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

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

September 2, 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

September 2, 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 of view. Previously, GPUs were only used for their actual purpose, for performing graphics manipulation for games and rendering. However, since around 2001, these cards have effectively been hijacked to perform all kinds of operations since they became stacked with programmable floating point operations. This general-purpose GPU programming, or GPGPU, has opened up a world of opportunity to take advantage of the hardware available and perform scientific operations. Nvidia now even offer cards which do not even provide any ports for graphical output. Nvidia have taken to developing their own set of mathematical libraries to be used with their cards and developed the CUDA language to implement this. Not only this, but in the last few editions of their architectures, since Fermi and Turing, they have began to add extra accelerator chips on to their cards for performing fast tensor operations and Ray Tracing for machine learning.

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, stress analysis crash simulations and modelling heat transfer. Its engineering foundations will become clear throughout the paper as much of the terminology has remained unchanged. Just as there is the libraries for linear algebra and other mathematical methods mentioned previously, certain libraries for applying the FEM already exist, such as FEniCS [CITE], FreeFEM [CITE] and even Wolfram Alpha have their own applications inbuilt to their robust Mathematica software [CITE]. Most of these implementation are all written for serial or CPU-based architectures.

This paper investigates current approaches to applying the finite element method on Nvidia GPUs and attempts 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 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,

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

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 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,

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

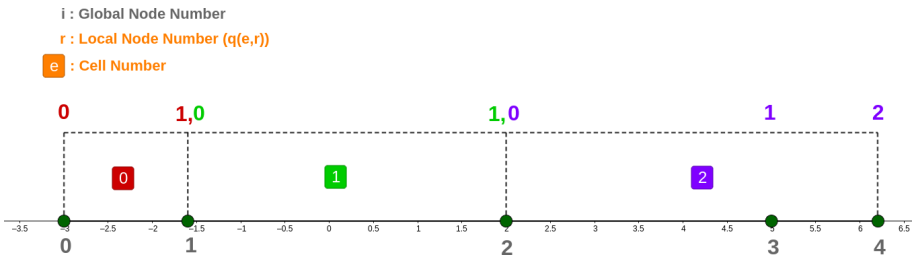


Figure 2.1: Illustration of three cell, 1D mesh with $\Omega_0 = \{0, 1\}$, $\Omega_1 = \{1, 2\}$, $\Omega_2 = \{2, 3, 4\}$.

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

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

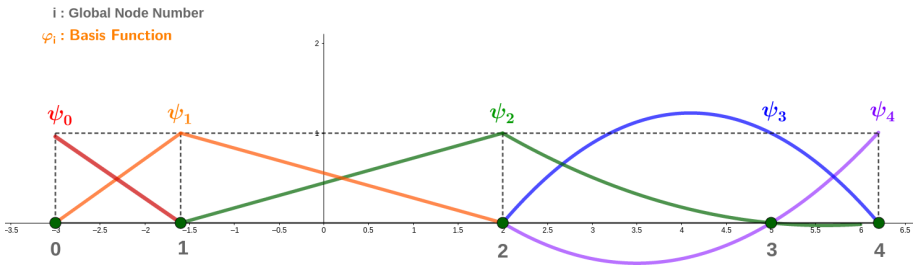


Figure 2.2: 1D mesh representation, with 1st order (P1) Lagrange polynomials in cells 0,1 and 2nd (P2) order in cell 2.

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, three 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.

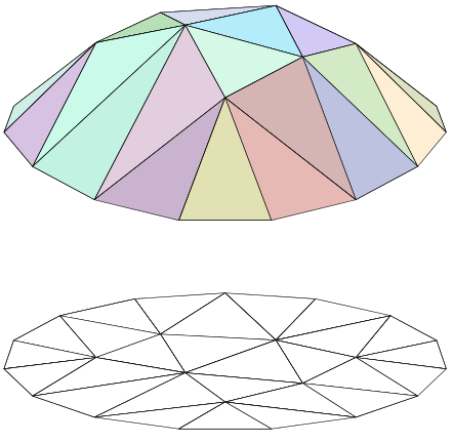


Figure 2.3: 2D mesh representation of P1 basis functions.

It was shown in Section 2.3.2, that a variational form PDE 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 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.

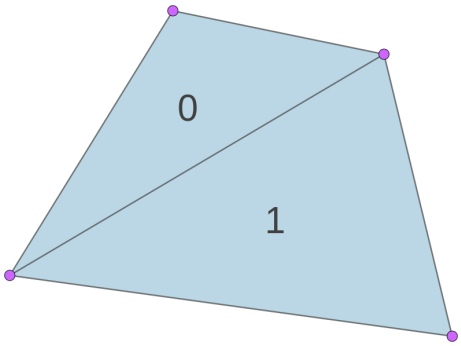


Figure 2.4: Small two cell example of a mesh.

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 Neumann condition applies. Subbing in g into the inner-product here has now in fact dealt with the Neumann 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 $\mathbf{x}$, 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

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

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

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

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

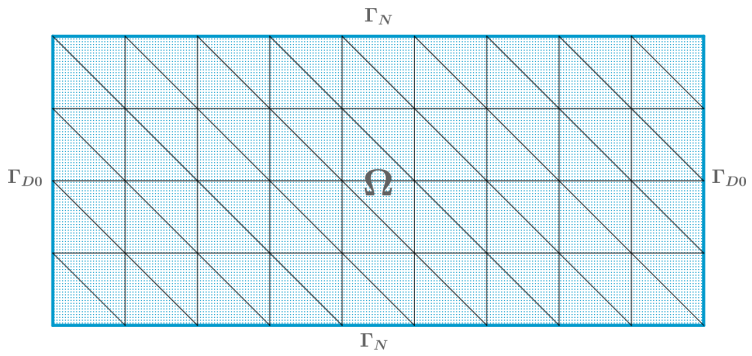


Figure 2.5: Domain of the Laplace problem to be solved.

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,

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

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

Analytical Solution

This given problem has the analytical solution,

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

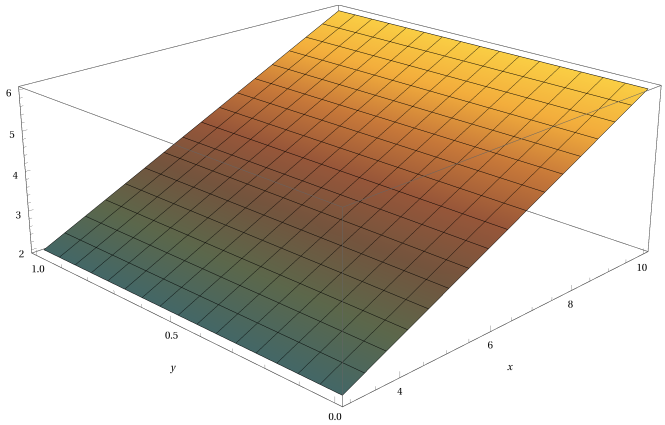


Figure 2.6: Plot of solution to PDE problem posed.

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.

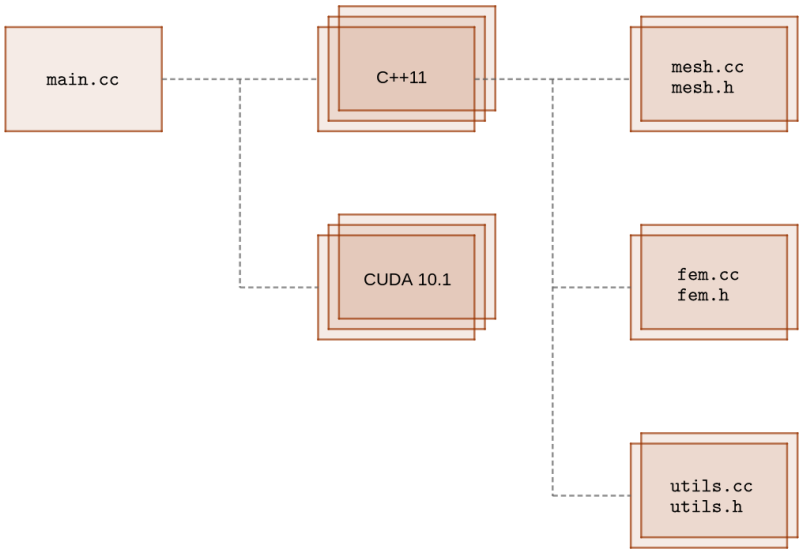


Figure 3.1: Code structure of the C++ code implementation.

- `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 the already stated reasons. 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

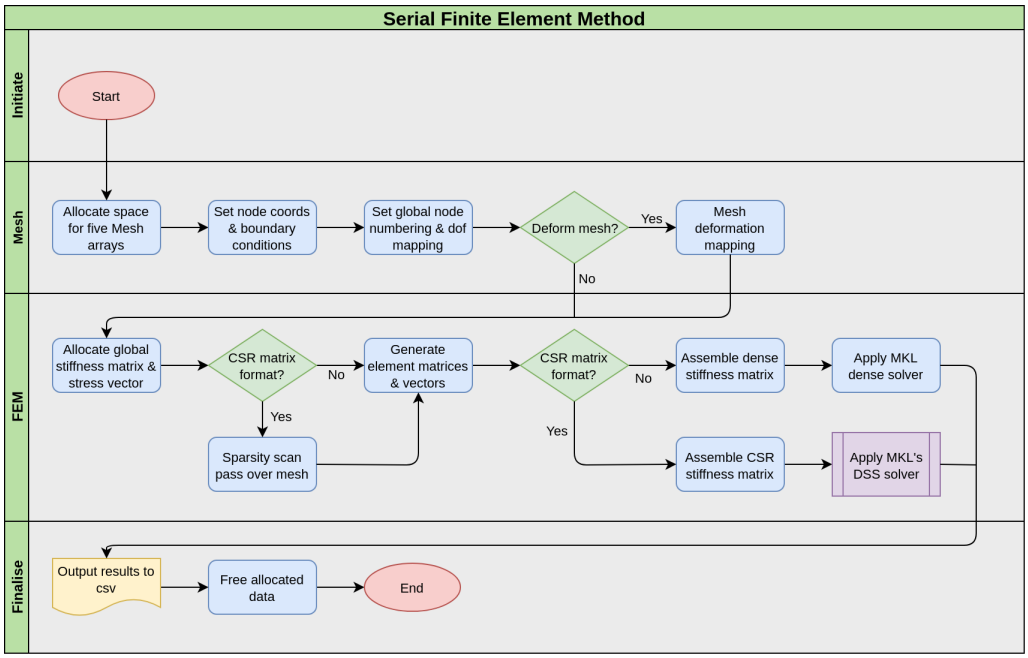


Figure 3.2: Flowchart of serial FEM implementation.

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 3.3 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,

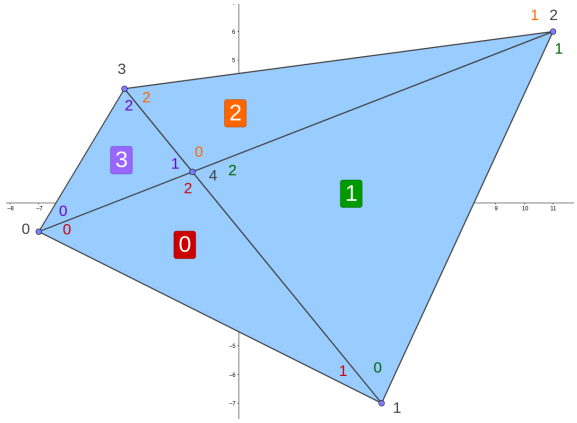


Figure 3.3: 2D, four cell, mesh representation with cell numberings and both local and global node numberings illustrated

Algorithm 1: Evaluation of element matrices & element vectors.

Input: V, e

for $r = 0 \dots 2$ **do**
  $\mathbf{x}_r = \begin{bmatrix} 1 & V_{r,0}^{(e)} & V_{r,1}^{(e)} \end{bmatrix}$
end
$\Delta = \frac{1}{2}|\mathbf{x}|$
for $r = 0 \dots 2$ **do**
  $\beta_r = x_{(r+1) \bmod 3,2} - x_{(r+2) \bmod 3,2}$
  $\gamma_r = x_{(r+2) \bmod 3,1} - x_{(r+1) \bmod 3,1}$
  $\mathbf{b}^{(e)} = \mathbf{0}$
  for $s = 0 \dots r$ **do**
    $L_{i,j}^{(e)} = \frac{1}{4}\Delta^3 \left(\beta_r \beta_r - \gamma_r \gamma_s\right)$
    $L_{j,i}^{(e)} = L_{i,j}^{(e)}$
  end
end
for $r = 0 \dots 2$ **do**
  $v = q(e, r)$
  if $v \in \mathcal{I}_b$ **then**
    $B = u_v$
    for $s = 0 \dots 2$ **do**
      if $r \neq s$ **then**
        $b_s^{(e)} \leftarrow b_s^{(e)} - L_{r,s}^{(e)}\ B$
        $L_{r,s}^{(e)} = 0$
        $L_{s,r}^{(e)} = 0$
      end
    end
    $L_{r,r}^{(e)} = 1$
    $b_r^{(e)} = B$
  end
end
return $\mathbf{L}^{(e)}, \mathbf{b}^{(e)}$

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 29, 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 2, 3 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.

Algorithm 2: Assembly of global stiffness matrix & stress vector in dense format.

Input: $\{\mathbf{L}^{(e)}\}_e, \{\mathbf{b}^{(e)}\}_e$
for $e = 0 \dots (num\_cells - 1)$ **do**
  for $r = 0 \dots 2$ **do**
    $v_r = q(e, r)$
    for $s = 0 \dots 2$ **do**
      $v_s = q(e, s)$
      $L_{v_r, v_s} \leftarrow L_{v_r, v_s} + L^{(e)}_{r,s}$
    end
    $b_{v_r} \leftarrow b^{(e)}_r$
  end
end
return $\mathbf{L}, \mathbf{b}$

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 it consists of

Algorithm 3: Assembly of global stiffness matrix & stress vector in CSR format.

Input: $\{\mathbf{L}^{(e)}\}_e, \{\mathbf{b}^{(e)}\}_e$
for $e = 0 \dots (num\_cells - 1)$ **do**
  for $r = 0 \dots 2$ **do**
    $v_r = q(e, r)$
    for $s = 0 \dots 2$ **do**
      $v_s = q(e, s)$
      $i = 0$
      $\text{ind} = AI_{v_r}$
      $\text{col} = IJ_{\text{ind}}$
      while $col \neq v_s$ **do**
        $i \leftarrow i + 1$
        $\text{col} = IJ_{\text{ind}+i}$
      end
      $\text{ind} \leftarrow \text{ind} + i$
      $L_{\text{ind}} \leftarrow L_{\text{ind}} + L^{(e)}_{r,s}$
    end
    $b_{v_r} \leftarrow b^{(e)}_r$
  end
end
return $\mathbf{L}, \mathbf{b}$

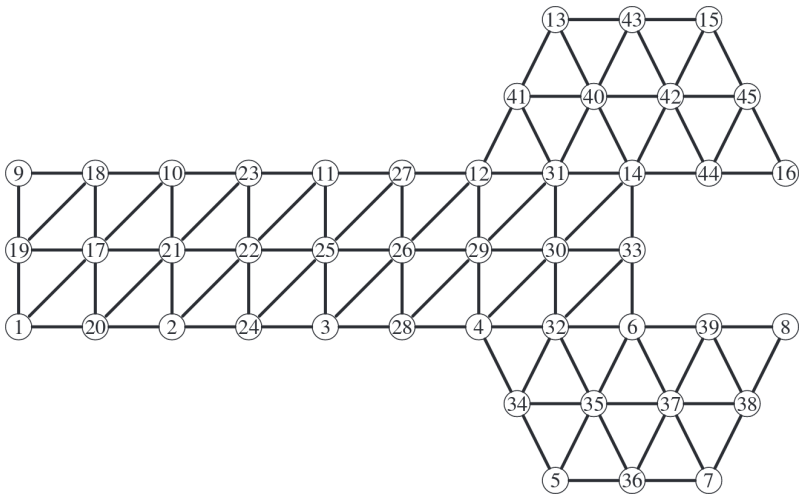


Figure 3.4: Mesh for generic FEM problem illustrated as a graph representation.

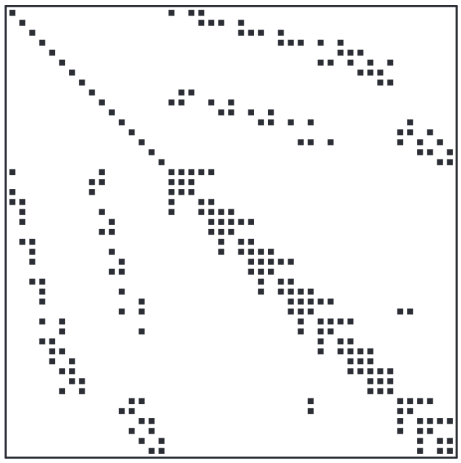


Figure 3.5: Sparsity pattern of FEM mesh.

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 3.3, 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 3.4, 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 3.5 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 4 details how this is explicitly performed.

Algorithm 4: Sparsity pass on mesh to populate row pointer and column indices vectors.

