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 20, 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 20, 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 **IV**

List of Tables **V**

1 Introduction **1**
1.1 Preliminaries . . . 2

2 Finite Element Method **3**
2.1 General Problem . . . 3
2.2 Approximation Theory . . . 3
2.2.1 Least Squares . . . 4
2.2.2 Galerkin . . . 5
2.2.3 Weighted Residuals . . . 5
2.3 Variational Calculus . . . 6
2.3.1 Weak Formulation . . . 6
2.3.2 Functionals . . . 6
2.4 Finite Elements . . . 7
2.4.1 Cells & Basis Functions . . . 7
2.4.2 Assembling the Stiffness Matrix . . . 8
2.4.3 Enforcing Boundary Conditions . . . 10
2.5 Implemented Example . . . 11
2.5.1 Problem Statement . . . 11
2.5.2 Analytical Solution . . . 12

3 Design & Implementation **13**
3.1 General Approach . . . 13
3.1.1 Design Structure . . . 13
3.2 Data Structures . . . 14
3.2.1 Mesh . . . 14
3.2.2 FEM . . . 14
3.3 Sparse Storage . . . 15
3.3.1 Graph Theory . . . 16
3.3.2 Sparsity Scan . . . 16
3.4 Solver Libraries . . . 16
3.4.1 Intel's MKL . . . 16

4 GPU Implementations **17**
4.1 GPU Programming & CUDA . . . 17
4.1.1 Blah blah . . . 17
4.1.2 CUDA Libraries . . . 17
4.2 Current FEM Approaches . . . 17
4.2.1 Linear System Decomposition . . . 17

4.2.2 Multigrid Techniques . . . 17
4.2.3 Domain Decomposition . . . 17
4.3 Basic FEM Implementation . . . 17
4.3.1 Overview . . . 17
4.3.2 Considerations . . . 17
4.3.3 Implementation . . . 17
4.4 FEMSES . . . 17
4.4.1 Overview . . . 17
4.4.2 Considerations . . . 17
4.4.3 Implementation . . . 17

5 Results **18**
5.1 Test-bed Architecture . . . 18
5.2 Serial Code Profiling . . . 18
5.3 GPU Performance . . . 18
5.3.1 Standard FEM . . . 18
5.3.2 FEMSES . . . 18
5.3.3 Comparison of GPU Architectures . . . 18

6 Conclusion **19**
6.1 Final Remarks . . . 19
6.2 Future Work . . . 19
6.3 Recommendations & Limitations . . . 19

A Appendix **21**

List of Figures

List of Tables

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.

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 behind the finite element method such as variational calculus, approximation theory and the finite elements themselves. The chapter also goes through the actual 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, let us first consider an n-dimensional, general n^{th} order PDE of the form,

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

where $\mathbf{x} = \{x_1, x_2, \ldots, x_n\}$. For a case of a 2-dimensional, 2^{nd} order PDE, we end 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, applied to a function to leave a PDE,

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

which is going to be our entire basis for this study - attempting to approximate the function u. 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 he arc length of the boundary C and $\mathbf{n}$ is a vector, externally normal to C. How these conditions are imposed and handled will be seen later in the report.

2.2 Approximation Theory

Suppose there exists some function $u(x)$ that we wish to approximate. The most common way to is to estimate the value of the function 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 obtain solutions to the approximation (REFERENCE EQ) 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

Suppose we are given a function $f(x)$ which needs to be approximated by a function $u(x)$ $\in V$ as defined above. 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 we are left with,

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

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

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, $\frac{\partial e}{\partial c_i}$. Evaluating and setting $\frac{\partial e}{\partial c_i}$, 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 seen 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, we do not actually know the true error as f is not explicitly known and so instead a residual R is used. Take for example, Eq. (REFERENCE HERE), 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,

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

and since any function in V can be approximated using (REFERENCE), 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 also 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 other space W, also known as the test space. This leaves the equation,

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

, 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 system of $N+1$ linear 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 (REFERENCE). What may not seem immediately obvious at a first glance, but an important thing to factor in, (REFERENCE) is not how the PDE problem is usually 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 derivative, as opposed to its strong form containing second-order derivatives. Usually, if only approximating a function, 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 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, it 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 entire PDE can be shown in terms of the inner product defined in (REFERENCE),

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

This PDE has now been transformed into its weak formulation. Looking at the (REF SECTION), 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 dimension.

