An Investigation of Decoupling Single Element Solutions from the Mesh in the Finite Element Method on Nvidia GPUs

MSc. High Performance Computing

Trinity College Dublin
Coláiste na Tríonóide, Baile Átha Cliath
The University of Dublin

Author: Alex Keating
Student ID: 17307230
Supervisor: Jose Refojo & Prof. Kirk Soodhalter

Department of Mathematics
Faculty of Engineering, Mathematics and Science
Trinity College Dublin, University of Dublin

August 26, 2019

Declaration

I have read and I understand the plagiarism provisions in the General Regulations of the University Calendar for the current year, found at `http://www.tcd.ie/calendar`.

I have completed the Online Tutorial in avoiding plagiarism 'Ready, Steady, Write', located at

Alex Keating

August 26, 2019

Acknowledgements

Hiermit versichere ich an Eides statt, dass ich die vorliegende Arbeit selbstständig und ohne die Benutzung anderer als der angegebenen Hilfsmittel angefertigt habe. Alle Stellen, die wörtlich oder sinngemäß aus veröffentlichten und nicht veröffentlichten Schriften entnommen wurden, sind als solche kenntlich gemacht. Die Arbeit ist in gleicher oder ähnlicher Form oder auszugsweise im Rahmen einer anderen Prüfung noch nicht vorgelegt worden. Ich versichere, dass die eingereichte elektronische Fassung der eingereichten Druckfassung vollständig entspricht.

Abstract

[Abstract goes here (max. 1 page)]

Contents

List of Figures

List of Tables

List of Algorithms

Listings

Chapter 1

Introduction

Nowadays, with the advent of modern computing, there has been a huge push towards taking advantage of these resources. In the scientific world, there is a constant drive towards building optimised libraries for performing operations, spanning areas such as, basic linear algebra [CITE], fast Fourier transformations [CITE] or genetic algorithms for machine learning [CITE]. All of these libraries have been written with the intentions of exhausting as much processing power, memory and parallelisation as possible. These libraries are available across all kinds of architectures, Intel's MKL for example built for Intel CPUs, Nvidia's CUDA SDK written for their own GPUs or MAGMA, a 3rd party library written for heterogeneous architectures. These advancements in scientific computing have not come from nowhere, but rather are becoming duly necessary as most modern problems in science become impossible to solve without taking advantage of modern computing resources. A clear example of this was seen last year when the first image of a black hole was rendered, using a novel image cleaning machine learning algorithm and 5 petabytes of data - clearly something that cannot be done without some form of distributed memory architecture.

Nvidia have been among the leaders of this charge from both a hardware and software point...

The need for taking advantage of parallelism in modern science now clear, one problem that crops up in a vast array of areas is solving partial differential equations, or PDEs. These are relationships between variables and their partial derivatives and can be seen in areas such as fluid dynamics with the Navier-Stokes equations, in finance with the Black-Scholes equation or even in engineering when modelling stress of a structure - an important one for the topic at hand in this paper. Unfortunately, while analytic solutions exist for theoretical problems of this nature, studied in undergraduate courses, in the real world most of these PDEs are not of this convenient solvable nature and thus require numerical methods to solve. There are many variants of numerical methods which one can use to solve PDEs, some more simple than others, such as the finite difference method which decomposes the domain into a simple grid and approximates the derivatives, and some more complex but robust, like meshfree methods which create a collection of Voronoi cells on the domain instead of a basic grid - allowing for more complex structures. Certain methods land somewhere in a middle ground of both complexity and adaptability such as the finite volume method and the finite element method - the later of which will form the

basis of this study.

The finite element method [CITE STRANG] is a numerical method developed originally for use in engineering for modelling stress on structures and has since quickly expanded to use in all branches of science such as electrostatics, something and something. Its engineering foundations will become clear throughout the paper as much of the terminology has remained unchanged. This paper will investigate current approaches to applying the finite element method on Nvidia GPUs and attempt at isolating and pinpointing certain bottlenecks for parallelism upon which may be improved. The importance of this is clear, as PDE-related problems get larger and more complex, the need for more computing power is evident to get approximate solutions in reasonable amounts of time.

SECTION ON EXISTING FEM LIBRARIES

1.1 Preliminaries

Maybe ???

Chapter 2

Finite Element Method

This paper focuses its mathematics on implementing the finite element method to solve PDEs. This chapter will go through the necessary mathematics behind each of the steps in the finite element method such as variational calculus, approximation theory and finite elements themselves. The chapter also goes through the approach itself, alongside the worked example for the case of this study of the Laplace equation.

2.1 General Problem

Before the nuts and bolts of the finite element method are discussed, first consider an n-dimensional, general n^{th} order PDE of the form,

$$f\left(\mathbf{x}; u(\mathbf{x}), \frac{\partial u}{\partial x_1}, \ldots, \frac{\partial u}{\partial x_n}; \frac{\partial^2 u}{\partial x_1 \partial x_1}, \ldots, \frac{\partial^2 u}{\partial x_i \partial x_j}, \ldots, \frac{\partial^2 u}{\partial x_n \partial x_n}; \ldots\right) = 0, \tag{2.1}$$

where $\mathbf{x} = \{x_1, x_2, \ldots, x_n\}$. For a case of a 2-dimensional, 2^{nd} order PDE, this results in ending up with the operator,

$$\mathcal{L} = a\frac{\partial^2 u}{\partial x^2} + b\frac{\partial^2 u}{\partial x \partial y} + c\frac{\partial^2 u}{\partial y^2} + F\left(x, y; u; \frac{\partial u}{\partial x}, \frac{\partial u}{\partial y}\right). \tag{2.2}$$

This can be used, by applying it to a function $u(\mathbf{x})$, leaving a PDE,

$$\mathcal{L}u = 0, \tag{2.3}$$

which is going to be basis for this study - attempting to approximate the function $u(\mathbf{x})$. This problem can then be bounded by three types of non-homogeneous boundary conditions in order to be properly posed and have a unique/non-trivial solution:

1. Dirichlet condition: $u = g(s)$.
2. Neumann condition: $\frac{\partial u}{\partial \mathbf{n}} = h(s)$.
3. Robin condition: $\frac{\partial u}{\partial \mathbf{n}} + \sigma(s)u = k(s)$,

where s is the arc length of the boundary C and $\mathbf{n}$ is a vector, externally normal to C. How these conditions are imposed and handled is discussed in Section later in the report.

2.2 Approximation Theory

Suppose there exists some function $u(x)$ which is required to be approximated. The most common way to is to estimate the value of the function is by using a collection of *basis functions* $\psi_i(x)$, and unknown coefficients, c_i, giving,

$$u(x) \approx \sum_{i=0}^{N} c_i \psi_i(x). \tag{2.4}$$

There are a collection of ways to construct your basis functions and create a linear system, and obtain solutions to the approximation Equation (2.4) and this paper looks at three in particular:

- least squares.
- Galerkin.
- weighted residuals.

There are other methods of approximation such as collocation and regression but they are not discussed here. Before these approaches are explained, two things must first be defined. Firstly, consider a function space V, defined by the span of set of basis functions,

$$V = \text{span}\{\psi_0, \ldots, \psi_N\}, \tag{2.5}$$

then it can be said the any function $u \in V$ can be written as a linear combination of the basis functions,

$$u(x) = \sum_i c_i \psi_i. \tag{2.6}$$

Consider now, functions f, g - the squared-norm or inner-product of these two functions is defined as,

$$\langle f, g \rangle = \int f(x) g(x) dx \tag{2.7}$$

2.2.1 Least Squares

Consider a function $f(x)$ which needs to be approximated by a function $u(x) \in V$ as defined in Equation (2.6). The most obvious way to approximate this function would be to minimise the differential between the two, $f - u$. Subbing this into the inner product defined in Equation (2.7) leaves the error,

$$e = \langle f - u, f - u \rangle = \langle f - \sum_i c_i \psi_i, f - \sum_i c_i \psi_i \rangle, \tag{2.8}$$

$$e = \langle f, f \rangle - 2 \sum_i c_i \langle f, \psi_i \rangle + \sum_{i,j} c_i c_j \langle \psi_i, \psi_j \rangle. \tag{2.9}$$

Of course, now as is well known from optimisation, to minimise this residual function, its derivative must be taken at each of the N points, $\left\{ \frac{\partial e}{\partial c_i} \right\}_N$. Evaluating and setting $\frac{\partial e}{\partial c_i} = 0$,

results in the equation,

$$-\langle f, \psi_i \rangle + \sum_j c_j \langle \psi_i, \psi_j \rangle = 0, \quad i \in \{0, \ldots, N\}, \tag{2.10}$$

which can equally be written as,

$$\sum_j A_{i,j} c_j = b_i, \tag{2.11}$$

where,

$$A_{i,j} = \langle \psi_i, \psi_j \rangle, \tag{2.12}$$

$$b_i = \langle f, \psi_i \rangle. \tag{2.13}$$

Now there is a system of linear equations which can be solved by usual means to obtain the approximation $u(x)$ of the function $f(x)$. Mathematically, it is equivalent to say that the method of least squares utilises the inner-product of the residuals, and solves by minimising it,

$$\min_{c_0,\ldots,c_N} \langle e, e \rangle, \tag{2.14}$$

giving the system of $N+1$ equations,

$$\left\langle e, \frac{\partial e}{\partial c_i} \right\rangle = 0, \quad i \in \{1, \ldots, N\}. \tag{2.15}$$

2.2.2 Galerkin

In the previous subsection, it was shown that the least squares method operates by minimising the error term between the two functions, or alternatively, forcing the error to be orthogonal to the function space V. However, in reality, the true error is not actually known, since f is not explicitly known, and thus instead a residual R is used. Take for example, Equation (2.3), if we sub in an approximation $\hat{u} = \sum_i c_i \psi_i$, we get,

$$R = \mathcal{L}\left(\sum_i c_i \psi_i\right) \neq 0. \tag{2.16}$$

Now, the residual R can be made orthogonal to the space V by imposing,

$$\langle R, v \rangle = 0, \quad \forall v \in V, \tag{2.17}$$

and since any function in V can be approximated using Equation (2.6), it can be said that,

$$\langle R, \psi_i \rangle = 0, \quad i \in \{1, \ldots, N\}, \tag{2.18}$$

thus leaving another system of linear equations to solve. This is alternatively known as projecting R onto V.

2.2.3 Weighted Residuals

The method of weighted residuals is a relatively simple generalisation of the standard Galerkin method. Rather than imposing that the residual is orthogonal to the space V known as the trial space, it is chosen to be orthogonal to some alternate space W, known as the test space. This leaves the equation,

$$\langle R, v \rangle = 0, \quad \forall v \in W, \tag{2.19}$$

where,

$$W = \text{span}\{w_0, \ldots, w_n\} \tag{2.20}$$

and so this leaves,

$$\langle R, w_i \rangle = 0, \quad i \in \{1, \ldots, N\}, \tag{2.21}$$

again, leaving a linear system of $N + 1$ equations.

2.3 Variational Calculus

2.3.1 Weak Formulation

As it should be clear by now, the overall aim of the finite element method is the minimise a residual in order to get an approximation of the function u as seen in Equation (2.3). What may not seem immediately obvious at a first glance, but an important thing to factor in, Equation (2.3) is not how the PDE problem is posed when trying to apply FEM. In fact, usually, the problem is posed in its *weak formulation*. The weak form of this equation is one which contains at most, first-order derivatives, as opposed to its strong form containing second-order derivatives. When only approximating a normal function defined in a Hilbert space, this issue wouldn't crop up. However, when dealing with calculus of variations, one must remember that boundary conditions must be imposed and reducing the order of derivatives weakens the demands on the test and trial functions, allowing them to not need be continuous in their second derivative.