Input: V, num_cells
$S = \bigcup_{v \in V} S^{(v)}$
for $e \dots num\_cells$ **do**
  for $r = 0 \dots 2$ **do**
    $v_r = q(e, r)$
    for $s = 0 \dots 2$ **do**
      $v_s = q(e, s)$
      $S^{(v_r)} \leftarrow S^{(v_r)} \cup \{v_s\}$
    end
  end
end
$c \leftarrow 0$
$n \leftarrow 0$
for $v \in V$ **do**
  for $v_r \in S^{(v)}$ **do**
    $AJ_c = v_r$
    $c \leftarrow c + 1$
  end
  $n \leftarrow n + \#(S^{(v)})$
  $AI_n = n$
end
return **AI**, **AJ**, n

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 Ph.D. 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 et cetera. The one which was used in the end was Intel's MKL.

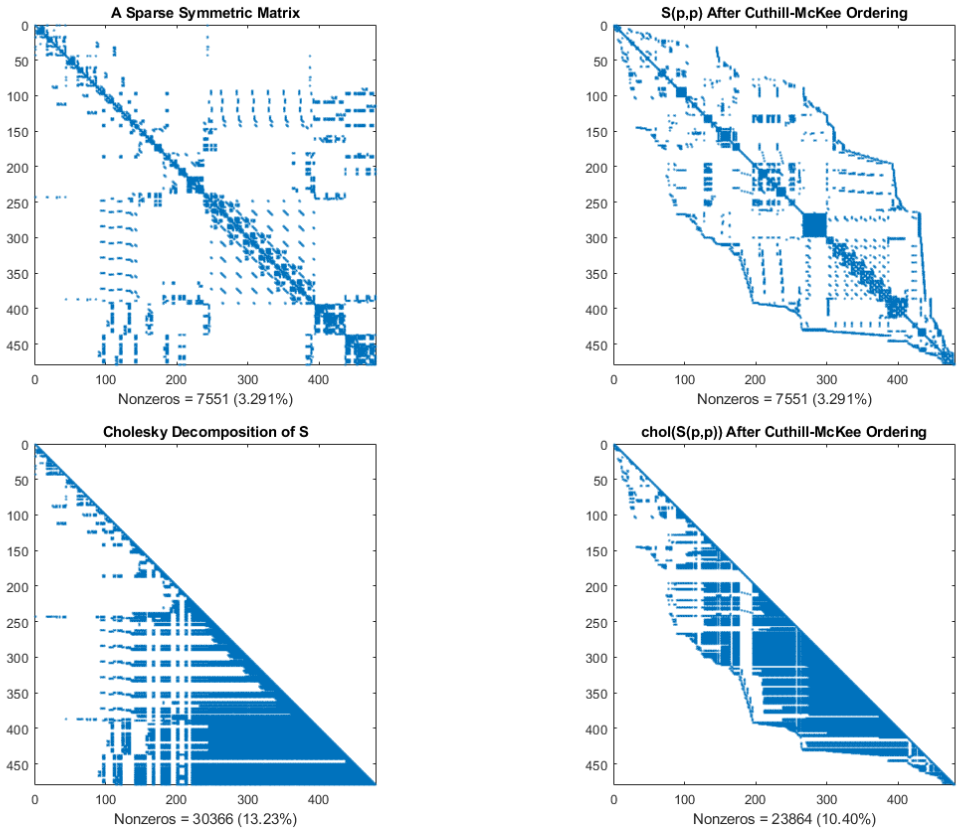


Figure 3.6: Illustration of a sparsity pattern of a matrix, pre and post Cuthill-McKee reordering, along with the resulting Cholesky decompositions of both matrices.

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 3.6 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

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 speed-ups 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 speed-ups, 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

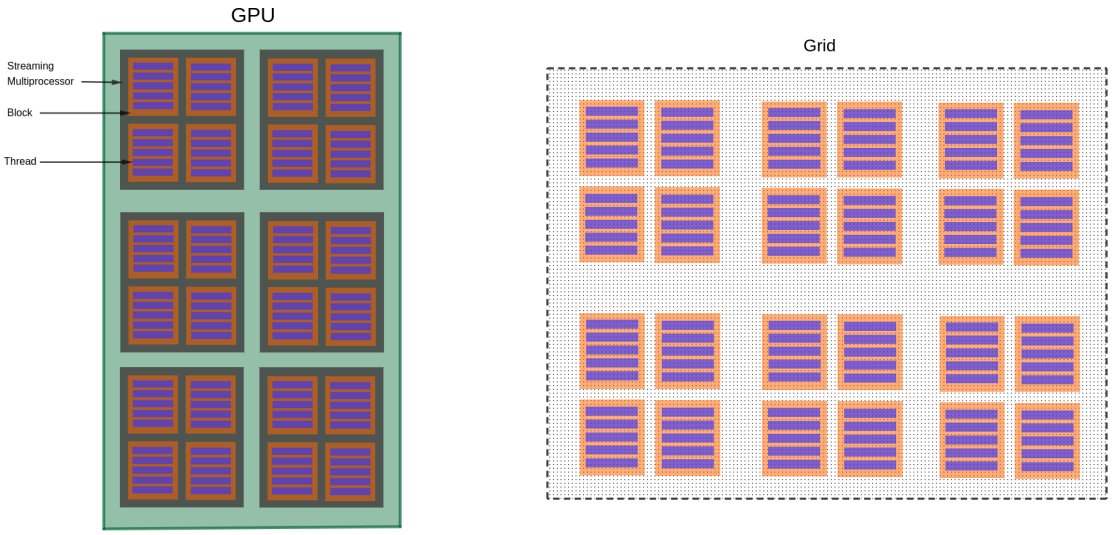


Figure 4.1: Illustration of a GPU's physical architecture and virtual 2D grid representation with dimensions 6×4 and block dimensions 5×1.

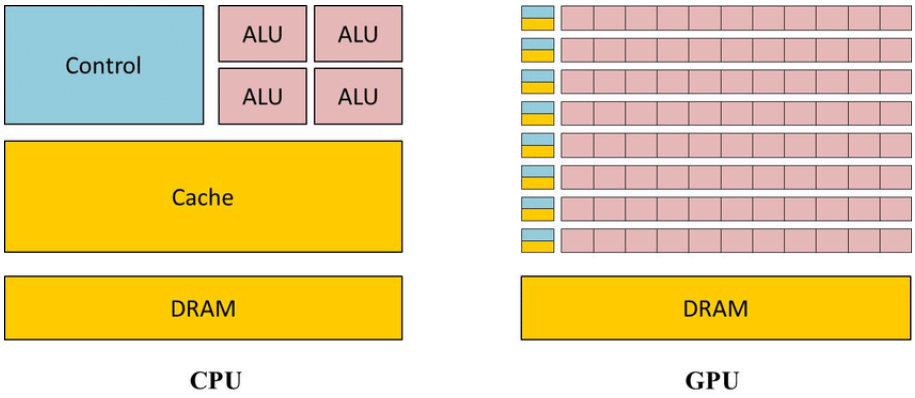


Figure 4.2: Simplistic comparison of CPU versus GPU architectures.

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 4.1 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 4.3 showing Nvidia's diagram of the Turing architecture, but these are relatively recent and not much used as of yet. Figure 4.2 shows a relatively basic illustration of the difference in architectures, illustrating the larger, more complex control unit, shared cache as well as the much larger *arithmetic logic units* - threads in a GPU's case.

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

Figure 4.3: Illustration of Nvidia's current Turing architecture, showing its SMs and Ray Tracing cores.

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 that 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 4.4 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 4.5 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

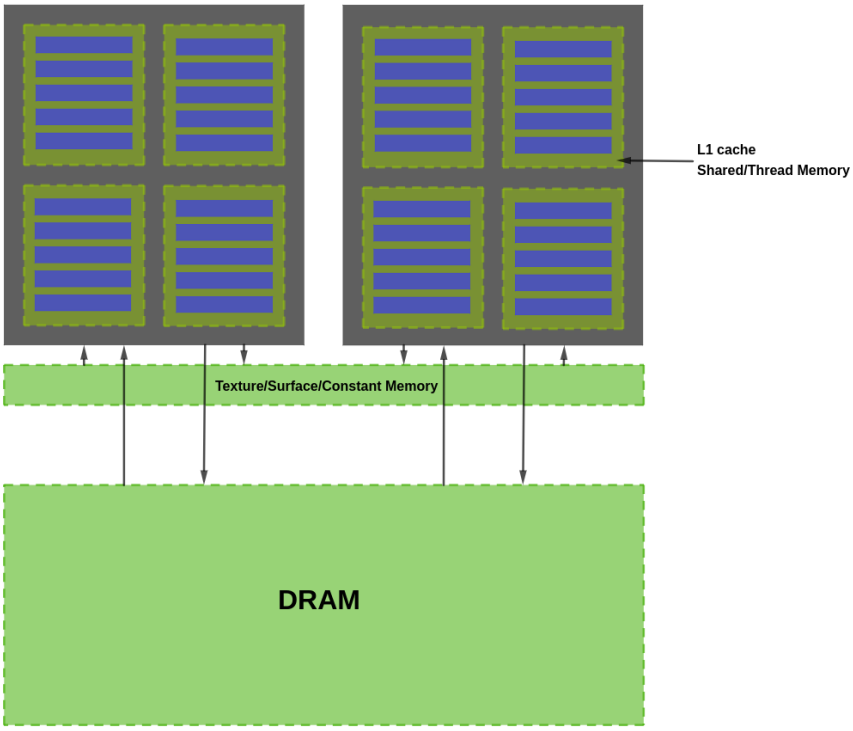


Figure 4.4: Illustration of the memory structure on an Nvidia GPU.

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

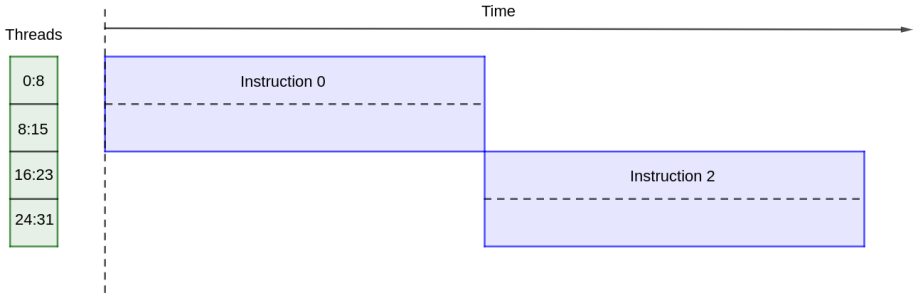


Figure 4.5: Figure illustrating thread divergence for a single warp of size 32 when two instructions are issued for only certain threads in the warp.

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 - UNFINISHED

Currently, there are three existing methods of parallelising the finite element method and implementing it on a GPU:

- Partitioning the linear system across the threads and solving.
- Applying a domain decomposition.
- Utilising multigrid techniques.

In this paper, the one implemented is the partitioning of the linear system across the processors and solving. This is, as it says on the tin; the element matrices and element vectors are assembled in parallel, the global stiffness matrix and stress vector are is then also assembled across the threads in parallel at which point this linear system is then solved using standard decomposition approaches. For this paper, the linear system decomposition was left to be handled by Nvidia's existing direct solver library cuSOLVER and the handling of the element matrices and assembly was implemented manually. These steps are no different really than discussed Section 3.1, barring needing to handle certain amounts of communication between threads handling nodes within the same cell, and also when mapping back to the global stiffness matrix/stress vector.

Domain decomposition is another technique applied to parallelising the FEM. The premise behind this approach is, if the mesh, or domain, Ω, upon which the PDE is defined is decomposed into n separate subdomains $\{\Omega_i\}_n$, these can each be solved iteratively using their own individual boundary conditions, as if they were independent PDEs. There are two main domain decomposition approaches, the Schwarz method for overlapping subdomains and the Schur method for disjoint domains. At each iteration there is corrections done

on the locals boundary conditions via communication from neighbours and the iterations continue. This can often, be quite expensive in terms of computation costs, and depending on the architecture might not always be the optimal choice. For a GPU, due to the lack of being able to communicate between threads in separate blocks, this could cause some bottlenecks in the approach.

The final main approach to parallelising the FEM is by implementing a multigrid approach. The idea behind the multigrid approach is to coarsen the grid, and use interpolation to find estimates to this courser grid by iterating until convergence. This can then be refined using prolongation to make a finer mesh, of just the error from the previous prolonged mesh. This finer mesh can then have the same approach applied until convergence and the error prolonged again. This can oscillate up and down between fine and coarse grids until a converged solution is found. In the case of the FEM, this is a natural and efficient implementation as the entire domain can be divided into coarse and fine grids, each of which can be solved in parallel across the threads.

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,

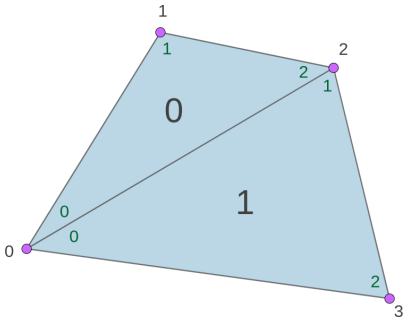


Figure 4.6: Small two cell example of a mesh with global and local node numberings illustrated.

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 4.6 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 4.7 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.

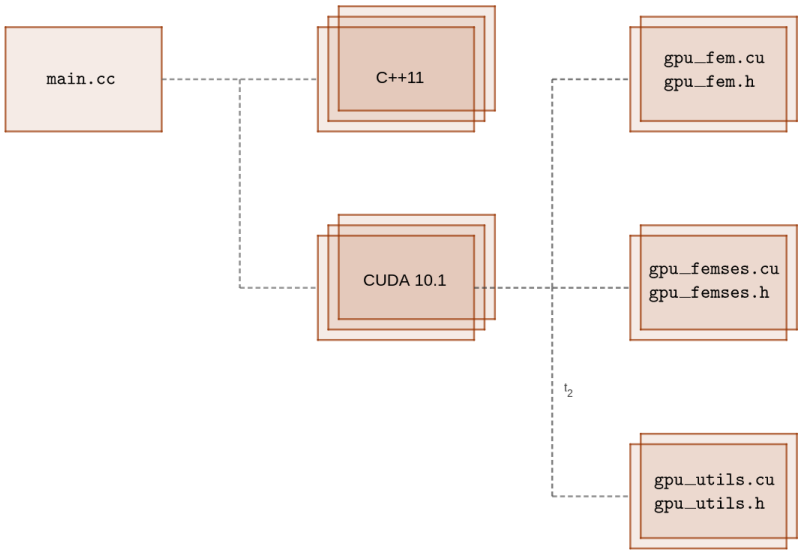


Figure 4.7: Structure of CUDA GPU code.

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.

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 should 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 4.8, a flow chart of the approach taken to code a basic FEM implementation on a GPU is illustrated. The coloured steps the ones which are executed involving CUDA code, the white steps are ones which are executed entirely 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

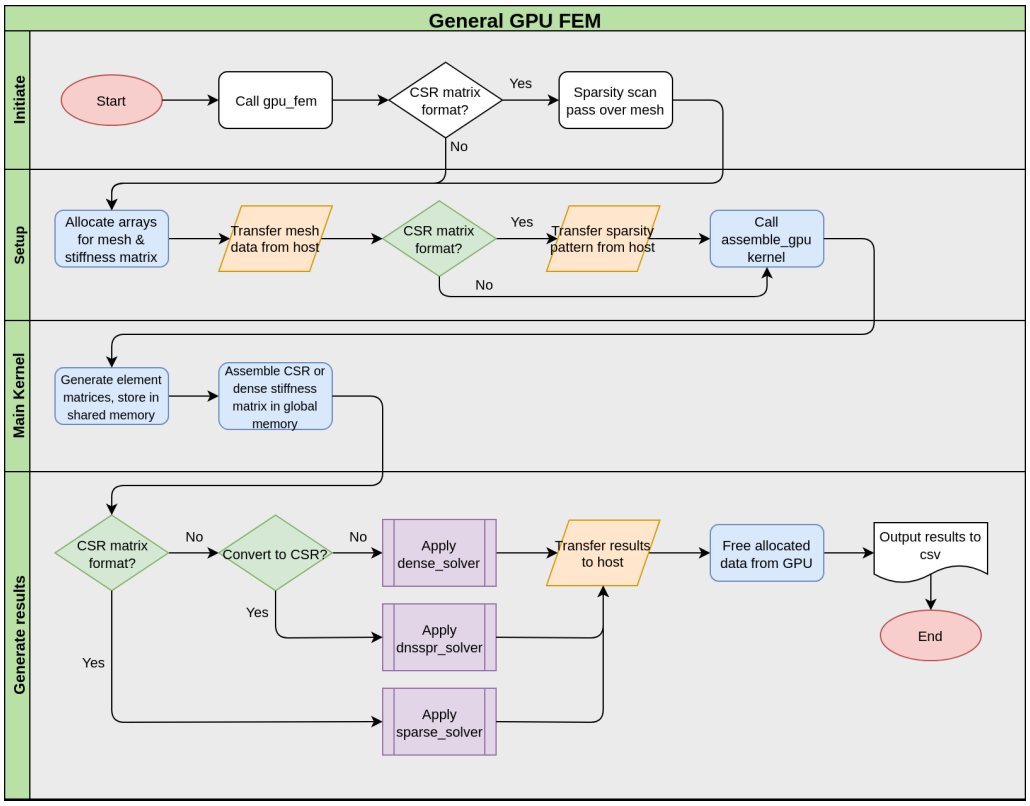


Figure 4.8: Flowchart of general GPU FEM implementation.