2.3.2 Functionals

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 if we consider 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 we saw in the previous sections. Consider 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 and contains 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 (REFERENCE) which we wish to minimise. This equation is equivalent to minimising the functional,

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

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 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,

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

where

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

Within this domain, define N_n nodes, equally spaced out. Suppose a node has a *global index* $i \in \{0, \ldots, d\}$ 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. No 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 (REFERENCE) demonstrates these definitions. Now, basis functions must 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, we define its corresponding basis function $\varphi_i(x)$ as a Lagrange polynomial of degree d, defined,

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

which is 1 at node r and 0 everywhere else. If the node is internal, than 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 triagular or tetrahedral shapes, however, they can if ones wishes be any shape one wishes such as squares or octagons as long as the degree of freedom mapping is correct. Figures BLAH BLAH 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 and most obvious step to do is a linear system must be assembled from these basis functions in order to solve for these unknowns $\{c)i\}_{\mathcal{I}_s}$ and achieve an approximation for u. It was shown in REFERENCE SECTION, that a variational form can be written in an abstract means form as,

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

With that in mind, consider the PDE (REFERENCE), 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 (REFERENCE), 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 (REFERENCE) should clearly demonstrate now why it was important to convert the PDE into its weak formulation as now we have 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 what the overall stiffness matrix will be calculated over the entire domain of but how does one achieve 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 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.

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

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

where $\tilde{A}^{(e)}$ and $\tilde{b}^{(e)}$ are known as the element matrix and element vector, respectively and r, are local node numberings as defined previously. Both of these evaluate the same integrals but over their compact support instead of the entire domain, so looking at equations (REFERENCE REFERENCE), both of their element evaluated counterparts on cell e become,

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

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

Take the small P1, 2D example seen in Figure (REFERENCE). Clearly, in this example, each cell contains three nodes, so the resulting element matrix $\tilde{A}^{(e)} \in \mathbb{R}^{3\times3}$ and element vector $\tilde{b}^{(e)} \in \mathbb{R}^3$. Evaluating both of these, by the nature of all the calculations being dot products, results in a symmetric positive definite element matrix,

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

and element vector,

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

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

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

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

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.

2.4.3 Enforcing Boundary Conditions

Boundary conditions have been conveniently ignored up to this point in the report for simplicity reasons when explaining how to assemble the linear system. However, they of course cannot in reality go without implementing them. Thankfully, certain boundary conditions are easier to handle than others. Looking at Eq (REFERENCE), 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 actually handles 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 you simply separate the two and handle as usual. There are a number of ways of handling the essential boundaries, one would be to compress your degrees of freedom set $\mathcal{I}_s$, such that it no longer contains any of the nodes which are boundaries and are this emitted from the stiffness matrix, since $A \in \mathbb{R}^{\dim\mathcal{I}_s \times \dim\mathcal{I}_s}$. In this instance, the function being approximated would look like,

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

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

Instead of this approach, modification of the linear system approach was taken, whereby all nodes remain. Since the boundary values are exact solutions at that particular point of the system, for that corresponding row and column 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. $A_{i,j} = A_{j,i} = 0$,
3. $A_{j,j} = 1$,
4. $b_j = U_j$.

Looking at the example in (REFERENCE), if supposing an essential boundary condition U_0 is enforced at local node 0, the updated element matrix would be,

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

and updated element vector,

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

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

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 EQREFERENCE, clearly the integrals or inner products are all calculated with respect to the pysical coordinates of x across the domain within 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 elements. 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 all the node's local coordinates to be (0,0), (0,1) and (1,0). For example, denote the local coordinate system by $\mathbf{X}$ and the new, easier local basis functions $\tilde{\varphi}_r(\mathbf{X})$, 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,

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

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 we get the equivalency,

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

where $\tilde{\Omega}^{(e)} = [0,1] \times [0,1]$. This mapping can easily be performed on all of the elements in the weak form equation. The clear advantage here is, previously the basis functions were written as Lagrange polynomials, no since the range of values only spans from 0 to 1,

$$\tilde{\varphi}_0(\mathbf{X}) = 1 - X_0 - X_1, \tag{2.62}$$
$$\tilde{\varphi}_1(\mathbf{X}) = X_0, \tag{2.63}$$
$$\tilde{\varphi}_2(\mathbf{X}) = X_1, \tag{2.64}$$