Take for example a general PDE defined,

$$\mathbf{v} \cdot \nabla u + \beta u = \nabla \cdot (\alpha \nabla u) + f, \quad \mathbf{x} \in \Omega \tag{2.22}$$

$$u = u_D, \quad \mathbf{x} \in \Gamma_D \tag{2.23}$$

$$-\alpha \frac{\partial u}{\partial \mathbf{n}} = g, \quad \mathbf{x} \in \Gamma_N, \tag{2.24}$$

multiplying this by a test function v, and getting the inner product over the domain Ω, just like was seen in the method of weighted residuals leaves the equation,

$$\int_\Omega (\mathbf{v} \cdot \nabla u + \beta u) v \; d\mathbf{x} = \int_\Omega \nabla \cdot (\alpha \nabla u) v \; d\mathbf{x} + \int_\Omega f v \; d\mathbf{x}. \tag{2.25}$$

Applying Green's lemma (CITE)? to the second-order term then results in the following

equation,

$$\int_\Omega \nabla \cdot (\alpha \nabla u) v \; d\mathbf{x} = -\int_\Omega \alpha \nabla u \cdot \nabla v \; d\mathbf{x} + \oint_\Gamma \alpha \frac{\partial u}{\partial \mathbf{n}} v \; ds + \int f v \; d\mathbf{x}. \tag{2.26}$$

Since the boundary integral is 0 on Γ_D and g on Γ_N, this can be rewritten as,

$$\int_\Omega \nabla \cdot (\alpha \nabla u) v \; d\mathbf{x} = -\int_\Omega \alpha \nabla u \cdot \nabla v \; d\mathbf{x} + \oint_{\Gamma_N} g v \; ds + \int f v \; d\mathbf{x}, \tag{2.27}$$

and the Equation (2.25) can be shown in terms of the inner product defined in Equation (2.7),

$$\langle \mathbf{v} \cdot \nabla u, v \rangle + \langle \beta u, v \rangle = -\langle \alpha \nabla u, \nabla v \rangle + \langle g, v \rangle_N + \langle f, v \rangle . \tag{2.28}$$

This PDE has now been transformed into its weak formulation. Looking at the Section 2.2.3, it's quite obvious now that u can be approximated by subbing in an estimate, applying the method of weighted residuals and achieving a system of linear equations. Thus the solution space has been reduced to a finite dimensional space.

2.3.2 Functionals

UNFINISHED SECTION

While regular calculus deals with changes in ordinary variables, calculus of variations by contrast deals with changes in functions - handling special functions known as functionals, which take functions as inputs compared to just variables. These are beneficial for the FEM as, considering the problem posed in the previous section, much of what is trying to be accomplished is finding the function which minimises an integral i.e. minimising a functional. Take for example, the functional

$$F[y(x)] = \int_\Omega f(x, y(x), y'(x)) \; dx, \tag{2.29}$$

in this instance, one would search for what function $y(x)$ would minimise the integral.

Before discussing how to derive these functionals, first look at an abstract notation for variational forms discussed in the previous sections. Take a function with trial functions u, and test functions v, supported on V, then define the problem as,

$$a(u, v) = L(v), \quad v \in V, \tag{2.30}$$

where $a(u, v)$ is in bilinear form, containing all terms which have both test and trial functions, whereas, $L(v)$ is linear, containing only test functions. This is equivalent to the integrals seen in Equation (2.28), which is looked to be to minimised. This equation is equivalent to minimising the functional,

$$I[v] = \frac{1}{2} a(u, v) - L(v). \tag{2.31}$$

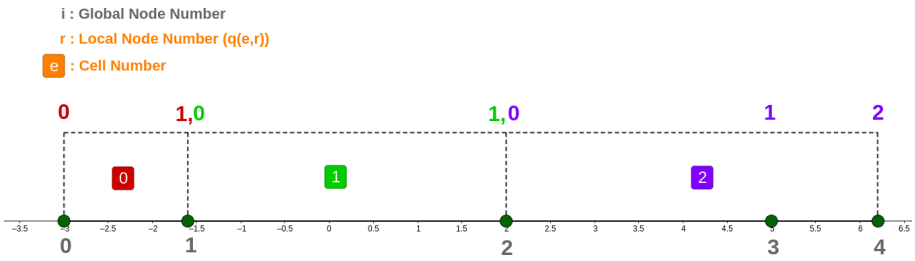

Figure 2.1: INSERT DETAILED CAPTION HERE

NEED TO CITE THIS.... In a multidimensional case, consider the PDE,

$$-\nabla(\alpha\nabla u) + \beta u = f, \quad \mathbf{x} \in \Omega \tag{2.32}$$

$$u = u_D, \quad \mathbf{x} \in \Gamma_D \tag{2.33}$$

$$-\alpha\frac{\partial u}{\partial \mathbf{n}} + \sigma(s)u = k, \quad \mathbf{x} \in \Gamma_R \tag{2.34}$$

ADD DERIVATION HERE. The resulting functional is,

$$I[u] = \int_\Omega (\nabla u \cdot \alpha\nabla u) + \beta u^2 - 2uf)\ d\mathbf{x} + \oint_{\Gamma_R} (\sigma u^2 - 2uk)\ ds \tag{2.35}$$

These functionals are an alternative and mathematically equivalent variant on attempting to minimising the problem at hand and a commonly seen in papers discussing the FEM. PAGE 234!!

2.4 Finite Elements

Up to this point, the basis functions that have been used have all been defined across the entire domain Ω. In this section, piecewise polynomial basis functions will be used, defined with *compact support* across what are known as *elements* or *cells*. These cells will contribute their own weighted value to the linear system in an assembly process, generating an overall linear system $Lu = b$, where L is known as the global stiffness matrix and b the stress vector. This system will result in the solution to the PDE problem at hand.

Remark. *As is convention in the FEM to refer to basis functions as $\varphi(x)$, from here on out in the paper, $\psi_i(x)$ will be replaced by $\varphi_i(x)$ - though note they are completely equivalent.*

2.4.1 Cells & Basis Functions

For illustrative purposes, a single-dimensional problem is used here to demonstrate the actual concept of the finite elements or cells. Consider now a domain,

$$\Omega = \Omega^{(0)} \cup \Omega^{(1)} \dots \Omega^{(N_e)}, \tag{2.36}$$

where

$$\Omega^{(i)} \cap \Omega^{(j)} = \emptyset \quad \forall i, j \in \{0, \dots, N_e\}. \tag{2.37}$$

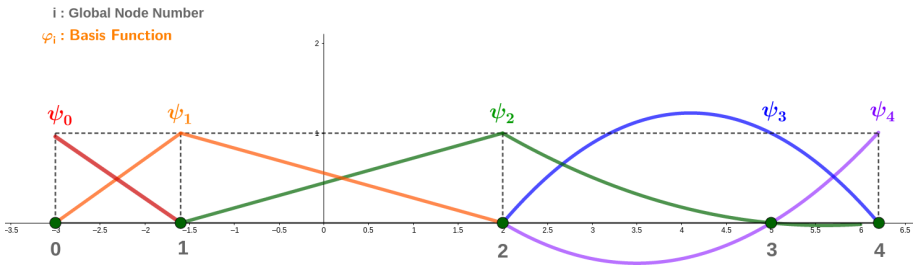

Figure 2.2: CAPTION THIS

Within this domain, define N_n nodes, spaced out across the domain any particular amount. Suppose a node has a *global index* $i \in \{0, \ldots, N_n\}$ and are laid out in no particular order. We define the node's *local index* in cell e as $r \in \{0, \ldots, d\}$, again these do not need to necessarily be in order or equally spaced. Now that the local and global numbering of the nodes are defined, we define the function,

$$i = q(e, r), \tag{2.38}$$

where $q(e, r)$ is a mapping function from local index r in cell e, back to the node's global index across the domain. Figure 2.1 demonstrates these definitions for a simple, 3 cell, 1D case. Basis functions must next be defined across the domain in order to make an approximation for u. Consider a node, globally numbered i, locally numbered r, on cell e, with $d+1$ nodes in said cell, its corresponding basis function $\varphi_i(x)$ is defined as a Lagrange polynomial of degree d,

$$l_d(x) = \prod_{\substack{0 \leq m \leq k \\ m \neq j}} \frac{x - x_m}{x_j - x_m} \tag{2.39}$$

which is 1 at node r and 0 everywhere else. If the node is internal, then the function is simply defined as is, if the node is shared with a neighbouring cell, then the basis function is a piecewise combination of the Lagrange polynomial from both cells which share the node. The value d is known as the degree of freedom and is related to the order of polynomial used as the basis function - the higher the order, the more nodes per cell. More often than not, cells are made up into triangular or tetrahedral shapes. However, they can be any shape such as squares or octagons, so long as the degree of freedom mapping is correct. Figures 2.2, 2.3 illustrate these piecewise polynomials in both 1D and 2D. Now it can be said that

$$u(x) \approx \sum_{\mathcal{I}_s} c_i \varphi_i(x), \tag{2.40}$$

where $\mathcal{I}_s$ is the set of indices for which $u(x_i)$ is unknown.

2.4.2 Assembling the Stiffness Matrix

The next piece to the FEM puzzle is a linear system must be assembled from these basis functions in order to solve for the unknowns $\{c_i\}_{\mathcal{I}_s}$ and achieve an approximation for u. It was shown in Section 2.3.2, that a variational form PDE can be written in an abstract

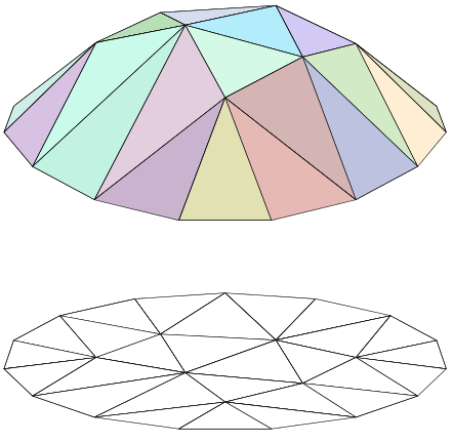

Figure 2.3: 2D representation of linear/P1 basis functions on a mesh.

means form as,

$$a(u, v) = L(v), \quad v \in V. \tag{2.41}$$

With that in mind, consider the PDE from Equation (2.28), moving all the terms with both test and trial function to the left-hand side, this can be rewritten as,

$$\langle \mathbf{v} \cdot \nabla u, v \rangle + \langle \beta u, v \rangle + \langle \alpha \nabla u, \nabla v \rangle = \langle g, v \rangle_N + \langle f, v \rangle \,, \tag{2.42}$$

or subbing in Equation (2.40), leaves,

$$\sum_{\mathcal{I}_s} \left(\langle \mathbf{v} \cdot \nabla \varphi_j, \varphi_i \rangle + \langle \beta \varphi_j, \varphi \rangle + \langle \alpha \nabla \varphi_j, \nabla \varphi_i \rangle \right) c_j = \langle g, \varphi_i \rangle_N + \langle f, \varphi_i \rangle \,, \tag{2.43}$$

which clearly demonstrates a linear system,

$$\sum_{\mathcal{I}_s} A_{i,j} c_j = b_i, \tag{2.44}$$

where,

$$A_{i,j} = \langle \mathbf{v} \cdot \nabla \varphi_j, \varphi_i \rangle + \langle \beta \varphi_j, \varphi_i \rangle + \langle \alpha \nabla \varphi_j, \nabla \varphi_i \rangle \,, \tag{2.45}$$

$$b_i = \langle g, \varphi_i \rangle_N + \langle f, \varphi_i \rangle \,. \tag{2.46}$$

Equation (2.43) shows clearly now why it was important to convert the PDE into its weak formulation. Now there are only first-order derivatives of the basis and trial functions which need to be continuous. Had it been left in strong form these would have been second-order.

