HIGH ORDER X-FEM : IMPROVING THE NUMERICAL INTEGRATION

Santiago Giraldo

ÉCOLE CENTRALE DE NANTES
EM MSc IN COMPUTATIONAL MECHANICS
NANTES
2011

HIGH ORDER X-FEM : IMPROVING THE NUMERICAL INTEGRATION

Santiago Giraldo

Thesis project for the degree of Master in Science

Advisor
Dr. Nicolas Chevaugeon

ÉCOLE CENTRALE DE NANTES
EM MSc IN COMPUTATIONAL MECHANICS
NANTES
2011

Grade

President of the Jury

Jury

Jury

Nantes, June 20 2011

Acknowledgments

To my family, their support has brought me here. I hope that I can repay you someday.

I would like to thank my advisor, Dr. Nicolas Chevaugeon, his guidance helped me move forward in this project.

To the European Commission who granted me the scholarship which made this experience possible.

To all the people across my studies which have helped to keep science fun.

To all my friends, wherever they are, a list falls incredibly short, then I feel blessed in many ways for this...

Abstract

An improvement to the integration scheme under the eXtended Finite Element Method is presented. The problem to be treated arrives when the order of the approximation functions are increased to accelerate convergence. A finer sub-mesh is defined to perform a more accurate approximation of the level set (defining geometrical characteristics such as holes, cracks or material interfaces) while preserving the computational mesh intact. The integration over the resulting elements is performed over the finer sub-mesh, the higher the order of the approximation functions, the increased number of gauss points required to do it.

The integration becomes operation intensive, increasing computational times. In order to make higher order approximation viable and interesting to pursue, special care must be taken in the integration procedure. Parallel computing models and strategies are explored and advantage of the possibilities of processing data on commodity graphic cards is presented for the current code suite.

Examples to demonstrate how the computational load can be alleviated with the use of GPU are presented. The driving concept is to execute parallelizable sections of code where possible to accelerate the solution of problems, maintaining accuracy or improving it. Future work for the challenges encountered and new strategies are suggested as well.

Contents

Objectives **17**
Specific Objectives. 17

1 Introduction **19**

2 Theoretical Framework **21**
2.1 Linear Elasticity 21
2.1.1 Kinematic equations 22
2.1.2 Internal Equilibrium 22
2.1.3 Constitutive equations 23
2.2 Finite Element Approximation 24
2.2.1 Discretization of the Domain Ω 26
2.3 An enriched Finite Element Method (X-FEM) 28
2.3.1 eXtended Finite Element Method 29
2.3.2 High Order X-FEM 29
2.3.3 Numerical Integration 30
2.4 General Purpose GPU Computing 30
2.5 GPU Computing Programming Language 34
2.6 Application possibilities of GPU in X-FEM 35

3 Implementations and Tests **39**
3.1 Cuda functions 39
3.1.1 Checking the Device 39
3.1.2 Checking Operability 40
3.2 Using CUBLAS Library 42
3.2.1 Interface to CUBLAS 43
3.3 Using OS X BLAS 48

4 Results **51**
4.1 Testing BLAS routines 51

5 Conclusions **57**

6 Perspectives **59**
6.1 Future Work 59

A Porting X-Fem Suite to Apple OS X **61**
A.1 Basic Requirements 61
A.2 Build Process 62
A.3 Porting The Project To Apple's XCode 66
A.3.1 Setup of the project in XCode: 67

B Setting Up CUDA In OS X **69**
B.1 Set Up Of CUDA On OS X 69
B.2 Adding CUDA Support To XCode 70

C Including CUDA in the X-Fem Suite **72**

D Annex of results table **75**

List of Tables

2.1 Characteristics of the numerical integration 35
2.2 Results of numerical integration for FE example 37

3.1 Hardware Characteristics 41
3.2 Vector addition test (N=5e6) 41
3.3 Vector addition test (N=1e7) 42
3.4 Performance for BLAS implementations 1 48
3.5 Performance for BLAS implementations 2 49

List of Figures

2.1 A deformable body 22
2.2 Balance of forces in the x direction for a differential element 23
2.3 Numerical integration: partition creation 30
2.4 Numerical integration with High order X-FEM -Element associated partition- 31
2.5 Scalability of GPU computation 32
2.6 Thread hierarchy 33
2.7 Programming model and execution 34

4.1 Array Fill time comparison for data preparation 52
4.2 Array Fill Gflops comparison for data preparation 53
4.3 Comparison of times for sgemm BLAS call 53
4.4 Comparison of Gflops for sgemm BLAS call 54

A.1 Necessary source code folder structure 63
A.2 Solver folder structure 64
A.3 SolverInterfaces folder structure 65
A.4 Trellis_darwin folder structure 66

B.1 **deviceQuery** output 70

Objectives

Objective.

To improve the integration algorithm executed by the X-Fem suite, improve computational times using parallelization strategies.

Specific Objectives.

1. To undertake a literature review on the subjects of high order X-Fem and applications of GPU computing to solid mechanics.
2. Fit the code to function under OS X with support for NVidia CUDA and OpenMP.
3. Modify the integration algorithm and explore parallelization options, taking advantage of shared geometrical information used for several integration steps.

1. Introduction

The eXtended Finite Element Method appears as an alternative to classic Finite Element to approach the problem of growing mesh sizes. Growing complexity of geometries makes it more challenging to deliver accurate result in engineering analysis nowadays. Current problems driven by industry's interest demand higher computational resources. High Performance Computing, which can provide the computational throughput to solve close to reality, close to real time problems is still an exclusive option.

HPC systems are bound by current processors, with current clock speed ceilings that appear to have stalled. Efforts have turned from clock speed to multicore processors, having reached as well a point where in current architecture is not useful to increase the number of cores. Then the efforts have gone from improving the memory transfer bandwidth to increasing the number of processors available in a server, even in the case of shared memory (one of the expensive options) Moore's law proves to be holding back this systems for more demanding problems being solved in less time.

Parallel computing strategies have been widely implemented were the problems allow so, and more optimized language instructions and architecture are bringing new functionality to this models. But still the number of processors would be holding back more accelerated developments. Adding more processors into and HPC machine is not an option most of the times, usually they are closed systems, being hardware bound by manufacturers. In the case of Blade-type clusters, adding more processors, memory and disk still is a matter of several thousands of dollars.

Modeling problems in engineering, where geometries are becoming even more complex, constraint the accuracy of possible simulations in great part to proper meshing. Having to conform to physical surfaces becomes a starting point for meshes of substantial size. X-FEM using Level Sets to define the geometrical and topological characteristics appears as an interesting option to tackle the computationally intensive process. Another strategy to improve the performance of the method is to use higher order approximation functions, in order to accelerate convergence maintaining acceptable ranges of accuracy [1].

The construction of the stiffness matrix yielded by the numerical integration process must be done with proper care. Increasing the degree of the approximation functions demands a higher number of integration (Gauss) points, this translates as an increased number of operations and data stored by finite element. On the provided routines, integration cells are created from the sub-mesh related to the Level-set. The geometrical

information is shared with the computational mesh, but the information required to integrate properties over each cell (up to the defined level of refinement) comes from the sub-mesh and the level-set. From this cell partition is where parallel computing strategies can profit.

After the advent of new GPUs with more horse-power, the advance in architecture and the development of a firm basis to enable users to develop applications to harness the power of GPUs for their applications, scientific computing capabilities became available for commodity graphic cards. The amount of cores available per multiprocessor and the amount of multiprocessors in a single GPU has increased dramatically over the last few years. This meant a steep climb in the number of floating point operations per second that it can be performed, aiming to desired TFLOP range. As well, efforts to lower transfer rates and increase the communications bandwidth has accompanied the evolution of GPUs. Nowadays, past the implementation of double precision operations -demand driven by the community- the power of scientific computing has become available to personal computers. Then HPC systems that involve several stacked high end GPUs (like Tesla WS) are available for a few thousands of dollars (in comparison to traditional HPC systems).

Given that the implementation of routines for the numerical integration has already been pursued, the aim of the manuscript is to provide founding basis to help the development of the x-FEM code at ECN that has been under development many years now.

2. Theoretical Framework

The topics reviewed for this document span along the Finite Element Method (FEM), the Extended Finite Element Method (X-FEM), High order X-FEM and GPU computing. Introductions to these topics are presented for clarity purposes.

In solid mechanics one widely used assumption that simplifies greatly the treatment of the equations involved is the one of linear elasticity. The outline and illustration of the methods derived to solve the differential equations for such problem, can profit from an introduction to linear elasticity.

2.1 Linear Elasticity

Linear elasticity is a common problem in solid mechanics, many practical structures behave as linearly elastic bodies -from aircraft to baby strollers-. Linear elasticity deals with bodies subjected to forces and their deformations. This physical phenomenon is governed by Partial Differential Equations (PDE) that describe deformation, internal equilibrium and material properties of the body. Once these equations are described they will lead to the variational formulation of the elasticity problem which is the starting point to obtain a finite element solution, thus further an extended finite element solution.

Let the solid body be represented by Ω. Then Ω refers to the domain on Figure 2.1 with its boundary Γ. The concepts presented deal with both $\mathbf{R}^2$ and $\mathbf{R}^3$ cases (two and three dimensions). However, this work is focused initially on two dimensional problems ($\mathbf{R}^2$).

Let there be a set $\Omega \subset \mathbf{R}^n$. A point $\mathbf{x} \in \mathbf{\Omega}$ is an internal point when in the neighborhood of $\mathbf{x}$ contains only points that belong to Ω. A point $\mathbf{y}$ is a boundary point $(\mathbf{y} \in \mathbf{\Gamma})$ if every neighborhood of $\mathbf{y}$ contains points that belong to Ω and some that do not belong to Ω. These concepts are illustrated in figure 2.1.

When a solid body (represented by Ω) is subjected to external forces F_i, each point of the body experiences a displacement u_i. The relative positions between the points of unloaded and loaded structure is called deformation. The aim of elasticity theory is to relate the forces experienced by the deformable body with the displacements caused by the loads when a certain behavior (elastic behavior) is assumed to be followed by the body [2].

The following section are introductory to the basic equations of elasticity. For more

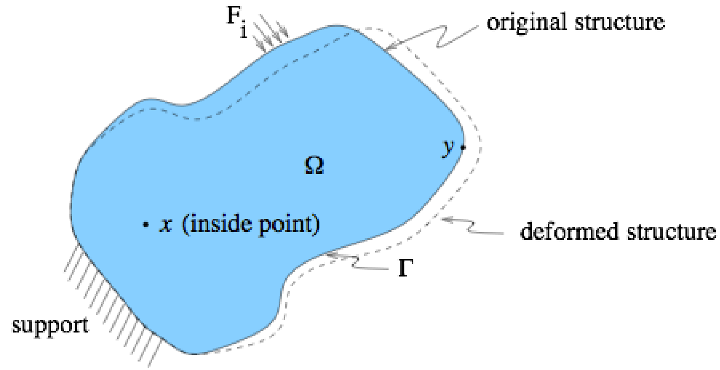


Figure 2.1: A deformable body

detail refer to the work of Kikuchi [3] or the book by Zienkiewicz [4].

2.1.1 Kinematic equations

The relative changes of position of the points of a body due to an applied load can be related to the changes of shape and volume. If the deformation of a body is infinitesimal, that is $|\partial u_i/\partial x_j| \ll 1$, then the relationship between displacement and strain are given by Equation (2.1).

$$\varepsilon_{ij}(\mathbf{u}) = \frac{\mathbf{1}}{\mathbf{2}}\left(\frac{\partial \mathbf{u_i}}{\partial \mathbf{x_j}} + \frac{\partial \mathbf{u_j}}{\partial \mathbf{x_i}}\right) \tag{2.1}$$

The tensor ε_{ij} was introduced by Cauchy for infinitesimal deformations and by Almansi and Hamel when the deformation is finite [5].

2.1.2 Internal Equilibrium

The balance of internal forces along the x axis for a differential volume inside of a deformed body is shown in Figure 2.2. In the same way, if the balance of forces was applied along the y and z axis, the equation of internal force equilibrium can be obtained as Equation (2.2).

$$\frac{\partial}{\partial x_j}\sigma_{ji} + \rho f_i = 0 \tag{2.2}$$

where:

σ_{ij} is the component of the stress tensor and represents the projection in the i direction of a traction on a plane with normal j. (i and j represent the direction of the x, y or z axis).

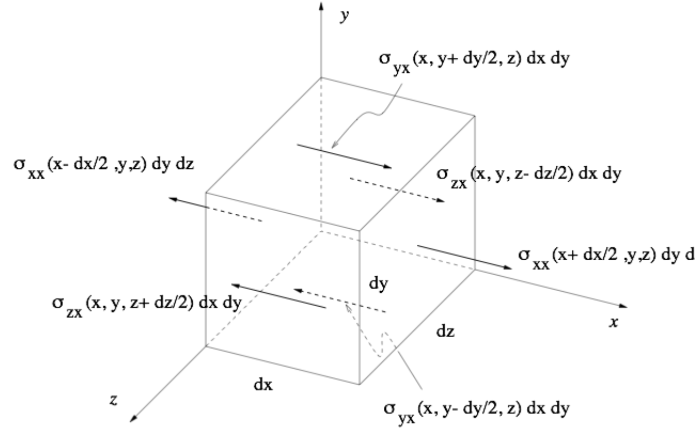


Figure 2.2: Balance of forces in the x direction for a differential element

ρ is the density of the body.

f_i represents the component i of the volume forces acting on the differential element.

The result of imposing moment equilibrium is the symmetry of the stress tensor which is given by Equation (2.3)

$$\sigma_{ij} = \sigma_{ji} \tag{2.3}$$

2.1.3 Constitutive equations

The constitutive equations describe the material characteristic of a solid body. For a linear elastic material the stress-strain relationship is given by Equation (2.4)

$$\sigma_{ij} = E_{ijkl}\,\varepsilon_{kl} \tag{2.4}$$

where $\mathbf{E}$ is a fourth order tensor of constant values that represent the physical properties of the material.

$\mathbf{E}$ is known as the elasticity tensor. In a three dimensional body subindexes i, j, k and l take range from 1 to 3 (x, y and z axis) and the elasticity tensor $\mathbf{E}$ comprises 81 constants. This number can be reduced by applying considerations of different kind, such as: symmetry of the strain tensor and existence of the strain energy density. The simplest form of the tensor $\mathbf{E}$ is found for isotropic materials were Equation (2.4) becomes:

$$\sigma_{ij} = \lambda\,\varepsilon_{kk}\,\delta_{ij} + 2\,\mu\,\varepsilon_{ij} \tag{2.5}$$

where λ and μ are the Lamé constants related to Young's modulus E and Poisson's ratio ν by

$$\lambda = \frac{E\nu}{(1-2\nu)(1+\nu)}, \qquad \mu = \frac{E}{2(1+\nu)} \tag{2.6}$$

2.2 Finite Element Approximation

The differential equations governing the behavior of a linear elastic body were described in the previous section. The Finite Element Analysis is a numerical method developed to approximate the solution to such equations.