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 rectangle 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 $\Omega = [a, b] \times [a, b]$ FIX HERE. 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 (REFERENCE) 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 (REFERENCE), P1 problems were used and some handy analytical solutions to the LHS integral which only work for 2D, P1, triangular meshes. If the physical, non-local, coordinates of the 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 the LHS integral,

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

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

2.5.2 Analytical Solution

Chapter 3

Design & Implementation

Now that the foundation 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 u se of linear algebra libraries, specifically Intel's MKL, are detailed as they were necessary 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 and structure it in 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 (REFERENCE) illustrates the overall design structure of the code - in both serial and parallel, though the GPU code will be discussed in more detail in Chapter (REFERENCE). The code is split into four main files and their respective headers,

1. `main.cc` - Main C++ code, used for invoking the FEM operation and calling results to be output.

2. `mesh.cc` - Contains the code for a Mesh class, storing all necessary operations and data structures to create a mesh.

3. `fem.cc` - Code for a FEM class which contains the routines needed to perform the method on a given mesh.

4. `utils.cc` - General utility functions, such as SSE, arguments parsing etc.

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 (REFERENCE) demonstrates a flowchart of the serial code process.

3.2 Data Structures

Objects are a massive sell for using C++ over C when programming up a numerical method, for reasons stated already. For this paper, two classes were written, one to handle the mesh creation and one to handle actually 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 multiple variables.

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 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. Looking back at the previous section 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 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 and 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 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 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 return various properties of the object, coordinates, pointers to the arrays (for CUDA purposes), number of nodes etc. There is also a routine to do a pass over the mesh and return its sparsity pattern which will be detailed in Section (REFERENCE).

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, and solve the given PDE - in this 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, reducing 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 CSR 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:

- element matrix generation.
- global stiffness matrix and stress vector assembly.
- linear system solver.

The element matrix calculation is detailed in Algorithm (REFERENCE), utilising the convenient formulas stated in (REFERENCE). 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. Algorithm (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 mostly 0 valued entries. 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 integral there will be 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 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 one was used in the implementation:

1. CSR.
2. CSC.
3. 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,

$$
\begin{aligned}
\text{A} &= [\ 3\ 1\ 20\ 15\ 1\ 4\ 1\ 9\ 1\ 6\ 8\ 7\ 11\ ] \\
\text{IA} &= [\ 0\ 2\ 5\ 7\ 11\ 12\ 13\ ] \\
\text{IJ} &= [\ 0\ 1\ 1\ 2\ 4\ 3\ 4\ 0\ 1\ 2\ 3\ 3\ 4\ ]
\end{aligned}
$$

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,

$$
\begin{aligned}
\text{A} &= [\ 3\ 9\ 1\ 20\ 1\ 15\ 6\ 4\ 8\ 7\ 1\ 1\ 11\ ] \\
\text{IA} &= [\ 0\ 3\ 0\ 1\ 3\ 1\ 3\ 2\ 3\ 4\ 1\ 2\ 5\ ] \\
\text{IJ} &= [\ 0\ 2\ 5\ 7\ 10\ 13\ ].
\end{aligned}
$$

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,

$$
\begin{aligned}
\text{A} &= [\ 3\ 1\ 20\ 15\ 1\ 4\ 1\ 9\ 1\ 6\ 8\ 7\ 11\ ] \\
\text{IA} &= [\ 0\ 0\ 1\ 1\ 1\ 2\ 2\ 3\ 3\ 3\ 3\ 4\ 5\ ] \\
\text{IJ} &= [\ 0\ 1\ 1\ 2\ 4\ 3\ 4\ 0\ 1\ 2\ 3\ 3\ 4\ ].
\end{aligned}
$$

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, can be equated to graph traversal problems in graph theory.

3.3.2 Sparsity Scan

3.4 Solver Libraries

3.4.1 Intel's MKL

CITE INTEL'S DOCUMENTATION

Chapter 4

GPU Implementations

4.1 GPU Programming & CUDA

4.1.1 Blah blah

4.1.2 CUDA Libraries

4.2 Current FEM Approaches

4.2.1 Linear System Decomposition

4.2.2 Multigrid Techniques

4.2.3 Domain Decomposition

4.3 Basic FEM Implementation

4.3.1 Overview

4.3.2 Considerations

4.3.3 Implementation

4.4 FEMSES

4.4.1 Overview

4.4.2 Considerations

4.4.3 Implementation

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

...