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 - Mention solver taking up space

In terms of storage, for a problem which scales like FEM does, managing one's resources correctly and not doing excessive memory allocations is important - along with, of course, 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 4.9 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

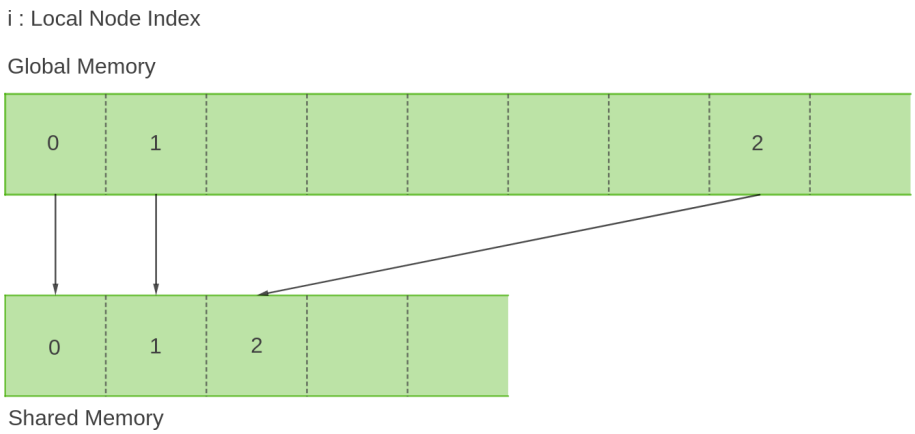


Figure 4.9: Illustration of global memory random access pattern being transferred to shared memory to ensure coalescence.

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 4.1; 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. 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 4.10 shows flowcharts of the necessary steps required in order to execute the three solver variants.

```
__global__ void assemble_gpu(
                float *L,
                float *b,
                float *vertices,
                int *cells,
                int *is_bound,
                float *bdry_vals,
                int order,
                int num_cells)
{
    // idx = cell number
    int idx = blockIdx.x * blockDim.x + threadIdx.x ;
    // idy = local node number
    int idy = blockIdx.y * blockDim.y + threadIdx.y ;
    // shared mem to store elem mats & constants
    extern __shared__ float temp1[];

    if(idx < num_cells && idy < blockDim.y ){

        assemble_elem(vertices, cells, is_bound, bdry_vals, temp1, idx, idy)
            ;
        __syncthreads();
        assemble_mat(L, b, vertices, cells, temp1, idx, idy, order);
    }
}
```

Listing 4.1: Main kernel in general FEM approach to generate element matrices and assemble global linear system.

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. The grid structure is defined as `block_size_X` FEM cells per block, within each block

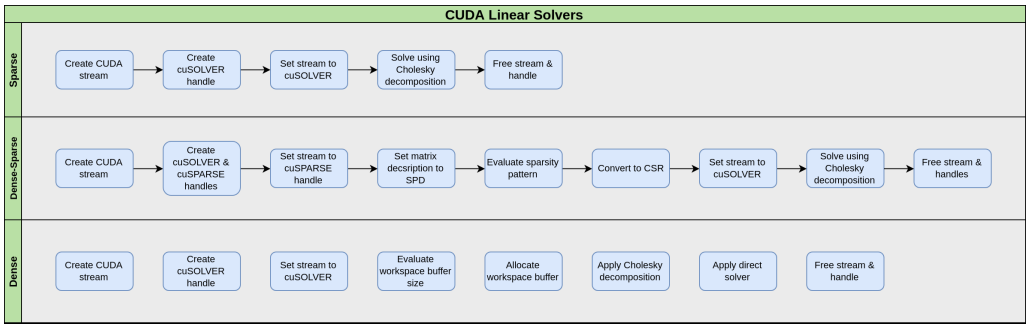


Figure 4.10: Flowchart illustrating the steps needed to implement solvers from CUDA's linear algebra libraries.

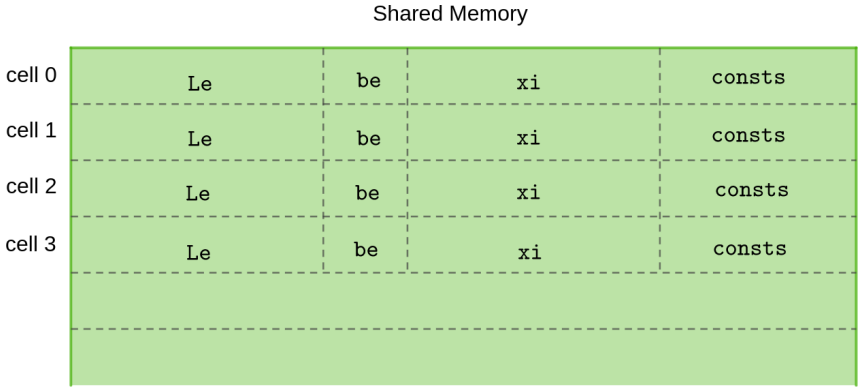


Figure 4.11: Illustration of shared memory structure for assembly kernel for a block containing four cells of the mesh.

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 4.11 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 29, 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

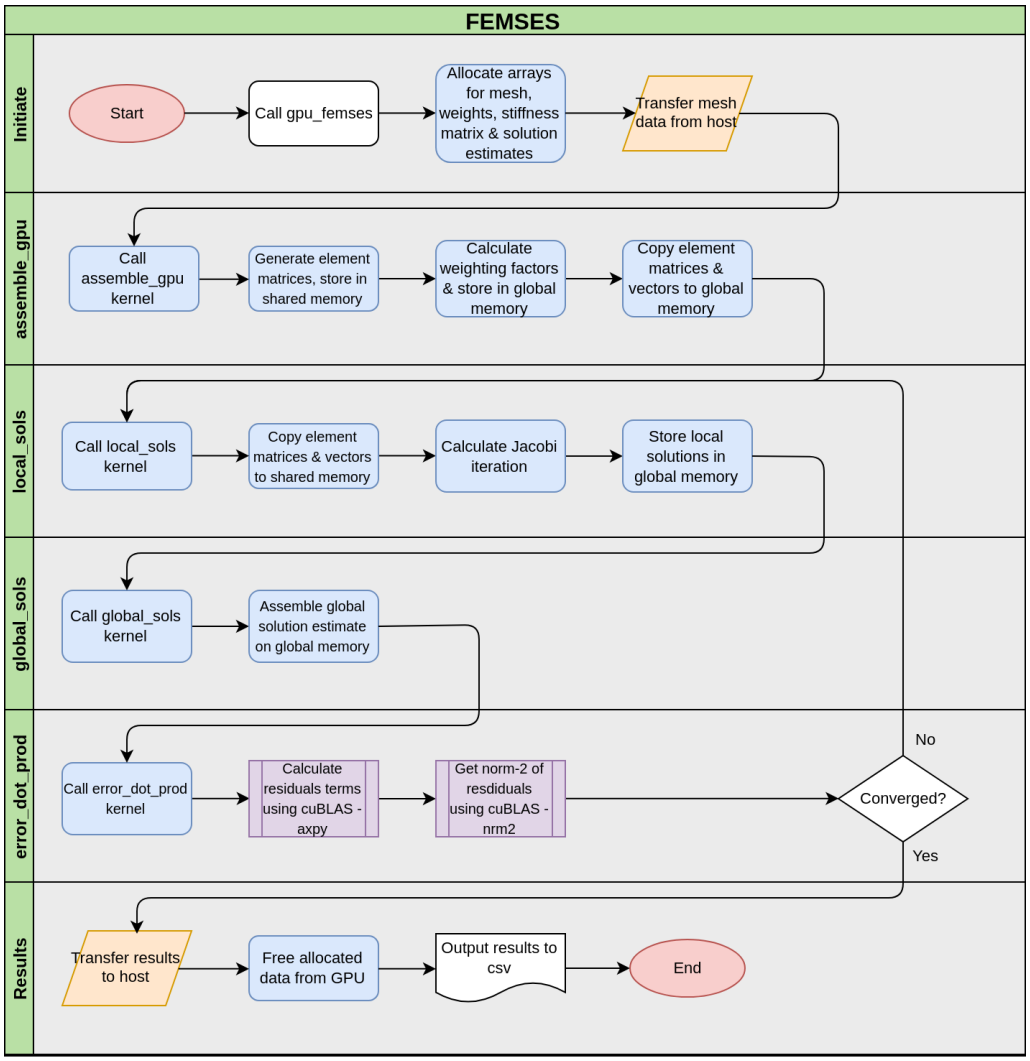


Figure 4.12: Flowchart of FEMSES approach.

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 2 or Algorithm 3, 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.

4.4.4 FEMSES

Figure 4.12 shows a float chart of the FEMSES process when applied with CUDA code. Like before, the coloured steps are carried out by/on the GPU, the white are the host C++ 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 speed-ups 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 4.12:

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 4.2, 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.

```
__global__ void assemble_elems_gpu(
                float *Le,
                float *be,
                float *w,
                float *u_glob,
                float *vertices,
                int *cells,
                int *is_bound,
                float *bdry_vals,
                int order,
                int num_cells)
{
    // idx = cell number
    int idx = blockIdx.x*blockDim.x + threadIdx.x;
    // idy = local node number
    int idy = blockIdx.y*blockDim.y + threadIdx.y;
    extern __shared__ float temp1[];

    if(idx < num_cells && idy < blockDim.y){
        // __device__ fn taken from other header to avoid code-reuse //
        assemble_elem(vertices, cells, is_bound, bdry_vals, temp1, idx, idy
            );
        __syncthreads();
        calc_weights(w, cells, temp1, idx, idy);
        elems_glob_cpy(Le, be, temp1, idx, idy);
    }
    // assigning initial guess for Jacobi to 0.0
    if( (idx*3) + idy < order){
        u_glob[(idx*3) + idy] = 1.0;
    }
}
```

Listing 4.2: Initial kernel in FEMSES to generate element matrices and store in global memory.

The second kernel, evaluates the local solution estimates. Illustrated in Listing 4.3, 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.

```
__global__ void local_sols(
                float *Le,
                float *be,
                float *ue,
                float *up_glob,
                int *cells,
                int num_cells)
{
    int idx = blockIdx.x*blockDim.x + threadIdx.x;
    int idy = blockIdx.y*blockDim.y + threadIdx.y;
    extern __shared__ float temp1[];

    if(idx < num_cells && idy < blockDim.y){
        elems_shared_cpy(Le, be, temp1, idx, idy);
        __syncthreads();
        jacobi_iter(ue, up_glob, cells, temp1, idx, idy);
    }
}
```

Listing 4.3: Kernel to evaluate local solutions in FEMSES using Jacobi iteration.

The final purpose-written kernel in this method assembles the global solution array using the weightings. Seen in Listing 4.4, 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.

```
__global__ void glob_sols(
                float *Le,
                float *w,
                float *u_glob,
                float *ue,
                int *cells,
                int num_cells)
{
    int idx = blockIdx.x*blockDim.x + threadIdx.x;
```

```
    int idy = blockIdx.y*blockDim.y + threadIdx.y;
    int v;
    float Lii, weight;

    if(idx < num_cells && idy < blockDim.y){
        v = cells[(idx*3) + idy];                // getting global vertex
            number
        Lii = Le[(idx*9) + (idy*3) + idy];

        weight = Lii/w[v];

        atomicAdd(&u_glob[v], weight * ue[(idx*3) + idy]);
    }
}
```

Listing 4.4: Kernel to assemble global solution vector in FEMSES from local solution estimates using weighting.

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 4.11, 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 4.5 details this. Taking the previous global solution estimate as an input, it reads this in as $u^{(\mathtt{old})}$, as well as reading in the element matrices and element vectors. Each thread will calculate its own individual $u^{(\mathtt{new})}$, 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.

```
__device__ void jacobi_iter(
                float *ue,
                float *up_glob,
                int *cells,
                float *temp1,
                int idx,
```

```
                int idy)
{
    // float *Le_shrd, *be_shrd, *ue_old;
    float ue_new;
    int v;
    int offset = 15*threadIdx.x;

    /*
    Le_shrd = &temp1[offset];
    be_shrd = &temp1[offset + 9];
    ue_old  = &temp1[offset + 12];
    */

    v = cells[(idx*3) + idy];

    ue_new = temp1[(offset + 9) + idy];
    temp1[(offset + 12) + idy] = up_glob[v];

    __syncthreads();

    ue_new -= temp1[offset + (idy*3) + ((idy+1)%3) ] * temp1[(offset + 12)
        + (idy+1) % 3];
    ue_new -= temp1[offset + (idy*3) + ((idy+2)%3) ] * temp1[(offset + 12)
        +  (idy+2) % 3];

    ue_new /= temp1[offset + (idy*3) + idy];

    ue[(idx*3) + idy] = ue_new;
}
```

Listing 4.5: Device code for the Jacobi iteration.

Chapter 5

Results

5.1 Test-bed Architecture

The problem stated in Section 2.5.1 was used to computationally test the topics discussed in this paper. In terms of test bed architecture, the timings were all conducted on a 3.4 Ghz Intel Core i5-3570K. with 6 Mb cache and 4 cores along with 16 GB of DDR3 memory. For the GPU times, two cards were used: a Kepler architecture, Tesla K40 clocking 745 Mhz-875 Mhz, carrying 15 SMs, each with 192 CUDA cores, it carries 12 GB of GDDR5 global memory clocking 1.5 GHz and a 288 GB/s bandwidth. The other card tested was the more modern, Turing architecture, RTX 2080 Super, clocking at 1.65 Ghz-1.815 Ghz, carrying 48 SMs, each with 64 CUDA cores, 8 GB of GDDR6 memory with a 496 GB/s bandwidth. Both cards have 48 Kb of shared memory per block and 16 Kb of per thread memory available. Note also, due to the limitations of the most recent version of Nvidia's Visual Profiler tool, installed on the system mentioned, and also the removal of some of the counters from the Turing architecture, for profiling, a different system with an older version of Nvidia's Visual Profiler and an older Kepler architecture GeForce GTX 780. The GTX 780 clocks at 869 Mhz-900 Mhz, has 2304 CUDA cores, and 3 GB of GDDR5 DRAM with 288 GB/s bandwidth. The system was also using an Intel Core i5 7600 with 3.5 Ghz clock speed, 6 cores and 6 MB L3 cache. Table 5.1 illustrates all the specifications for the setup. From a software perspective, all serial code was written in C++11 and compile using `gcc5.4.0` with `-O3` flag enabled. The `-std=c++11` flag was of course enabled to ensure that the correct version of C++ and the STL was being utilised. In order to take advantage of the linear solvers, the Intel's MKL 18.0.4 library also needed to be called compiled with the

GPU	*Arch*	*Clock Speed*	*SMs*	*CUDA Cores*	*DRAM*	*Bandwidth*	*Shared*
Tesla K40	*Kepler*	745 *Mhz*	15	2880	12 *GB GDDR*5	288 *GB/s*	46 *Kb*
RTX 2080 Super	*Turing*	1.65 *Ghz*	48	3072	8 *GB GDDR*6	496 *GB/s*	46 *Kb*
GeForce GTX 780	*Kepler*	869 *Mhz*	12	2304	3 *GB GDDR*5	288 *GB/s*	46 *Kb*

Table 5.1: Testbed architecture.

serial code. To enable this, the binary executable and library files needed to be appended to the `PATH` and `LD_LIBRARY_PATH` environment variables, this was done by either calling the correct modules, if available or manually appending by,

```
export PATH=$PATH:/home/support/apps/intel/18.0.4/bin/
export LD_LIBRARY_PATH=$LD_LIBRARY_PATH:/home/support/apps/intel
   /18.0.4/mkl/lib/intel64/intel64/
```

. The list of flags then needed for the Make were as follows,

- `-lm`
- `-lmkl_intel_lp64`
- `-lmkl_sequential`
- `-lmkl_core`.

For the GPU code, all of it was written in CUDA with the most recent 10.1 SDK. The code was compiled with `nvcc` and `-O3` flag enabled again. In the implementation, certain device functions are shared across different headers so the `-relocatable-device-code=true` flag had to be enabled. the last consideration for the setup is the CUDA SDK 10.1, and its libraries used for linear solutions and BLAS operations. To enable these, again the `PATH` and `LD_LIBRARY_PATH` environment variables had to be appended, in this instance by,

```
export PATH=$PATH:/usr/local/cuda-10.1/bin
export LD_LIBRARY_PATH=$LD_LIBRARY_PATH:/usr/local/cuda-10.1/lib64
```

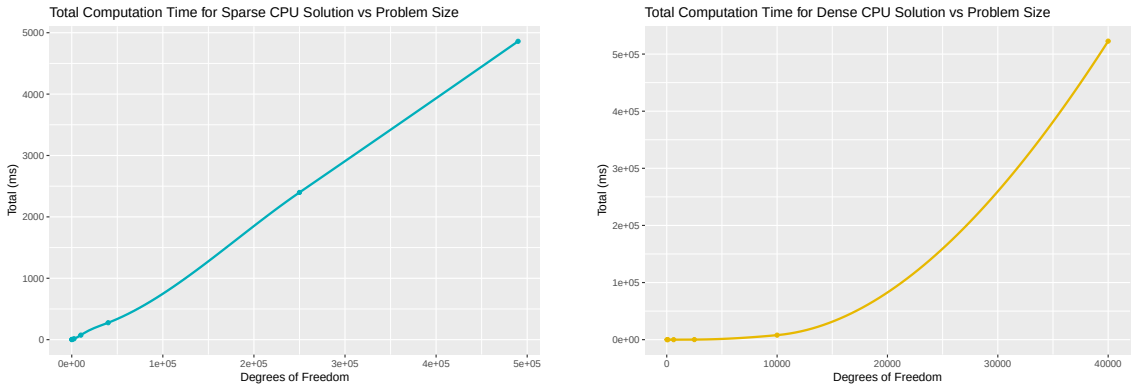
. The list of flags then needed:

- `-lcusolver`
- `-lcusparse`
- `-lcublas`.