The stationary boundary value problem for a linear elastic body is given by the equilibrium (2.2), kinematic (2.1) and constitutive (2.4) equations and is summarized by Equation (2.7) with boundary conditions given by equations (2.8).

$$\begin{aligned} -\frac{\partial}{\partial x_j}\sigma_{ji} &= \rho f_i \\ \varepsilon_{ij} &= 1/2\left(\frac{\partial u_i}{\partial x_j} + \frac{\partial u_j}{\partial x_i}\right) \\ \sigma_{ij} &= E_{ijkl}\,\varepsilon_{kl} \end{aligned} \tag{2.7}$$

and the boundary conditions

$$\begin{aligned} u_i &= g_i \quad on \quad \Gamma_{i1} \\ \sigma_{ji} n_j &= h_i \quad on \quad \Gamma_{i2} \end{aligned} \tag{2.8}$$

where

$\Gamma = \Gamma_{i1} \cup \Gamma_{i2}$ for each i = 1, 2, 3.

Γ_{i1} denotes the boundary where the i component of the displacement is known

Γ_{i2} denotes the boundary where the i component of traction is known. It is assumed that boundaries Γ_{i1} and Γ_{i2} are mutually disjointed.

In PDE literature the first condition is called the Dirichlet boundary condition and corresponds to the displacement boundary condition. That is, if u_i is the i component of the displacement, then, along the boundary Γ_{i1}, u_i is equal to g_i. When the body is fixed over Γ_{i1}, g_i is equal to zero. That is, if no movement is allowed in any direction then $g_i = 0$ for each i = 1, 2, 3.

The second condition corresponds to the i component of traction h_i on boundary Γ_{i2}. The traction represents the force per unit area or volume normal to the surface. In case of a free boundary $h_i = 0$.

Another kind of boundary condition in elasticity problems corresponds to the spring support. For simplicity purposes this condition is not considered.

To approximate a solution by the Finite Element Method, a weak form of the boundary problem defined by equations (2.7) and (2.8) is necessary [6]. This form can be obtained by multiplying the first of equations (2.7) by a virtual displacement $\mathbf{fv}$ and integrate over the domain Ω. The resulting term is integrated by parts to yield the terms of Equation (2.9).

$$\int_{\Omega} \sigma_{ji}(\mathbf{u})\mathbf{v_{i,j}d\Omega} - \int_{\mathbf{\Gamma}} \sigma_{\mathbf{ji}}(\mathbf{u})\mathbf{n_j v_i d\Gamma} = \int_{\mathbf{\Omega}} \rho \mathbf{f_i v_i d\Omega} \tag{2.9}$$

where $\sigma_{ji}(\mathbf{u})$ represents the component of the stress caused by the actual displacement $\mathbf{u}$.

The first term of Equation (2.9) can be further simplified by applying the symmetry of the stress tensors as follows

$$\begin{aligned}
\sigma_{ji}(\mathbf{u})\mathbf{v_{i,j}} &= \frac{1}{2}\left(\sigma_{ij} + \sigma_{ji}\right) v_{i,j} && (2.10)\\
&= \frac{1}{2}\sigma_{ij}\, v_{i,j} + \frac{1}{2}\sigma_{ij}\, v_{j,i} && (2.11)\\
&= \sigma_{ij}\,\frac{1}{2}\left(v_{i,j} + v_{j,i}\right) && (2.12)
\end{aligned}$$

And according to the definition of strain in Equation (2.1)

$$\varepsilon_{ij}(\mathbf{v}) = \mathbf{\frac{1}{2}}\left(\mathbf{v_{i,j}} + \mathbf{v_{j,i}}\right) \tag{2.13}$$

then

$$\sigma_{ji}(\mathbf{u})\mathbf{v_{i,j}} = \sigma_{\mathbf{ij}}(\mathbf{u})\varepsilon_{\mathbf{ij}}(\mathbf{v}) \tag{2.14}$$

replacing this result into Equation (2.9) and combining this with boundary conditions from Equation (2.8), the weak form of the boundary problem is obtained as follows:

$$\begin{gathered}
u_i = g_i \quad \text{on} \quad \Gamma_{i1}\\
\int_{\Omega} \sigma_{ij}(\mathbf{u})\varepsilon_{\mathbf{ij}}(\mathbf{v})\mathbf{d\Omega} = \int_{\mathbf{\Omega}} \rho \mathbf{f_i v_i d\Omega} + \int_{\mathbf{\Gamma_{i2}}} \mathbf{h_i v_i d\Gamma}\\
\forall \mathbf{v} \ni \mathbf{v_i} = \mathbf{0} \quad \text{on} \quad \mathbf{\Gamma_{i1}}
\end{gathered} \tag{2.15}$$

The LHS of Equation (2.15), $\int_{\Omega} \sigma_{ij}(\mathbf{u})\varepsilon_{\mathbf{ij}}(\mathbf{v})\mathbf{d\Omega}$, represent the internal work done due to a virtual displacement $\mathbf{v}$ from the equilibrium configuration defined by the displacement field $\mathbf{u}$. The first term of the right hand side $\int_{\Omega} \rho f_i v_i d\Omega$ is the *virtual work* produced by the volume force ρf_i. The second term $\int_{\Gamma_{i2}} h_i v_i d\Gamma$ is the *virtual work* by the boundary traction h_i. As such, the weak form (2.15) is the *principle of virtual work* in solid mechanics.

In contracted notation, the weak form of the boundary problem, Equation (2.15), reads:

$$\int_{\Omega} \sigma_r(\mathbf{u})\varepsilon_{\mathbf{r}}(\mathbf{v})\,\mathbf{d\Omega} = \int_{\mathbf{\Omega}} \rho \mathbf{f_i v_i}\,\mathbf{d\Omega} + \int_{\mathbf{\Gamma_{i2}}} \mathbf{h_i v_i}\,\mathbf{d\Gamma} \tag{2.16}$$

For linear elastic materials the stress can be expressed as a linear combination of the strain, Equation (2.4). Therefore, the weak form for linear elastic materials becomes:

$$\begin{gathered} u_i = g_i \quad \text{on} \quad \Gamma_{i1} \\ \int_{\Omega} C_{rs}\varepsilon_s(\mathbf{u})\varepsilon_{\mathbf{r}}(\mathbf{v})\,\mathbf{d\Omega} = \int_{\mathbf{\Omega}} \rho \mathbf{f_i v_i}\,\mathbf{d\Omega} + \int_{\mathbf{\Gamma_{i2}}} \mathbf{h_i v_i}\,\mathbf{d\Gamma} \\ \forall \mathbf{v} \ni \mathbf{v_i} = \mathbf{0} \quad \text{on} \quad \mathbf{\Gamma_{i1}} \end{gathered} \tag{2.17}$$

This equation is usually referred as the finite element equation and is the starting point to obtain the finite element solution.

2.2.1 Discretization of the Domain Ω

In order to solve Equation (2.17) by the Finite Element Method, the domain Ω is discretized into a set of elements, each of them with shape functions $\{N_\alpha(\mathbf{x})\}$ that must be defined. Shape functions describe how the displacement $\mathbf{u}(\mathbf{x})$ is interpolated from the values of the displacement ($\mathbf{u}^\alpha$) at the nodes of the elements.

The displacement $\mathbf{u}$ and the virtual displacement $\mathbf{v}$ can be approximated by linear combinations of their node values and the shape functions.

$$\mathbf{u}(\mathbf{x}) = u_j^\alpha N_\alpha(\mathbf{x})\,\mathbf{e_j} \tag{2.18}$$

$$\mathbf{v}(\mathbf{x}) = v_i^\beta N_\beta(\mathbf{x})\,\mathbf{e_i} \tag{2.19}$$

where

$i, j = 1, 2, 3$ represent the $x,\ y,\ z$ components of the displacement.

$\alpha, \beta = 1, \ldots nn$ are the nodal values. (nn is the number of nodes).

$\mathbf{e_i}, \mathbf{e_j}$ are unit vectors in the $i,\ j$ direction.

The discretization of the LHS of the weak form (Equation (2.17)) derives from expressing the displacement $\mathbf{u}$, and $\mathbf{v}$ in terms of the approximated displacement.

$$\int_{\Omega} C_{rs}\varepsilon_s(\mathbf{u})\varepsilon_{\mathbf{r}}(\mathbf{v})\mathbf{d\Omega} = \int_{\mathbf{\Omega}} \mathbf{C_{rs}}\varepsilon_{\mathbf{s}}(\mathbf{u_j^\alpha N_\alpha e_j})\varepsilon_{\mathbf{r}}(\mathbf{v_i^\beta N_\beta e_i})\mathbf{d\Omega} \tag{2.20}$$

Then, by applying the linearity of the strain, Equation (2.20) becomes

$$\int_{\Omega} C_{rs}\varepsilon_s(\mathbf{u})\varepsilon_{\mathbf{r}}(\mathbf{v})\mathbf{d\Omega} = \mathbf{u_j^\alpha v_i^\beta} \int_{\mathbf{\Omega}} \mathbf{C_{rs}}\varepsilon_{\mathbf{s}}(\mathbf{N_\alpha e_j})\varepsilon_{\mathbf{r}}(\mathbf{N_\beta e_i})\mathbf{d\Omega} \tag{2.21}$$

where $\varepsilon_s(N_\alpha \mathbf{e_j})$ and $\varepsilon_r(N_\beta \mathbf{e_i})$ are defined in terms of the $\mathbf{B}$ matrix as

$$\boldsymbol{\varepsilon}(N_\alpha \mathbf{e_j}) = \mathbf{B} \begin{Bmatrix} N_\alpha \delta_{1j} \\ N_\alpha \delta_{2j} \\ N_\alpha \delta_{3j} \end{Bmatrix} \qquad \boldsymbol{\varepsilon}(\mathbf{N}_\beta \mathbf{e_i}) = \mathbf{B} \begin{Bmatrix} N_\beta \delta_{1i} \\ N_\beta \delta_{2i} \\ N_\beta \delta_{3i} \end{Bmatrix} \tag{2.22}$$

where δ_{1j} is the Kronecker's delta.

In a similar way, the discretization of the RHS of the weak form of the boundary problem, Equation (2.17), can be obtained by expressing displacements $\mathbf{u}$ and virtual displacements $\mathbf{v}$ in terms of the approximated displacement defined by Equation (2.18). That is,

$$\begin{aligned} \int_\Omega \rho f_i v_i d\Omega + \int_{\Gamma_{i2}} h_i v_i d\Gamma &= \int_\Omega \rho f_i v_i^\beta N_\beta \, d\Omega + \int_{\Gamma_{i2}} h_i v_i^\beta N_\beta \, d\Gamma \\ &= v_i^\beta \left(\int_\Omega \rho f_i N_\beta \, d\Omega + \int_{\Gamma_{i2}} h_i N_\beta \, d\Gamma \right) \end{aligned} \tag{2.23}$$

The discrete version of the weak form of the boundary problem for a linear elastic material can be obtained by replacing Equations (2.23) and (2.21) into Equation (2.17).

$$u_j^\alpha v_i^\beta \left(\int_\Omega C_{rs} \varepsilon_s(N_\alpha \mathbf{e_j}) \varepsilon_\mathbf{r}(\mathbf{N}_\beta \mathbf{e_i}) \mathbf{d\Omega} \right) = v_i^\beta \left(\int_\Omega \rho f_i N_\beta d\Omega + \int_{\Gamma_{i2}} h_i N_\beta d\Gamma \right) \tag{2.24}$$

Since the virtual displacement $\mathbf{v}$ is arbitrary, Equation (2.24) yields the finite element equation

$$u_j^\alpha \left(\int_\Omega C_{rs} \varepsilon_s(N_\alpha \mathbf{e_j}) \varepsilon_\mathbf{r}(\mathbf{N}_\beta \mathbf{e_i}) \mathbf{d\Omega} \right) = \left(\int_\Omega \rho f_i N_\beta d\Omega + \int_{\Gamma_{i2}} h_i N_\beta d\Gamma \right) \tag{2.25}$$

or in a simplified way

$$K_{\alpha\beta ji} \, u_j^\alpha = L_i^\beta \qquad i, j = 1 \ldots 3 \quad \alpha, \beta = 1..nn \tag{2.26}$$

where

$$K_{\alpha\beta ji} = \int_\Omega C_{rs} \varepsilon_s(N_\alpha \mathbf{e_j}) \varepsilon_\mathbf{r}(\mathbf{N}_\beta \mathbf{e_i}) \mathbf{d\Omega} \tag{2.27}$$

and

$$L_i^\beta = \int_\Omega \rho f_i N_\beta d\Omega + \int_{\Gamma_{i2}} h_i N_\beta d\Gamma \tag{2.28}$$

Equation (2.25) constitutes a linear system of equations where the displacement at the nodes u_j^α are the unknown variable. The matrix $K_{\alpha\beta ji}$ is known as the stiffness matrix and the vector L_i^β is referred as the vector of loads. In order to compute the stiffness matrix and the vector of loads, it is necessary to calculate the integral over the whole domain Ω. In finite element analysis, the domain is subdivided into a set of elements $\Omega = \sum_e \Omega_e$, so

the stiffness matrix can be calculated as the sum of the integrals over all the elements in the domain, that is:

$$K_{\alpha\beta ji} = \sum_e \int_{\Omega_e} C_{rs}\varepsilon_s(N_\alpha \mathbf{e_j})\varepsilon_\mathbf{r}(\mathbf{N}_\beta \mathbf{e_i})\mathbf{d\Omega_e} \tag{2.29}$$

Computation of the integrals over the elements depends upon the geometry of the element and shape functions are selected to approximate the displacement. The following section introduces the representation of the domain under the extended finite element proposal and presents the integration of the shape functions using high order polynomials.

2.3 An enriched Finite Element Method (X-FEM)

To deal with complex geometries and the subsequent problems with numerical integration and treatment of essential boundary conditions, the the eXtended Finite Element Method (XFEM) was developed around the year 2000, with the aim of mesh simplification in crack propagation problems [7]. In this method, some mesh boundaries are implicitly represented by the iso-zero values of a level set function. This has proven useful when following the evolution of cracks as moving interfaces.

The basic notion of partition of the unity [8] is used to enrich the finite element approximation with additional functions, the modeling of crack tip expansion serves as example of this treatment and its consequences on convergence rates [9]. To solve problems that involve topological and geometrical changes X-FEM presents as a good alternative to classical FEM.

Level Set functions are used to define the enrichment functions, allowing to determine the distance to the interface on a given point. The integration must be done carefully over elements that comprise a discontinuity and are enriched with additional functions to form correctly the element stiffness matrix, this can be done without meshing the interfaces. Partition cells are the elements that are split for separate regular integration on each of the sides of the iso-zero of the LS.

The LS is used to determine wether the partitioned integration cell is within the material or not when the case of a free surface. But when multiple interfaces or crack propagation is being considered, additional functions for the displacement field are needed [10].

The integration procedure then becomes under X-FEM a challenge comparable to surface representation under FEM. Replacing the discontinuous, non-differentiable functions with equivalent polynomials is presented to avoid the subdivision of elements during integration of the stiffness matrix. This proposal works only with straight discontinuity across an element and linear shape functions. Yet, good result with linear LS in linear elements have been achieved, even applied to curved geometries [11].