This has now shown how the overall stiffness matrix would be calculated over the entire domain Ω, but is this achieved this from the elements and their basis functions? The main sell of the FEM is that the domain can be decomposed into elements with easier to evaluate integrals and are robustly able to form many complex overall domain shapes, and then assemble these results into the global stiffness matrix. Remembering that these basis functions are compactly supported, it can be seen that $A_{i,j}$ can be assembled by

incrementing the integral result from all cells which contain nodes i, j i.e.

$$A_{i,j} \coloneqq A_{q(e,r),q(e,s)} + \widetilde{A}^{(e)}_{r,s} \tag{2.47}$$

$$b_i \coloneqq b_{q(e,r)} + \widetilde{b}^{(e)}_r, \tag{2.48}$$

where $\widetilde{A}^{(e)}$ and $\widetilde{b}^{(e)}$ are known as the element matrix and element vector, respectively, and r, s are local node numberings as defined in Equations (2.38). Both of these evaluate the same integrands, but over their compact support instead of the entire domain. Thus, looking at Equations (2.47), (2.48), both of their element evaluated counterparts on cell e become,

$$\widetilde{A}^{(e)}_{r,s} = \langle \mathbf{v} \cdot \nabla\varphi_j, \varphi_i \rangle_{\Omega^{(e)}} + \langle \beta\varphi_j, \varphi_i \rangle_{\Omega^{(e)}} + \langle \alpha\nabla\varphi_j, \nabla\varphi_i \rangle_{\Omega^{(e)}}\,, \tag{2.49}$$

$$\widetilde{b}^{(e)}_r = \langle g, \varphi_i \rangle_{\Gamma_N \cap \Omega^{(e)}} + \langle f, \varphi_i \rangle_{\Omega^{(e)}}\,. \tag{2.50}$$

Take the small P1, 2D example seen in Figure 2.4. Clearly, in this example, each cell contains three nodes, so the resulting element matrix $\widetilde{A}^{(e)} \in \mathbb{R}^{3\times 3}$ and element vector $\widetilde{b}^{(e)} \in \mathbb{R}^3$. Evaluating both of these, by the nature of all the calculations being dot products, results in a symmetric positive definite element matrix,

$$\widetilde{A}^{(e)} = \begin{bmatrix} \widetilde{A}^{(e)}_{0,0} & \widetilde{A}^{(e)}_{0,1} & \widetilde{A}^{(e)}_{0,2} \\ \widetilde{A}^{(e)}_{1,0} & \widetilde{A}^{(e)}_{1,1} & \widetilde{A}^{(e)}_{1,2} \\ \widetilde{A}^{(e)}_{2,0} & \widetilde{A}^{(e)}_{2,1} & \widetilde{A}^{(e)}_{2,2} \end{bmatrix}, \tag{2.51}$$

and element vector,

$$\widetilde{b}^{(e)} = \begin{bmatrix} \widetilde{b}^{(e)}_0 \\ \widetilde{b}^{(e)}_1 \\ \widetilde{b}^{(e)}_2 \end{bmatrix}, \tag{2.52}$$

Now it can be illustrated quite easily how to assemble the global stiffness matrix A and stress vector b from this,

$$A = \begin{bmatrix} \widetilde{A}^{(0)}_{0,0} & \widetilde{A}^{(0)}_{0,1} & \widetilde{A}^{(0)}_{0,2} & 0 \\ \widetilde{A}^{(0)}_{1,0} & \widetilde{A}^{(0)}_{1,1} + \widetilde{A}^{(1)}_{0,0} & \widetilde{A}^{(0)}_{1,2} + \widetilde{A}^{(1)}_{0,2} & \widetilde{A}^{(1)}_{0,1} \\ \widetilde{A}^{(0)}_{2,0} & \widetilde{A}^{(0)}_{2,1} + \widetilde{A}^{(1)}_{2,0} & \widetilde{A}^{(0)}_{2,2} + \widetilde{A}^{(1)}_{2,2} & \widetilde{A}^{(1)}_{2,1} \\ 0 & \widetilde{A}^{(1)}_{1,0} & \widetilde{A}^{(1)}_{1,2} & \widetilde{A}^{(1)}_{1,1} \end{bmatrix}, \tag{2.53}$$

$$b = \begin{bmatrix} b^{(0)}_0 \\ b^{(0)}_1 + b^{(1)}_0 \\ b^{(0)}_2 + b^{(1)}_2 \\ b^{(1)}_1 \end{bmatrix}. \tag{2.54}$$

Normally, A would be a sparse SPD matrix, but since this is a small case the matrix is more or less dense. More on that later.

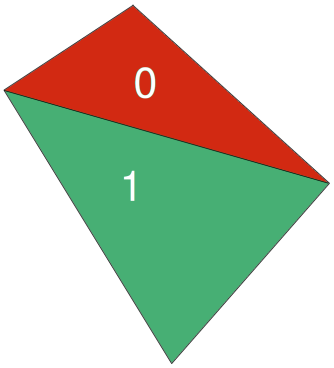

Figure 2.4: Small two cell example of a mesh with global and local node numberings illustrated. NOT FINISHED

2.4.3 Enforcing Boundary Conditions

Boundary conditions have been conveniently ignored up to this point in the report for simplicity's sake when explaining how to assemble the linear system. However, they of course cannot in reality go without being implemented. Thankfully, certain boundary conditions are easier to handle than others. Looking at Equations (2.43), the first term on the right-hand side is a contour integral over the boundary where the Nuemann condition applies. Subbing in g into the inner-product here has now in fact dealt with the Nuemann condition - this is known as a *natural boundary condition*, where it is handled naturally as part of the integration by parts or Green's lemma.

The Dirichlet conditions on the other hand, are a little more tricky - these must be manually enforced and are thus known as *essential boundary conditions*. Robin conditions contain both essential and natural components so won't be looked at here as these are simply separated into the two components and handled as usual. There are a number of ways of handling the essential boundaries, one would be to restrict your degrees of freedom set $\mathcal{I}_s$, such that it no longer contains any of the nodes which are boundaries and are thus emitted from the stiffness matrix as they are not unknowns, since $A \in \mathbb{R}^{\dim\mathcal{I}_s \times \dim\mathcal{I}_s}$. In this instance, the function being approximated would look like,

$$u(x) \approx \sum_{j \in \mathcal{I}_b} U_j \varphi_j(x) + \sum_{j \in \mathcal{I}_s} c_j \varphi_{\widetilde{j}}(x), \tag{2.55}$$

where $\mathcal{I}_b$ is the set of all boundary points, U_j is the boundary value at node j and $\widetilde{j}$ is the updated node index for $\mathcal{I}_s$, less the boundary nodes. This can cause some hassle with bookkeeping as the nodes' indices are being messed around with which can cause headaches when mapping back to the global indices.

Instead of this method, modification of the linear system approach was taken, whereby all nodes remain in the set of unknowns' indices. Since the boundary values are exact solutions at that particular point of the system, the corresponding row and column with that node are set to be all 0, barring 1 on the diagonal, and the RHS to be equal to the boundary value. This will force the system to give, $1 \times c_j = U_j \implies u(x_j) = u_j$ for

a boundary value at global node number j. To achieve this, the element matrices are assembled as normal, and then a four step linear algebra operation is performed,

1. $b_i \leftarrow b_i - A_{i,j}U_j \quad \forall i \in \mathcal{I}_s$,
2. $A_{i,j} = A_{j,i} \leftarrow 0$,
3. $A_{j,j} \leftarrow 1$,
4. $b_j \leftarrow U_j$.

Looking at the example in Equation 2.51 and Equation 2.52, if supposing an essential boundary condition U_0 is enforced at local node 0, the updated element matrix would be,

$$\widetilde{A}^{(e)} = \begin{bmatrix} 1 & 0 & 0 \\ 0 & \widetilde{A}^{(e)}_{1,1} & \widetilde{A}^{(e)}_{1,2} \\ 0 & \widetilde{A}^{(e)}_{2,1} & \widetilde{A}^{(e)}_{2,2} \end{bmatrix}, \tag{2.56}$$

and updated element vector,

$$\widetilde{b}^{(e)} = \begin{bmatrix} U_0 \\ \widetilde{b}^{(e)}_1 - \widetilde{A}^{(e)}_{0,1}U_0 \\ \widetilde{b}^{(e)}_2 - \widetilde{A}^{(e)}_{0,2}U_0 \end{bmatrix}. \tag{2.57}$$

These can now be assembled into the global stiffness matrix and stress vector just as before without any other changes necessary.

2.4.4 Physical or Local Coordinates

The last outstanding piece of mathematics that needs to be touched upon in the FEM is the coordinate mapping of the nodes. In Equation 2.43, clearly the integrals or inner-products are all calculated with respect to the physical coordinates of $mathbfx$, across the domain upon which they are defined - compact or whole domain. However, as was mentioned, much of the advantage of the FEM is that you can shape a complex domain into many smaller domains with easier to calculate integrals. In this case, it might be much more within one's interest to locally define the coordinates in each cell and then map back. For example, take a 2D, P1 case again, it might be much more convenient for the node's local coordinates to be (0,0), (0,1) and (1,0) in any cell. For example, to apply this mathematically, denote the local coordinate system by $\mathbf{X}$ and the new, easier local basis functions $\widetilde{\varphi}_r(\mathbf{X})$, and now consider the integral,

$$\int_{\Omega^{(e)}} \alpha(\mathbf{x}) \nabla\varphi_i(\mathbf{x}) \cdot \nabla\varphi_j(\mathbf{x}) \, d\mathbf{x}. \tag{2.58}$$

using the transformation,

$$\nabla_{\mathbf{X}}\widetilde{\varphi}_r(\mathbf{X}) = \mathbf{J} \cdot \varphi_i(\mathbf{x}), \tag{2.59}$$

where $\mathbf{J}$ is the Jacobian,

$$\mathbf{J} = \begin{bmatrix} \frac{\partial x_0}{\partial X_0} & \frac{\partial x_1}{\partial X_0} \\ \frac{\partial x_0}{\partial X_0} & \frac{\partial x_1}{\partial X_1} \end{bmatrix}. \tag{2.60}$$

Subbing these in results in the equivalency,

$$\int_{\Omega^{(e)}} \alpha(x)\nabla\varphi_i(\mathbf{x}) \cdot \nabla\varphi_i(\mathbf{x})\ d\mathbf{x} = \int_{\widetilde{\Omega}^{(e)}} \alpha(\mathbf{x}(\mathbf{X}))(\mathbf{J}^{-1} \cdot \nabla_{\mathbf{X}}\ \widetilde{\varphi}_r(\mathbf{X})) \cdot (\mathbf{J}^{-1} \cdot \nabla_{\mathbf{X}}\ \widetilde{\varphi}_s(\mathbf{X}))\ |\mathbf{J}|\ d\mathbf{X}, \tag{2.61}$$

where $\widetilde{\Omega}^{(e)} = [0,1] \times [0,1]$. This mapping can easily be performed on all of the terms in Equation (2.43). The clear advantage here is, previously the basis functions were written as Lagrange polynomials, now since the range of values only spans from 0 to 1,

$$\widetilde{\varphi}_0(\mathbf{X}) = 1 - X_0 - X_1, \tag{2.62}$$

$$\widetilde{\varphi}_1(\mathbf{X}) = X_0, \tag{2.63}$$

$$\widetilde{\varphi}_2(\mathbf{X}) = X_1, \tag{2.64}$$

thus leaving far easier integrals to evaluate.

2.5 Implemented Example

2.5.1 Problem Statement

Given that the intent of this paper is to investigate novel approaches to solving PDEs using the FEM on GPUs, it wouldn't make much sense to code a very complicated PDE for testing purposes. Instead, the 2D Laplace equation was used with a standard rectangular boundary - mainly as it is non-transient/steady state. It is defined as,

$$\nabla \cdot (\alpha \nabla u(\mathbf{x})) = 0, \quad \mathbf{x} \in \Omega, \tag{2.65}$$

$$u(\mathbf{x}) = U_0, \quad \mathbf{x} \in \Gamma_{D0}, \tag{2.66}$$

$$u(\mathbf{x}) = U_1, \quad \mathbf{x} = \Gamma_{D1}, \tag{2.67}$$

$$\frac{\partial u}{\partial \mathbf{n}} = 0, \quad \mathbf{x} \in \Gamma_N, \tag{2.68}$$

where

$$\begin{aligned} \Omega &= [a, c] \times [b, d], \\ \Gamma_{D_0} &= \{\mathbf{x} : x_0 = a\}, \\ \Gamma_{D_1} &= \{\mathbf{x} : x_0 = b\}, \\ \Gamma_N &= \{\mathbf{x} : x_1 = c \ ||\ x_1 = d\}. \end{aligned}$$

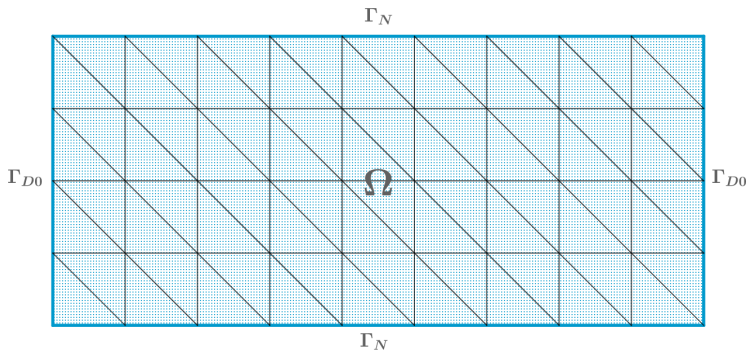

Figure 2.5: Domain of the Laplace problem to be solved.