Remark. *For all the testing mentioned in the rest of this chapter, each data point, or setup combination was run five times and the mean score taken. All models were plotted using `ggplot2` and loess smoothing fitted with linear, quadratic or logarithmic regression depending on the data. All grey ribbons represent 95% confidence intervals for these models.*

5.2 Serial Code Profiling

Figure 5.1 shows the total time taken in ms for the serial code to complete in both dense matrix and CSR variants. Clearly, the sparse case achieves a near linear scaling and appears to perform quite efficiently as the number of unknowns increases. The dense case on the other hand appears to have quadratic scaling, and takes orders longer time to complete as it gets larger. As mentioned in Section 3.3, the benefit of using CSR can be huge by comparison to solving a dense system, particularly for systems as predictably sparse as the FEM. Figure 5.2 shows a logarithmic speed-up scaling of the sparse solution over the dense,



(a) Total time taken for sparse case.

(b) Total time taken for dense case.

Figure 5.1: Total time taken for serial sparse and dense cases versus problem size.

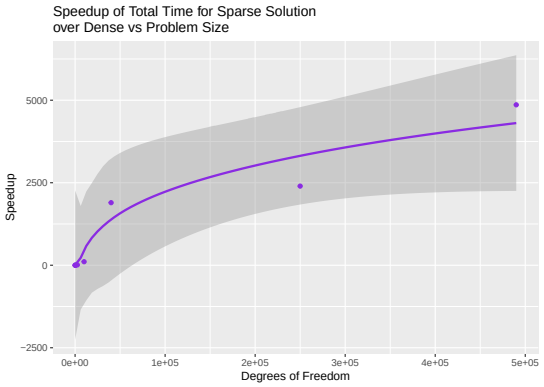
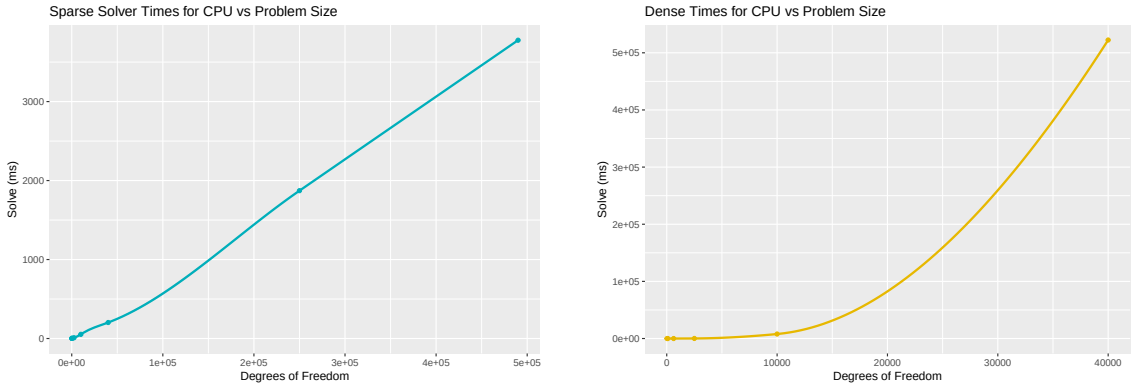


Figure 5.2: Speed-up of serial sparse solution total time over dense solution versus problem size.

reaching up to 5000× faster. As it turns out, there is a couple of contributing factors here, the first being that the direct dense solvers simply do not actually scale very well in any case, and the second is that, not alone do sparse direct solvers scale better, but also, the Intel's DSS solver is a very efficient and well written piece of software compared to their dense LAPACK library.

The logic behind this inference, that the solvers are what is causing the non-linear scaling, is relating to their dominance in the total computation time. Figure 5.3 shows the times taken for both sparse and dense solvers. Both Figures are near identical to Figure 5.1, clearly demonstrating that the solvers are taking up almost all the computation time and driving the scaling pattern. Figure 5.4 actually illustrates far better this dominance, showing the proportion of computation time taken by the various kernels in the FEM. It also demonstrates the better scaling of the sparse solver, the proportion of which remains relatively constant as the problem size gets larger, unlike the dense solver which tends towards 100%. The smallest problem sizes were removed from both charts as the timing was too small to actually get an accurate representation.

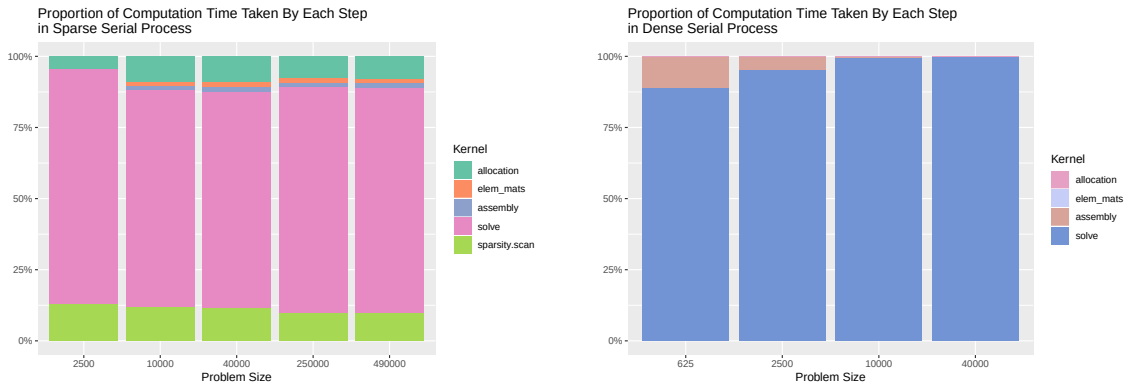
In terms of the profiling and performance of the other kernels involved in the process, Figure 5.5 shows the timings taken for the generation of the element matrices, assembling both CSR and dense global stiffness matrices and the sparsity scan performed a priori.



(a) Time taken for sparse solver.

(b) Time taken for dense solver.

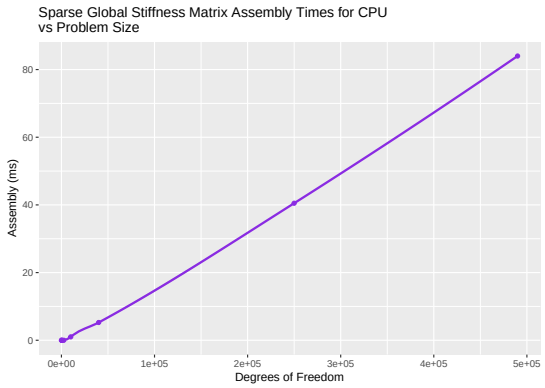
Figure 5.3: Time taken for linear solvers in serial sparse and dense cases versus problem size.



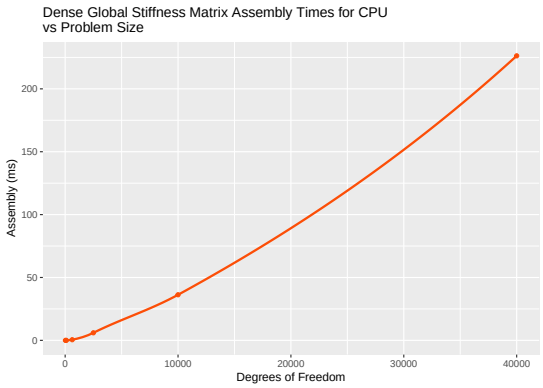
(a) Proportion of time taken for each kernel in sparse solution.

(b) Proportion of time taken for each kernel in dense solution.

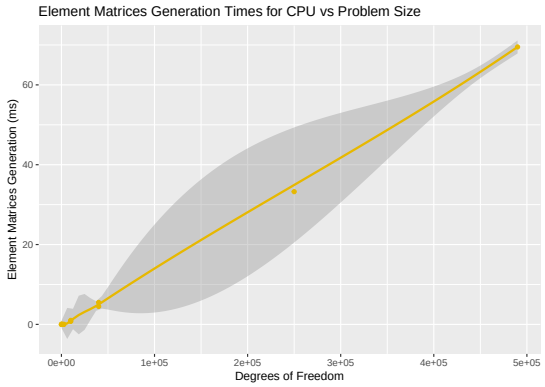
Figure 5.4: Proportion of computation time taken per kernel for sparse and dense CPU cases.



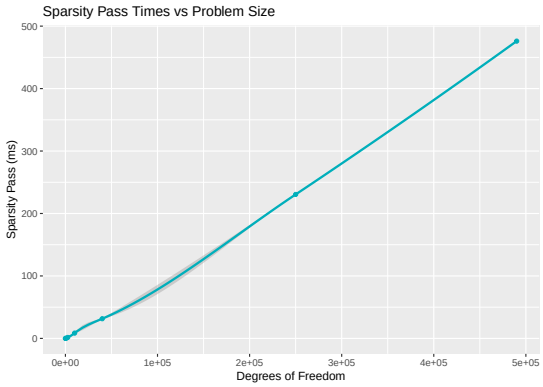
(a) Time taken for assembly of CSR global stiffness matrix.



(b) Time taken for assembly of dense global stiffness matrix.



(c) Time taken for generation of element matrices.



(d) Time taken to perform sparsity pass.

Figure 5.5: Timings taken versus problem size for various kernels involved in CPU FEM method.

Most of the Figures show a linear scaling, barring Figure 5.18b, showing a slight quadratic scaling - to be expected given the quadratic scaling in matrix size.

Results

Figure 5.6 shows a 3D plot of the resulting solution to the PDE achieved by the serial implementation for a 200×200 case.

5.3 GPU Performance

5.3.1 Standard FEM

The standard FEM GPU approach achieved some varying times, depending largely on whether or not sparse or dense solvers were used. The generation of the element matrices, and assembly of the stiffness matrices reached some quite substantial speed-ups. However, as is detailed in this section, these kernel timings were dominated by the far more computationally expensive linear solvers.

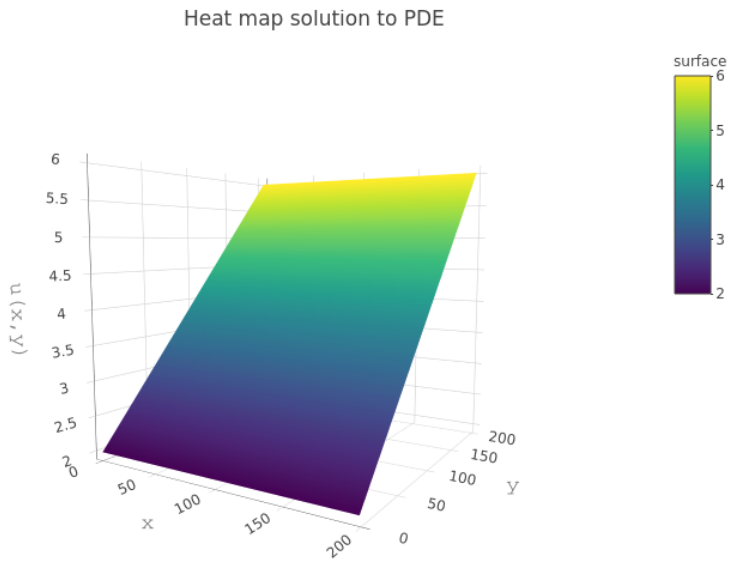
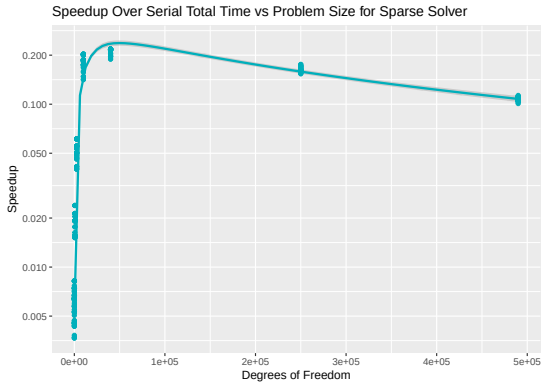


Figure 5.6: Plot of solution to Laplace equation posed in Section 2.5.1 generated by serial code.

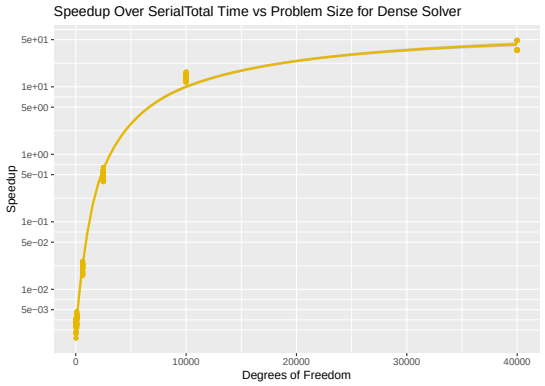
Total Timings

Figure 5.7 illustrates the speed-ups of the total timings taken to complete the FEM for the various solver approaches against problem size. Looking initially at the sparse or CSR variant, Figure 5.7a shows the speed-up over the sparse serial case. Clearly it does not actually achieve a speed-up at all, but is in fact slower for all problem sizes. It does demonstrate and initial spike, but this can be written down to simply a factor of GPU programming. Often, for small problem sizes, there are initial costs of transferring data from host to device, allocating memory on the device's DRAM and even just the card needing to "warm-up" - for lack of a better term. For small problem sizes, this cost is proportionally much larger, and clearly as the number of unknowns grows, this factors in less. There is also, naturally, much less parallelisation achievable at smaller problem sizes than larger ones. Thus, this spike is natural and to be expected and not really a result to be concerned about. What is more concerning was the lack of positive scaling. As it turns out, the total time is quite substantially dominated by the cuSOLVER sparse direct solver, which happens to not perform anywhere near as optimally as Intel's DSS, and even sees a decreasing scaling as problem size gets larger - though there is little can be done about that unfortunately.

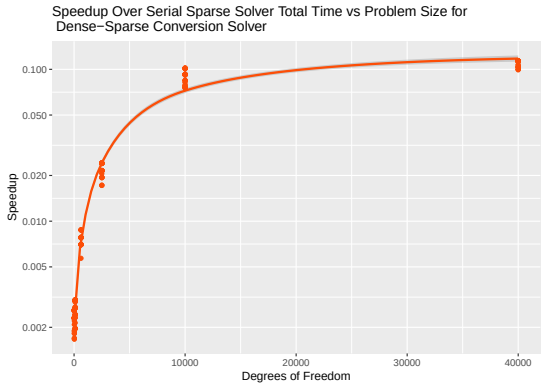
The dense solver, from a glance at Figure 5.7b, demonstrates far more promising results. There is a clear logarithmic scaling present in the graph. However, again from the offset with small problem sizes, parallelising the problem is not worth it, and results in slower times. However, evidently, this improves quite rapidly and as the degrees of freedom approach 40,000, the solver outperforms the serial code by a speed-up of around 50×. Again here, there is a substantial proportion of the computation time taken up by the linear solver which is shown later.



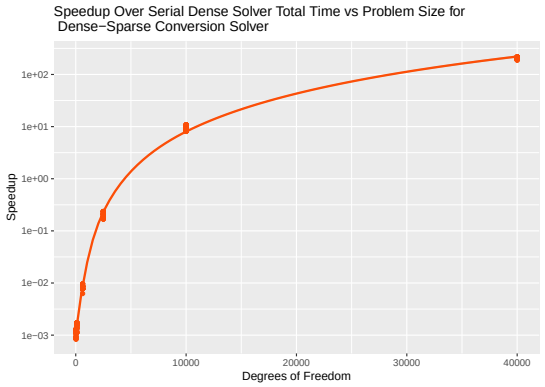
(a) Total time speed-up over sparse serial solution.



(b) Total time speed-up over dense serial solution.



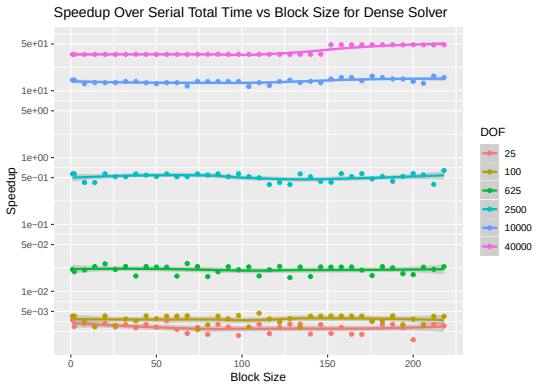
(c) Total time speed-up dense-sparse conversion method over sparse serial solution.