The previous approach proposes a representation of the LS on a finer submesh parallel to the finite element mesh, the representation of the LS is maintained as piecewise

linear on each element, but no DOF are added to the actual mechanical field. A strategy to use high order X-FEM is introduced, this is also applied to domains with curved boundaries.

As the order of the polynomials is raised the integration procedure becomes computation intensive. A solution to this problem is explored with the aid of current parallel computing capabilities that could alleviate the load and computation times significantly.

2.3.1 eXtended Finite Element Method

The problem is treated under the assumptions of a linear elastic, isotropic homogeneous solid, the same conditions mentioned on section 2.2 are applied, from which the same treatment and governing equations as in FEA are used. Small strains and no volumic forces are considered.

As discussed on the FEA section, the domain Ω is discretized into elements -finite elements- which constitute a mesh. Over these elements and approximation of the displacement field is done:

$$\mathbf{u}(\mathbf{x}) = \sum_{\mathbf{i}}^{\mathbf{n}} \phi_{\mathbf{i}}(\mathbf{x})\mathbf{u}_{\mathbf{i}} \tag{2.30}$$

where $\phi_i(x)$ are the finite element shape functions, and $\mathbf{u_i}$ are nodal displacements. Using the X-FEM approach, conforming the mesh to internal geometric characteristics such as holes, cracks, material interfaces is not mandatory. Hence, the possibility of using structured or unstructured simple meshes. Then, the FEA approximation is enriched with additional functions that approach the behaviour of boundaries as:

$$\mathbf{u}^{\mathbf{h}}(\mathbf{x}) = \sum_{\mathbf{i}\in\mathbf{N}} \phi_{\mathbf{i}}(\mathbf{x})\mathbf{u}_{\mathbf{i}} + \sum_{\mathbf{j}\in\mathbf{N_g}} \phi_{\mathbf{j}}(\mathbf{x})\ \mathbf{F}(\mathbf{x})\mathbf{a}_{\mathbf{j}} \tag{2.31}$$

where $F(x)$ is the enrichment function, $\mathbf{a_j}$ are the additional DOFs for enriched nodes, N is the mesh node set, and N_g the enriched nodes set. When the mesh nodes conform to geometrical characteristics, the classical finite element approximation is recovered.

2.3.2 High Order X-FEM

Classically , linear approximation shape functions $\phi_i(x)$ and linear LS are used in X-FEM. Rates of convergence close to optimal or optimal are achieved for given cases like material interfaces or cracks ([11], [10]).

High order shape function applied to the FE approximation have been used to accelerate the convergence of the method. Yet, the convergence rates are highly dependent on the quality of the geometrical representation, unless the description is improved, geometrical errors surpass the approximation errors [1].

The approach presented in such paper is to improve the description of the discontinuity to conform more to the real geometry maintaining a linear representation. A finer mesh to represent the level set is used, preserving the number of DOF of the real mesh, hence keeping the mechanical field intact. This approach allows to improve the representation of discontinuities and boundaries in a simple manner. This mesh is refined recursively up to a determined depth (as can be seen on Figure 2.4) to conform to the geometry but staying parallel to the computational mesh that remains unchanged.

2.3.3 Numerical Integration

Direct integration over discontinuous elements is not feasible, special care is taken over the cells where the iso-zero value level set intersects, a partition is created to generate integration cells (see Figure 2.3). These parted elements are tied to the computational mesh elements [7]. Then, a simple Gauss integration can be done on the cell that corresponds to the side on the element with material.

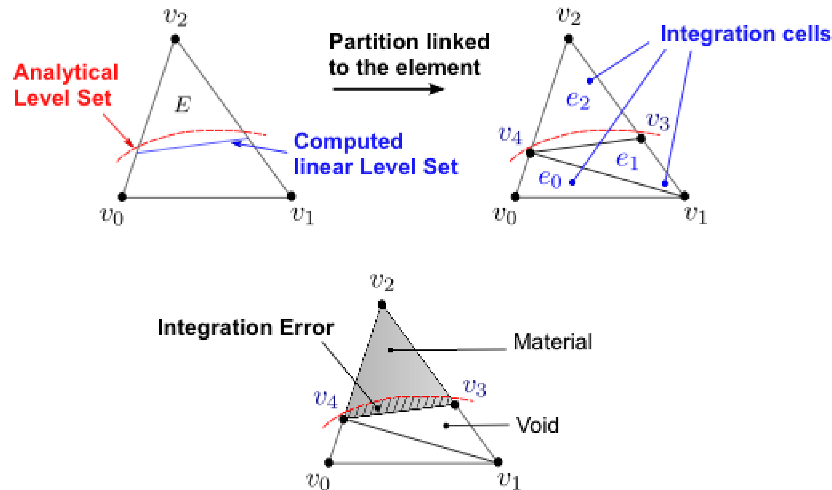


Figure 2.3: Numerical integration: partition creation

The resulting partition cells are classified depending on their distance to the interface, wether there are on the material or the void side (See Figure 2.4). The validity of this process on a benchmark problem is shown by the convergence of the energy norm on [1].

2.4 General Purpose GPU Computing

Current roofs on clock speeds for processors of major chip companies pushed the efforts from achieving higher clock speed to increasing the number of cores available in a

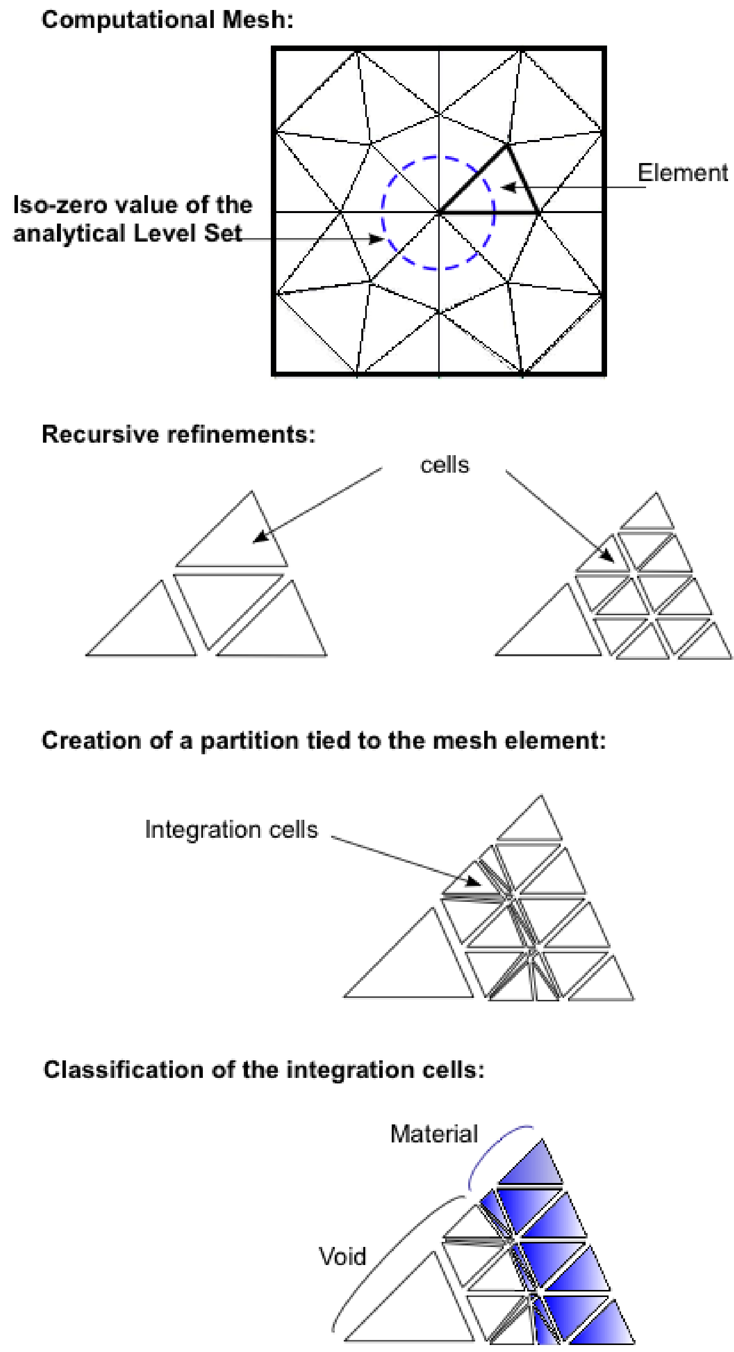


Figure 2.4: Numerical integration with High order X-FEM -Element associated partition-

processor. As Intel is bounded to top clock speed of 4GHz and AMD to 3GHz due to technical and power constraints. Though placing more cores into the processor boosts performance, this performance is not significantly improved unless the algorithms are developed with parallel computation in mind.

Parallel computing has been tackled over the past years with notable results, yet the access to HPC computing lies only by the ones with the monetary capacity to acquire quite expensive systems (i.e: Clusters, either blade-type or shared memory). Such conditioning has pushed the development of new strategies to approach scientific computing in a different manner.

One of these approaches, and one that has gotten a lot of attention is General Purpose Computations using Graphics Processing Units (GP-GPU). GPU have increased in capacity, precision, performance and programmability at incredible rates. Being one of their main driving forces the multi-billion dollar gaming industry, innovations in hardware architecture and capabilities are increasing at elevated rates.

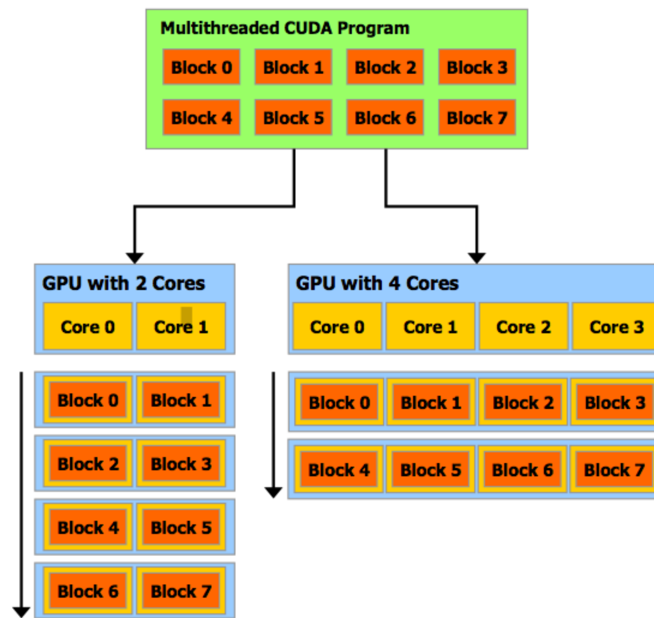


Figure 2.5: Scalability of GPU computation

Given the specialized architecture of GPUs it is possible to add more processors to perform computations, and increasing memory transfer bandwidths and cache sizes keep bringing great computation power to commodity graphic cards. Making GPUs and GPU oriented code highly scalable (See Figure 2.5). Recent advances brought double precision floating point operations on graphic cards, one much expected characteristic to finally give the status of scientific computing capable to GPUs. Even so, using single precision, a wide set of applications on scientific computing were already developed (i.e Physically based problems, linear systems solution and PDE treatment).

With problems being solved in engineering increasing in size due to modeling even more characteristics, taking advantage of coming supercomputers using GPUs for a fraction of the cost and with increased FLOPs (Currently on the TFLOPS range) becomes an appealing solution. But it is not as simple as porting code to a different language, but where the problem to treat involves several operations over the same or shared data, and parallelizable parts, then an attempt at developing a solution using GPGPU might prove worthy. Each of the cores on the card can execute simultaneously an amount of threads (process/operation) on data. Problems are split into Grids, which contain blocks, which execute within them the number of threads set by GPU capabilities. All threads within a block are expected to be within the same core (currently at 1024 threads) then if the problems demands less threads more blocks can be adjusted to fit in each core, and is this architecture that takes advantage at maximum of the available resources and of parallel computing techniques (see Figure 2.6)

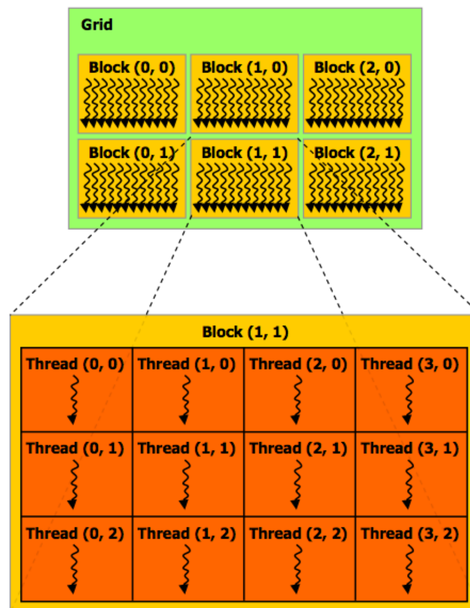


Figure 2.6: Thread hierarchy

“Driven by the insatiable market demand for realtime, high-definition 3D graphics, the programmable Graphic Processor Unit or GPU has evolved into a highly parallel, multithreaded, many-core processor with tremendous computational horsepower and very high memory bandwidth ”[12].

Not every problem or code can be ported to the GPU for processing, there are problems that are inherently serial. In cases where the majority of the problem cannot take advantage of the GPU raw power, there are portions that can still be though of in parallel and thus can be solved using the GPU, still achieving speed ups that are well worth the

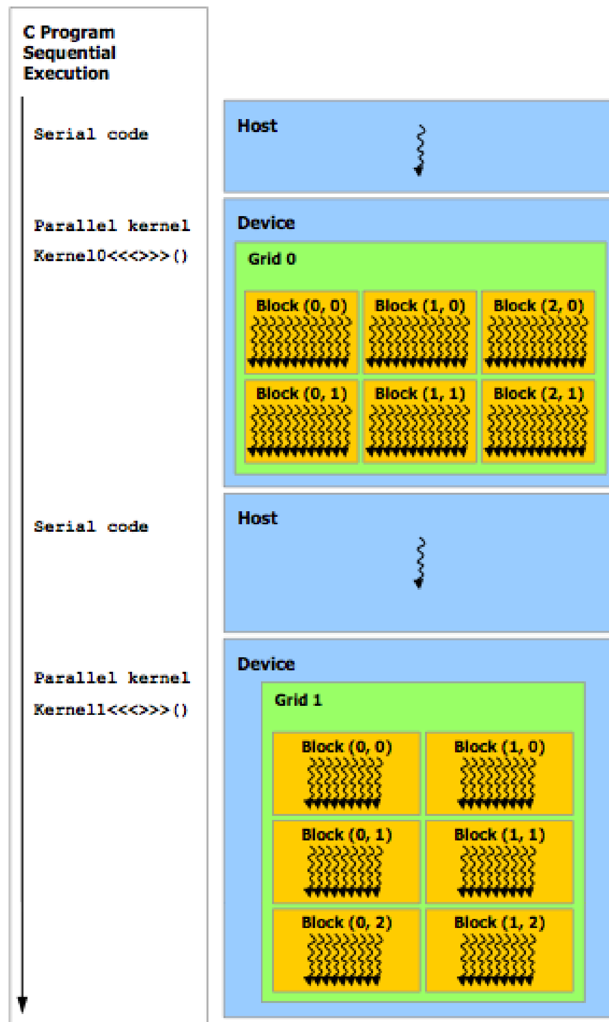


Figure 2.7: Programming model and execution

thinking. Portions of code that can be treated in parallel are compiled into *kernels* which are to be executed directly onto the cores of the GPU (See Fig. 2.7). Currently there is still strong coupling within CPU and GPU when solving a given problem, communication is unavoidable but should be spared to the minimum. There are current efforts by the major GPU manufacturers to develop more coupled CPU/GPU chips that improve the bus bandwidth or even remove the barrier currently between GPU and CPU.