This, when converted into its weak formulation by Green's identity gives,

$$\int_\Omega \nabla \cdot (\alpha \nabla u(\mathbf{x}))\ v\ d\mathbf{x} = -\int_\Omega \alpha \nabla u \cdot \nabla v\ d\mathbf{x} + \int_{\Gamma_N} \alpha \frac{\partial u}{\partial \mathbf{n}}, \tag{2.69}$$

$$= -\int_\Omega \alpha \nabla u \cdot \nabla v\ d\mathbf{x}. \tag{2.70}$$

So it can be said that,

$$A_{i,j} = \langle \alpha \nabla \varphi_j, \nabla \varphi_i \rangle \tag{2.71}$$

$$b_i = 0. \tag{2.72}$$

Figure 2.5 demonstrates the mesh being arranged into downward sloping, triangular cells where the node numbering is done in anti-clockwise direction to maintain orientation of the integrals. For more convenience, for the sake of the paper, to avoid manually calculating the Jacobian and performing numerical integration, as seen in Section 2.4.4, P1 problems were used and some handy analytical solutions to Equation (2.71) exist which only work for 2D, P1, triangular meshes. If the physical, non-local, coordinates of a nodes in a cell are defined by, $x_i, y_i\ \forall i \in \mathcal{I}_s$, then define two constants,

$$\beta_i = y_j - y_k, \tag{2.73}$$

$$\gamma_i = x_k - x_j, \tag{2.74}$$

and the area of the cell,

$$\Delta = \frac{1}{2} \begin{vmatrix} 1 & x_i & y_i \\ 1 & x_j & y_j \\ 1 & x_k & y_k \end{vmatrix}. \tag{2.75}$$

Without going into large amounts of detail, it can be deduced mathematically that Equation (2.71), on a compact support,

$$\int_{\Omega^{(e)}} \alpha \nabla \varphi_j \cdot \nabla \varphi_i\ d\mathbf{x} = \int_{\Omega^{(e)}} \frac{\Delta^2}{4} (\beta_i \beta_j + \gamma_i \gamma_j)\ d\mathbf{x} \tag{2.76}$$

$$= \frac{\Delta^3}{4} (\beta_i \beta_j + \gamma_i \gamma_j). \tag{2.77}$$

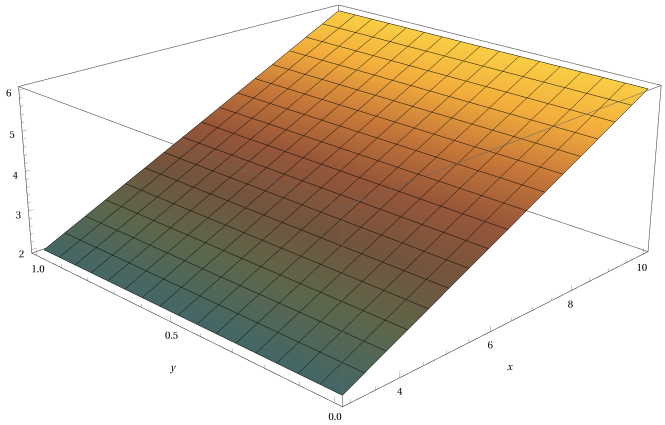

Figure 2.6: MATHEMATICA SOLUTION CAPTION

2.5.2 Analytical Solution

This given solution has the analytical solution,

$$u(x,y) = U_1 + \left(\frac{U_1 - U_0}{b-a}\right)(x-a). \tag{2.78}$$

Figure 2.6 illustrates this solution plotted in Wolfram Mathematica.

Chapter 3

Design & Implementation

Now that the fundamental mathematics of numerical technique has been covered, in this chapter, the general approach to implementing the finite element method in serial is discussed. The design structure of the code is explained, such as data structures and classes used, and storage approaches such as the benefits of applying sparse storage formats and sparse solvers. The use of linear algebra libraries, specifically Intel's MKL, are detailed as they were utilised to avoid reinventing the wheel when solving the final linear system.

3.1 General Approach

Given that there is a lot of machinery involved in programming the FEM as a numerical method, steps were taken to break the code up into as many logical partitions as possible and structure it in a such a way which made sense from a glance. All the serial code was written in C++11, allowing use of object oriented programming and many of the newer features of the standard library (STL) such as auto types and ordered sets. A large benefit here also was the inbuilt RAII features of C++, reducing the need for manual memory management - important here when it will become clear there is much memory to be allocated to code a FEM implementation.

3.1.1 Design Structure

Figure 3.1 illustrates the overall design structure of the code - in both serial and parallel, though the GPU code will be discussed in more detail in Section 4.4. The code is split into four main files and their respective headers,

- `main.cc` - Main C++ code, used for invoking the FEM operation and calling results to be output.
- `mesh.cc` - Contains the code for the Mesh class, storing all necessary operations and data structures to construct a mesh.
- `fem.cc` - Code for a FEM class which contains the routines needed to perform the method on a given mesh.

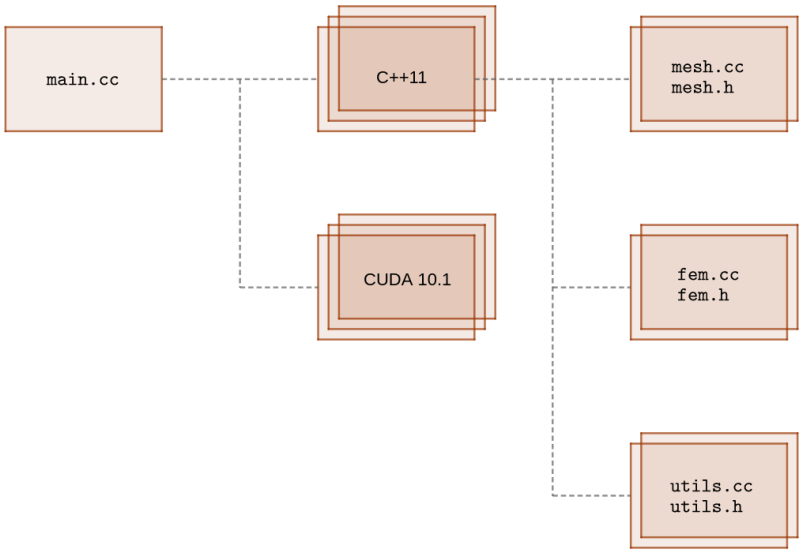

Figure 3.1: Code structure of the C++ code implementation.

- `utils.cc` - General utility functions, such as SSE, parsing of command line arguments et cetera.

The approach here would be that a Mesh object be created, passed to a FEM object as an input parameter to a constructor, the FEM would complete the routine and the utility functions would perform sundry operations after this to tidy up. Figure 3.2 demonstrates a flowchart of the serial code process.

3.2 Data Structures

Objects are a massive sell for using C++ over C when programming a numerical method, for reasons stated already. For this paper, two classes were written, one to handle the mesh creation and one to handle performing FEM. In terms of other data structures, there was also a struct `Tau` to manage timings of all the operations without needing to pass around multiple variables in as function parameters.

3.2.1 Mesh

The mathematics of the mesh has been illustrated, the next logical step would be to port it over to code. The data structure for the mesh holds a handful of key components and routines needed for its generation. Given the mesh itself is of course, a collection of interconnected nodes on a plane, upon each of which numerical calculations must be performed, it was naturally necessary to stores these nodes and their corresponding coordinates. In the implementation of this paper, a 2D mesh was used, though the extension to higher dimensions is rather arbitrary - another benefit of the FEM. The nodes' coordinates were stored in an array, `vertices`, which had a dimension of `order`, where order is the number of nodes or number of unknowns. Looking back at the Section 2.4, however, simply knowing the nodes coordinates for the FEM isn't enough, one also needs to know their global indices and degree of freedom mapping back to the stiffness matrix. These were respectively

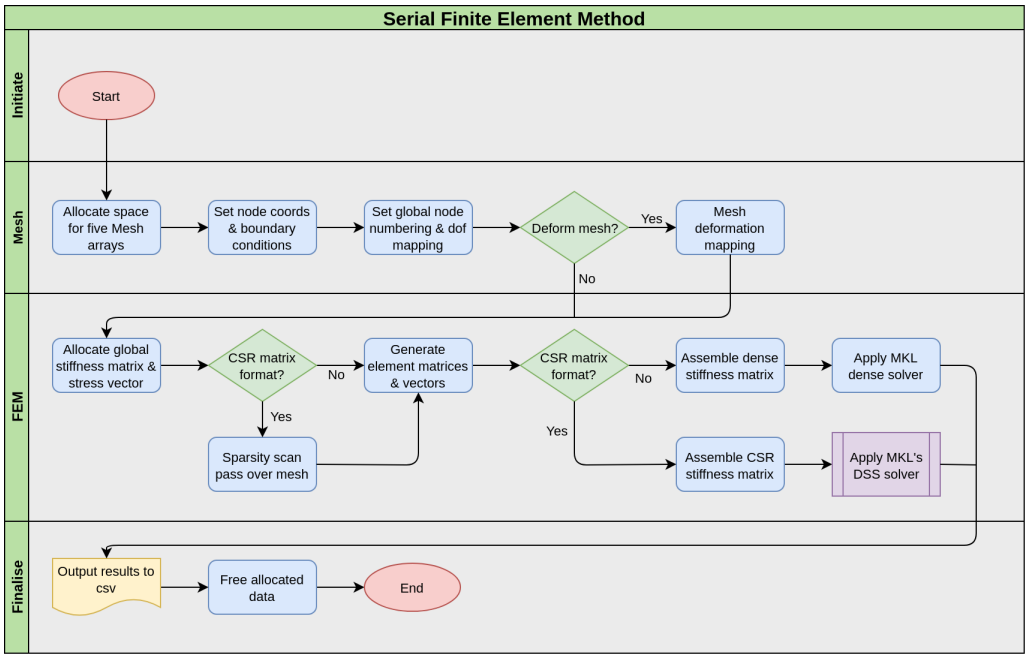

Figure 3.2: Flowchart of serial FEM implementation.

stored in two 2D arrays, `cells` and `dof`. Note that this is the exact same as performing the mapping function $q(e, r)$ seen previously. For example, `dof[e][r]` will provide the global node index of local node r in cell e. Figure (REFERENCE) gives an example of a small mesh, the populated version of these three arrays are described in (LISTING). Note also in this example that each cell's array is numbered in an anti-clockwise direction - this is important for the orientation of the integrals calculated later. Since in this paper, only P1 examples were used, the degree of freedom mapping and global node indices are actually equal and so having two separate arrays for these was rather unnecessary but it allows for potential further expansion of the code with ease.

Remark. *Standard library vectors were not used for the 2D arrays as the contiguity of the data was important for transferring the data to the GPU and for certain linear solvers.*

In terms of what routines are contained in the Mesh class, during instantiation, the constructor creates a generic, rectangular mesh and populates the three arrays. There is a routine, whereby one can pass a mapping function as a parameter and deform the mesh. The aim of the deformation function is simply the fact that most FEM meshes aren't standard rectangles, apart from this particular test case - since the focus here is on a GPU optimisation. Other routines in the class then are merely to return various properties of the object: coordinates, pointers to the arrays (for CUDA purposes), number of nodes et cetera. There is also a routine to do a pass over the mesh and return its sparsity pattern which is detailed in Section 3.3.2.

3.2.2 FEM

Once the mesh has been generated, the next aim for the software was to create as generic a FEM class as possible, one which could take any mesh in the structure of the Mesh class,

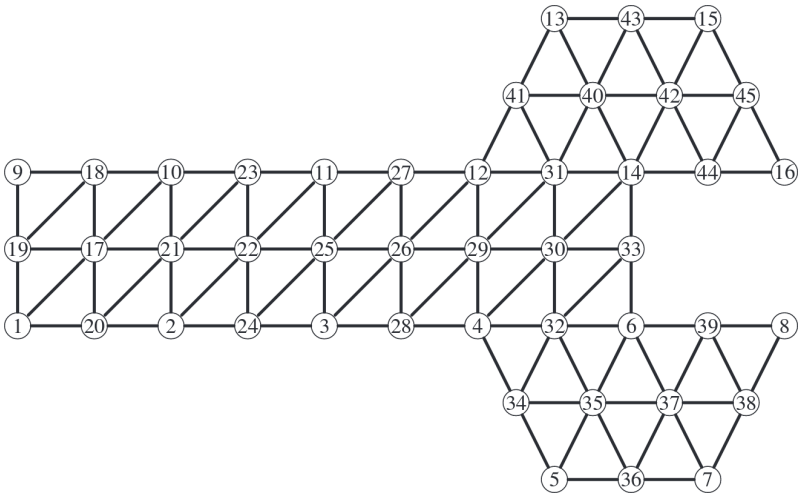

Figure 3.3: Mesh for generic FEM problem illustrated as a graph representation.

and solve the given PDE - in this report's case the Laplace equation. The FEM class will, of course, need to allocate data to store the global stiffness matrix and global stress vectors. The stress vector is simply stored in `b` and has a dimension of `order`, the stiffness matrix on the other hand can be stored in either dense or CSR format. Since CSR is stored in three separate, single dimensional arrays, this allowed the benefit of using standard library vectors, also eliminating the need for `new` and `delete`. The dense format on the other hand is 2D and has the same issue with contiguity as mentioned in the remark above. The CSR matrix is stored in `valsL`, `rowPtrL` and `colIndL` and the dense matrix is stored in `L`. The class then also stores certain properties needed, such as the dimensions of the problem `order`, the number of cells in the mesh and the number of non-zeros in the sparse matrix.