(d) Total time speed-up dense-sparse conversion method over dense serial solution.

Figure 5.7: Speed-ups of total GPU times over serial code vs. problem size for sparse, dense and dense-sparse conversion solutions.

(a) Total time speed-up over sparse serial solution.



(b) Total time speed-up over dense serial solution.

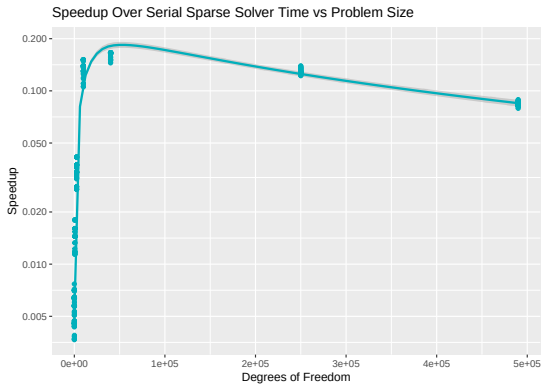
Figure 5.8: Speed-ups of total GPU times over serial code versus block size, for sparse and dense solutions.

The dense-sparse conversion was compared to both the serial CSR solver and the serial dense solver in Figures 5.7c 5.7d, since it sits somewhere in between both, having a dense assembly and sparse solver. Given the proportion of computation time is considerable, it is to be expected that the dense-sparse solver performs poorly also by comparison to Intel's DSS - which it indeed does. The scaling has the same spike as the sparse-sparse comparison, though it is not immediately obvious due to memory limitations not permitting allocation of larger problem sizes - which the pure CSR solver had the benefit of avoiding to a larger extent. This solver drastically outperforms the purely dense serial solution, achieving large speed-ups of up to 100×.

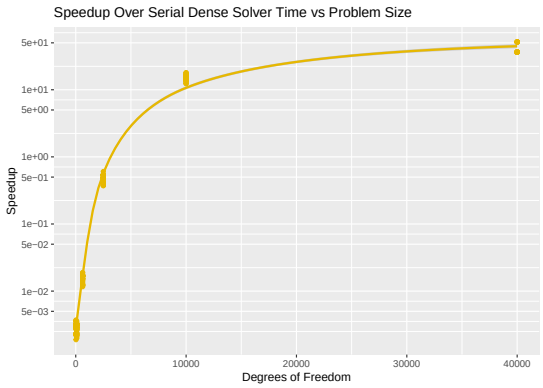
Figure 5.8 shows the speed-ups seen over the CPU code for sparse and dense solutions versus the choice of block size input from the command line. Without yet going into the details of the linear solvers computational dominance, in both cases, the stability in timings with variation of block size should tell enough that, the purpose-written code has little actual impact on the resulting final timings. A benefit of the linear solvers is that they are optimised such that the best choice of block size to reduce warp divergence and maximise shared memory usage is selected for the user. Thus, meaning the choice of block size, should have not much impact - shown very much so in Figure 5.8. Rather interestingly though, it does show the nice scaling with problem size for the dense case in Figure 5.8b and the decreasing scaling for the sparse case in Figure 5.8a, where 40,000 degrees of freedom achieves more speed-up than 490,000.

Linear Solvers

Looking at the actual performance of the linear solvers themselves, Figures 5.9a, 5.9b almost identically replicate the results seen in Figures 5.7a, 5.7b - just more illustrating how dominant these are towards the total time taken. In Figure 5.10, this can be seen from screen-shots taken from Nvidia's Visual Profiler where the massive chunk of computation time taken by these solvers, for a relatively small problem size of 2,500, compared to the rest the FEM process is shown - the kernel written for this report takes 0.4%. It isn't



(a) Sparse solver speed-up over Intel's MKL DSS.



(b) Dense solver speed-up over Intel's MKL LAPACKE.

Figure 5.9: Speed-ups of cuSOLVER's solver times over Intel's MKL versus problem size, for sparse and dense solutions.

Figure 5.10: Proportion of computation time taken by various kernels involved shown in Nvidia's Visual Profiler for a problem size of 2,500 unknowns for sparse GPU method.

that these wholly perform poorly, more-so that Intel's DSS solve for CSR linear systems is optimised to very impressive levels, and simply outperforms Nvidia's cuSOLVER and cuSPARSE solutions. The dense solver on the other hand actually performs quite well compared to Intel's MKL library. It was not the premise of this paper to improve on existing direct linear solvers so this is simply a result that needs to be taken with a pinch of salt.

While it may seem that Nvidia's CSR solver is actually a poorly written and poorly performing piece of software, and one would be better off using the dense solver - this is certainly not the case, it just happens to not be as optimised as Intel's. Figure 5.11 illustrates the speed-ups of the sparse solver over the dense solver as problem size increases. Notably, it gets up to around 10× speed-up and is, in fact, a better performing solution than assembling a dense system and solving. This is the benefits mentioned in Section 3.3 visibly in action.

Allocations & Transfers

Figure 5.12 shows the total allocation times for both sparse and dense cases. The dense case seen in Figure 5.12b, is not expected to have a linear scaling but more so a polynomial scaling. It illustrates a 2^{nd} order polynomial scaling, this is to be expected since, excluding

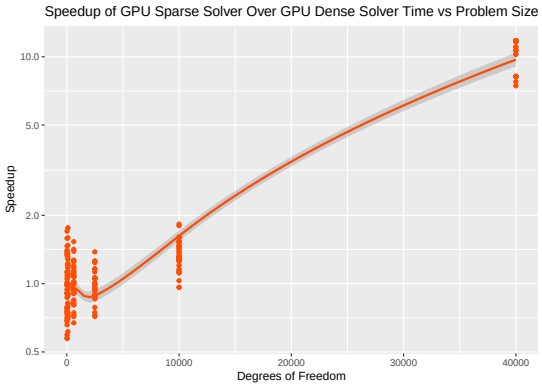
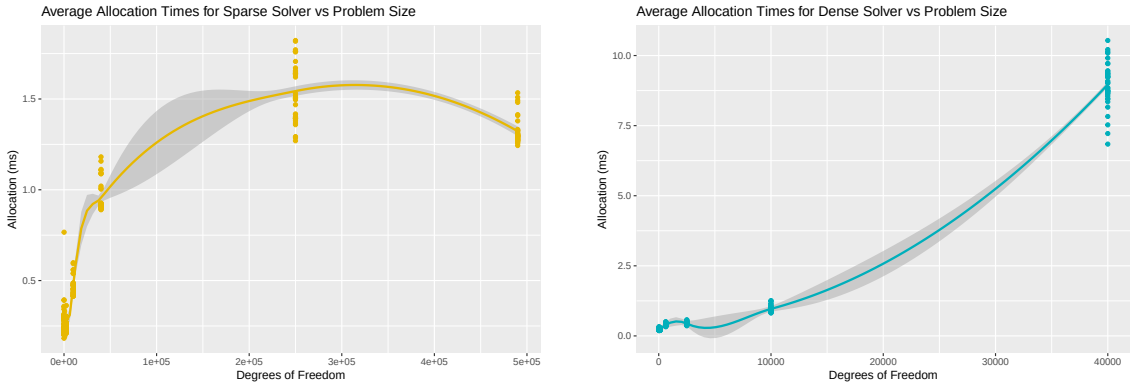


Figure 5.11: Speed-ups of cuSOLVER's sparse linear solver over cuSOLVER's dense solver against problem size.



(a) Allocation time taken for sparse case.

(b) Allocation time taken for dense case.

Figure 5.12: Total allocation time taken for both dense and CSR GPU approaches.

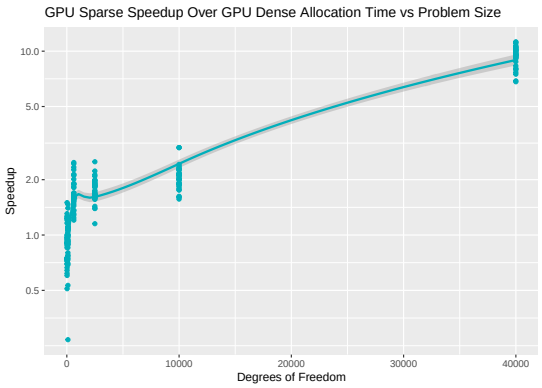
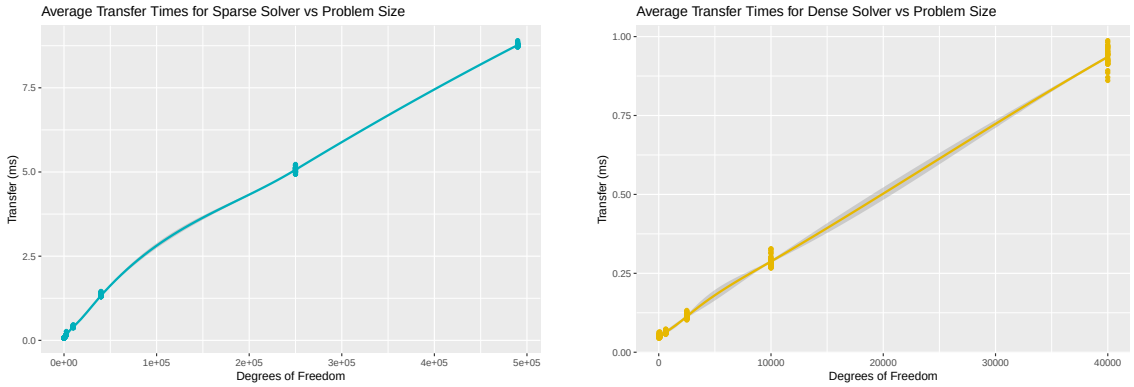


Figure 5.13: Speed-up of allocation time taken for sparse case over dense versus problem size



(a) Transfer time taken for sparse case.

(b) Transfer time taken for dense case.

Figure 5.14: Total transfer time taken for both dense and CSR approaches.

the mesh, the dominant array needing to be allocated is of size `order` $\times$ `order`. The sparse case in Figure 5.12a, on the other hand, is harder to predict the scaling of. This is due to the fact that as the matrix gets larger, the number of non-zeros present in the matrix will be totally dependent on the both the shape of the mesh, and the degrees of freedom chosen for the basis functions - be it P1 or P2 et cetera. The timing does appear to decrease, though the inclination here is that this it is due to variance and the small sample size of five per data point - in this case five runs per block size and problem size. Figure 5.13 shows the speed-up seen in performing the sparse case over the dense. Since both variants allocate the five arrays needed for the mesh as well as the stiffness matrix, naturally the expected, and observed, result here is sparse speed-up over dense.

Unlike in the allocation case, the sparse timings can be a bit more predictable. For allocating, all three of the arrays for the stiffness matrix have to be allocated, as well as the array for the stress vector. For the transfer, the mesh of course, as in allocation, needs to be transferred, and solution vector or former stress vector, but only two of the three CSR arrays. This should give rise to a much more linear scaling, as demonstrated in Figure 5.14a. It is not a perfect linear scaling, nor should it be, but the reduced effect of the irregular scaling of the CSR matrix is definitely evident. The dense case on the other hand, only transfers the mesh and the stiffness matrix, all of which scale in size linearly and thus,

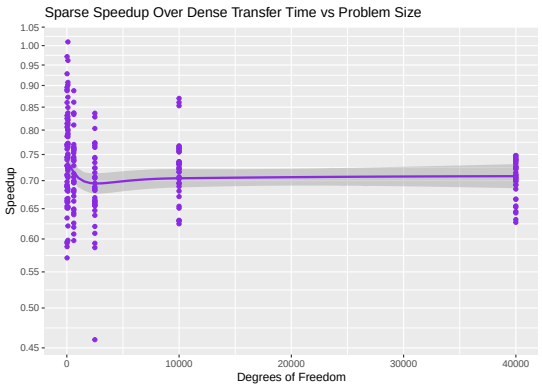
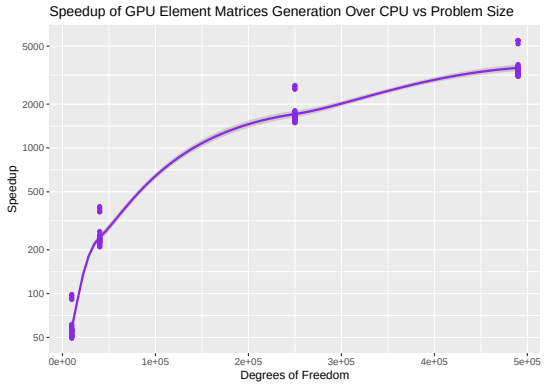
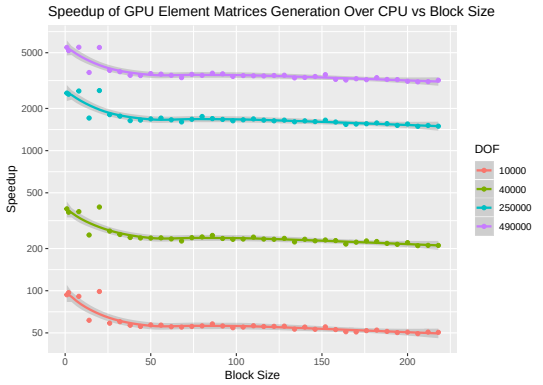


Figure 5.15: Speed-up of transfer time for sparse case over dense case versus problem size.



(a) Speed-up over serial generation of element matrices versus problem size.



(b) Speed-up over serial generation of element matrices versus blocks size.

Figure 5.16: Speed-ups of GPU generation of element matrices and vectors over serial code against problem size ans block size.

as is seen in Figure 5.14b, observe a nice linear scaling. Figure 5.15 shows the speed-up seen in the sparse case over the dense. In this case, as would be entirely presumed, the dense case takes about 70% of the time. Of course this would be expected given there is no sparsity pattern to transfer from the host.

Generation of Element Matrices & Vectors

One of the two most relevant timings in this section is the generation of element matrices. As was discussed, there is a limited amount can be done about pre-existing linear solvers. However, there is a great deal of parallelism can be achieved when generating the element matrices and element vectors for the FEM. This is one of two of the the areas into which most of the thought regarding optimisation was put. All the considerations mentioned in Section 4.1.1 were taken into account here: optimising shared memory and registry usage, minimisation of thread divergence from warps, coalescing memory when at all possible et cetera.

Figure 5.16 shows the massive speed-ups seen over the CPU against both problem size and block size when the device function was measured. There is a massive logarithmic scaling as the degrees of freedom increases, reaching up to 5000× faster than the serial

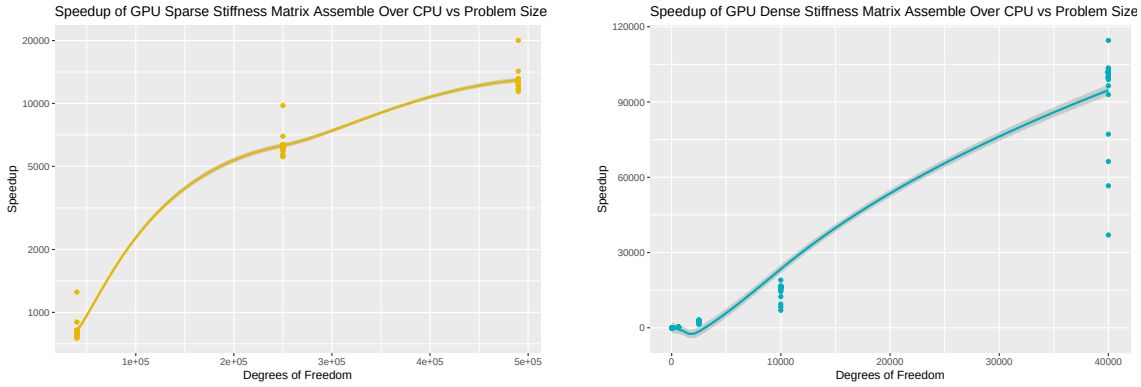
code for the largest problem size tested. In terms of the comparison with block size, unlike in the pre-existing linear libraries, the choice of block size does matter here and is visible in Figure 5.16b, where there is a dip as block size increases - albeit it is not very clear what differences it makes after that. Figure 5.17 demonstrates these same graphs but faceted by problem size. For considerations of shared memory, the CUDA devices tested in this paper, all come with a 1:3 ratio of thread registers to shared memory - specifically 16 Kb:48 Kb. There is quite a substantial amount of machinery needed for the FEM, resulting in a large amount of pointers to global and shared memory. Due to this, the thread registers end up overflowing. The conscious effort was made to use only a single shared memory pointer and avoid any unnecessary variables within device functions if at all possible. However, this can only help so much. Given that shared memory is quite large for the smaller block sizes, an added optimisation performed here was checking outside the kernel to deduce if there was shared memory spare, if there was, the memory was reconfigured to a 1:1 or 3:1 ratio to avoid and hedge against this overflow. The benefits are very clear in Figure 5.16, observing more speed-up seen for all problem sizes. The overflow of thread registers is clear from the off as the drop-off seen in speed-up is observed from the smallest problem sizes.

Regarding thread divergence, it can be loosely seen that the speed-up appears to increase in and around multiples of 32. However, it isn't definitive for sure, and is almost certainly outweighed by the cost of memory transfer, to and from the thread registers overflowing into L2-cache.