2.5 GPU Computing Programming Language

CUDA is a general purpose parallel computing architecture and programming model. A new programming model and set of instructions allows the user to take advantage of

the raw power on nVidia GPUs, to solve many complex computational problems more efficiently than in a CPU.

A software environment and the necessary tools to profit from the GPU are provided by nVidia, C is used as a high level programming language -on which most of the implementations and core developing has been done-. As well, other APIs have been made available to allow more programming flexibility, such as CUDA Fortran, DirectCompute and OpenCL. Aside from the core components, community interest has played a role in porting and developing multiple sets of tools and instructions. Widely used routines and libraries, like BLAS, FFT, Sparse Matrix treatment and RNG are provided within the framework.

Improvements in GPUs and their architecture brought double precision capacity, which made possible a new standard of super computing for scientific simulations. With the advent of CUDA enabled GPUs into personal computing, the capacity to do scientific computing in personal computers or laptops became feasible. It is of interest to explore the advantages of such language extension developed by NVidia. The availability of many cores for data processing on a single machine (way more than typical CPUs) gives the developer the possibility to create and evaluate the performance and scalability of parts of the code subject to parallelization.

2.6 Application possibilities of GPU in X-FEM

The numerical integration that takes part in the calculation of the stiffness matrix is an operations intensive procedure. Depending on how the basic operators are arranged and coded, treatment of the shape functions, derivatives, determinant calculation and Jacobian are all recurrent operations. Regardless of how the problem is solved, they all depend on the geometry of the element, number of nodes and amount of shape functions and their degree. As well, the complexity of the problem to solve and the approximation used determines how computational intensive the calculation of the solution becomes.

	Degree of the approximation						
Number of:	1	2	3	4	5	6	7
Shape Functions:	6	18	40	75	126	196	288
Nb. $\boldsymbol{A}^e$ Entries:	36	324	1600	5625	15876	38416	82944
Gauss Points:	6	18	48	80	150	231	336

Table 2.1: Characteristics of the numerical integration

The case of interest involves using higher order polynomials and special elements from which the integrals are computed. They must be computed at runtime, and can demand a significant portion of the process in the FE solution. The parameters that shape the computational characteristics of the numerical integration are the amount of

shape functions used per element, which as well determine the amount of entries of the local element stiffness matrix, and the number of gauss points needed for the integration depending on the degree of the polynomial are summarized on Table 2.1 for the 3D case.

Treating the computation of shape functions together for each element can increase the memory needs with the increase of p. With the constraint of limited fast memory resources shared for each thread within a core in a GPU, a proper strategy is necessary. Most cases in-house implementations, tailored to the algorithms are developed to harness the maximum computing power [13].

The typical approach to numerical integration where subsequent loops are used will render a number of operations that depend on the number of integration points and the degree of the shape functions. Thinking in parallel architecture as specially in CUDA GPU, kernels, which are operations that can be performed for several elements -the concept of the loop- take their place but must be carefully planned to have the proper information available. Depending on the geometry of the finite elements, the type of problem being solved and the approximation used, the precise numbers of operations vary. Even for larger problems, having the option of massive amounts of processors available is a tempting alternative to speed up the solution.

An interesting approach to the integration procedure is presented in [14]. Suggesting an algorithm that is planned for the architecture of the GPU computing model. Given the limited resources available for a single thread, the most used information should be available in the fast shared memory that is associated to each multiprocessor core. Allowing entries to be calculated concurrently and place the necessary data available on the shared memory, also make access patterns as optimal as possible. The main characteristics noted and that come from the nVidia standards are -as shown in section 2.4-:

- A warp is a set of 32 threads that are simultaneously executed by hardware.
- Blocks (or threadblocks) are groups of warps that are synchronized by operations and access the same shared memory.
- Groups of blocks will form a Grid, which executes the same kernel for all composing threads.

Limitations on resources are:

- The amount of registers that are allocated by the compiler.
- The size of the shared memory (fast) is 16KB.
- The constant memory as cached device memory filled by the running host code, yet slow for transfers between main memory and device memory.

To optimize resource utilization some steps must be followed. Have as many possible threads being filled with data, this improves the performance of the thread scheduling. Also if multiple blocks are concurrently processed on a MP the computation efficiency is

increased. Since threadblocks are multiple of 32, there will always be a chance for idle threads depending on the mesh, or that a threadblock is not enough for the given stiffness matrix entry.

The results rendered by the model presented on [14] on a simple physical problem are encouraging. The hardware used dates from 2009-2010, in the past year chip technology, memory bandwidth and multiprocessor count has seen significant increases. In their results, speed-ups of up to 20x vs. the CPU are observed. In Table 2.2 are shown the execution times in milliseconds and speed-up GPU versus CPU for the numerical integration algorithm with different orders of approximation p using GeForce 8800GTX GPU and AMD X2 CPU.

	Degree of the approximation						
	p=1	p=2	p=3	p=4	p=5	p=6	p=7
CPU core time per element(s):	0.006	.103	1.164	6.543	33.834	125.039	390.167
GPU multiprocessor time per element(s):	0.006	0.046	0.466	6.596	46.271	257.341	889.719
Speed up (1 core, 1 multiprocessor):	1.07	2.25	2.50	1.0	0.73	0.49	0.44
Speed up (2 core, 16 multiprocessor):	8.56	18.0	19.98	7.97	5.85	3.89	3.51

Table 2.2: Results of numerical integration for FE example

3. Implementations and Tests

3.1 Cuda functions

Once the functionality of CUDA was ported to the X-Fem suite, proper tests were done to corroborate correct communication with the device(s), memory copying back and forth the device, basic operations and overall performance. The routines implemented were an interface that allows the xfem classes and methods to gain use of CUDA functions. Issues like memory allocation, memory copying and correct type casting -when absolutely necessary- were considered in the process.

3.1.1 Checking the Device

Using some CUDA set of functions, `checkCuDevices` included a query on the device name and capabilities. It was used to prove the X-Fem suite was properly initializing the CUDA environment, detecting the device(s) and communicating with it. A small modification to a class in the file *xForm.h* was done to call the function and check both the X-Fem library was accessing CUDA properly and the implemented functions were working correctly.

Using properly the capabilities of CUDA depends highly on the correct planning of the implementations and proper practices. As any parallel oriented software development, scalability and compatibility must be considered to create a sustainable computational tool. Acquiring proper information from the characteristics of the system in place, its capabilities, and making this variables determine how the application will handle the resources is of great importance.

Using a pre-compiled binary from the CUDA source code examples `deviceQuery`, the characteristics of the graphic card and the CUDA environment are displayed for information to be considered in the context of the tests and benchmarks.

deviceQuery Output:

```
CUDA Device Query (Runtime API) version (CUDART static linking)
   There is 1 device supporting CUDA
```

Device 0:	"GeForce 9400M"
CUDA Driver Version:	3.20
CUDA Runtime Version:	3.20
CUDA Capability Major/Minor version number:	1.1
Total amount of global memory:	266010624 bytes
Multiprocessors x Cores/MP = Cores:	2 (MP) x 8 (Cores/MP) = 16 (Cores)
Total amount of constant memory:	65536 bytes
Total amount of shared memory per block:	16384 bytes
Total number of registers available per block:	8192
Warp size:	32
Maximum number of threads per block:	512
Maximum sizes of each dimension of a block:	512 x 512 x 64
Maximum sizes of each dimension of a grid:	65535 x 65535 x 1
Maximum memory pitch:	2147483647 bytes
Texture alignment:	256 bytes
Clock rate:	1.10 GHz
Concurrent copy and execution:	No
Run time limit on kernels:	Yes
Integrated:	Yes
Support host page-locked memory mapping:	Yes
Compute mode:	Default
Concurrent kernel execution:	No
Device has ECC support enabled:	No
Device is using TCC driver mode:	No

deviceQuery, CUDA Driver = CUDART, CUDA Driver Version = 3.20, CUDA Runtime Version = 3.20, NumDevs = 1, Device = GeForce 9400M

It displays that the latest CUDA driver and runtime are being used (to date 02/2011), and that the card capabilities regarding architecture are of 1.1, which means doubles are not supported. Also another interesting aspect is the number of multiprocessors, which in this card is 2 (with 8 cores per MP). From the Nvidia reference [12], a number of maximum 1024 threads (lightweight processes) are available per multiprocessor on current GPUs, which on a dedicated card becomes an appealing number of processes available.

The characteristics of the computer used during the development and its hardware are shown to set the parameters of the test.

3.1.2 Checking Operability

Once the detection and communication to the device had been proven, the following step was to use a tweaked example of the vector addition provided by the CUDA source code examples to corroborate the proper use of the X-Fem library of this environment and its

Machine	Apple MacBook Pro
Operating System	OS X (10.5.8)
Processor	Intel Core 2 Duo P8400 @ 2.26 GHz
RAM	6 GB DDR3
Bus Speed	1.07 GHz
L2 Memory	3 MB
Graphic Card	Nvidia GeForce 9400M (256 MB VRAM) 64 MB

Table 3.1: Hardware Characteristics

capacities. Given the card used has not Fermi architecture and does not support double precision operations, *float* type is used throughout the tests.

Vector Addition

The *vectorAddition* example allocates memory for an arbitrary vector size in the host and device, it initializes each vector with a set of random values and copies this set to the device memory before the kernel is called to perform the operation over it. Once the operation has been performed, a comparison is made between the computation from the CPU and the GPU to a tolerance of $1e-5$.

Vector Size (N)	5000000
GFlops to perform	0.005000
Memory allocation time (Device) [s]	0.385615
Memory copying time (To Device) [s]	0.024941
Memory copying time (To Host) [s]	0.043163
Processing Time (GPU) [s]	0.000177
Processing Time (CPU) [s]	0.017934
Total time (GPU) [s]	0.471841
GFlop/s (GPU)	28.225464
GFlop/s (CPU)	0.278802

Table 3.2: Vector addition test (N=5e6)

Given the total time to allocate and copy memory to the device are to be added to consider overall performance, the total time rendered by the vector addition on the GPU is over 26 times greater than the time needed by the CPU (see Table 3.2). Considering that raw processing times are considerably different, the GPU is over 101x faster to perform

the calculations alone. The main concern is to take advantage of the raw processing power without having high memory transfer loads.

Vector Size (N)	10000000
GFlops to perform	0.010000
Memory allocation time (Device) [s]	0.383140
Memory copying time (To Device) [s]	0.049882
Memory copying time (To Host) [s]	0.086439
Processing Time (GPU) [s]	0.000181
Processing Time (CPU) [s]	0.036724
Total time (GPU) [s]	0.556377
GFlop/s (GPU)	55.188211
GFlop/s (CPU)	0.272301

Table 3.3: Vector addition test (N=1e7)

Using a set of vectors of one order of magnitude greater, the results are comparable and as expected, the increased number of elements reflect on higher times for copy to the device (memory allocation time is preserved). As well, the actual time to perform the operations increase proportionally. The processing time is above 200x faster in the GPU than the CPU, which confirms the tendency when using GPU resources in highly parallel operations.

One of the main concerns when attempting to use the GPU to co-process is the actual time that is spent in data transfer. The approach is justified only when the amount of operations done over the data is large enough to compensate for the time spent in transfer. From this basic example we can obtain key information. It is a simple example and the level of complexity of the operations performed over the data is quite basic, only one addition per entry is performed. The time spent allocating memory for the vectors and its copy to the device has to be taken into account when attempting to improve operations in the code, it is not just the amount of floating point operations that can be performed by the device.

3.2 Using CUBLAS Library

CUBLAS is an implementation of the widely used Basic Linear Algebra Subprograms (BLAS) on top of the CUDA driver, it has no direct interaction with the CUDA driver and it is handled at the API level. CUBLAS is one of the libraries that has been implemented in CUDA to take advantage of the GPU computing power.

The basic usage model for applications using CUBLAS is:

1. Create either matrix or vector objects in GPU memory.

2. Fill the created objects with data.

3. Perform a call to any CUBLAS function.

4. Upload the results from the GPU memory back to the host memory.

In order to do this, CUBLAS provides a set of helper functions that allow to create and destroy objects in GPU memory space. Also, helper functions to write and retrieve data are provided. The interface to the CUBLAS library is the header file *cublas.h*, the application to be compiled must link to *cublas.dylib* in OS X (See Appendix C). With the proper definitions and aware of the workflow, a C/C++ interface of the CUBLAS functions for the developed application can be done.

3.2.1 Interface to CUBLAS

The workflow for any CUBLAS call follows the same guidelines each time independent of the routine to be called. The variables to be passed to the CUBLAS function will be taken from the environment in which the library is linked to. It is important to be able to maintain the coherence of the variables and structures being used within X-Fem while implementing the calls to CUBLAS.

Using cublasDgemm

One of the routines that are widely used along the X-Fem suite is the general matrix-matrix multiplication output or DGEMM by its initials. An interface definition is provided for the X-Fem suite in the header file *xBlasDef.h*, this definitions link to the local BLAS version or system BLAS (given the case that the environment is Linux and system BLAS is proved compatible). The system BLAS routines are also provided on OS X, including the proper header files and making the function calls is treated later. This definitions were written to provide a proper port of the variables and data structures used within X-Fem and use them in BLAS function calls.

Maintaining the structure of the provided code and its variables, an interface to cublas-Dgemm was developed to pass correctly the variables and performing the necessary procedures over the data.

```
1 void cublasDgemm (char transa, char transb, int m, int n, int k, ↩
    double alpha, const double *A, int lda, const double *B, int ldb, ↩
    double beta, double *C, int ldc)
```

This calls are equivalent to the one made on `dgemm`, where a small difference in the input is to be observed in reference to the OS X system BLAS call. In the CUDA implementation the column major storage is preserved from BLAS, whereas in OS X the ordering of the data must be specified.

```
1 void cuDgemmInterface(CUresult &error, char transa, char transb, int ↩
    m, int n, int k, double alpha, const double *h_A, int lda, const ↩
    double *h_B, int ldb, double beta, double *h_C, int ldc)
```

To maintain a black-box approach to the CUDA environment when profiting from libraries such as CUBLAS, a function that performs the required memory allocation and copying to the device was written. Where the variables that hold the data: `const double *h_A`, `const double *h_b` and `double *C`. Represent memory space in the host and have to be treated to operate on the device. Am overall outline of the operations can be described as:

1. Initialize CUBLAS environment (`cublasInit()`).
2. Initialize device memory space for the matrices to operate (`cublasAlloc(...)`).
3. Transfer the data of the matrices to the device memory space (`cublasSetVector(...)`). Also `cublasSetMatrix` can be used, it depends on the storage form and indexing being used. All matrices must be initialized, including matrix C.
4. Call to `cublasDgemm(...)`.
5. Retrieve output matrix from device memory space.
6. Allocated memory clean up.