From a routine perspective, this is where all the real nuts and bolts of the FEM come into play. There are three primary routines here which need to be completed in order to obtain a solution:

1. element matrix generation.
2. global stiffness matrix and stress vector assembly.
3. linear system solver.

The element matrix calculation is detailed in Algorithm (REFERENCE), utilising the convenient formulas stated in Section 2.5.1. The assembly routine on the other hand, has two different variants, one to handle the dense matrix assembly and one to handle assembling in CSR. Algorithms (REFRENCEx2) show both variants of this assembly. The last main routine, solving the linear system was handled by Intel's MKL library, taking advantage of pre-existing, optimised kernels.

3.3 Sparse Storage

The idea of sparse matrices and CSR storage has been thrown about in this paper a handful of times. The idea here is that a matrix is considered sparse, if if consists of

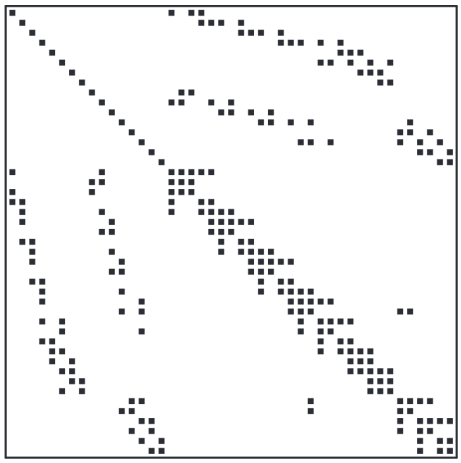

Figure 3.4: Sparsity pattern of FEM mesh.

mostly 0 valued entries - in some papers it is defined as having over 50% 0 entries but this is a rather loose definition. This is important in the FEM as, if the basis functions are considered, one must remember that these are defined on a compact support $\Omega^{(e)}$, and will be 0 everywhere outside of this domain. Due to this, the global stiffness matrix will be a SPD, sparse matrix. Consider again Figure (REFERENCE), at no point is there any connection between nodes X and Y, thus the resulting inner product $\langle \varphi_X, \varphi_Y \rangle = 0$. Why is this important? It can be hugely beneficial to computation time if one can take advantage of the sparsity pattern of a matrix, clearly as there is going to be potentially orders less null operations to iterate through.

These matrices can be stored in a number of different ways, three will be looked at here, although only CSR was used in the implementation:

- CSR.
- CSC.
- COO.

CSR or *compressed sparse row* storage is the most commonly used variant of sparse storage. As mentioned, it works by storing the matrix into three separate arrays, A, containing all the non-zero values, IA, the indices of the beginning of each row in the array A and has a length of $n + 1$ where n is the number or rows, and finally JA, the indices of the column each value in A is in. For more clarity, consider the matrix,

$$A = \begin{bmatrix} 3 & 1 & 0 & 0 & 0 \\ 0 & 20 & 15 & 0 & 1 \\ 0 & 0 & 0 & 4 & 1 \\ 9 & 1 & 6 & 8 & 0 \\ 0 & 0 & 0 & 7 & 0 \\ 0 & 0 & 0 & 0 & 11 \end{bmatrix}, \tag{3.1}$$

in CSR format this can be written as,

```
 A = [  3  1  20  15  1  4  1  9  1  6  8  7  11  ]
IA = [  0  2  5  7  11  12  13  ]
IJ = [  0  1  1  2  4  3  4  0  1  2  3  3  4  ]
```

In the case of this implementation, `valsL` = A, `rowPtrL` = AI and `colIndL` = IJ. An important thing to note is that these can be either 0-based indexing or 1-based indexing. Since this report didn't utilise any FORTRAN code, 0-based indexing was used in all cases.

Compressed sparse column is almost identical to CSR storage, except it is column-major format instead. The same example matrix in CSC would give,

```
 A = [  3  9  1  20  1  15  6  4  8  7  1  1  11  ]
IA = [  0  3  0  1  3  1  3  2  3  4  1  2  5  ]
IJ = [  0  2  5  7  10  13  ].
```

Clearly here, IJ has dimensions $m + 1$, where m is the number of columns.

The last commonly used sparse storage format is *coordinate lists* or COO, which stores the values and their coordinates in both row and column. Keeping with the same example gives,

```
 A = [  3  1  20  15  1  4  1  9  1  6  8  7  11  ]
IA = [  0  0  1  1  1  2  2  3  3  3  3  4  5  ]
IJ = [  0  1  1  2  4  3  4  0  1  2  3  3  4  ].
```

This can of course be done in either row-major or column-major directions.

3.3.1 Graph Theory

Looking at the sparsity of a matrix, this can be equated to graph traversal problems in graph theory. If we consider a graph problem $G(V, E)$, defined as a set of vertices,

$$V = \{v_0, v_1, \ldots, v_n\}, \tag{3.2}$$

and a set of edges E, consisting of pairs of vertices v_i, v_j, such that,

$$E \subseteq V \times V. \tag{3.3}$$

From a sparse matrix perspective, this graph can be thought of as the connectivity between elements in the matrix, i.e. if $\{v_i, v_j\} \in E$, then it can be shown that for a sparse matrix A, $A_{i,j}$ is non-zero. Looking at Figure 3.3, demonstrating a mesh for a FEM problem in the form of a graph. Clearly all the edges and connectivity are clear from the figure, but this also gives us the inherent sparsity patter in the resulting matrix. Figure 3.4 demonstrates this resulting global stiffness matrix. Thinking back to how the FEM, particularly its elements were defined, this result makes sense. The basis functions were defined on a compact support of touching nodes, so clearly if $\{v_i, v_j\} \notin E$, then the resulting inner-product,

$$\langle \varphi_i, \varphi_j \rangle = 0. \tag{3.4}$$

3.3.2 Sparsity Scan

Naturally, the next question needing to be posed is, how does one take advantage of this inherent sparsity pattern available in the stiffness matrix. The easiest, although there have been papers published on more advanced methods (CITE), is to run a single pass over the mesh to begin, taking the computational hit, and using this to create a sparsity pattern. In this implementation, the pass created a vector of STL sets, one set for each node, and iterated through each, amending to the set if there was an edge between the two nodes. Using this, one now has the number of non-zeros and can populate quite easily, IA and IJ, in the CSR matrix. Algorithm (REFERENCE) details how this is explicitly performed.

3.4 Solver Libraries

A substantial part of the computation time in the FEM, relies on actually solving the final linear system. While this report is aimed at optimising the FEM in general, it does not aim at optimising linear solvers - something which has been covered by countless PhD students and doctors over many years. Instead, a decision was made to simply use a pre-existing library for both convenience and efficiency's sake. There are many variants that could have been used, GNU Standard Library, MAGMA, LAPACK etc. The one which was used in the end was Intel's MKL.

3.4.1 Intel's MKL

The Intel library was split into two separate sub-libraries: their LAPACK routines for dense systems, and their Direct Sparse Solver (DSS). Both were used in this paper for comparison reasons. In either case, given that the stiffness matrix A, is symmetric positive definite (SPD), both dense and sparse solvers work by initially factorising the matrix using Cholesky decomposition, $A = L^T L$, where L is lower-triangular. This new system is now solved using a direct solver. The DSS also has an option of providing a reordering of the sparse matrix, prior to the Cholesky decomposition, such as Cuthill-McKee (CITE), if one so wishes to make the sparsity pattern more compact, and thus more efficient for solving. Figure 3.5 demonstrates the potential benefits of reordering, a much more compact data set, allowing for far more coalesced operations and less distance to travel along memory (CITE MATLAB SITE). This option was left on "auto" for the purposes of this report. CITE INTEL'S DOCUMENTATION

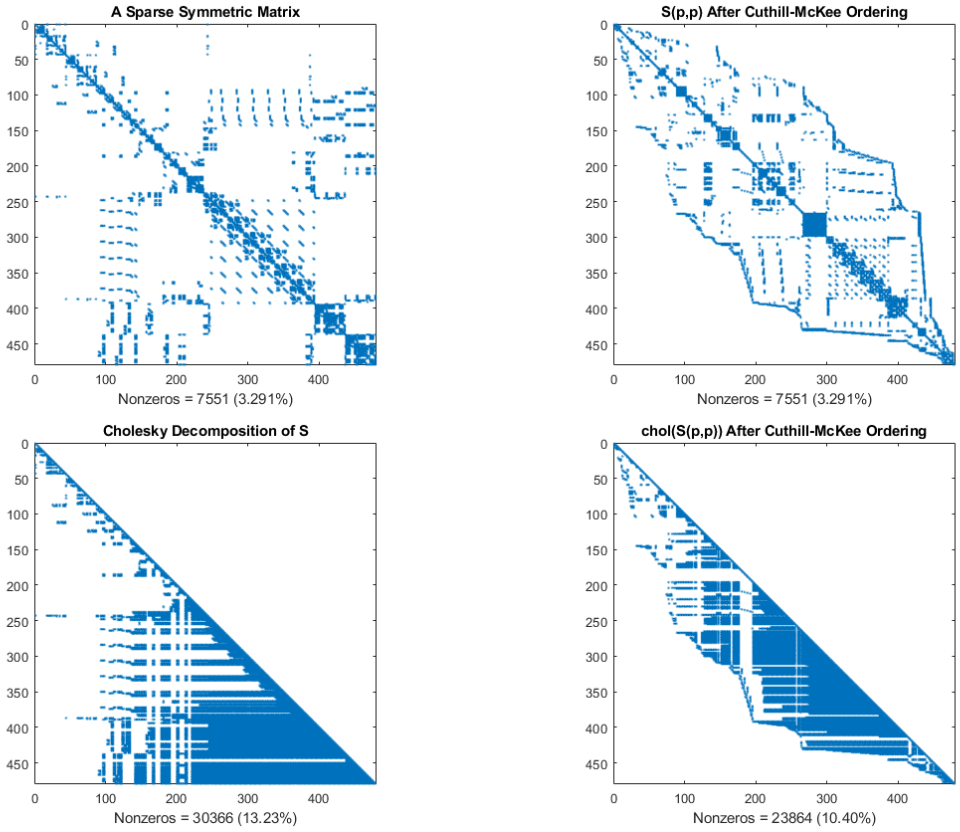

Figure 3.5: Illustration of a sparsity pattern of a matrix, pre and post Cuthill-McKee reordering, allong with the resulting Cholesky decompositions of both matrices.

Chapter 4

GPU Implementations

In scientific computing, GPU programming can be hugely beneficial to large-scale problems. Due to the highly parallelisable architecture of GPUs, massive speedups can be seen. In this chapter, the heterogeneous, GPU programming model is discussed, as well as how this can be then implemented to apply the FEM. The design structure of the code written for this paper is detailed, as well as going through the theory of existing approaches.

4.1 GPU Programming & CUDA

In GPU programming, there are multiple options currently available. Currently on UNIX systems, the main two would be CUDA and openCL, others exist such as openGL and DirectX. They each have their own benefits, openCL is open source and can be used on any device with graphical processing abilities, from a phone to a cluster, while CUDA is only available solely for Nvidia devices. While all having their differences, they do all share the same general approach, this section details the model needed to program a GPGPU over a CPU.

4.1.1 GPU Programming Model

While, as stated, GPUs have the potential to provide huge speedups, this is not necessarily a given. There is much to consider as a programmer when writing code which are not traditionally have to consider when writing code for CPUs - in serial, shared memory or even distributed architectures. CPUs traditionally consist of multiple cores, usually to the order of four to sixteen, sitting on a chip, with CISC architectures - meaning it is written to perform complicated instructions at with high clock speeds. These chips often contain up to three levels of automated caches, with buffers, preventing thrashing by first-in-first-out or least-recently-used strategies. They regularly have extra chips called accelerators for performing specific calculations like cryptography or data compression and can have quite complex scheduling algorithms. In short, CPU chips are very clever, but fall down when it comes to high levels of parallelism.