Global Stiffness Matrix & Stress Vector Assembly

The assembly of the global stiffness matrix and stress vector are another important point of optimisation in this study, as it is another area which can provide huge amounts of parallelisation. Figure 5.18 shows the speed-ups seen for both sparse and dense variants of the device function, achieving even more impressive results than the element matrices generation. Both show massive speed-ups of $20,000\times$ and $100,000\times$, respectively. It might be an easy thought to assume that the sparse assembly would, in fact, be faster, but it turns out, to be slower than the dense assembly. Figure 5.19 shows this fact. There are two contributing factors to this unexpected anomaly. The first being, there are far less variables, particularly pointers, needed in the dense assembly function and so the overflow of the, already-packed, registry memory is less of a factor. This can actually be seen in the initial spike, and quick drop off in speed-up seen in Figure 5.19, before the thread registers overflow in the sparse function. The second, main contributing factor to this, is the lack of coalescence and irregular memory access pattern of the CSR matrix. The algorithm requires jumping back and forward through three separate arrays to find the correct position in the matrix to perform an atomicAdd. This wouldn't particularly be an issue in the shared memory, but in global memory, this will provide quite a slow down. The dense assembly, however, is more rudimentary, and is also sequential accessing so does not suffer from these bottlenecks. So it then begs the question, which should have already an obvious answer, is the trade off for a sparse solver better than a dense assembly?

Figure 5.17: Speed-up of GPU generation of element matrices over CPU versus block size for various problem sizes and memory configurations.



(a) Speed-up over serial assembly of CSR matrix.

(b) Speed-up over serial assembly of dense matrix.

Figure 5.18: Speed-ups of GPU generation of assembly of global stiffness matrix and stress vector over serial code against problem size for CSR and dense matrices.

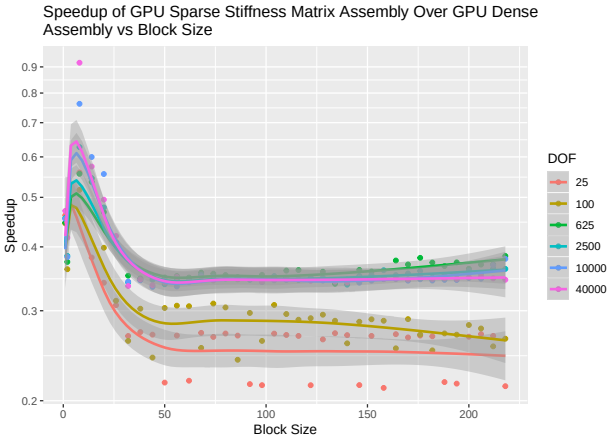


Figure 5.19: Speed-ups of GPU sparse global stiffness matrix assembly over dense assembly versus block size.

Like in the element matrices, choice of block size does in fact matter - both for sparse and dense assembly. Figures 5.20, 5.21 both show this. Like in the element matrices' device functions, there isn't a clear indication from the plots of thread divergence causing a large effect, but more so, oscillates slightly in and around multiples of 32 where the speed-ups increase. Again, also, reconfiguring the memory resulted in some positive results, barring overlaps in the smallest problem size. This could be down to the fact that all the threads in the block potentially not being used as the problem was not large enough.

Main Kernel Total

A downside to the timings for the two device functions is the lack of ability to use cudaEvents - which can only be used to time an kernel in its entirety. Instead, `clock64()` must to be utilised. This takes the clock ticks on each thread, which then must be written back to a global memory array, and then using cuBLAS, the max of this array calculated - giving the theoretical maximum time taken for the device function to finish. Unfortunately, in reality, that isn't how perfectly kernels and device functions run, as threads end up stalling while waiting for others and the kernel cannot complete until all the threads are finished. This is a bottleneck of CUDA programming and unavoidable. For example, when taking the cudaEvents time for the entire kernel, the irregular memory access pattern of writing these times into global memory must be taken into account as all the threads cannot finish the kernel until the entire array of clock cycles has been populated.

With that being said, for a more realistic speed-up time, the entire kernel was timed and these times are illustrated in Figures 5.22a, 5.22b. Even with the device timings being taken, the kernel still demonstrates faster execution times than in serial for both sparse and dense cases, reaching up to around 20× and 300×, respectively. These times can be effectively taken as a minimum speed-up as removing the clock cycle evaluation will increase these even further.

Overall then, considering that the only part of the entire process that was attempted to be parallelised took up around 0.4% of the entire computation time, the amount of speed-ups achieved in that small portion proved quite substantial.

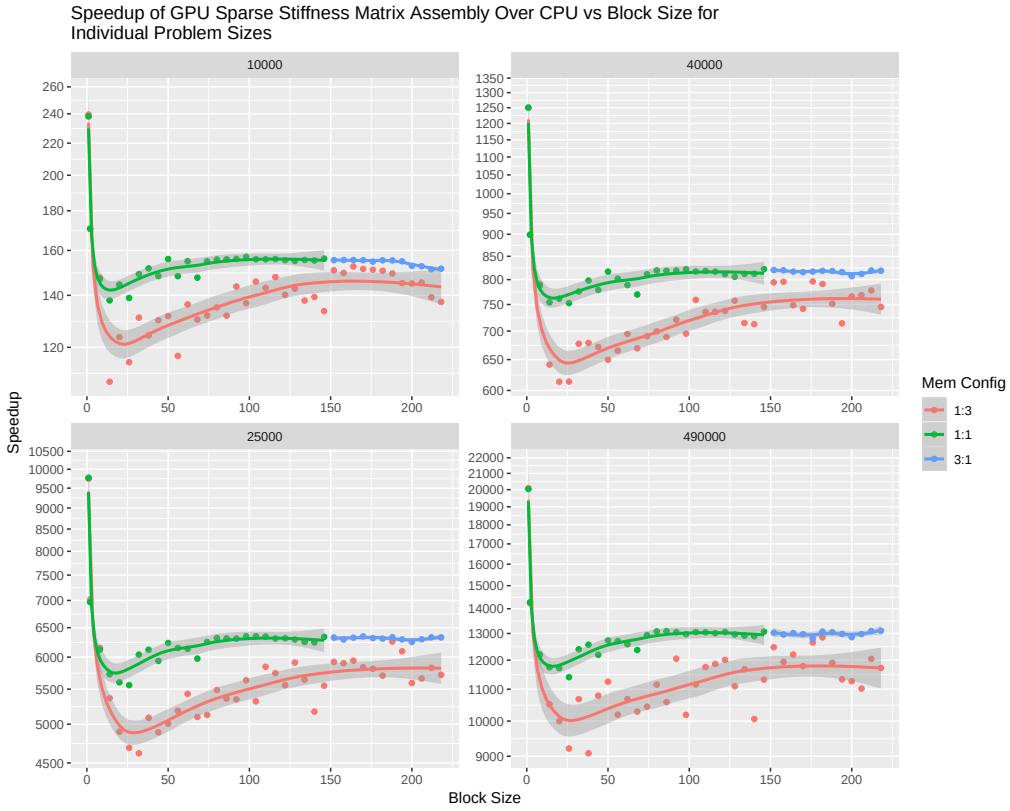


Figure 5.20: Speed-up of GPU assembly of sparse global stiffness matrix and stress vector over CPU versus block size for various problem sizes and memory configurations.

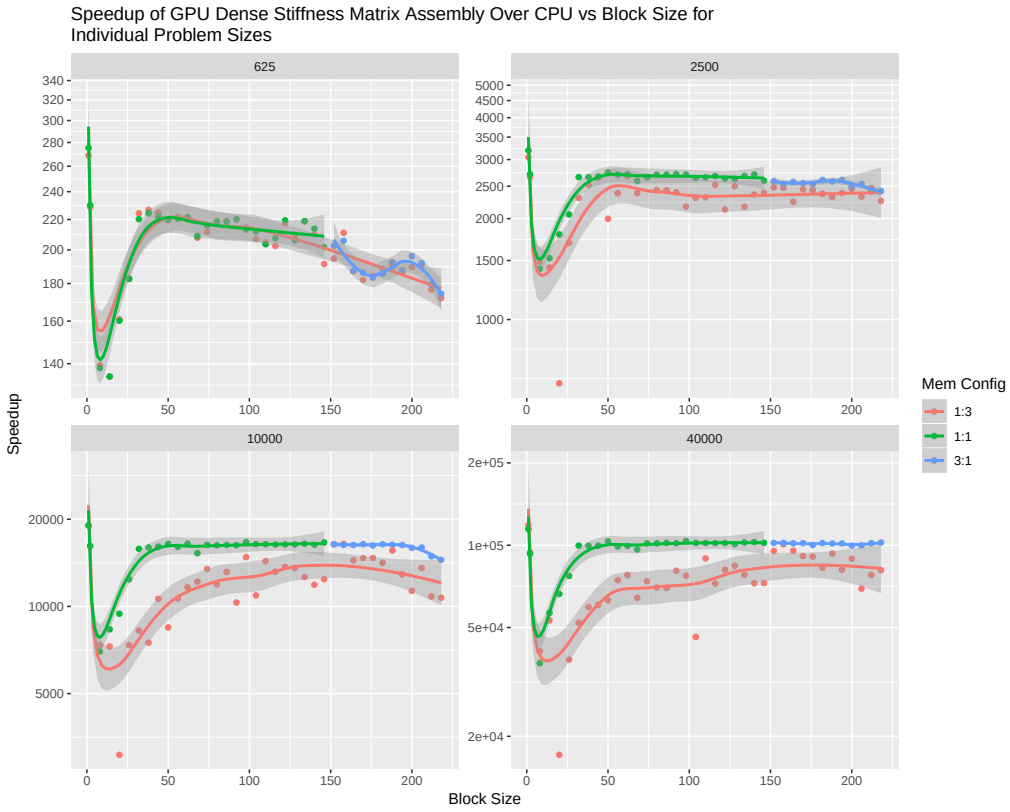
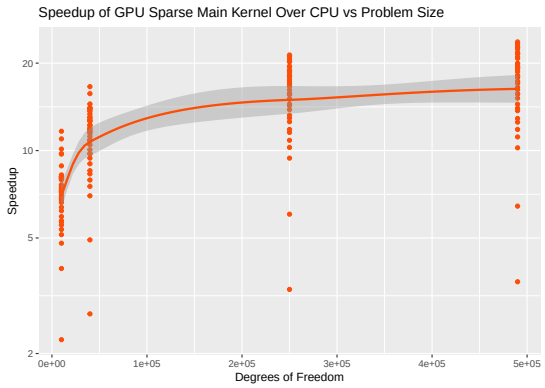
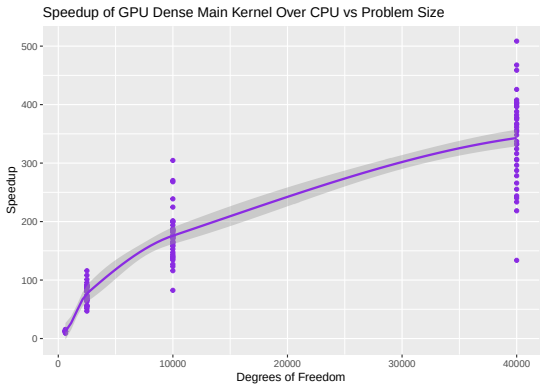


Figure 5.21: Speed-up of GPU assembly of dense global stiffness matrix and stress vector over CPU versus block size for various problem sizes and memory configurations.



(a) Speed-up of GPU main kernel over serial code for sparse case.



(b) Speed-up of GPU main kernel over serial code for sparse case.

Figure 5.22: Speed-ups of GPU main kernel over serial code against problem size for CSR and dense matrices.

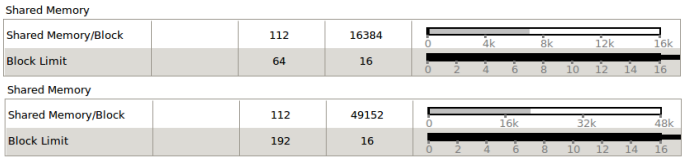
NVVP Profiling

For some extra profiling and analysis of the tool's performance, and deducing the levels of improvement seen by the various optimisations, Nvidia's Visual Profiler was used. Looking firstly at the memory usages, Figure 5.23, shows the shared memory occupation for four different block sizes, 1, 32, 100, and 175, with memory reconfiguration both on and off. The smallest block size of 1 shows an obvious massive waste in available shared memory. This small block size does obviously mean more space for thread registers. However, without after reconfiguration it can be seen that the shared memory available goes from 49,152 bytes to 16,384 - gifting the registers the difference. It would make little to no sense to not do this when only 112 bytes per block are being used. As the block size increase, more optimal used of shared memory can be seen, stretching it towards its upper limits. For block size of 175 it is seen to switch to a 1:1 split since the shared memory required is actually above the 16 Kb available in the 3:1 split.

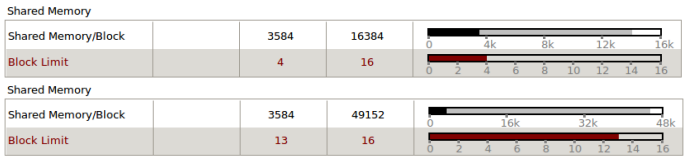
It was briefly touched upon that thread divergence is slightly visible from the plots earlier in this section, albeit only slightly. Figure 5.24 shows the thread divergence seen for three different block sizes 1, 32, 150 and 175 - all using the same problem size of course. It is evident here that multiples of 32 or even, the closer one is to multiples to 32, the less divergence is seen. The block size of 1 has the largest divergence with 150 and 175 more or less sharing 2nd place. With that being said though, the difference is rather minimal and the gain is only in fact a few percent.

A big factor in writing efficient parallel code is managing to balance the workload across the processing units as evenly as possibly. This seems entirely logical of course, as, if one thread has to do twice as much work as another, then the time taken will be slower than if they split 50:50. Figure 5.25 shows the load balance across the 12 SMs on the GTX 780 card for the main kernel in the standard GPU approach. The results are quite even, with none of the SMs appearing to drop below 95%.

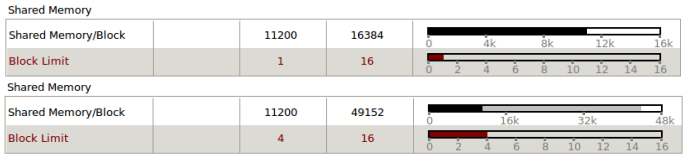
The final thing looked at in NVVP was the instruction latency. Deducing the cause of your main latencies in your code can be a very good indication for why your CUDA code



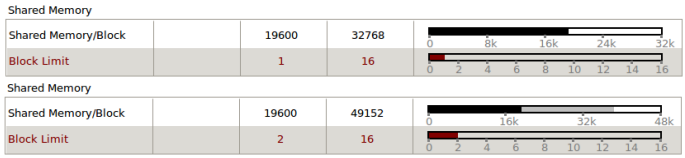
(a) Block size: 1



(b) Block size: 32



(c) Block size: 100



(d) Block size: 175

Figure 5.23: NVVP screenshots of shared memory occupation for four different block sizes, and with reconfiguration on and off.

Line / File	NA
NA	Divergence = 33.2% [44551 divergent executions out of 134101 total executions]
NA	Divergence = 39.9% [89253 divergent executions out of 223952 total executions]
NA	Divergence = 36.2% [89255 divergent executions out of 246452 total executions]
NA	Divergence = 25% [150 divergent executions out of 600 total executions]
NA	Divergence = 25% [150 divergent executions out of 600 total executions]

(a) Block size: 1

Line / File	NA
NA	Divergence = 3.8% [318 divergent executions out of 8406 total executions]
NA	Divergence = 17.9% [3275 divergent executions out of 18273 total executions]
NA	Divergence = 18.1% [3565 divergent executions out of 19662 total executions]

(b) Block size: 32

Line / File	NA
NA	Divergence = 23.3% [4809 divergent executions out of 20676 total executions]
NA	Divergence = 11.1% [1043 divergent executions out of 9411 total executions]
NA	Divergence = 20.9% [4496 divergent executions out of 21558 total executions]

(c) Block size: 150

Line / File	NA
NA	Divergence = 22% [4296 divergent executions out of 19512 total executions]
NA	Divergence = 19.8% [4010 divergent executions out of 20291 total executions]
NA	Divergence = 8.8% [769 divergent executions out of 8714 total executions]

(d) Block size: 175

Figure 5.24: NVVP screenshots of shared thread divergence for different block sizes.

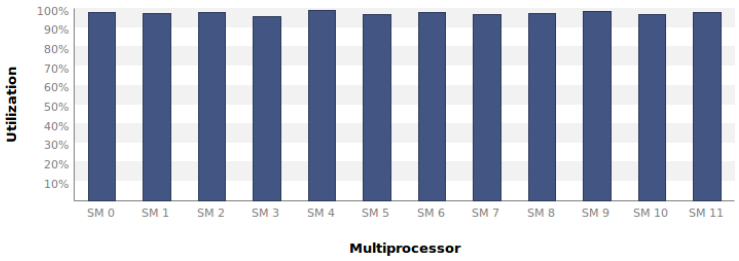


Figure 5.25: NVVP screenshot of load balance across SMs for main kernel of standard GPU approach.