Using cublasSgemm

Since the hardware available during the development was not able to handle doubles, for proof of concept purposes the same interface adapted to `cublasSgemm` was implemented.

```
1 void cublasSgemm (char transa, char transb, int m, int n, int k, ↩
    float alpha, const float *A, int lda, const float *B, in float *C,↩
     int ldc)
```

As before, an interface to handle the cublas call, memory allocation and copying to device was made. The outline and operations performed are equivalent to the ones of the `cuDgemmInterface`.

```
1 void cuSgemmInterface(CUresult &error, char transa, char transb, int ↩
    m, int n, int k, float alpha, const float *h_A, int lda, const ↩
    float *h_B, int ldb, float beta, float *h_C, int ldc)
```

Using cublasDaxpy

Another routine that is frequently used across the X-Fem code, and is also defined in the interface header file *xBlasDef.h* is DAXPY. This is a double precision vector-scalar multiplication and addition method. The operation performed is alpha $*$ x $+$ y. The basic CUBLAS call needs as well of the same outline described previously and an interface to handle this operations transparently for the X-Fem environment was also implemented.

```
1 void cublasDaxpy (int n, double alpha, const double *x, int incx, ↩
    double *y, int incy)
```

As in the previous implemented methods, the CUBLAS environment must be initialized, the memory space in the device must be allocated and contents must be copied, all this operations are performed inside a function that provides the interface to X-Fem.

```
1 void cuDaxpyInterface(CUresult &error, int n, double alpha, const ↩
    double* x, double* y)
```

The same basic steps from the outline are taken, both double vectors are initialized in device memory space, vector *y* is to be rewritten and retrieved from the device.

Using cublasSaxpy

For the same reasons stated earlier, the hardware available during development was not able to handle double precision. Hence, for proof of concept purposes and testing results the same interface done for `cublasDaxpy` was done for `cublasSaxpy`.

```
1 void cublasSaxpy (int n, float alpha, const float *x, int incx, float↩
     *y, int incy)
```

The same interface is provided to handle single precision as in the previous examples.

```
1 void cuSaxpyInterface(CUresult &error, int n, float alpha, const ↩
    float* x, float* y)
```

Using CUBLAS in the X-Fem Suite

Bilinear operators are defined in the header file *xForm.h*, within some of them BLAS function calls are used to perform operations between vectors and tensors. Within the methods implemented in the classes that use BLAS functions the CUBLAS interfaces can be implemented.

As an example, the class `xFormBilinearSymetricWithLawTensor2BlasCUDA` is a mere transcript of the class `xFormBilinearSymetricWithLawTensor2Blas` which implements

BLAS calls to improve the performance of the integration scheme. Both DGEMM and DAXPY routines are used in the method `xFormBilinearSymetricWithLawTensor2Blas::accumulate( xGeomElem* geo_integ )`

```
1  template <typename OperatorLeft, typename MaterialLaw>
2    class xFormBilinearSymetricWithLawTensor2Blas: public xFormBilinear↩
       {
3    ...
4   public:
5    ...
6    void accumulate(xGeomElem*  geo_integ){
7      ...
8
9      double *testpp = new double[test_size*test_size];
10     double *vals_left = new double[9*test_size];
11     double *vals_right= new double[9*test_size];
12
13     ...
14     for(int k = 0; k < nb; k++){
15         ...
16         //Geometric data handling
17         for (int i = 0; i < test_size; ++i){
18           ...
19           //Shape function evaluation in the domain
20         }
21         int nop = 9;
22         int ts = test_size;
23         dgemm_(&TRANSPOSE, &NOTRANSPOSE, &ts, &ts, &nop, &ONE, ↩
              vals_left, &nop, vals_right, &nop, &ONE, testpp, &ts);
24       }
25       axpy(1., testpp, Matrix);
26       ...
27   }
28 };
```

The values of the shape functions by left and right at the integration points are stored for every element in arrays. These arrays are taken as input in the `dgemm_(...)` call.

Where there were local BLAS calls, which used the interface defined in *xBlasDef.h*, the CUBLAS interface is replaced. This reads:

```
1  template <typename OperatorLeft, typename MaterialLaw>
2   class xFormBilinearSymetricWithLawTensor2BlasCUDA
3   : public xFormBilinear
4   {
```

```
5   private:
6     const OperatorLeft  Left, Right;
7     MaterialLaw&  Law;
8     typename MaterialLaw::result_type phys;
9   public:
10    xFormBilinearSymetricWithLawTensor2BlasCUDA(MaterialLaw& law) : ↩
         Left(OperatorLeft()),Right(OperatorLeft()),Law(law) {}
11    xFormBilinearSymetricWithLawTensor2BlasCUDA(OperatorLeft& l, ↩
         MaterialLaw& law) : Left(l), Law(law), Right(l) {}
12
13    void accumulate(xGeomElem*  geo_integ){
14      CUresult error;
15      ...
16
17    #pragma omp parallel
18      {
19        ...
20
21        for (int i=0; i< test_size*test_size; ++i) testpp[i] = 0.;
22      #pragma omp for
23          for(int k = 0; k < nb; k++){
24            ...
25
26            for (int i = 0; i < test_size; ++i){
27              ...
28            }
29
30            char TLeft = 'T';
31            char TRight = 'N';
32            int nop = 9;
33            double one =1.;
34            int ts = test_size;
35
36            cuDgemmInterface(error, TRANSPOSE, NOTRANSPOSE, ts, ts, ↩
                 nop, ONE, vals_left, nop, vals_right, nop, ONE, testpp,↩
                  ts);
37          }
38        #pragma omp critical
39            cuDaxpyInterface(1., testpp, Matrix);
40           ...
41      }
42    }
43  };
```

The same variables and outline is kept, only an error checking for the CUDA environment is set both for DAXPY and DGEMM interfaces. The variables that reflect the

performance of such changes can be monitored in the integration time. Careful testing of memory allocation and copying to memory space in the device was done. This provides a detailed map of the performance of this alternative.

Both versions of SAXPY and SGEMM were implemented in order to prove functionality in the code. The single precision available for the hardware used during the development did not allowed the use of the actual code data types. A working example became necessary to benchmark BLAS and CUBLAS as it was done with the vector addition example.

3.3 Using OS X BLAS

The development included porting the X-Fem suite to OS X system. Apple's SDK comprises a set of frameworks that offer commonly used functions and custom compiled libraries. It also offers a BLAS implementation compiled under the darwin architecture included in the ***vecLib*** framework or also in the ***Accelerate*** framework. The functions declared in the header files *cblas.h* and *vblas.h*, contain the interfaces for Apple's implementation of the BLAS API.

Given a system architecture it is recommended to use system specific libraries to increase performance, with minor editing on the code the adequate function calls can be made. After including the header file *cblas.h* and declaring the appropriate variables, calls to DAXPY and DGEMM can be done quite straightforward. The usual template for using BLAS routines is cblas_<Blas routine name>(...). To call DGEMM from *xForm.h*, certain version specific variables must be declared, these variables play the same role as the flags in BLAS or CUBLAS calls. In OS X the method can be oriented to be row major or column major, the latter is the one by default used in CUBLAS and BLAS.

```
1 CBLAS_ORDER Order = CblasColMajor;
2 CBLAS_TRANSPOSE TransA = CblasTrans;
3 CBLAS_TRANSPOSE TransB = CblasNoTrans;
4 cblas_dgemm( Order, TransA, TransB, ts, ts, nop, ONE, vals_left, nop,↩
      vals_right, nop, ONE, testpp, ts);
```

Routine	Integration time
cblas_dgemm	0.613498
dgemm_	0.611556

Table 3.4: Performance for BLAS implementations 1

Other widely used routines like DAXPY had to be adapted in a different way. There was an interface already programmed that used function overloading depending on the input data type. *axpy* is the interface defined in the header *xFemMatrix.h* which calls the BLAS interface definition *daxpy_* from *xBlasDef.h*.

```
1 inline void axpy(const double & a, const xFemMatrix &X, xFemMatrix &Y↩
    ){
2   int one =1;
3   int N = X.nbrow*X.nbcol;
4   daxpy_( &N, const_cast< double *> (&a), const_cast<double *> (&X.↩
      vals[0]), &one, &Y.vals[0], &one);
5 };
6
7 inline void axpy(const double & a, const double * X, xFemMatrix &Y){
8   int one =1;
9   int N = Y.nbrow*Y.nbcol;
10  daxpy_( &N, const_cast< double *> (&a), const_cast<double *> (X), &↩
      one, &Y.vals[0], &one);
11 };
```

There are 2 definitions that are overloaded depending on the input data type, the difference lies wether the vector addition is performed between a `double *` and `xFemMatrix` or betweeen two `xFemMatrix`. Inside the overloaded function the call to *daxpy_* is done to match the interface data types for BLAS. The same can be done to use *cblas_daxpy*, adding the header *cblas.h* and editing these inline methods gives the intended result.

```
1 inline void axpy(const double & a, const xFemMatrix &X, xFemMatrix &Y↩
    ){
2   int one =1;
3   int N = X.nbrow*X.nbcol;
4   cblas_daxpy( N,  a, const_cast<double *> (&X.vals[0]), one, &Y.vals↩
      [0], one);
5 };
6 inline void axpy(const double & a, const double * X, xFemMatrix &Y){
7   int one =1;
8   int N = Y.nbrow*Y.nbcol;
9   cblas_daxpy( N,  a, const_cast<double *> (X), one, &Y.vals[0], one)↩
      ;
10 };
```

Routine	Integration time
cblas_daxpy	0.616095
daxpy_	0.619433

Table 3.5: Performance for BLAS implementations 2

4. Results

With proper interfaces and wrappers for the necessary functions, the test to some of the most used routines could be done. Among the routines used from BLAS, corresponding CUBLAS wrappers were developed to handle the memory passing and operations.

Given the nature of the test, the routines are not deployed in their most optimized form, but they still yield proportional results. As well, given the asynchronous nature of the execution of the kernels in the GPU cores, thread synchronization control was extensively used to maintain timing coherency.

Tests were developed for variable array sizes, in the case of sgemm, and for simplicity purposes in implementation square matrices were employed. Given the data structures in the code, a different approach is taken regarding the operations. The linear algebra operations are batched as much as possible to improve efficiency.

4.1 Testing BLAS routines

Within the provided code, interfaces were already programmed to provide support for a compiled version of BLAS. Extensive use of such operation along the many classes that comprise the code were traced down to evaluate possible porting and implementation into CUDA. Enabling the CUBLAS routines within the core functionality demanded the creation of interfaces, the process is not as straightforward as it would be expected, and they also had to allow their use with existing data structures. Some of the most employed routines are axpy, gemv, gemm, dsyrk and norm.

From expected behavior with scalable computing routines, increasing size of the data should prove the viability of the usage of the GPU for computations at points were the CPU is no longer optimal.

When running release code in CUDA, methods like *cudaThreadSynchronize* are not implemented. Usually a core runs the same kernel for all data going to it, meaning most of the times all threads will finish simultaneously. The synchronization of the execution of the kernel on each thread is not guaranteed, the kernel can still exit the routine and threads can be executing within the GPU, even long after the serial CPU code has taken over the execution. A good code design would take advantage of hardware implementations to guarantee the highest level of synchrony possible, yet sparing the use of time consuming routines. Once again, for testing purposes and in order to be able to time correctly

the operations being performed, methods to record the events when threads have been synchronized are used.

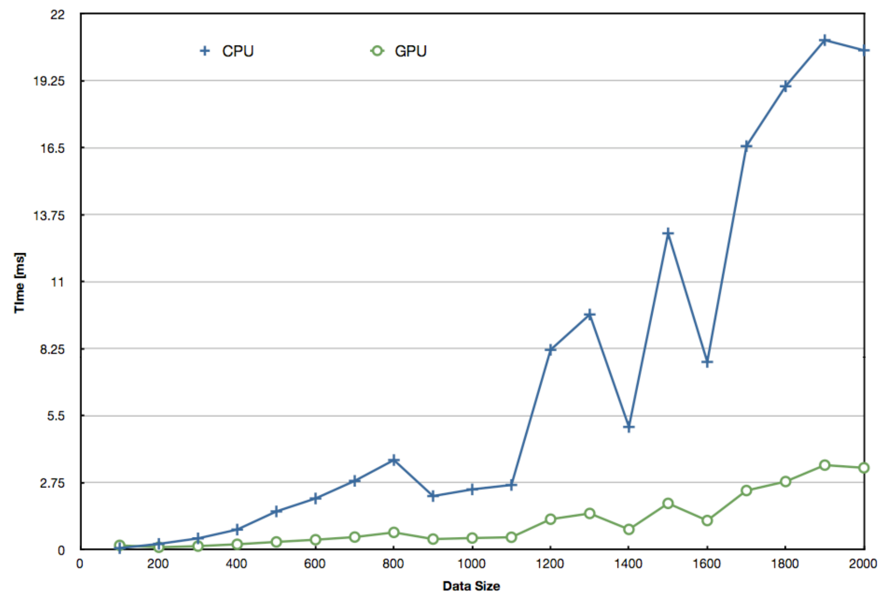


Figure 4.1: Array Fill time comparison for data preparation

In Figure 4.1 an example where an array is filled with constants is shown. A square array of *nxn* is passed to the routine, when the size of the array is small (n=200, 40000 elements) the difference is not as steep as when the size of the array goes over 1.5 million elements. Each element is a single precision float -which means 4B, around 5.5 MB-, this is important since the amount of local fast memory within each GPU core is limited. It can be seen how out of independent operations the most profit can be made out of the GPU, the array is scattered in blocks that are sent along each core. In this way we see differences in average of 6x better times on the GPU when the array size is big enough. As discussed on Section 3.1.2 in the vector addition evaluations.

Even if the amount of GFLOPs is not as impressive as would be expected, the superiority of the GPU is clearly observed (See Figure 4.2). The oscillating pattern on the GPU could be a matter of data handling, as mentioned, to each core where the kernel is executed a number of blocks is assigned, each block comprised of an amount of warps, which contain 32 threads -constants set by nVidia in their architecture-. A kernel that is executed over data of size multiple of 32, or data that is more likely to fill all of the cores it is executed in, is more likely to be highly efficient, full blocks and full cores with no idle threads make a solid base for highly optimal running code.

The timings that could be observed with the BLAS calls of sgemm between the CPU and the GPU are not far apart (See Figure 4.3). The difference in milliseconds does not encourage the use of CUBLAS given the overhead cost of pushing the data to the GPU back and forth on the process.