GPUs, on the other hand, are made up of thousands of CUDA cores, sitting on what are known as Streaming Multiprocessors - often about eight or so per SM. Each of these

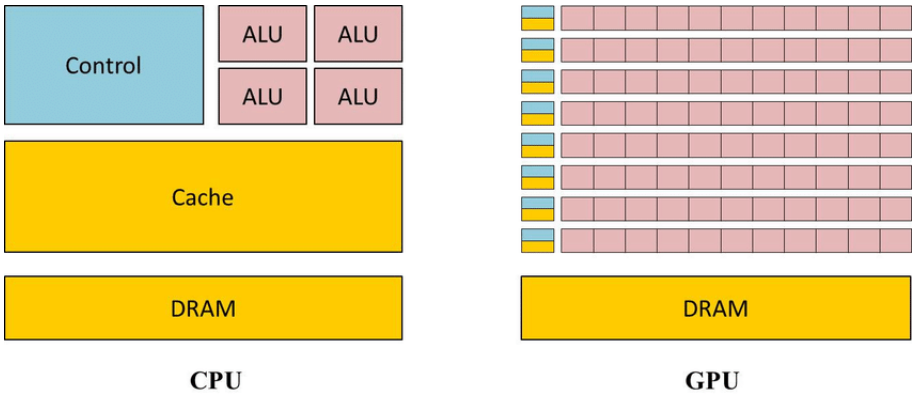

Figure 4.1: Simplistic comparison of CPU vs. GPU architectures.

Figure 4.2: Illustration of Nvidia's current Turing architecture, showing its SMs and Ray Tracing cores.

SMs are arranged into blocks of threads, making up an overall grid. The threads inside each block are what allow these massive amounts of parallelisation. On the current Turing architectures, a block can have a max of up to 1024 threads per block, for example. Considering that there can be thousands of CUDA cores, it is now how this can be hugely beneficial. Figure (REFERENCE) shows this architectural design structure. So what is the downside? The first, and most evident thing to consider, is these cores are very basic. They have SIMD instruction sets, similar to RISC on CPUs, multiple less complex control units and have slower clock rates. GPUs do also have accelerators on them, such as the newer Tensor Cores and Ray Tracing cores, as seen in Figure 4.2 showing Nvidia's diagram of the Turing architecture, but these are relatively recent and not much used as of yet. Figure 4.1 shows a relatively basic illustration of the difference in architectures.

While slow clock-speed and simple instruction sets are the main drawbacks of using a GPU, there are other considerations to be made. The first, and almost certainly the most important when it comes to writing fast CUDA code is memory management. Unlike in

Von Neumann, and other more modern CPU architectures, all of the memory management is left up to the programmer on a GPU and comes in various hierarchical levels. The largest bank of memory is *global memory*, this is akin to the RAM of the GPU in terms of capacity order and is the slowest memory to access as it is physically furthest from the cores. The benefit of global memory is it is both the largest and also accessible to all cores on the card. There are other faster memory banks such as constant, texture, and surface which have various advantages and disadvantages such as being closer to the chip but non-programmable once set et cetera. However, there are two main types of memory which are most advantageous to take advantage of, *shared memory* and *thread registers*. Register memory is stored per thread and sits physically on top of the core. It is as fast as one can possibly get, but is limited in size. Similarly, shared memory also sits on top of the core, the main difference is that shared memory is distributed across all threads in a given block and is usually around 48KB in modern architectures. Figure (REFERENCE) illustrates the memory hierarchy on a GPU. Unlike in CPUs, the programmer must decide where the various data needed to run the program will be stored, so clearly managing this correctly will be hugely important.

The final, main thing to consider, is what are known as warps. On the GPU, threads are organised into groups of 32. These groups of 32 threads all receive the same instructions at the same time. If some of the threads do not need to perform a given instruction, they will wait until the rest of the warp has completed the instruction until receiving another. Clearly, this can cause some bottlenecking. Figure (REFERENCE) demonstrates how this can apply to a specific case. The best thing one can do when taking warps into consideration is to attempt to set block sizes as multiples of 32 - since usually blocks are all performing the same or similar instructions.

In terms of actually putting this all to use, in CUDA, the code is structured by splitting the functions up into three classifications:

1. `__host__` functions, are functions executed on the CPU or host. These are usually left without being denoted.

2. `__global__` functions, also known as kernels. These are invoked by the host, and are a way of initiating the CUDA code - these are executed on the GPU. Usual inputs into these include, block and grid dimensions, shared memory allocation and streams.

3. `__device__` functions are GPU functions which are called on by the device - these cannot be called by the host.

Clearly then, the main difference between programming on a CPU compared to on a GPU is the onus put on the programmer to manage scheduling and memory management amongst other things without the assistance of the device. Thus, CUDA code must inevitably be more basic and more-so a collection of simple executions - it lends itself best to large scale operations of basic calculations.

4.1.2 CUDA Libraries

While many of the GPU programming variants like openCL and openGL have their pros and cons, the main reason for choosing CUDA for this paper is, for one, Nvidia cards were used for testing so the performance would be optimal, but more-so due to the extensive availability of libraries for CUDA - something which openCL does not have. Intel's MKL was discussed in Section 3.4.1, and likewise, CUDA has its own collections libraries of mathematical functions. In Nvidia's case, these span from LAPACK, fast-Fourier transforms to image decomposition. In the case of this paper, cuBLAS, cuSOLVER and cuSPARSE were used. This is discussed in the breakdown of the GPU code's design structure in Section 4.1.2. Similarly to the serial code, these were used for getting the Cholesky decomposition of a linear system and applying a direct solver. The cuBLAS package was used for finding the 2-norm of the error for each iteration seen in FEMSES.

4.2 Current FEM Approaches

4.2.1 Linear System Decomposition

4.2.2 Multigrid Techniques

4.2.3 Domain Decomposition

4.3 FEMSES

The finite element method single-element solutions (or FEMSES) approach is the key study in this paper. Developed with the aim of decoupling solutions to the element matrices, from assembling the global stiffness matrix, aiming to reduce a large computational cost from the existing methods seen in Section 4.2.

Traditionally, the FEM has four main steps: create the mesh, generate the element matrices and element vectors, assemble the global stiffness matrix and stress vector and lastly, solve the linear system. Of course, the global stiffness matrix and stress vector can be quite a large linear system, the idea of FEMSES is to decouple this from the overall process by implementing a 2-step iterative relaxation scheme on the element matrices, applying a Jacobi iteration to get estimates of the local solutions, and then assembling these using a weighting to achieve an estimate of the global solution. This step is repeated until convergence, thereby relieving the need to assemble the global stiffness matrix. Once convergence is achieved, the global solution is passed back to the host in one single transfer. The approach is identical to ones seen previously up until the point of actually assembling the global stiffness matrix.

4.3.1 Two-Step Iterative Relaxation

The two-step iterative relaxation scheme is, naturally, going to be the most dominant kernel seen in the approach. However, the premise is that it should be more efficient than needing to apply the Cholesky decompositions and a direct solver on the entire system.

Consider the Jacobi iterative scheme,

$$x^{(\text{new})} = M^{-1}(b - Nx^{(\text{old})}), \tag{4.1}$$

for the linear system $Ax = b$, where $M = \text{diag}(A)$, $N = A - M$ is a matrix splitting. For our linear system for the FEM, $Lu = b$, now define local element solutions as u_e. If now, the Jacobi method is modified slightly, we can write it as,

$$u_e^{(\text{new})} = M^{-1}(b - Nu^{(\text{old})}). \tag{4.2}$$

It may seem strange here, to have an estimate to the global solution on the RHS and a the local solutions' estimate on the left. However, failing this, what will result is if one takes, for example, the Laplace equation where the RHS of the linear system b, is mostly sparse, barring cells which have boundary conditions, a collection of trivial matrices. The solution will be given for all internal cells as the initial and so in order to take account for the entire mesh, the global solution must be used for the RHS.

The second step in the 2-step iterative relaxation is to take these local solutions, and assemble them into a global solution. This is done by creating a weighting array, summing up the contribution of each node on the denominator. Mathematically, the weighting and assembly is given by,

$$u_i^{(new)} = \sum_{e}^{q(e,r)=i} \frac{A_{r,r}^{(e)}}{w_i} (u_r)_e^{(old)}, \tag{4.3}$$

where,

$$w_i = \sum_{e}^{q(e,r)=i} A_{r,r}^{(e)} \tag{4.4}$$

Figure (REFERENCE), illustrates a small two-cell implementation. In this example, the Jacobi relaxation for cell 0 is given by,

$$\begin{bmatrix} (u_0)_e^{(new)} \\ (u_1)_e^{(new)} \\ (u_2)_e^{(new)} \end{bmatrix} = \begin{bmatrix} \frac{1}{A_{0,0}^{(0)}} & 0 & 0 \\ 0 & \frac{1}{A_{1,1}^{(0)}} & 0 \\ 0 & 0 & \frac{1}{A_{2,2}^{(0)}} \end{bmatrix} \times (-\mathbb{I}_3) \begin{bmatrix} 0 & A_{0,1}^{(0)} & A_{0,2}^{(0)} \\ A_{1,0}^{(0)} & 0 & A_{1,2}^{(0)} \\ A_{2,1}^{(0)} & A_{2,1}^{(0)} & 0 \end{bmatrix} \begin{bmatrix} u_0^{(old)} \\ u_1^{(old)} \\ u_3^{(old)} \end{bmatrix} \tag{4.5}$$

and the assembly using the weighting for global node 2,

$$u_2 = \frac{A_{2,2}^{(0)}}{A_{2,2}^{(0)} + A_{1,1}^{(1)}} (u_2)_0 + \frac{A_{1,1}^{(1)}}{A_{2,2}^{(0)} + A_{1,1}^{(1)}} (u_1)_1. \tag{4.6}$$

4.3.2 Advantages & Limitations

The advantages of this approach is it improves on the parallelism seen in other methods. Of course, like the other approaches, the element matrices may be built in parallel and stored in global memory. However, after this, the assembly and solution of the linear system requires quite a substantial amount of communication between threads and synchronisation whereas

the local solution estimates may be calculated entirely in parallel. The assembly of the global solution from the weightings can also be done substantially in parallel barring some scattered atomic functions when adding back to the global solution array. These atomic functions should not cause much overhead either as they are not in sequential order, they are ordered depending on the shape of the mesh. This is detailed in Section 4.4.4.

On the other hand, in terms of drawbacks, the Jacobi iteration isn't the most efficient iterative method. It bodes well here due to needing to use two different vectors on the LHS and RHS, which might cause some complications in something like the Gauss-Seidel, as you don't have the most recent update until the global solution is assembled. However, it is rather slow to converge. There is also somewhat of a communication hit in the process. After each iteration is completed, convergence must be checked, since kernels cannot call recursively, the convergence check must pass back the error term to the host, in order to test if the loop shall continue or not. It isn't hugely costly as it is only a single float, but it is still a consistent amount of device-host synchronisations.

4.4 Implementations

As mentioned, there is a lot of machinery and parts to move around in applying the FEM, especially when applying it on a GPU. Due to this, all of the code was written in CUDA 10.1, to avoid unnecessary complications with openCL and to take advantage of the Nvidia CUDA libraries. Unfortunately, CUDA cannot handle C++ code like the serial approach can, but rather only in C, thus the operations had to be made as simple as possible, no STL features were used in the CUDA code.

4.4.1 Design Structure

Figure 4.3 illustrates the overall design structure of the CUDA code. The code was largely divided up into three main files and headers:

1. `gpu_fem.cu` - Contains the `__host__` functions and all the `__global__` and `__device__` functions for the general, FEM GPU approach - applying a linear decomposition to the linear system.

2. `gpu_femses.cu` - Contains the `__host__` functions and all the `__global__` and `__device__` functions for the FEMSES GPU approach.

3. `gpu_utils.cu` - Contains general utility functions for use in the two main methods, such as LAPACK and BLAS operations.

Both `__host__` functions are `extern` functions and so either can be invoked from the C++ code in `main.cc`. Note also that there is code reuse of `__device__` functions, so the flag `-relocatable-device-code=true` had to be set during compilation. Other flags for the CUDA libraries also had to be enabled, similar to the MKL library flags seen in the serial code.

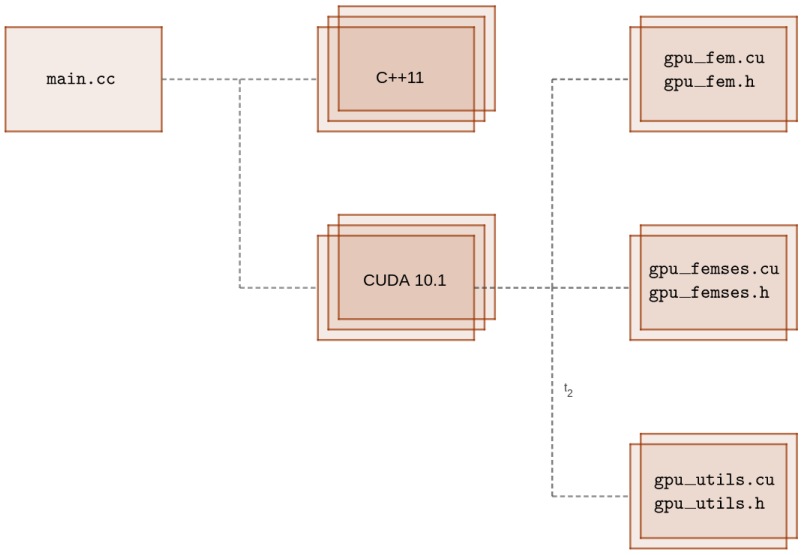

Figure 4.3: Structure of CUDA GPU code.