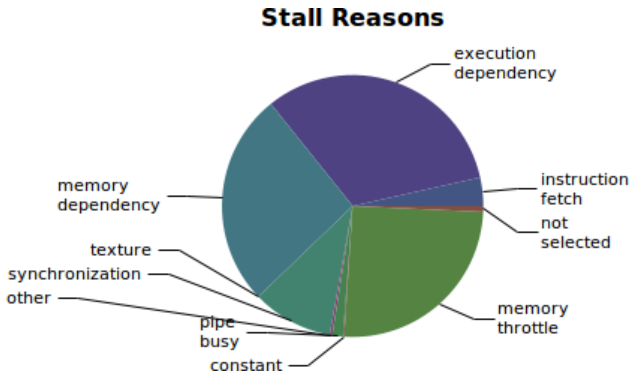


Figure 5.26: NVVP screenshot of latencies in main kernel of standard GPU approach.

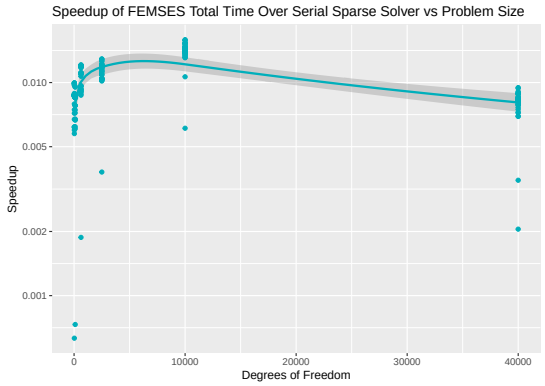
is not performing as optimally as you would have thought it to. Figure 5.26 shows the latencies in the main kernel for the standard GPU approach. These are mainly dominated by memory latencies. The explanation of each of these terms are explained in Appendix (REFERENCE).

5.3.2 FEMSES

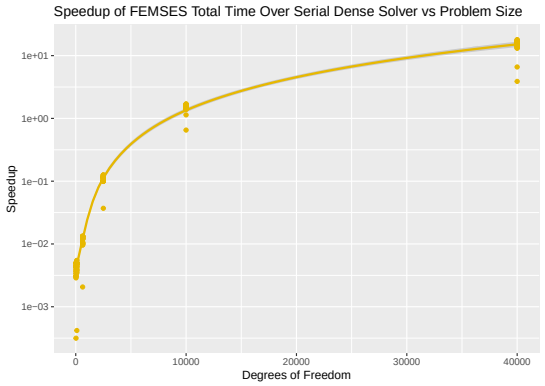
The FEMSES approach, also achieved some varying results in comparison to both the sparse and dense serial and standard GPU implementations. The main concern seen here, and also the main bottleneck was the slow rate of convergence. by comparison to the rates seen in Fernandez (CITE). This is address later in the section.

Total Timings

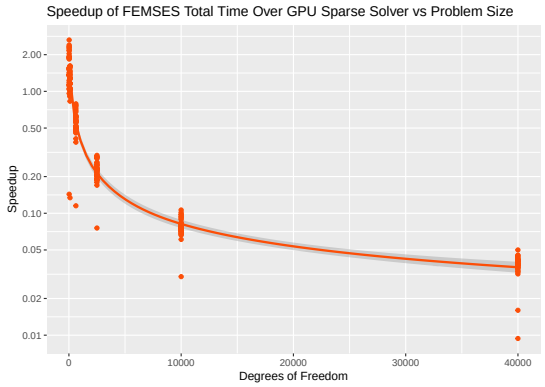
For the overall timings, the scaling trends were not wildly different to the ones seen in the other GPU implementation. Figure 5.27 illustrates these speed-ups against both serial and alternate GPU approaches. For the sparse serial case, in Figure 5.27a, the results are much the same as the other GPU approach where the parallel version is way outperformed by the serial due to Intel's DSS package. The scaling here even appears to decrease as problem size gets larger, after an initial increase. In Figure 5.27b, the story is much more positive, showing FEMSES to outperform the serial dense case at a logarithmic rate - reaching up to 10× as fast. For comparison against the standard GPU approach, the sparse case in Figure 5.27c shows quite poor scaling, initially outperforming the other approach, but as



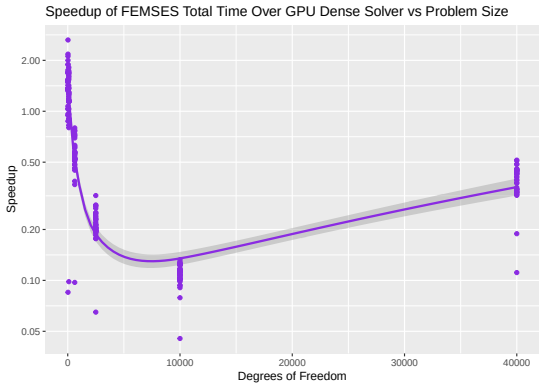
(a) Total time speed-up of FEMSES over serial sparse solution.



(b) Total time speed-up of FEMSES over serial dense solution.



(c) Total time speed-up of FEMSES over GPU sparse solution.



(d) Total time speed-up of FEMSES over GPU dense solution.

Figure 5.27: Speed-ups of total times over serial code and standard GPU code vs. problem size for FEMSES solution.

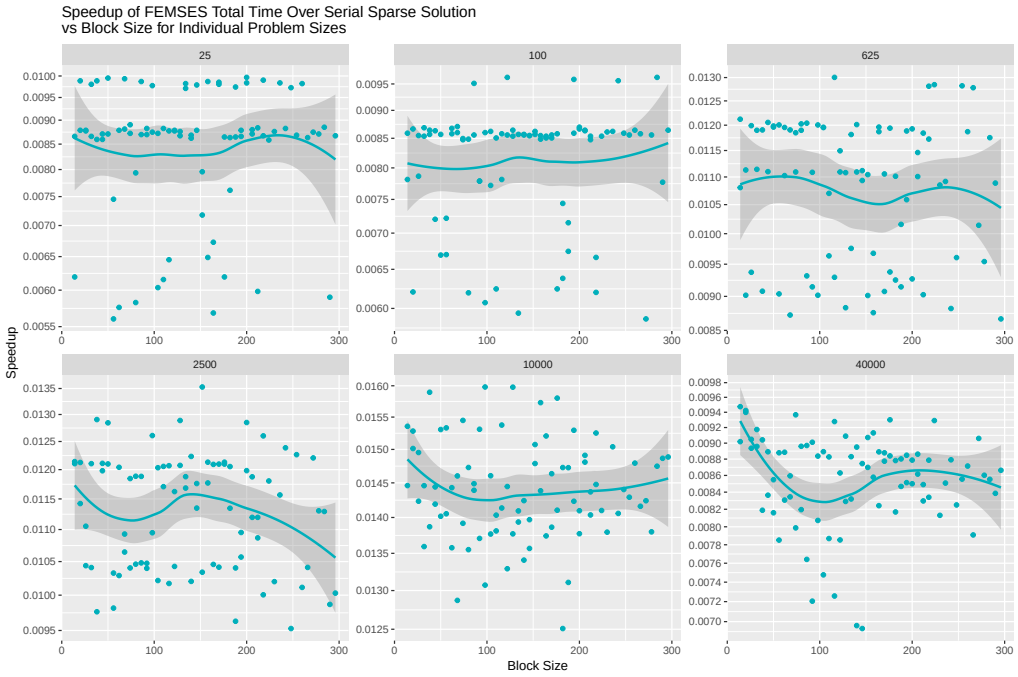


Figure 5.28: Speed-up of FEMSES total time over serial sparse case versus block size for various problem sizes.

the solver kernel becomes more dominant as the problem size increase, there is an evident exponential decay in speed-up, to the point of being orders slower. The comparison to the GPU dense case in Figure 5.27d shows much the same results but begins to improve again as the problem size increases. This is more than likely down to the fact that the linear solver has less of an impact for the smaller problem sizes, and becomes more influential as the problem size increases. FEMSES, should, ideally scale better than the dense direct solver if its convergence achieved is at a short rate. Unfortunately, further problem sizes for the dense case couldn't be tested to check if this increase in speed-up is an anomaly due to restrictions on problem size in Intel's LAPACK library.

Contrary to what was seen in the other GPU FEM implementation, there is a lot more variation in results as block size changes for FEMSES. This is to be expected since out of the four kernels in the process, three of them are dependent on the block size - the exception being cuBLAS to evaluate the error at each iteration. Figures 5.28, 5.29, illustrate this massive amount of variability. This is far more erratic then the timings seen in the evaluation of the element matrices and assembly device functions in the previous section - even the confidence intervals seen by the shaded grey area of the loess regression are far larger. There isn't much inference can be made from the erraticity of these results, there definitely isn't any clear indication of thread divergence. One thing that can be deduced from them is there isn't any evidence of slowdown due to overflow of registers. It was difficult to deduce accurately what the cause of the scattered results is without proper per-kernel profiling and timings. However, it is discussed in the next section why this wasn't practical.

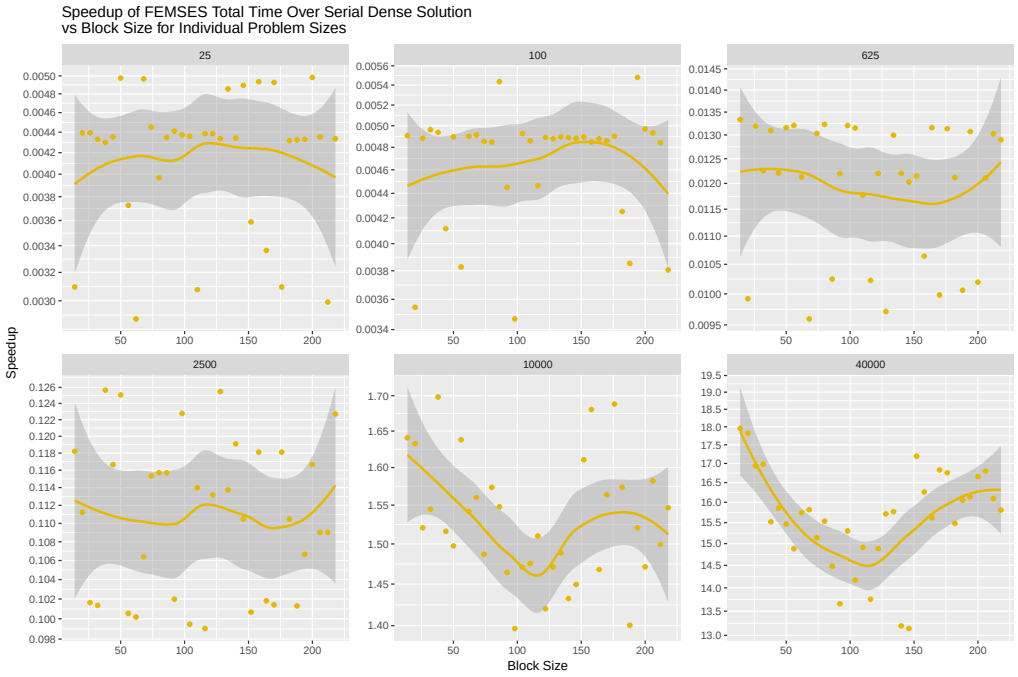


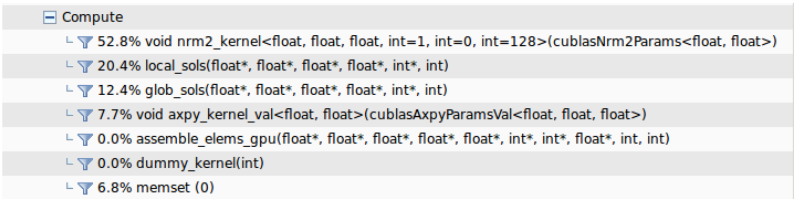
Figure 5.29: Speed-up of FEMSES total time over serial dense case versus block size for various problem sizes.

Solver

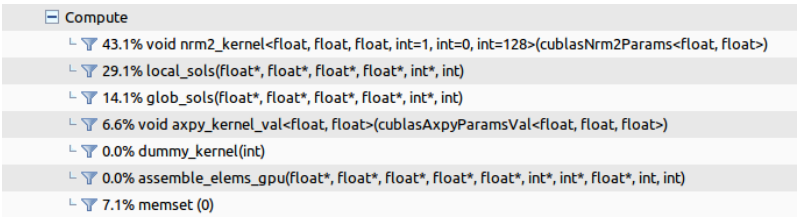
The FEMSES solver, like the linear system solvers in the previous sections, is the most dominant kernel in the process. It is, of course, divided up into three separate kernels for the GPU, the Jacobi iteration, the global solution assembly and the error evaluation. Unfortunately, to get any decent timings on these would involve adding cudaEvents to the code. These would be placed inside a while loop, which, while testing this in fact added up to 30% extra computation time and so didn't actually prove a practical solution for profiling. Instead, the total solver time was timed for comparison with the other solutions, and the three individual kernels were profiled using Nvidia's Visual Profiler. Figure 5.30 shows the proportion of computation time taken by each of the kernels in the FEMSES approach individually. For small problem sizes, the error calculation is the most dominant. However, as this increases, the Jacobi solver becomes the dominating kernel. Figure 5.31 shows the speed-ups over both serial and GPU solutions for the FEMSES solver. Again, due to the dominance of the solver step of the entire process, it has almost replicated the results seen in Figure 5.27. Unfortunately, the performance over the sparse cases leave a lot to be desired, but again, this is going to be primarily due to the unforeseen slower rate of convergence than anticipated.

Allocations & Transfers

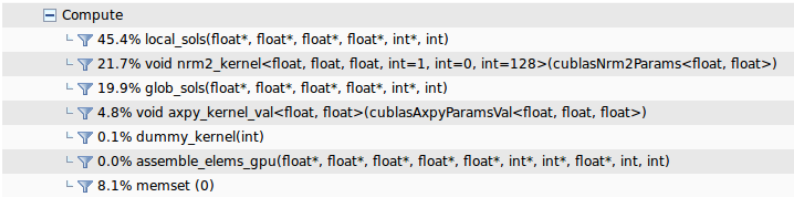
The allocation time for the FEMSES approach is shown in Figure 5.32. It demonstrates an expected linear scaling (barring one outlier within the 95% confidence allowance). The allocation here involves, as before, the five mesh arrays, as well as an array containing all the element matrices and the element vectors, as well as the weightings. All of these



(a) Degrees of Freedom size: 900



(b) Degrees of Freedom size: 2,500



(c) Degrees of Freedom size: 10,000

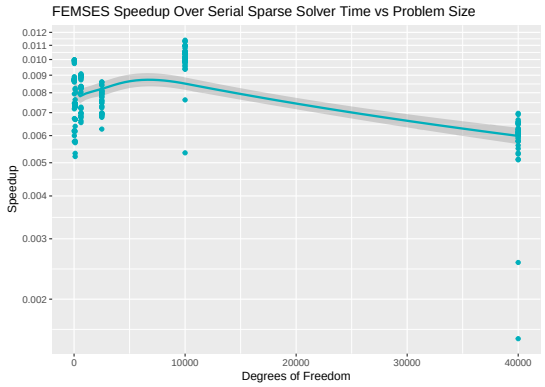
Figure 5.30: NVVP screenshots of proportion of computation time taken by each of the FEMSES kernels for different problem sizes.

arrays have linear scaling and there is no dense global stiffness matrix so the linear scaling was to be expected. Figure 5.33 shows the speed-up over the other two GPU implementations. FEMSES manages to take roughly the same amount of allocation time as the sparse approach at small problem sizes, but as this grows, the scaling of the CSR matrix scales better than the array of all the element matrices and element vectors along with the weighting vector - hence the evident decline as the degrees of freedom grows. The dense case, on the other hand, scales far worse than FEMSES - to be expected since there is no dense stiffness matrix allocated. There is a nice logarithmic scaling, reaching up to 7.5× speed-up.

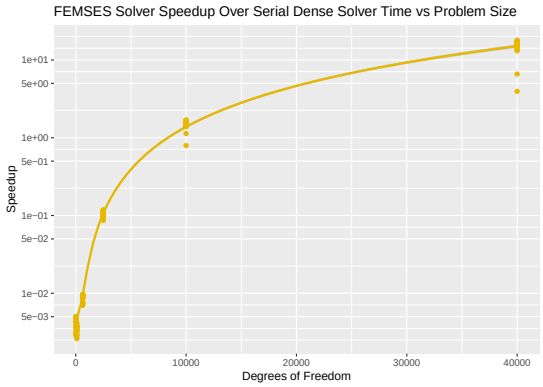
There was no transfer times taken for FEMSES as the amount of data required to be transferred is identical to that of the GPU dense approach - the mesh and solution vector.