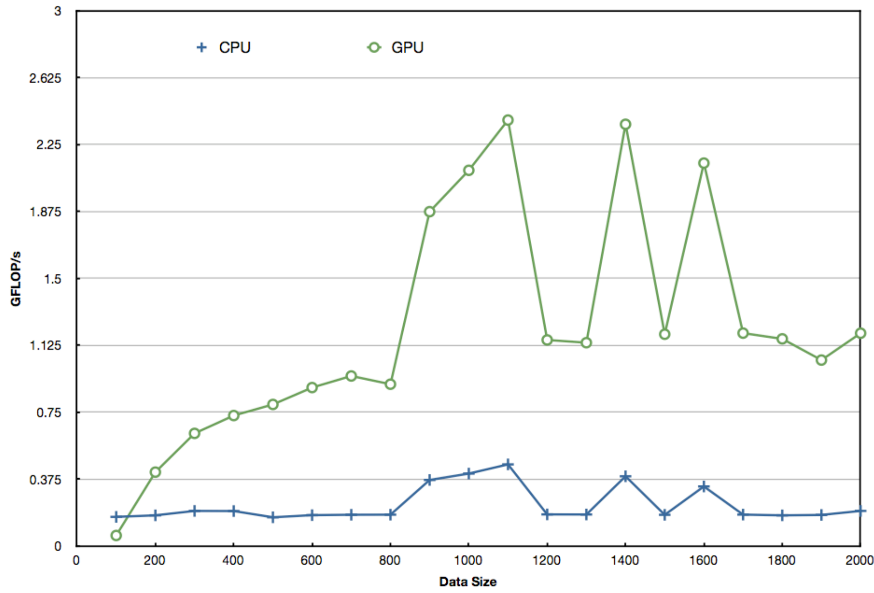


Figure 4.2: Array Fill Gflops comparison for data preparation

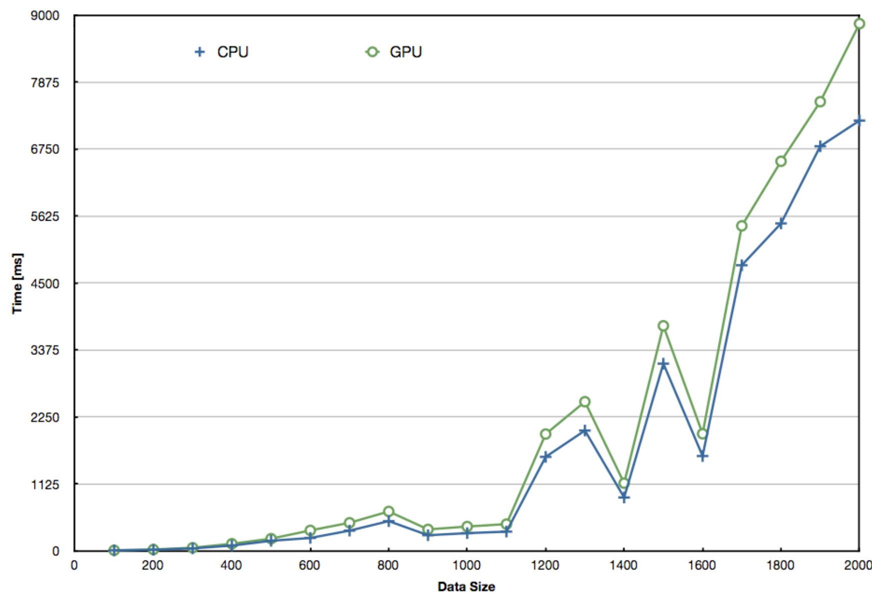


Figure 4.3: Comparison of times for sgemm BLAS call

One reason encountered that was predominant throughout all of the tested applications was the overhead comprised by an idle card and the type of memory being used.

A significant difference is noticed when changing from shared to pinned memory, for

which some implementation to call the proper memory allocation and deleting functions were made (see *cublasHostAlloc()* on [15]). The drawback of using pinned memory is its limited availability -depends on the system- and increased cost of allocation. If the type of problem allows so, and the amount of data to be processed does not exceed system limitations, pinned memory offers great advantage in memory transfer. This presented problems in the current settings given the available GPU is not fully dedicated.

Testing beyond 4 million element arrays was not possible with current hardware capabilities, and as seen on 2.1 the number of entries to the arrays involved in the operations escalates rapidly when increasing the degree of the approximation. On Figure 4.3 the execution of both BLAS and CUBLAS routines can be followed, as described in previous sections the BLAS implementation is hardware oriented and optimized for the apple's intel processor. The previous characteristic makes the scheduling tasks more efficient in the CPU than the GPU, giving the CPU still an advantage unless other implementations are done over the GPU code.

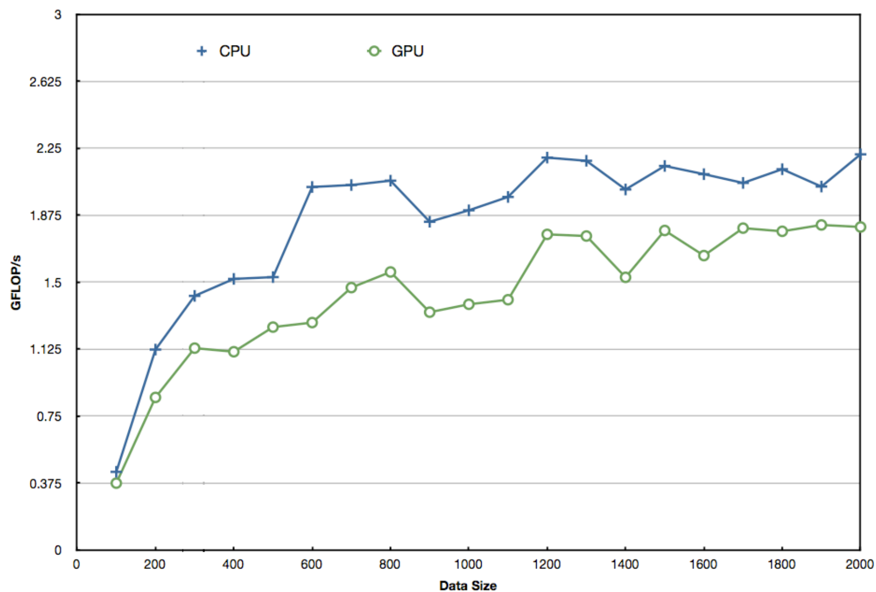


Figure 4.4: Comparison of Gflops for sgemm BLAS call

The same behavior is observed on the overall comparison on Figure 4.4, GFLOPs are measured in the following manner:

$$\text{GFLOPs} = \frac{\text{FLOPS}/10^9}{\text{Time}[s]}$$

Where the time is measured from memory allocation to the copying back to the device, In the case where the copying time could be avoided the performance in GFLOPS of the GPU surpasses greatly the CPU. On the used hardware, the theoretical limit of GLOPS

is calculated as:

$$\text{Theoretical_GFLOPs} = \text{ClockSpeed}[GHz] * \text{Nb. Cores} * (FADD + FMUL)$$

Where the FADD and FMUL stand for the amount of floating point sums and multiplications a core can make in a clock cycle (3 in current hardware). This renders a theoretical ceiling of 54 GFLOPS, a measure way below current market standards on dedicated cards. Yet still superior to the CPU performance even if bandwidth penalization is made over this number and have it reduced to 1/10.

5. Conclusions

Using the GPUs for numerical integration applied to Finite Element problems has been proven in [14] and [13]. The applicability to the X-FEM suite is justified up to some extension. Given the nature of the code and data structures, to fully profit from the computing power of the GPU, an important overhaul of the code must be done.

The numerical integration which carries information from the sub-mesh and the computational mesh has to be tailored to function under the GPU architecture. The implementation of the CUBLAS library to use routines like gemm, gemv, axpy, dot, norm and syrk was tested and the results given are not particularly encouraging. From the initial testing and development some conclusions and solutions were drawn.

1. There was an initial delay that would render any operation surprisingly slow. Searching for the origin of such behavior, it was found that due to hardware power consumption policies, the cores remain idle unless being called to process. Allocating memory for declared variables and memory transfers were significantly faster once a single event was created at the beginning of the code. It works like a wake up call to the GPU and changes radically initial response.

2. With the some of the operators for the integration provided by the X-FEM code that were modified, initial tests proved that memory transfers were considerably expensive to prove viable. single and small transfers proved to be the slowest, as could be expected for the amount of communication against the data being actually moved.

3. Batching memory transfer calls as much as possible proved to be a more efficient strategy when dealing with subsequent operations. Yet, the memory copying stage and communications took a great portion of the overall computation time by the GPU.

4. From all of the tests done in the available system, the superior processing power of the GPU is clear, the amount of GFLOPs performed by the GPU surpass in at least 50x to a maximum of 202x times more operations than the CPU in examples as vector additions and axpy.

5. When the use of memory is unavoidable and intensive, the performance is improved when using pinned memory. This type of memory is page-locked and more expensive to allocate, but this drawback is compensated with higher transfer rates. Some

modifications to functions must be done since this is not the default method to allocate memory (API default is *-shared-*).

6. Even though the implementation of BLAS routines in CUDA has great performance, they lack the tailored aspect that would make the use of the GPU fully viable for the proposed problem, determining precisely how the problem is split in blocks and that it corresponds to a single element like proposed in [14] demands further control over the operations. Keeping data coalescence and use of virtual data to hold the proper structure becomes necessary to fully revamp the performance of the algorithms.

7. The lack of C++ support in the current driver and API still constraint the use of current code, forcing the copying and casting of much of the data to move to the GPU, which is under any point of view inappropriate.

8. The use of lighter parallelization options like OpenMP offers increased efficiency but not up to expected behavior, and yet it would be constrained to a single Thread per core available. It would require a massive amount of CPUs to achieve results comparable to the ones that could be obtained with the GPU. Increasing the number of CPUs would mean increasing the communication between them, which in the end will not yield the expected results and will prove as a time/resource consuming option.

Given the limited amount of resources in local, fast memory to each core of the multiprocessor, the approach must be to reorganize the data structures to keep the necessary data to perform the calculations in the GPU as much as possible. Transfer rates are still to low to consider migrating code if overall times are not reduced significantly, even with the breach of GLOPS performed.

Not all code is subject to use of the GPU, and there will always be serial code, reducing the amount of data that has to be transfered from the CPU to the GPU is key to maximize efficiency. It was proven that for recurrent transfers it is not worthy to make use of the GPU for the computations.

It is but an initial approach that renders questioning but promising results, a change of culture to promote development in this area would prove very fruitful. Further and more extensive research would be a mandatory step to profit from all the new capacities that GPUs have to offer. It is a rapidly growing field that is yielding surprising results in scientific computing, it is a topic that should be studied further.

It is necessary to prove the implementations in more powerful graphic cards, with greater dedicated real memory, not the shared type that even dedicated graphics cards use in laptops.

6. Perspectives

The challenge with code suites that have been developed over the years is to make them sustainable over time. Allow new developers to be up to date and get deep knowledge of the routines in order to contribute efficiently. Most of the times data structures have become overly rigid and do not allow easily new implementations that change substantially the treatment of the computation.

Some of the implementations necessary to profit from the computing capacity of the GPU would be to code tailored routines to the x-FEM suite, like it has been done to match their own applications on [14]. More than just the numerical integration can profit from parallelization strategies. Mesh creation and processing, equation systems solvers are just but a few options.

6.1 Future Work

1. Proper versioning of the code, cross-platform and architecture support should be guaranteed. This means that when a change in the code or any of the linked libraries or used APIs will not be affected by necessary recompiling when using different architectures.
2. Implement tailored basic linear algebra operations, designed for the problems dealt with in X-FEM, considering the mesh handling, operators, data- and matrix structures used. This spans along the most used routines like axpy, gemm, gemv, norm and syrk.
3. Testing of the new routines using double precision operations. Perform benchmarking regarding the desired accuracy related to the used level of precision in accordance to processing time.
4. Improve memory handling given the structure of the problem, implement padding on elements to use fully populated blocks.
5. Hardware beyond a laptop computer is necessary to perform the proper testing, access to better graphic cards[1] or even the acquisition of a CPU-GPU enabled super-

[1] Nvidia is quite eager to provide some hardware for development when proven to push the use of GPU use further.

computer (which has a cost range of a fraction of the cost of clusters being currently purchased by universities).

6. A deeper overhaul of the code to orient it more to C for the necessary portions of code that rely greatly on C++ structures.

7. Porting and evaluation of the newly released, driver, plug-ins and API from Nvidia last May.

A. Porting X-Fem Suite to Apple OS X

The process of porting code to other platforms can be time consuming. Although the original X-Fem suite was developed as multi platform code, this did not include Apple's proprietary operating system. Darwin is built on top of the proven solid building blocks of UNIX, the similarities with BSD makes the porting from Unix/Linux operating systems a closer task.

The process of finding the information and solutions necessary to set up the developing environment implies extensive forum browsing and reference documentation searches.

Any seasoned programmer can understand that even between different Linux flavors code porting can be a demanding task. The process of correct library compiling and linking, adapting compiler/architecture dependent flags proves to be a arduous process.

The X-Fem suite was also flavor dependent even on Linux platforms (experienced between Ubuntu and Suse), this difficulted the porting procedure, given the non-compatibility of in-house code with minor reference and dependency changes.

Another difficulty that can be encountered when a suite comprehends a massive amount of in-house and external code, is the lack of control the developers have along the span of the code dependencies and references. Both minor and substantial changes had to be done to the sources in order to allow compilation under darwin.

A.1 Basic Requirements

The compilation of the source code and the use of the software will require for certain libraries and programs that are not usually available out of the box for many Linux distributions or under Darwin.

The OS X version under which the X-Fem suite was compiled is Leopard (10.5.8). The system compiler is gcc 4.0 and is set as default. In order to have openMP support, all related system defaults for g++, gcc, cpp (and any remaining) must be redirected to gcc 4.2 (which is also provided as supported within the system). Do this either exporting the variables on your *.bash_profile* file or remove and create the symbolic links under /usr/bin. Fortran compilation has been added to gcc compilers, yet to the provided version under Darwin, download the packages and patches required to enable this feature (It is supposed to be supported under OS X 10.6).

Other software and library requirements can be fulfilled by installing macports and using the available ports from Linux. By properly setting up the system variables, the installed ports are available to the system. See (put the macports http). Install macports.

sudo port search <portname> to look for a given port, the <portname> does not have to be the exact software name, it can be a fragment of the name.

Some useful pieces of software that will be needed for the compilation or can be useful for developing are:

- **ndiff** (necessary). File comparison tool used by the code. There is an available port.
- **gmsh** (necessary). Pre and Post-processing tool used by the code. There is no available port, yet there is an OS X version of the binary (gmsh-2.5.0-MacOSX.dmg). and the provided source code [16], [17] can be compiled using CMake. Download the source code and place it in a folder, it is recommended the /Users/Shared/dports folder, it does not require super user permissions and is the folder created typically to install deprecated or alternative ports.
  - **CMake**. Cross-platform Make. Can be obtained using macports.
  - **FLTK**. Fast Light ToolKit. It is also necessary to compile the graphical interface of gmsh, there is an available port.
  - Configure properly the build version of gmsh using ccmake (create a build directory inside the source and call cmake from that location).
- **Boost**. Well adjusted set of libraries in C++ with wide support. There is an available port, this will also install other dependencies and boost-jam.
- **Mtl**. High-performance generic component library that provides comprehensive linear algebra functionality for a wide variety of matrix formats.
- **tcl/tk**. Language support in case your developing environment requires so. Also necessary for InsightToolkit compilation.

