quadratic elements.

4.3 Femur analysis based on a medical image

The final test in this Chapter consists of a medical image-based analysis. This example will check the speed-up of the specific improvements made for this special feature of cgFEM, particularly those made on the FEM resolution block.

Figure 4.7a shows the medical image on which the problem is based. The boundary conditions are applied through curves defined by the user. In this case there is a straight line where the Dirichlet constrain is imposed and a pressure is applied on the arc on the top.

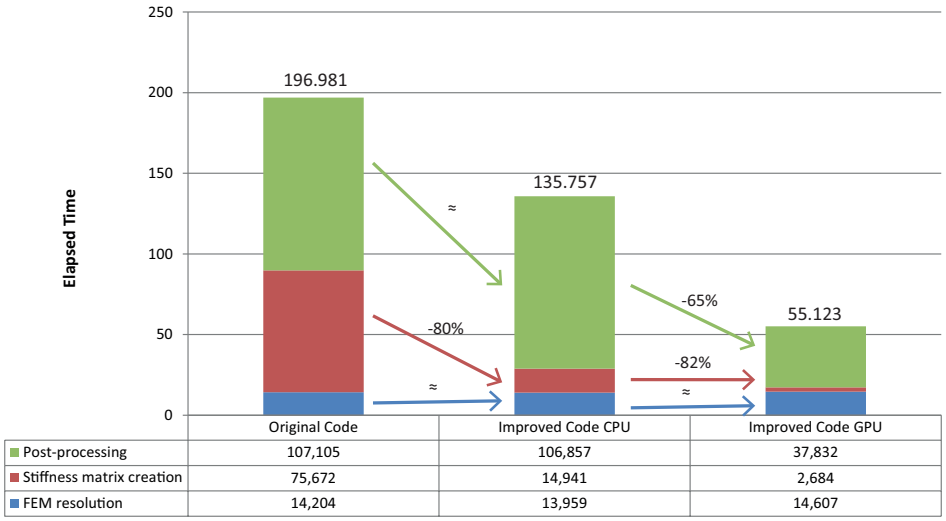


(a) (b)

Figure 4.7: Medical image of a femur and calculation mesh for the analysis.

There is no contour in this model to intersect with the Cartesian grid pile, and the analysis consists of one only iteration because the h-adaptive refinement based on the quality of the solution has not yet been implemented for medical image analyses. Therefore the pre-processing block is negligible in terms of computational cost, and only the resolution and the post-processing block are considered. The element stiffness matrices procedure is shown here apart

from the rest of the resolution block, as major changes were performed in this routine. The results of the test are shown in Figure 4.8.



	Original Code	Improved Code CPU	Improved Code GPU
Post-processing	107,105	106,857	37,832
Stiffness matrix creation	75,672	14,941	2,684
FEM resolution	14,204	13,959	14,607

Figure 4.8: Time results for a medical image-based analysis.

Some of the benefits of the Cartesian grids regarding the computational efficiency disappear when it comes to an analysis based on a medical image. This is because all elements have different material properties. Therefore, the element characteristics cannot be linked to the *reference element's* characteristics, as it happened with the internal elements in standard cgFEM. In this case all elements have to be treated as if they were "boundary elements", meaning that all characteristics have to be calculated element-wise. This explains why the original code test consumed almost 200 seconds to solve a 86×10^3 thousand dofs problem, which is a small amount of dofs considering the size of the previous tests.

The vectorization of the element stiffness matrices report a 80% reduction on the time consumed by this task. This result demonstrates that *struct* type

variables are inefficient in the access of the data, and a change of this data structure could provide an improvement on the computational cost of the program.

The GPU implementations for medical image analysis procedures involve all elements of the mesh. All element stiffness matrices, stresses and energy norms can be calculated using GPU computation, so the impact of this improvements is higher than in the other type of tests. GPU computing gives in this case a total 71% time reduction, showing the potential of this techniques for high performance computation.

4.4 Performance comparison with ANSYS®

The tests made in this section do not pretend to be an accurate comparison between both codes, but a qualitative measure about where is cgFEM in terms of computational cost with respect to commercial FE codes.

There were some tests performed in [7] that compared the efficiency of the old cgFEM code with ANSYS® release 12.1. The results showed that cgFEM could be up to two times faster than ANSYS® for fine meshes with a high amount of degrees of freedom.

In this Thesis the new cgFEM code is compared with ANSYS® Academic Research Mechanical, release 14.5. There has been a remarkable improvement in the efficiency of ANSYS® through this time, so this test will show if cgFEM is still as efficient as the new code of ANSYS®.

The problem used to compare the performance of both codes was the hollow cylinder used in section 4.1. All analyses were performed using uniform meshes, because the mesh h-adaptive refinement procedures are not robust enough in ANSYS®. Moreover, it is worth to remind that the post-processing procedure used in cgFEM is considerably more accurate than the post-processing routine of ANSYS®, as it is explained in [7].

The tests were performed using only one CPU core. In the case of cgFEM , both runs, with and without GPU computing, were executed. However, there is no possibility to enable GPU computing on ANSYS® using the same GPU devices. ANSYS® only allows NVIDIA® Tesla and Quadro GPU devices, which are more powerful but considerably more expensive as well. Table 4.5 shows the results of the tests performed.