4.4.2 Considerations

There is a lot of data structures needed, so managing data reuse and reduction of transfers was a big consideration when designing the CUDA code. If something need not be transferred from host to device, it would be avoided. As well as this, the aim was to optimise shared memory use and minimise excessive synchronisation. For non-shared memory, attempting to keep the data coalesced is an important consideration as global memory access is quite slow. Fancy STL functionality couldn't be used so workarounds there had to be used - for example in the case of the sparsity pass function, this could not be performed on the GPU so it had to be done in serial and the resulting vectors transferred over. The last thing was to try minimise thread divergence from warps - reducing the amount of if statements is usually a good place to start along with making block sizes multiples of 32.

4.4.3 Basic FEM

In Figure 4.4, a flow chart of the approach taken to code a basic FEM implementation on a GPU is illustrated. The steps coloured in green are the ones which are executed on the GPU, the grey steps are ones executed on the host. The generation of the mesh and all its pertaining information needed, is all completed in serial, which is then passed down to the device, and for the most part, barring a sparsity scan, the rest of the entire FEM process is completed on the device. The element matrices creation and stiffness matrix assembly is all completed in a single kernel. The linear system solution is then solved using cuSOLVER - and cuSPARSE depending on the matrix structure flagged at run-time. Once this is complete, the final solution is passed back to the host and the CUDA code is complete.

Storage & Memory Management

In terms of storage, for a problem which scales like FEM does, managing your resources correctly and not doing excessive memory allocations is important - along with, of course,

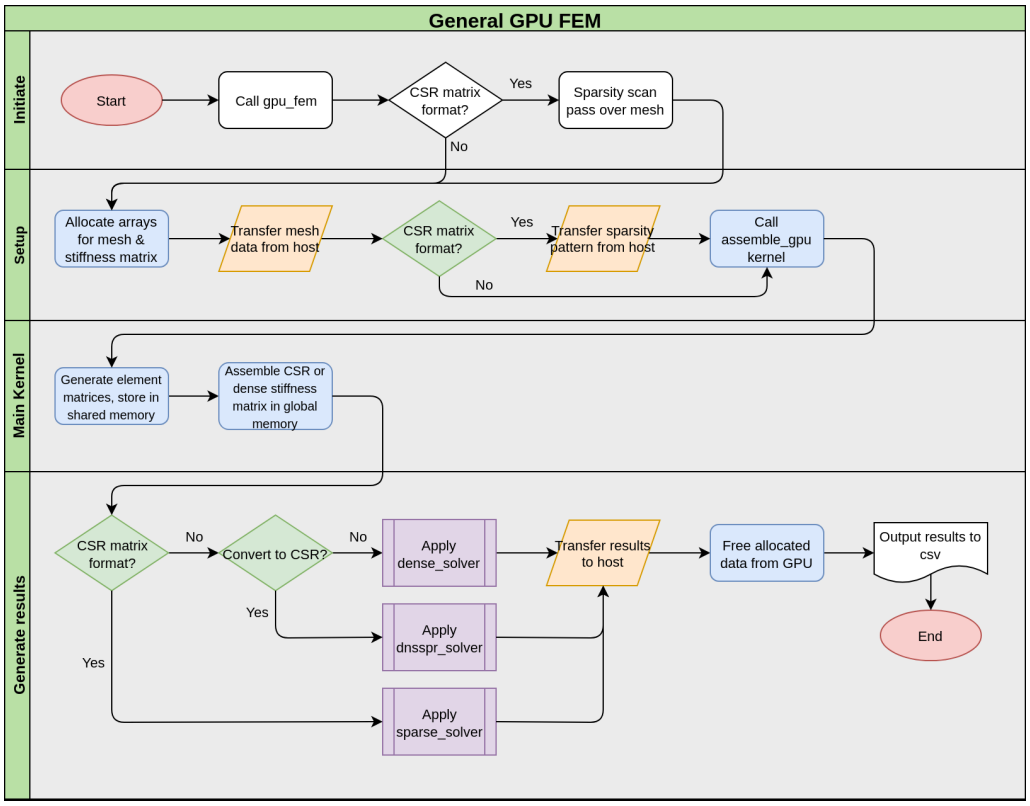

Figure 4.4: Flowchart of general GPU FEM implementation.

freeing any allocations which are completed. The first thing needed to be allocated on the GPU is the five arrays to store the mesh. Exactly like was mentioned in Section 3.2.1, memory needs to be allocated to store `vetices_gpu`, `cells_gpu`, `dof_gpu`, `is_bound_gpu` and `bdry_vals_gpu`. In terms of memory being coalesced, all five arrays are indeed contiguous, and in both `cells_gpu` and `dof_gpu`, the memory access pattern is sequential so this won't pose any coalescence problems. However, the other three arrays, the memory access pattern jumps up and down the array, completely dependent on the shape of the mesh, and the connectivity of the nodes. Thus to handle this, as detailed in the device functions, the necessary values are read into shared memory in single writes to hedge this bottleneck. Figure (REFERENCE) illustrates this, showing the lack of coalescence and the solution by writing to shared memory once.

Naturally, the next thing which needs to be allocated is the actual stiffness matrix itself. For this, unlike in serial, only the global stiffness matrix and stress vector need be allocated on global memory, since the local element matrices will be stored on shared memory. Depending on the choice to assemble a sparse or dense matrix will decide what must be allocated. For a dense case, a single dimensional array `L`, will be allocated, for the sparse case, three arrays must be allocated, `valsL`, `colIndL` and `rowPtrL`, as seen in Section 3.3. The CSR matrix, unfortunately does not have coalesced memory access, but it makes up for this by the huge reduction in the actual size of the matrix, and so, the timing should scale far better than dense. Both dense and sparse allocated the same space for the stress vector, as it is stored in dense format. This will be overwritten with the resulting solution, saving the space for needing an extra array.

There is quite a large amount of data needed to be transferred from the offset in the FEM from device. All five of the mesh's arrays must be transferred from host to device (`dof_gpu` technically does not when using P1 cases, but as stated this is to leave scope for further expansion). On top of this, as mentioned, the GPU cannot perform the sparsity scan algorithm, thus this must be done on the host. This algorithm populates both the column-index array and row-pointer array in the CSR structure, both of these must be transferred at the beginning when a sparse solution is chosen.

In terms of transfers, in both sparse and dense cases, the five arrays for the mesh need to be transferred. If `order` and `num_cells` represent the number of unknowns/nodes and the number of cells respectively, then these five arrays take up,

$$\texttt{mem} = (3 \times \texttt{order} \times \texttt{sizeof(float)}) + (((6 \times \texttt{num\_cells}) + \texttt{order}) \times \texttt{sizeof(int)}), \tag{4.7}$$

$$\texttt{mem} = (14 \times \texttt{order}) + (12 \times \texttt{num\_cells}), \tag{4.8}$$

where `mem` is measured in bytes. The two approaches will then differ when it comes to storing the stiffness matrix and transfers. For both cases, the global solution array will need to be stored and also transferred back to the host, which will take up `order` × `sizeof(float)`, or $4 \times$ `order`, bytes. This is the same array used to store the stress vector - it will simply be overwritten by the solution. For transfers, the sparse matrix must receive two arrays from the host after the sparsity pattern has been completed, the column-index array and row-pointer array, these are sized `nnz` × `sizeof(int)` and `(order+1)` × `sizeof(int)` respectively, where `nnz` is the number of non-zero entries achieved from the sparsity scan. The sparse approach also must store the values in `valsL`, taking up `nnz` × `sizeof(float)` too. There are no extra transfers needed for the dense approach but the memory needed to store the stiffness matrix is far larger, coming in at `order` × `order` × `sizeof(float)`. Therefore it can be said that the total amount of data needed to be transferred,

$$\texttt{mem\_transf\_sparse} = (20 \times \texttt{order}) + (12 \times \texttt{num\_cells}) + (2 \times \texttt{nnz}) + 2, \tag{4.9}$$

$$\texttt{mem\_transf\_dense} = (18 \times \texttt{order}) + (12 \times \texttt{num\_cells}), \tag{4.10}$$

and the total amount of memory allocated on global memory,

$$\texttt{mem\_alloc\_sparse} = (20 \times \texttt{order}) + (12 \times \texttt{num\_cells}) + (2 \times \texttt{nnz}) + 2, \tag{4.11}$$

$$\texttt{mem\_alloc\_dense} = (18 \times \texttt{order}) + (12 \times \texttt{num\_cells}) + (4 \times \texttt{order} \times \texttt{order}). \tag{4.12}$$

This is an important factor to take into account when deciding what problem size to calculate as memory on the GPU is limited. In the case of the two GPUs tested in this report, the Tesla K40 has 12GB of global memory, while the GeForce RTX2080 Super has 8GB.

Kernels

There are two main kernels for this approach: the first kernel, `assemble_gpu` or in its sparse form, `assemble_gpu_csr`, will take the input mesh as transferred prior, create the element matrices and assemble the global linear system, storing it back into global memory, this is demonstrated in Listing (REFERENCE); the second kernel, utilises CUDA libraries to solve the linear system. The assembly kernel was written specifically for this implementation, thus the setup of the grid and block dimensions must be set by the user. In order to take as much advantage of shared memory as possible, it was noted that the only real communication at this stage in the FEM process, is between nodes in the same cell. The setup of the kernel is then set that there are `block_size_X`, cells per block, in the x direction, and 3 nodes per cell, in the y direction. This gave the convenient benefit of `idx` being equal to the cell number, and `idy` being the local node numbering. This mapping is illustrated in Figure (REFRENCE). The kernel itself, runs two primary device functions, the first one generates the element matrices on shared memory, and the second then assembles these from shared memory into the global stiffness matrix, writing back to global memory. For two reasons, this is where the kernel had to terminate: you cannot globally synchronise the entire CUDA device within a kernel, only individual blocks, moving on to solve the linear system from here would almost certainly cause race conditions. The second reason is simply to avoid having to rewrite linear solver libraries that have already been optimised.

The linear solver kernel, comes in three different variations: one for CSR, one for dense matrices, and one which converts a dense matrix into CSR and solves. Unlike the assembly kernel, however, there is no need to decide how to organise the blocks and threads as this is taken care of by the library itself inside a variable called the handle, which is invoked at the beginning and passed into all library functions. In terms of what actual library functions are used, for the sparse solver, cuSPARSE is used to define the matrix structure, such as number of non-zeros, zero-based indexing, symmetric positive definite et cetera. This is used in tandem with cuSOLVER's single-precision, sparse SPD, Cholesky decomposition direct solver, `cusolverSpScsrlsvchol`. Unlike in the serial version, and Intel's DSS, the cuSOLVER kernel needs to store the entire matrix, as opposed to only the upper or lower half. This is an unnecessary bottleneck but one which cannot be avoided. The dense-CSR solver kernel, applies the same functionality as the sparse kernel, except for executing two other library kernels prior to the solver, `cusparseSnnz`, to deduce the sparsity pattern of the matrix, and `cusparseSdense2csr`, to convert from dense to CSR. The dense solver on the other hand, naturally does not use cuSPARSE. It does, however, split its solver kernel into two separate operations, the first being the actual Cholesky factorisation using `cusolverDnSpotrf` and the second applies the solver, `cusolverDnpotrs`. Figure 4.5 shows flowcharts of the necessary steps required in order to execute the three solver variants.

Device Functions

There are two main device functions written for the assembly kernel. Prior to running these, when invoking the kernel the required amount of shared memory must be allocated.