A.2 Build Process

The general structure needed to obtain a functional version of the X-Fem suite is the following (See Fig. A.1):

- The ***Util*** folder.
- The ***Solver*** folder.
- The ***SolverInterfaces*** folder.

- The ***Trellis_darwin*** folder.
- The ***Xfem*** folder.

The folder ***Trellis_darwin*** has the exact same source code than the folder ***Trellis***. The structure of ***Trellis*** which is divided in ***Util***, ***model*** and ***AOMD*** to be compiled as separate module libraries is put together as a whole block of code under darwin, given some cross referencing of compiled objects to create the module libraries. There is a structure that has to be respected on each folder, enclosed folders named the same are there for a reason.

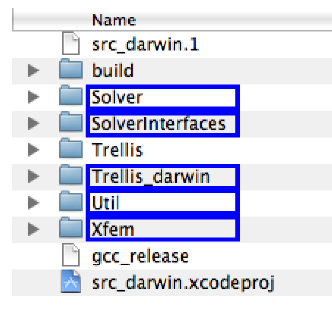


Figure A.1: Necessary source code folder structure

The file hacking process that was made is described:

1. The getarch script located on the `$(DEVROOT)/Util/buildUtil/buildUtil` folder was modified to fit another architecture other than Linux based or Windows. Proper naming of $(ARCH) and $(OS) where made.

2. A configuration file for the current architecture was made, named i386_darwin as would be the output of the *getarch* script. This file is located on `$(DEVROOT)/Util/buildUtil/buildUtil/config` folder, given the similarities with Linux based systems a copy of *x86_linux* file was made and the proper hacks to fit the architecture requirements were done.

    - Proper C/C++ flags were provided to enable compilation of objects under the given architecture.
    - Adequated C++ flags were changed to enable dynamic library creation, even though *-shared* flag is accepted by current customized gcc and g++ compilers on darwin the *-dynamiclib* is the proper flag that will allow better backward compatibility and the creation of dynamic libraries that form frameworks (apple's developing environments).
    - The *-fPIC* flag that corresponds to Position Independent Code, which allows easier cross-referenced compilation of objects, is complemented on darwin with *-fno-common*.

3. On the general structure of the provided compilation tools, the implementation of two general files in the build process of all pieces of code are extensively used. Both *make.initial* and *make.common* located on `$(DEVROOT)/Util/buildUtil/buildUtil` were edited to provide support for the new architecture. Both *make.initial* and *make.common* use the *getarch* script and also use variable definitions provided in the *i386_darwin* configuration file.

    - The use of the *getarch* script is implemented on the *make.initial* file to provide earlier architecture dependent linking options (library extension, paths). Given that *make.initial* is invoked at the beginning of every *Makefile* it is practical to add those lines.

4. The build process starts with the first building block from bottom to top. The folder Solver contains basic foundations that are referenced all across the code. Inside the Solver folder, darwin versions of superlu were compiled inside SuperLU/SuperLU_3.0 (see Fig. A.2).

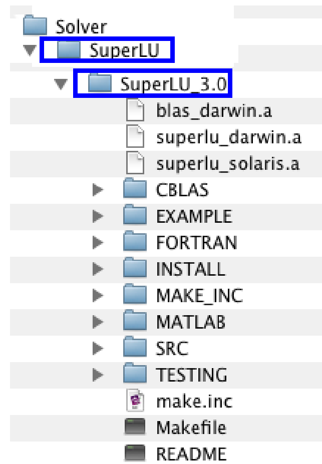


Figure A.2: Solver folder structure

5. The compilation process follows with the ***SolverInterfaces*** source code. In the *Makefile* used by X-Fem, the libraries called in the linking process are: ***Lapack SuperLu*** and ***SolverBase*** (See Fig. A.3).

    - The first library to be compiled was ***SolverBase***, since it is used by the other ones. The flags $(ADDLIBSTATIC) and $(REALAR) include the call to BLAS under darwin for the object compilation and subsequent linking. This was found to be under the framework architecture used on darwin. The use of proprietary implementations of BLAS is encouraged due to their tailoring to the architecture. The proper compilation flag " -DYA_BLAS -lblas -framework vecLib " has to be included as command line during the compilation of objects and the linking process.

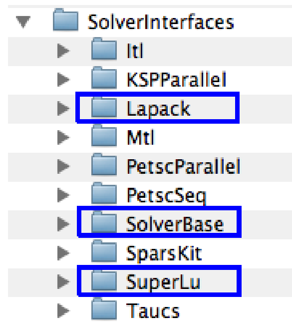


Figure A.3: SolverInterfaces folder structure

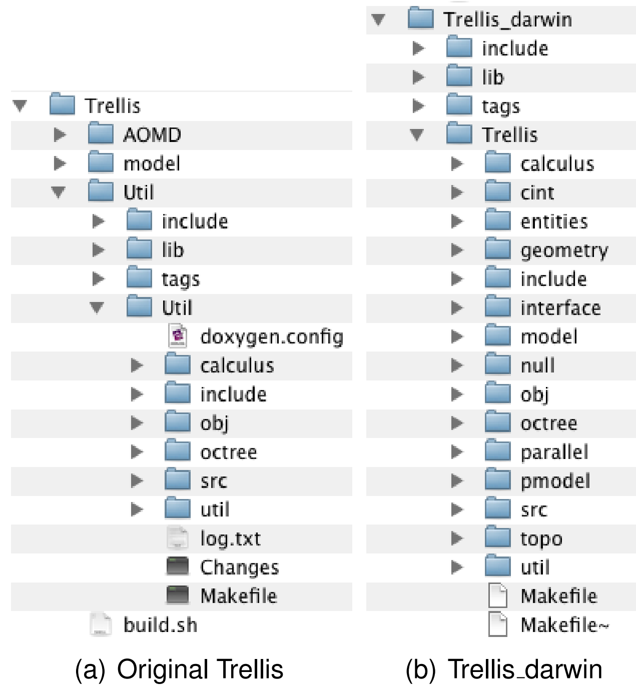
- The $(CXXFLAGS) were also updated to enable openMP support for #pragma directives.
- The library ***SuperLu*** references the ***SolverBase*** library and proper linking flags had to be provided for the ***SuperLu*** Makefile in order to enable compilation.
- The compilation of libraries ***SuperLu*** and ***Lapack*** was straightforward from the known changes and correct referencing to ***SolverBase***, BLAS, LAPACK, gfortran and SuperLU libraries.

6. The following module of the code to be compiled is the ***Trellis_darwin*** folder. As described before, the change made to the sub-module structure inside this folder in its original version obeys to the inability to compile the libraries using cross references to other objects outside the sub-module under darwin (See Fig. A.4).

   - The source code from ***model Util*** and ***AOMD*** folders was placed concurrently as a large ***Trellis*** under the ***Trellis_darwin*** folder. Namesake folders under all sub-modules (include, util, src, etc.) were carefully copied and its contents preserved (See Fig. A.4).
   - In the *Makefile* the variables $(SUBSYSNAME) and $(MODULENAME) were changed accordingly.
   - The expanded list of directories used on the object compilation was edited on *dirs* variable.
   - Also de $(DEPS) variable was edited to match the changes and for later dependency setup when compiling X-Fem.

7. The last module to be compiled its the Xfem source code itself, the compiled library will link against all of the previously described and built libraries. The following changes were made to the *Makefile*:

   - The variable $(DEPS) was changed to be architecture dependent, under Linux platforms the dependencies point to ***Trellis*** modules, while under Darwin they point to ***Trellis_darwin***.



(a) Original Trellis (b) Trellis_darwin

Figure A.4: Trellis_darwin folder structure

- Both $(ADDLIB) and $(SUPERLU) variables point correctly to the libraries needed by X-Fem.
- $(ADDSTATICLIB),$(CXXFLAGS) and $(REALAR) were changed properly depending on the architecture to offer openMP support and correct pointing to libraries at linking phase.

A.3 Porting The Project To Apple's XCode

Once the code was compiled, porting it into the Apple's programming environment XCode was considered the natural step to be taken. Using a developing environment makes the building, linking and debugging processes easier for the programmer. Navigating projects that have acquired a considerable size in source code becomes a difficulty that can be circumvented by the use of a developer environment such as XCode. Looking for functions on the linked libraries, navigating classes and definitions becomes easier and less time consuming.

It was important to have the same source working under Linux and Darwin operating systems for developing and benchmarking purposes. Also it was of interest to have under Darwin the same set of code building and running from the command line as well as an XCode project.

A.3.1 Setup of the project in XCode:

1. In the Command Line Utility project templates, a new project for a Standard Tool was created. This builds a command-line tool written in C.

2. Once the project was created (including its folders and dependencies), the target created by default was deleted (including its dependencies), also the default executables created were deleted.

3. Under the group "Target", a new target was created (right click over the group Target-Add - New Target...), Go to the category "Other" and select "External Target". This will allow to use the current ***Makefile*** structure and files.

4. The necessary source code was imported to the project (drag and drop), allowing for it to copy the files, not just referencing them.

5. Once the file groups were properly imported and created, the settings for the target were edited. By double clicking on the Target name the settings appear in a new window (it is not the same as the Target Info). The following settings are available for editing:

   - Under the "Custom Build Command", the "Build Tool" is invoked, by default points to /usr/bin/make, but this can be edited to gnumake from macports or any other build tool.
   - Under the "Custom Build Command", "Arguments" can be passed to the build tool. These arguments can be defined as environment variables in "Build Settings". Arguments used under the Xfem framework to compile and run the application can be defined here. Environment variables like *$(DEVROOT)* and common arguments such as "GCC_OPTIMIZATION_LEVEL", "setup", "VERS=opt" or "checkone" and "DIR" can be passed to the build tool easily as if it were from the command line.

6. To enable openMP support for external targets which call an external build system, the same process done to enable openMP support for the system had to be done on `/Developer/usr/bin`. By some reason the symbolic links done on `/usr/bin` to gcc, g++ and cc do not apply on XCode projects. When an external build tool is used, the project template calls the system default compiler, even if the template has been modified itself to add a $GCC_VERSION flag.

B. Setting Up CUDA In OS X

Greater and more powerful GPUs and architecture improvements that brought double precision to GPUs made possible a new standard of super computing for scientific simulations. With the advent of CUDA enabled GPUs into personal computing, the capacity to do scientific computing in personal computers or laptops became feasible. It is of interest to explore the advantages of such language extension developed by NVidia. The availability of many cores for data processing on a single machine (way more than typical CPUs) gives the developer the possibility to create and evaluate the performance and scalability of parts of the code subject to parallelization.

B.1 Set Up Of CUDA On OS X

Making CUDA development available on OS X comprehends mostly an straightforward task. The steps taken to do so are described:

1. The latest CUDA Toolkit (includes compiler, tools, libraries, header files and others), developer drivers and Software Development Kit (SDK) code samples from the developer site were downloaded, select `http://developer.nvidia.com/object/cuda_3_2_downloads.html#MacOS`.

2. A previous version of the CUDA Toolkit and GPU Computing SDK was uninstalled. The files from `/usr/local/cuda` and from `/Developer/GPU Computing`, the default installation locations were deleted. (Older versions of the SDK installed into `/Developer/CUDA` by default rather than `/Developer/GPU Computing`.)

3. All installers work out of the box and as long as the recommended paths are kept, not many hacks must be done on Makefiles and environment variables. The Driver was installed first, then the Toolkit.

4. The *.bash_profile* file under the home folder was edited and the following environment variables:

    - The PATH variable was appended `/usr/local/cuda/bin`.
    - The DYLD_LIBRARY_PATH variable was appended `/usr/local/cuda/lib`.

5. The last package to be installed was the CUDA SDK.
6. An installation check was done to verify the operability of the installed packages. A check to determine the tools could communicate properly with the CUDA enabled hardware, some of the examples were compiled and run for such purpose.
7. It was verified that the kernel extension was being loaded at boot-time.
8. **deviceQuery** was run from `/Developer/GPU Computing/C/bin/darwin/release` to probe the used device (see Fig B.1).

```
ext-130-66-205-64-wifi:release muerzo$ ./deviceQuery
./deviceQuery Starting...

 CUDA Device Query (Runtime API) version (CUDART static linking)

There is 1 device supporting CUDA

Device 0: "GeForce 9400M"
  CUDA Driver Version:                           3.20
  CUDA Runtime Version:                          3.20
  CUDA Capability Major/Minor version number:    1.1
  Total amount of global memory:                 266010624 bytes
  Multiprocessors x Cores/MP = Cores:            2 (MP) x 8 (Cores/MP) = 16 (Cores)
  Total amount of constant memory:               65536 bytes
  Total amount of shared memory per block:       16384 bytes
  Total number of registers available per block: 8192
  Warp size:                                     32
  Maximum number of threads per block:           512
  Maximum sizes of each dimension of a block:    512 x 512 x 64
  Maximum sizes of each dimension of a grid:     65535 x 65535 x 1
  Maximum memory pitch:                          2147483647 bytes
  Texture alignment:                             256 bytes
  Clock rate:                                    1.10 GHz
  Concurrent copy and execution:                 No
  Run time limit on kernels:                     Yes
  Integrated:                                    Yes
  Support host page-locked memory mapping:       Yes
  Compute mode:                                  Default (multiple host threads can use this device simultaneously)
  Concurrent kernel execution:                   No
  Device has ECC support enabled:                No
  Device is using TCC driver mode:               No

deviceQuery, CUDA Driver = CUDART, CUDA Driver Version = 3.20, CUDA Runtime Version = 3.20, NumDevs = 1, Device = GeForce 9400M


PASSED

Press <Enter> to Quit...
-----------------------------------------------------------
```

Figure B.1: **deviceQuery** output

B.2 Adding CUDA Support To XCode

Once the basic operability of the CUDA Toolkit has been proven other interest is to add the developing environment capabilities of XCode. For this purpose some other configuration steps must be taken. There is a plugin that enables the creation of project templates inside XCode. This allows to create basic code templates for CUDA projects, also sets by default the build options inside the project. Once the **NVCuda.pbplugin** (which can be found on: [18] - thread #79) was copied to `~/Library/Application Support/Developer/`

`Shared/XCode/Plug-ins`, under the File→New Project, User Templates section a CUDA project option appears. This allows to create the file structure and groups to compile the C/C++/CUDA tool, using the **nvcc** compiler and linking to the CUDA framework and libraries. Some of other configuration steps necessary to have full compatibility to build CUDA enabled code with some other characteristics are the following:

1. To enable openMP support and add that capability to the CUDA project build options, the default system compiler under OS X 10.5.x must be redirected to gcc 4.2 (system specific version offered by Apple as well within the system). See section A.1 for the changes to be made to the system compiler.

2. Also a simple project template can be found on: CUDA_XCode_Test_Project.zip (also [18]) with basic necessary file structure to start a project, kernel code template and main files both for host and device. Also the proper library includes are available, yet the build options had to be corrected. The build options set by default by the plugin are more accurate yet the compilation problem with the plugin is not encouraging. Also the plug-in saves the changes made to the build options template.