Element type	DOF	cgFEM time with GPU	cgFEM time without GPU
Linear	322446	156	239
Quadratic	965470	278	338

Element type	DOF	ANSYS® time
Linear (Plane42)	382128	286
Quadratic (Plane82)	1181468	335

Table 4.5: Comparison between ANSYS® and cgFEM

Some previous tests with a small amount of dofs report that ANSYS® is quite more effective than cgFEM when meshes are coarse. But there is little difference when analyzing fine meshes with a large amount of dofs. Results show that cgFEM is still, at least, as efficient as ANSYS® when using fine meshes. As it was said, this comparison only pretends to establish how close is cgFEM to commercial FE codes in a qualitative way. It is worth to remind that cgFEM is implemented in an interpreted language, where as ANSYS® is

implemented in a compiled language. Therefore, it seems that the cgFEM code has a lot of potential regarding the reduction of the computational cost.

Chapter 5

Conclusions

The results shown in this Thesis demonstrate that the efficiency of the cgFEM code in terms of computational cost has been enhanced by improving all the different areas of the program: pre-processing, resolution and post-processing routines. It is worth to remark that the effectiveness of cgFEM code for medical image-based analyses has undergone a significant improvement thanks to the usage of GPU computing.

The procedures shown in this Thesis have settled the basis for an efficient programming of future developments in the Department of Mechanical and Materials Engineering, and more specifically for the 3D version of cgFEM , FEAVox. Some optimization changes have already been performed on the FEAVox code, as well as the introduction of GPU computing.

It has been proven in this Thesis that the hierarchical structure of Cartesian grids existing in cgFEM is an appropriate environment to develop this High Performance Computing technologies. Moreover, thanks to the research made in this Thesis, the data structure of FEAVox has been changed from the original structure in cgFEM to ease the application of vectorization and GPU computing techniques.

Finally, some comparison tests with the commercial FE code ANSYS® show that the cgFEM code, despite being implemented in an interpreted language, is still in the same order of magnitude regarding computational cost than commercial compiled codes.

Bibliography

[1] J A Cottrell, T J R Hughes, and Y Bazilevs. *Isogeometric Analysis: Toward Integration of CAD and FEA*. Wiley, 1st edition, 2009.

[2] L Giovannelli, O Marco, J M Navarro, and E Giner. Direct creation of finite element models from medical images using Cartesian grids. *VipImage 2013, IV Eccomas thematic conference on computational vision and medical image processing*, 2013.

[3] L Giovannelli, J J Ródenas, J M Navarro-Jimenez, and M Tur. Element stiffness matrix integration in image-based Cartesian Grid Finite Element Method. *CompIMAGE 2014, Computational modeling of objects presented in images. Paper accepted.*, 2014.

[4] The Mathworks Inc. MATLAB Documentation Center. http://www.mathworks.com/help/matlab/, 1994-2014.

[5] K Karimi, NG Dickson, and Firas Hamze. A performance comparison of CUDA and OpenCL. *eprint arXiv:1005.2581*, (1), 2010.

[6] F Magoulès, A C Ahamed, and R Putanowicz. Auto-tuned Krylov Methods on Cluster of Graphics Processing Unit. *International Journal of Computer Mathematics*, (June 2014):1–30, June 2014.

[7] E Nadal. *Cartesian grid FEM (cgFEM): High performance h-adaptive FE analysis with efficient error control. Application to structural shape optimization.* PhD thesis, Universidad Politécnica de Valencia, 2014.

[8] E Nadal and O A Gonz. Error estimation and error bounding in quantities of interest based on equilibrated recovered displacement fields. (Eccomas), 2012.

[9] E Nadal, J J Ródenas, J Albelda, M Tur, J E Tarancón, and F J Fuenmayor. Efficient Finite Element Methodology Based on Cartesian Grids: Application to Structural Shape Optimization. *Abstract and Applied Analysis*, 2013:1–19, 2013.

[10] J M Navarro-Jiménez. Obtención de modelos de elementos finitos con mallados cartesianos independientes de la geometría a partir de imágenes médicas y su uso en la simulación de interacción entre prótesis y tejidos vivos. *Proyecto Final de Carrera. ETSII-UPV.*, 2013.

[11] J J Ródenas, C Corral, J Albelda, J Mas, and C Adam. Solver directo de división recursiva en subdominios basado en un programa de refinamiento h adaptable de estructura jerárquica. *Métodos Numéricos e Computacionais em Engenharia. CMNE CILAMCE*, 2007.

[12] J J Ródenas, L Giovannelli, E Nadal, J M Navarro, and M Tur. Creation of patient specific finite element models of bone-prosthesis—simulation of the effect of future implants. *VipImage 2013, IV Eccomas thematic conference on computational vision and medical image processing*, 2013.

[13] J J Ródenas, J E Tarancón, J Albelda, A Roda, and F J Fuenmayor. Hierarchical properties in elements obtained by subdivision: A hierarquical h-adaptivity program. In Pedro Díez and Nils-Erik Wiberg, editors, *Adaptive Modeling and Simulation 2005*, 2005.

[14] O C Zienkiewicz and J Z Zhu. A simple error estimator and adaptive procedure for practical engineering analysis. *International Journal for Numerical Methods in Engineering*, 24(2):337–357, 1987.

[15] O C Zienkiewicz and J Z Zhu. The superconvergent patch recovery and a posteriori error estimates. Part 1: The recovery technique. *International Journal for Numerical Methods in Engineering*, 33(7):1331–1364, 1992.