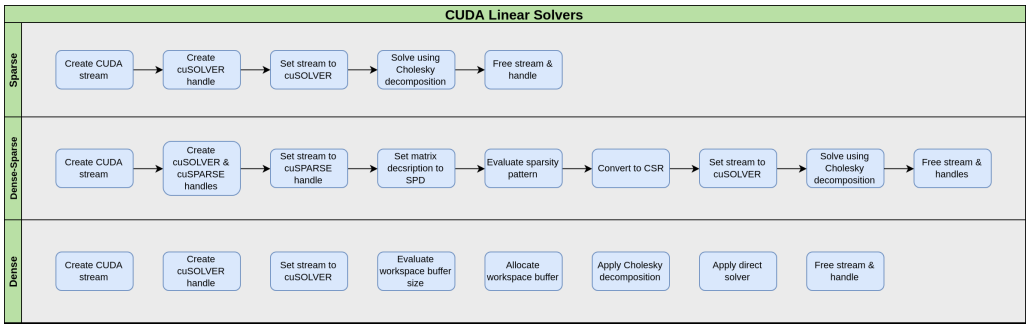

Figure 4.5: Flowchart illustrating the steps needed to implement solvers from CUDA's linear algebra libraries.

The grid structure is defined as `block_size_X` FEM cells per block, within each block there is one FEM cell designated per row of threads in the x-axis, and 3 nodes per row i.e. one node per thread in the y-axis. The shared memory needs to be large enough to store the element matrix `Le`, the element vector `be` as well as some other constants which need reuse such as the coordinates of each node in the cell `xi`, the global node numberings `dof_r` and the constants seen in Equation (2.77), for each cell in the block. Thus the amount of shared memory required is $28 \times$ `block_size_X`. This memory structure is displayed in Figure (REFERENCE) for an example block size of 4 in the x-axis. This would need to be larger of course, for non-P1 type problems but for this implementation it is sufficient.

The first device function, `assemble_elem`, starts off by setting pointer values for the element matrices, element vectors and constants in shared memory. It reads in the global node numbering associated with the node for that particular thread into the thread register. It then reads in the coordinates of the three nodes in that cell into shared memory so that data is now coalesced and then synchronises the threads in the block. The area of the cell is calculated by the thread associated with local node 0, and then the β and γ values are calculated in parallel. After this point, the device function runs Algorithm (REFERENCE), where each thread calculates a single row in the element matrix and writes them to the shared memory array. Notice that there is need for atomic functions when enforcing the boundary conditions, this is to prevent race conditions as the threads need to write to the same memory addresses.

A large speed-up point here is that the element matrices never need to be written to global memory, but rather the second kernel which assembles the global stiffness matrix simply assigns the pointers to the same location as the previous device function. For the assembly function, the global node numbers for the node associated with each thread are read in from global memory once and written over the constants in the previous function. Algorithm (REFERENCE) or Algorithm (REFERENCE), is then applied in parallel, depending on whether the matrix is dense or CSR, and written back to global memory. Again, unlike in the serial case, for the CSR matrix the entire matrix must be assembled and not simply its upper or lower-echelon form due to limitations of cuSOLVER.

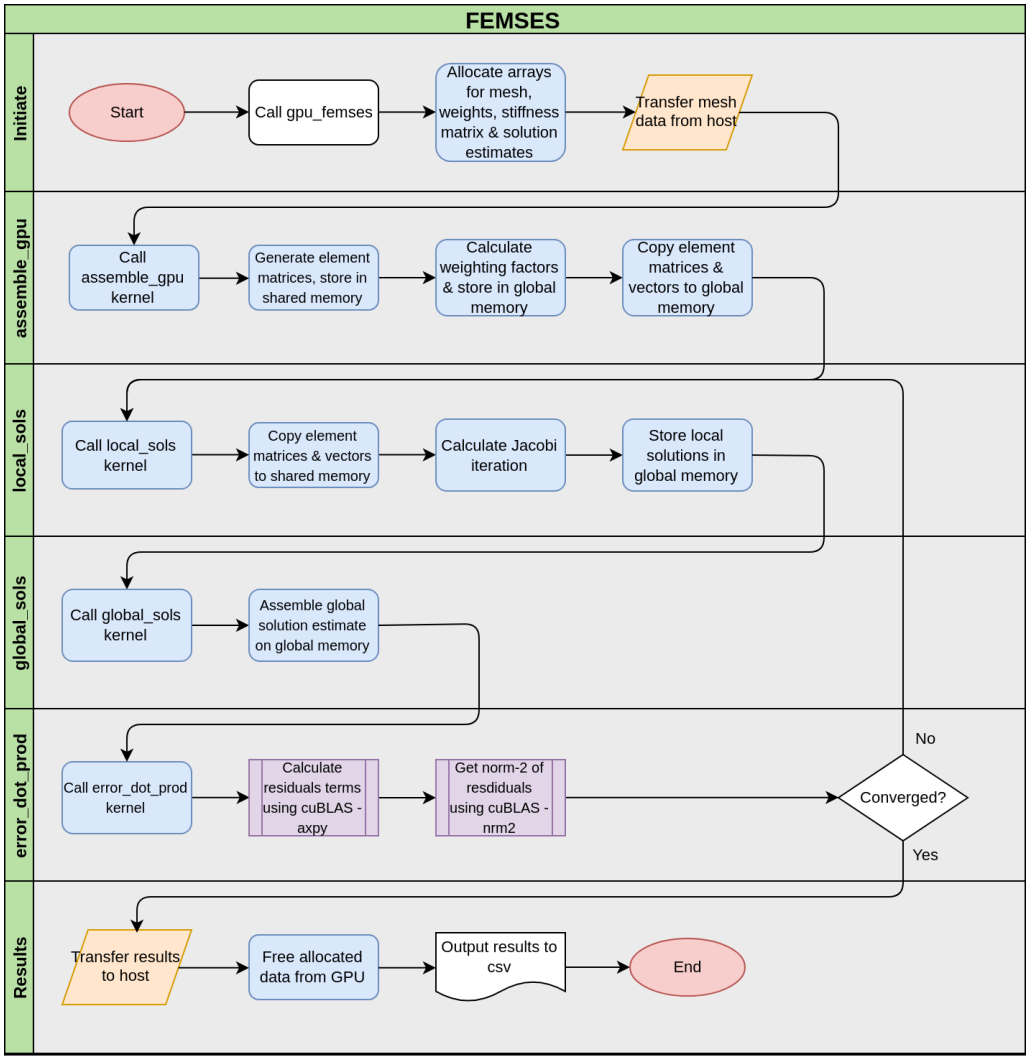

Figure 4.6: Flowchart of FEMSES approach.

4.4.4 FEMSES

Figure 4.6 shows a float chart of the FEMSES process when applied with CUDA code. Like before, the green coloured steps are carried out on the GPU, the grey are the host functions. Unlike in Fernandez (CITE), the assembly of the element matrices is carried out on the GPU, and stored into global memory instead of being transferred by the host to device - this should show some speedups over the results seen. Just like in the standard FEM implementation, all the mesh creation operations are carried out on the host and transferred down, in this case excluding the sparsity scan as there is no global matrix ever assembled in this method. The kernels are divided up in quite a logical manner, but this is simply due to synchronisation limitations on the CUDA cores. Everything that can be completed in a single kernel without needing a global synchronisation, is completed as such. Unfortunately, any point that needs global memory synchronisation, will require a kernel to terminate. This can be seen in the flow chart, where the element matrices are assembled and stored into global memory, and using these, local solutions are calculated using the Jacobi relaxation and stored into global memory. Following that, another kernel assembles the global solution, again writing back to global memory. Lastly, cuBLAS is called in order to calculate the 2-norm of the error and check for convergence. Clearly at each of these steps, global memory must synchronise and so there was no option but to write these all in separate kernels.

Storage & Memory Management

The FEMSES approaches shares the exact same memory allocations as the standard GPU FEM approach does, when it comes to allocating for the mesh. The same five arrays must be allocated and transferred taking up an identical amount of bytes as Equation (4.7). After this point though, the two differ. Clearly, since the aim of the FEMSES approach is to avoid generating a global stiffness matrix and stress vector, then there will be no need to allocate space for them. Instead rather, two arrays must be allocated, both containing all of the element matrices and element vectors, `Le`,`be` - these will be of size $\texttt{num\_cells} \times 9 \times \texttt{sizeof(float)}$ and $\texttt{num\_cells} \times 3 \times \texttt{sizeof(float)}$. There must also be space allocated to store an array of all the local element solutions `ue`, for when they are written back from shared memory, which is the same size as `be`. Finally there must be memory allocated to store both previous and new global solution estimates in the iteration, `up_gpu`, `un_gpu`, and an array containing the denominator of all the weightings, `w` as seen in Equation (4.3) - all three of these are size $\texttt{order} \times \texttt{sizeof(float)}$ each. In total then, the global memory needed to perform FEMSES is,

$$\texttt{mem\_alloc\_femses} = (26 \times \texttt{order}) + 72 \times \texttt{num\_cells}). \tag{4.13}$$

For transfers, all of the data that is allocated besides the mesh is calculated on the GPU, and only the resulting global solution needs to be transferred. Besides this, there is also one single float transferred at every iteration to check for convergence, of course this is not known a priori so for arguments sake, if there is `iter` iterations before convergence

then the total data transferred is,

$$\texttt{mem\_transf\_femses} = (18 \times \texttt{order}) + (12 \times \texttt{num\_cells}) + (4 \times \texttt{iter}). \tag{4.14}$$

Kernels

The FEMSES approach is divided up into four component kernels as seen in Figure (REFERENCE):

1. generate element matrices.
2. calculate local element solutions.
3. assemble global solution estimate.
4. calculate 2-norm of error.

In the cases of the first three kernels, the grid structure is identical to what was seen in the general FEM approach, allowing maximum use of shared memory and attempts to prevent thread divergence.

The first kernel, listed in Listing (REFERENCE), is called on to generate the element matrices. It uses the same device function as the main kernel in the alternate approach to generate the element matrices on shared memory. After this, it calls on a device function to calculate the weightings and save them onto global memory. Lastly it writes the element matrices and element vectors to the array in global memory and terminates the kernel.

The second kernel, evaluates the local solution estimates. Illustrated in Listing (REFERENCE), this has two main device functions: the first reads in the element matrices and element vectors from global to shared memory. The second device function is the key computational step of this approach, calculating the Jacobi iteration estimate, as seen in Equation (4.2), to generate local solutions and writes these back to global memory.

The final purpose-written kernel in this method assembles the global solution array using the weightings. Seen in Listing (REFERENCE), there was in fact no need for extra device functions here simply due to the fact that it was a read/write operation and one line of calculation - an atomic addition did sufficiently here. Thankfully, due to the irregular memory access pattern the atomic add should not actually pose much bottleneck. In Fernandez (CITE), the residuals were also calculated in this kernel, which is believed to be an oversight or error since this will potentially cause a race condition as there is no guarantee the entire global solution has been assembled until the kernel terminates.

The final calculation kernel here is utilising cuBLAS. The kernel uses two BLAS Level-1 operations, the first being the vector product, $y \leftarrow \alpha x + y$, or `cublasSaxpy`, to evaluate the residual vector and write it over y. The second operation evaluates the 2-norm of the residual, $e \leftarrow \sqrt{\langle r, r \rangle}$, or `cublasSnrm2`. The resulting error term is passed back to the device.

Device Functions

The first device function utilised in the FEMSES approach is actually code reuse from the standard approach. It takes the mesh as input, uses the same shared memory structure as Figure (REFERENCE), and generates the element matrices and vectors in parallel, writing them to shared memory. See Section 4.4.3 for more details on this device function.

The next device function evaluates the weightings. This functions takes the cell indices and element matrices as input and calculates the denominator weighting seen in Equation (4.3). In Fernandez (CITE), each node in each cell was assigned its own individual weighting value in an array. This would involve a global synchronisation (something they did not do) to ensure the denominator values were fully added up and to avoid a race condition. Instead, the weighting array was allocated as one value per node as opposed to three per cell. In this paper's approach, the denominator was the weight stored and this was simply used to divide the node's corresponding diagonal value in its element matrix later on. Of course this means an extra read from global memory but it avoids the need for a global synchronisation.

The key device function in the entire approach is the Jacobi iteration function. Listing (REFERENCE) details this. Taking the previous global solution estimate as an input, it reads this in as $u^{(\texttt{old})}$, as well as reading in the element matrices and element vectors. Each thread will calculate its own individual $u^{(\texttt{new})}$, this will be written back to global memory then, to be assembled into a new global solution estimate using the succeeding kernel.

There are also another couple of device functions not mentioned here or in the previous section but these are merely read/write functions or area calculations et cetera.

Chapter 5

Results

5.1 Test-bed Architecture

5.2 Serial Code Profiling

5.3 GPU Performance

5.3.1 Standard FEM

5.3.2 FEMSES

5.3.3 Comparison of GPU Architectures

Chapter 6

Conclusion

6.1 Final Remarks

6.2 Future Work

6.3 Recommendations & Limitations

Appendix A

Appendix

...