3. The most significant changes done to the build options within the code template are the following:

   - The option **Enable OpenMP Support** under build settings was activated.
   - The compiler version was set to GCC 4.2.
   - Added -fopenmp on **Linker Flags**.
   - Under linking options also the **Runpath Search Paths** was defined to: `/usr/local/cuda/lib`.
   - The **Header Search Paths** variable was set to: `/usr/lib/gcc/i686-apple-darwin9/4.2.1/include /Developer/GPU Computing/C/common/inc /usr/local/cuda/include`.
   - The **Library Search Paths** was set to: `/Developer/GPU Computing/C/lib /usr/local/cuda/lib`.
   - Under NVidia CUDA code generation section, the variable **Device Emulation** was checked off.
   - Any other libraries required for code to compile can be dragged and dropped into the project window and the correct links are made by the developing environment.

C. Including CUDA in the X-Fem Suite

CUDA is an architecture for parallel computing developed by NVIDIA which includes a set of instructions that allow the programmer to use the engine of the GPUs to perform massive amounts of computation [12].

To include the functionality offered by Nvidia's CUDA into the X-Fem library, certain preparations had to be done on the existing *Makefile* structure before recompiling the library.

CUDA has been designed to support different Application Programming Interfaces, the C language is one of the high-level languages in which routines have been implemented. The C syntax is maintained and the CUDA routines also resemble greatly the C programming model, the file extension is different though (*<filename>.cu*). Any source that contains this extension (*.cu*) must be compiled using **nvcc**, basically **nvcc** is a compiler driver that invokes all the required tools and compilers to successfully produce the binaries that run using the GPU resources [12]. It can output PTX or object code directly, as well as C code that has to be subsequently compiled using another tool. Is this latter output that is to be used the X-Fem code sources and compilation framework.

The provided source code and examples are meant to be compiled using a template *Makefile* by Nvidia, which uses a *make.common* that includes the API driver and the necessary dependencies. To have the capability to compile the object correctly using nvcc and linking them to correctly to the CUDA necessary libraries, as well as to the other X-Fem objects, the proper additions to the own *Makefile* structure from the X-Fem suite must be implemented.

The steps taken to do follow:

1. The *make.common* file was modified and the required lines and inference rules for compiling *.cu* source files to objects were added.

   - In order to detect the *.cu* source files and create the inference rule for object compilation the following variables were added:
     tempcu := $(foreach dir,$(dirs2), $(wildcard $(dir)/.cu))*
     srcscu := $(notdir $(tempcu)). This variable will hold the names of the *.cu* files to use in:
     objs := $(srcscpp:.cpp=.o) $(srcscxx:.cc=.o) $(srcsc:.c=.o) $(srcsf:.f=.o) $(srcsf2:.F=.o) $(srcscu:.cu=.o)

- Then the compilation rule is stablished for cuda files as:

  ```
  #For CUDA %:  %.cu
  @ echo ``--- Compiling'' $<
  ifeq ($(PURIFY),1)
  $(PURIFYCMD) $(NVCC) $(NVCCFLAGS) $(DEFS) $(NVINCLUDES) $(INCLUDES) $<
  -o $@ $(LDFLAGS)
  else
  $(NVCC) $(NVCCFLAGS) $(NVINCLUDES) $(INCLUDES) $< -o $@ $(LDFLAGS)
  endif
  ```

  The variable $(NVCC) defines the nvcc compiler -which should already be present after the driver API and Toolkit installation-. $(NVCCFLAGS) is defined wether for debug or release versions. Both variables are defined in the configuration file i386_darwin. The $NVINCLUDES and $LDFLAGS variables are modified respectively in the X-Fem *Makefile*. The proper header files with the respective paths and the necessary libraries are declared at this point. Also a $WCUDA flag is declared to append the $NVINCLUDES and $NVLDFLAGS to $INCLUDES and $LDFLAGS in order to provide support when compiling objects from the library that include CUDA routine calls and link them properly.

In the current implementation only one *.cu* file is used and both device and host code are implemented there, usually on cuda projects the device code -or kernels- are placed in different *.cu* files. Following the Nvidia's Makefile policy these files are not compiled as objects but rather treated as dependencies, it is also clear from how they are treated as header files and included on the main cuda file.

To allow the C/C++ files that comprise the Xfem suite to access the functions implemented in CUDA, external definitions of all functions are provided in a header file. Also the adequate header files are included here to provide access to definitions and functions from the CUDA Toolkit. This header file is meant to be included in any of the test tools developed under the X-Fem suite (See *cuda_interface.h*).

Once the library has been compiled, the implemented functions can be used in the test tools, given access to the capabilities of the GPU device to use in the computations performed by the X-Fem library.

D. Annex of results table

	m= 100	m= 200	m= 300	m= 400
	[ms]	[ms]	[ms]	[ms]
Memory allocation time (Device):	0.2135742186	0.195703125	0.244189453	0.3109375002
Memory alloc time (Device - cuEvents):	0.2039615988	0.1938304008	0.2414655986	0.3015168012
Memory init time (Dev C):	0.167138672	0.097998047	0.1436523438	0.2197753908
Memory init time (Dev C - cuEvents):	0.174015999	0.0969599994	0.1429248006	0.2192896008
Memory copying time (To device):	0.9111816408	0.366015625	0.697607422	1.2370605468
Memory copying time (To device - cuEvents):	0.9681280136	0.3641600012	0.6962752104	1.2272448062
Processing Time BLAS call (GPU):	5.3651855466	18.7578125	47.866210937	115.43535156
Processing Time BLAS call (GPU) - cuEvents:	5.362918377	18.76398735	48.331564331	115.47447052
GFlops to perform:	0.002	0.016	0.054	0.128
Memory copying time (To Host):	0.0959960936	0.2060058594	0.6575683594	0.4763671874
Memory copying time (To Host - cuEvents):	0.0956799996	0.2053248048	0.3335871994	0.4756672024
Total time (GPU):	7.0981933594	19.845800781	50.309033203	118.00673828
GFlop/s (GPU):	0.3728	0.853	1.129	1.1088
Memory allocation time (Host):	0.0240046978	0.0126187562	0.0108261584	0.011374593
Memory init time (Host C):	0.0618117334	0.2348300936	0.4612339974	0.8230043174
Processing Time BLAS call (CPU):	4.600557828	14.470757437	38.002832031	84.476017976
GFlop/s (CPU):	0.436	1.121	1.4216	1.5168
Total time (CPU):	4.6863742592	14.718206287	38.474892187	85.310396886

m= 500	m= 600	m= 700	m= 800	m= 900	m= 1000
[ms]	[ms]	[ms]	[ms]	[ms]	[ms]
0.353955078	0.3494140624	0.3678710938	0.3682128906	0.350078125	0.35790624996
0.335507196	0.276691197	0.2947199984	0.346566397	0.31100031792	0.312897021264
0.316796875	0.4075683592	0.5157714846	0.7071777344	0.4334179688	0.4761464844
0.3157184004	0.406086403	0.515027207	0.7068863988	0.432601602	0.47526400224
2.1405761718	3.0313476564	4.0276367186	4.9711914062	3.08156249996	3.450462890592
2.1412799836	3.031775999	4.0296512604	4.9734015464	3.08067071912	3.451355901704
200.55419922	339.72358398	467.5184082032	658.16484375	356.27927734372	404.448062499984
200.8629303	339.8274292	467.7926574708	658.3432373046	356.46014495848	404.657279846176
0.25	0.432	0.686	1.024	0.504	0.5792
0.718359375	0.948876953	1.2563964844	1.633886719	1.00677734376	1.112859375032
0.718368006	0.9499136092	1.2604672194	1.6381376026	1.00851072792	1.115079433024
204.68037109	344.8418457	474.2450683594	666.255029297	361.60581054692	410.325625000064
1.2468	1.2722	1.468	1.5556	1.33028	1.374576
0.0117507936	0.0114453314	0.011414051	0.011414838	0.0114799214	0.01150098708
1.5679960966	2.0996327878	2.8218437672	3.6768859386	2.19787258152	2.472846234344
164.9484129	212.95803945	335.9376074312	495.4425956964	258.7525346898	293.6078380326
1.5268	2.0316	2.0426	2.067	1.83696	1.900992
166.52815979	215.06911757	338.7708652494	499.130896473	260.96188719264	296.092185254048

m= 1100	m= 1200	m= 1300	m= 1400	m= 1500	m= 1600
[ms]	[ms]	[ms]	[ms]	[ms]	[ms]
0.358696484352	0.4072265624	0.3972167968	0.3742248437024	0.4244628904	0.39236551553088
0.3083749863168	0.3684096036	0.3746815976	0.33507270534016	0.349516803	0.34721113917139
0.50801640628	1.2498046876	1.4857910154	0.830635312496	1.8987792968	1.1946053437152
0.507173122608	1.2485055926	1.4851072074	0.8297303053696	1.898041582	1.19371156199552
3.7124402343504	10.9563476564	12.4996093754	6.74008453134048	16.3958007812	10.0608565157382
3.7133710853248	10.9622652054	12.5058752058	6.74270762346976	16.4027713774	10.0653980994789
445.226835156261	1957.520605469	2500.8203613282	1132.85902835943	3775.3268066408	1962.35072739074
445.416149755851	1958.2426757814	2501.8929687502	1133.33384381842	3777.1339355468	1963.20391473053
0.64504	3.456	4.394	1.915648	6.75	3.4321376
1.1917593750384	3.5654296876	4.0910156248	2.19356828124608	6.8892089846	3.5861963906569
1.1944217184288	3.5730624198	4.0986047744	2.19793581471456	6.6640512466	3.54561519478867
451.454675781277	1974.6135742186	2520.5321777344	1143.70637265625	3803.0518554688	1978.67173117187
1.4001312	1.7662	1.757	1.52563744	1.788	1.647393728
0.011451025776	0.0113742354	0.0111738444	0.0113960028112	0.0142328026	0.01192558219744
2.6538162618928	8.201579714	9.653156638	5.03585428595136	12.984777379	7.70583685576883
319.33972306008	1575.097989869	2017.4050156354	892.840620257376	3141.4348011972	1589.22363000381
1.9758304	2.1962	2.1784	2.01767648	2.1494	2.103501376
322.004990347818	1583.3109438182	2027.069346118	897.887870546141	3154.4338113782	1596.94139244167

m= 1700	m= 1800	m= 1900	m= 2000
[ms]	[ms]	[ms]	[ms]
0.4330078126	0.4654296876	0.458203125	0.5119628906
0.3071616022	0.3394751964	0.3386752038	0.4117888004
2.4247070312	2.6712402346	3.0166503906	3.359423828
2.4252287388	2.7942719936	3.4691520212	3.3581120016
21.678125	23.716455078	28.1324218752	28.5581054686
21.6848190308	23.7210819244	28.0572288514	28.5642498016
5454.7731933592	6539.4611328124	7540.2666503906	8851.769970703
5456.5692382812	6541.424609375	7542.3635742188	8852.937109375
9.826	11.664	13.718	16
7.8656249998	8.8755371094	10.4126464842	9.9907714844
7.8698623656	8.8789825438	9.4030591964	9.7258750916
5490.2454589844	6577.6160156248	7584.74453125	8895.076513672
1.8012	1.7834	1.819	1.8076
0.014827347	0.0159582138	0.0151960848	0.015562892
16.566174984	19.023039055	20.915775013	20.498177719
4794.478401947	5497.5916125058	6793.6134413958	7224.7155558586
2.0544	2.1312	2.034	2.2146
4811.059404278	5516.6306097746	6814.5444124938	7245.2292964696

Bibliography

[1] K. DRÉAU, N. CHEVAUGEON, and N. MOES, “Studied x-fem enrichment to handle material interfaces with higher order finite element,” *Computer Methods in Applied Mechanics and Engineering*, 2010.

[2] M. GARCIA, “Fixed grid finite element analysis in structural design and optimization,” Ph.D. dissertation, Department of Aeronautical Engineering, The University of Sydney, March 1999.

[3] N. KIKUCHI, *Finite Element Methods in Mechanics*, 1st ed. Cambridge University Press, 1986.

[4] O. ZIENKIEWICZ and R. TAYLOR, *The Finite Element Method - The Basis 1*, 5th ed. Butterworth-Heinemann, 2000.

[5] A. E. H. Love, *A Treatise on the Mathematical Theory of Elasticity*, 4th ed. New York: Dover publications, 1944.

[6] S. Brenner and L. R. Scott, *The Mathematical Theory of Finite Element Methods*. Springer-Verlag, 1994.

[7] N. MOES, J. DOLBOW, and T. BELYTSCHKO, “A finite element method for crack growth without remeshing,” *International Journal on Numerical Methods in Engineering*, no. 46, pp. 131–150, 1999.

[8] J. MELENK and I. BABUSKA, “The partition of unity finite element method: basic theory and applications,” *Computer Methods in Applied Mechanics and Engineering*, no. 139, pp. 289–314, 1996.

[9] N. SUKUMAR, N. MOES, T. BELYTSCHKO, and B. MORAN, “Extended finite element method for three-dimensional crack modeling,” *International Journal on Numerical Methods in Engineering*, no. 48, pp. 1549–1570, 2000.

[10] N. MOES, P. CLOIREC, P. CARTRAUD, and J.-F. REMACLE, “A computational approach to handle complex microstructure geometries,” *Computer Methods in Applied Mechanics and Engineering*, no. 192, pp. 3163–3177, 2000.

[11] N. SUKUMAR, D. CHOPP, N. MOES, and T. BELYTSCHKO, “Modeling holes and inclusions by level sets in the extended finite-element method,” *Computer Methods in Applied Mechanics and Engineering*, no. 190, pp. 6183–6200, 2001.

[12] *NVIDIA CUDA C Programming Guide*. Nvidia, 2010.

[13] D. GODDEKE and R. STRZODKA, “Performance and accuracy of hardware-oriented native-, emulated- and mixed-precision solvers in fem simulations,” *International Journal of Parallel, Emergent and Distributed Systems. Special Issue: Applied Parallel Computing.*, vol. 22, no. 4, pp. 221–256, 2009.

[14] P. MACIOL, P. PLASZEWSKI, and K. BANAZ, “3d finite element numerical integration on gpus,” *International Conference on Computational Science ICCS - Procedia on Computer Science*, no. 1, pp. 1093–1100, 2010.

[15] *CUDA Toolkit Reference Manual*. Nvidia, 2010.

[16] @gmsh, “Gmsh source code,” Web, Nov 2010, available at: http://geuz.org/gmsh/.

[17] C. GEUZAINE and J. REMACLE, “Gmsh: a three-dimensional finite element mesh generator with built-in pre- and post-processing facilities,” *International Journal for Numerical Methods in Engineering*, vol. 79, no. 11, pp. 1309–1331, 2009.

[18] @NVidiaForums, “Enabling cuda support on xcode,” Web, Nov 2010, available at: http://forums.nvidia.com/index.php?s=&showtopic=60336&view=findpost&p=562247.