Convergence

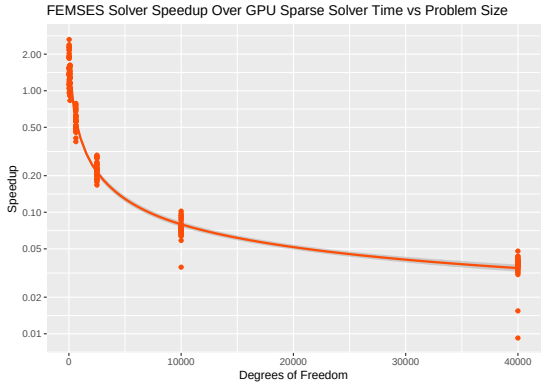
The poor rate of convergence was the main concern regarding the FEMSES implementation. Figure 5.34 illustrates the convergence as problem size increases and shows the super-linear rate which it achieves. When investigating why these results may be happening, one thing which was noticed, as the errors got smaller, they often reached orders of 10^{-5} and 10^{-6} and then proceeded to oscillate in and around these values, never reaching convergence within maximum iterations. The error then had to be truncated to 10^{-4} to avoid this. An inclination here was that the paper may have been using double precision and this was down to accuracy lost from utilising single precision values instead. This was tested and the results proved equal. The current proposal as to the reasoning for



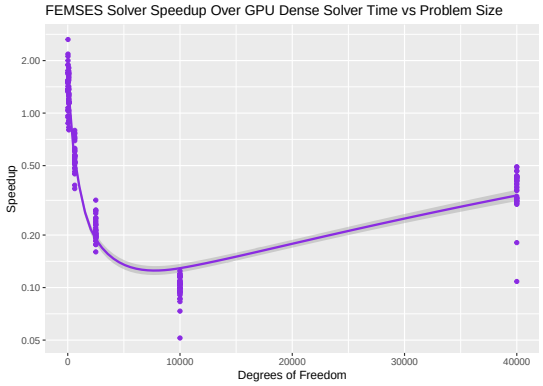
(a) FEMSES solver speed-up over Intel's MKL DSS.



(b) FEMSES solver speed-up over Intel's MKL LAPACKE dense solver.



(c) FEMSES solver speed-up over cuSOLVER's sparse solver.



(d) FEMSES solver speed-up over cuSOLVER's dense solver.

Figure 5.31: Speed-ups of FEMSES solver times over Intel's MKL and cuSOLVER versus problem size.

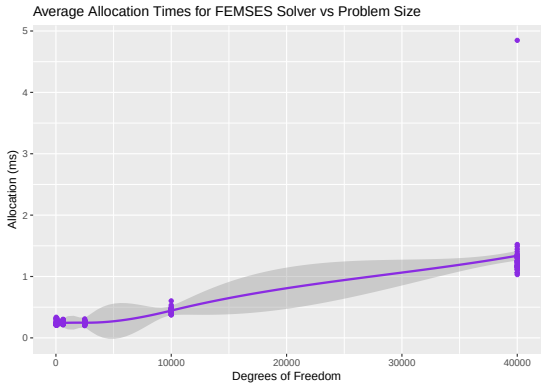
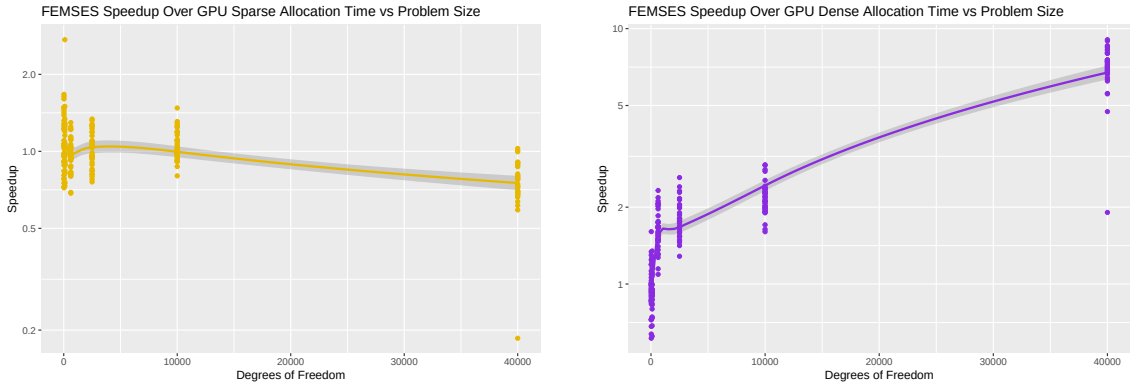


Figure 5.32: Allocation time for FEMSES versus problem size



(a) Speed-up over sparse case.

(b) Speed-up over dense case.

Figure 5.33: Speed-ups of FEMSES allocation time over GPU sparse & dense solutions.

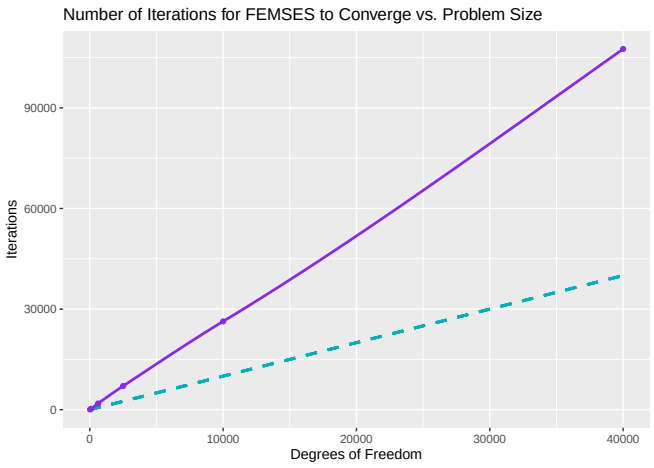
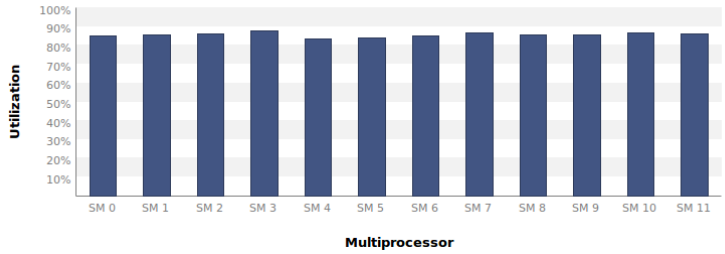
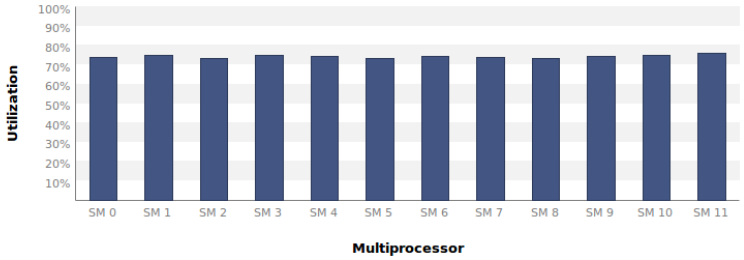


Figure 5.34: Number of iterations for convergence of the FEMSES approach versus problem size with abline of linear scale for reference.

Figure 5.35: NVVP screenshots showing the lack of thread divergence for the all FEMSES kernels.



(a) Load balancing for local solutions kernel.



(b) Load balancing for global solutions kernel.

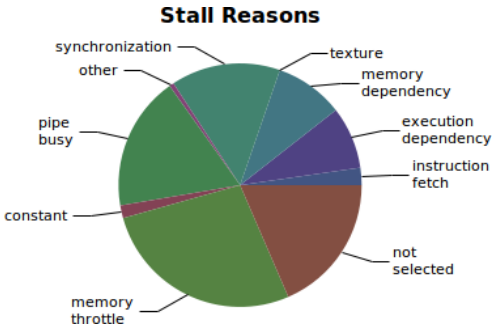
Figure 5.36: NVVP screenshots of shared load balance across SMs for two of the main kernels in the FEMSES approach.

these unexpected results is in Fernandez (CITE), the PDE used is a rectangular pipe, with non-homogeneous Dirichlet conditions on the outer boundaries and homogeneous Neumann on the bore hole, thus causing propagation from four directions. In the example in Section 2.5.1, the rectangular mesh has homogeneous Neumann conditions on the upper and lower edges and non-homogeneous Dirichlet conditions on the left and right boundary, therefore propagation will occur in only two directions, from left and right inwards. This theory has yet to be tested but would be an interesting result to deduce if this is indeed the root cause of the sub-par performance of FEMSES.

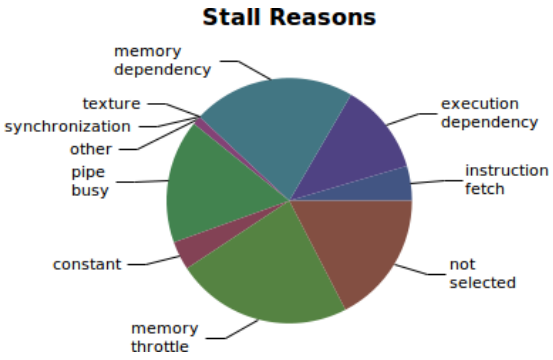
NVVP Profiling

Regarding warps, it was stated in this section that there really wasn't any clear indication of the presence of thread divergence in the FEMSES code. This was profiled with NVVP and Figure 5.35 shows the output from this. It was tested for various block sizes, multiples of 32 and not, and for all contributing kernels, and in all cases there was no resulting thread divergence indicated by the profiler. This would explain to a certain extent why the erratic graphs didn't show any pattern.

For load balancing the code, Figure 5.36 shows the distribution of workload across the SMs for two of the kernels written for FEMSES - cuBLAS nrm2 was ignored as it wasn't the job of this report to optimise it and the element matrices kernel as it was profiled



(a) Latencies for Jacobi iteration kernel.



(b) Latencies for global solutions kernel.

Figure 5.37: NVVP screenshots of computation latencies for two of the main kernels in the FEMSES approach.

in the standard GPU approach. Both kernels are quite load balanced, though the Jacobi iteration kernel does show a certain degree of lacking in compute utilisation.

Lastly, the stalling profiling. Figure 5.37 illustrates the latency reasons for both of the main kernels in the FEMSES approach - again, excluding the element matrices and error kernels. These latencies are quite evenly distributed across various reasons. Again, these are all explained in Appendix (REFERENCE).

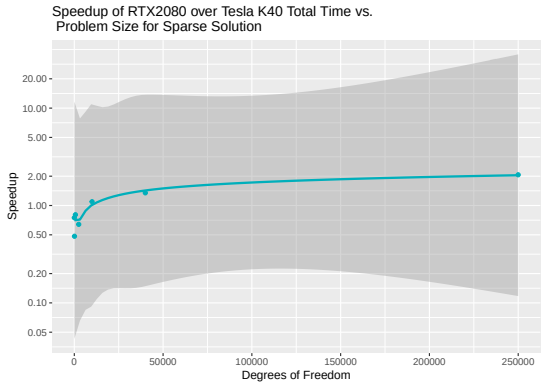
5.3.3 Comparison of GPU Architectures

Two different GPUs were used in testing for this paper. All the results seen in the previous section are all taken from the Tesla K40. The RTX 2080 Super was installed recently and due to time constraints didn't complete an entire test suite, but rather only completed tests for a single block size. The problem size was also limited to smaller values than the Tesla due to memory limitations of the RTX 2080 having smaller DRAM. In this section, the performance of the RTX 2080 Super is compared against the older Tesla K40.

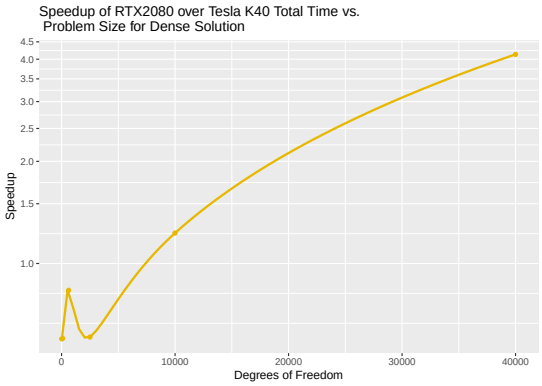
Total Timings & Solvers

The total times appeared to achieve mostly more positive results on the RTX 2080 than on the Tesla. The speed-ups over the Tesla versus problem size are illustrated in Figure 5.38. The sparse case demonstrates a nice stable speed-up of around 2×, while the dense case appears to experience some form of linear or possibly slightly logarithmic scaling, to a higher degree of up to 4×, roughly. FEMSES on the other hand, actually appeared to perform worse on the newer architecture. It does experience a positive scaling trend but never appears to actually outperform the older card.

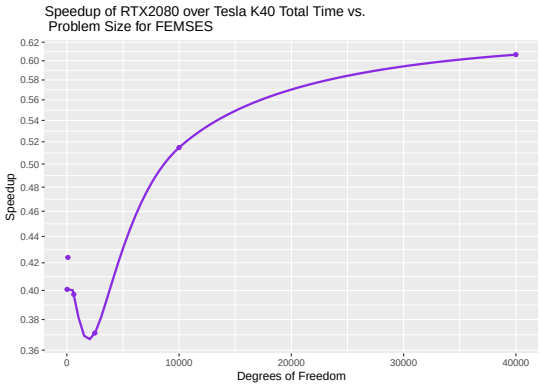
In Figure 5.39, the speed-ups are shown against problem size for the solver kernel in the process. Again, just like the results seen on the Tesla, the solver is evidently the dominant kernel in the method and the resulting graphs are near identical. These results do somewhat explain the poorer performance in the FEMSES approach than the other two methods. It is largely possibly that the cuSOLVER libraries have been optimised to work well with the newer Nvidia architectures and take advantage of whichever technologies



(a) Total time speed-up of RTX 2080 over Tesla K40 for sparse solution.

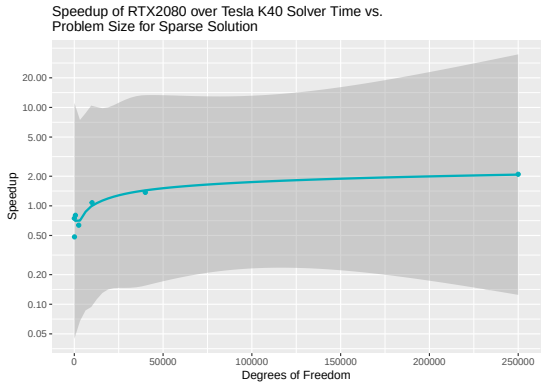


(b) Total time speed-up of RTX 2080 over Tesla K40 for dense solution.

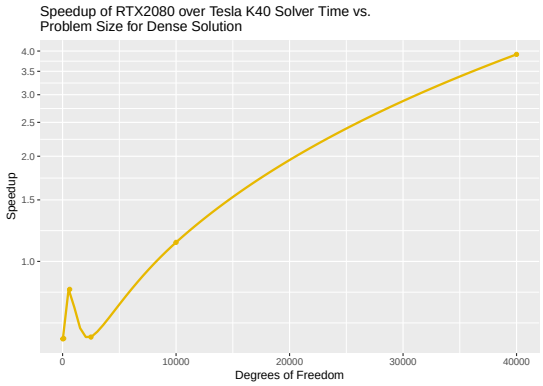


(c) Total time speed-up of RTX 2080 over Tesla K40 for FEMSES.

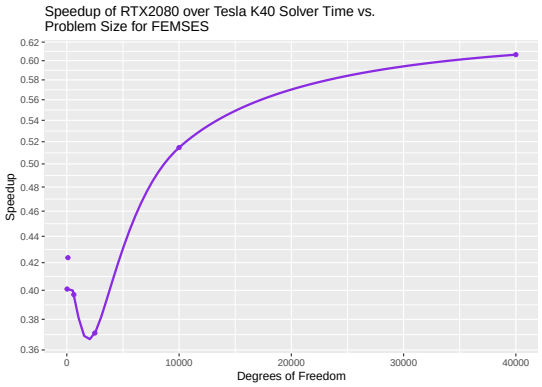
Figure 5.38: Speed-ups RTX 2080 over Tesla K40 of total times versus problem size for three main approaches.



(a) Solver time speed-up of RTX 2080 over Tesla K40 for sparse solution.

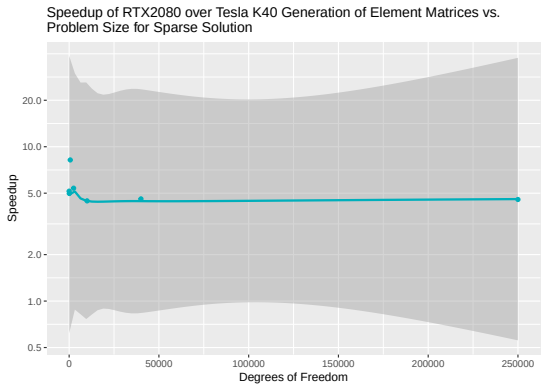


(b) Solver time speed-up of RTX 2080 over Tesla K40 for dense solution.

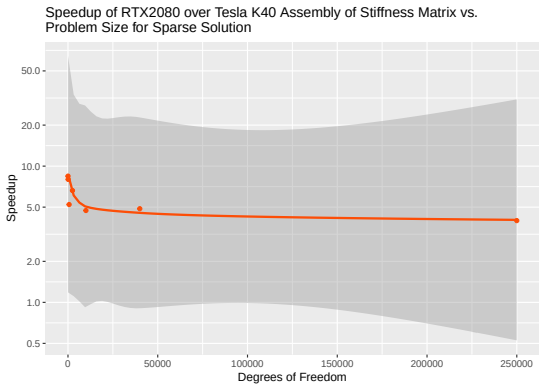


(c) Solver time speed-up of RTX 2080 over Tesla K40 for FEMSES.

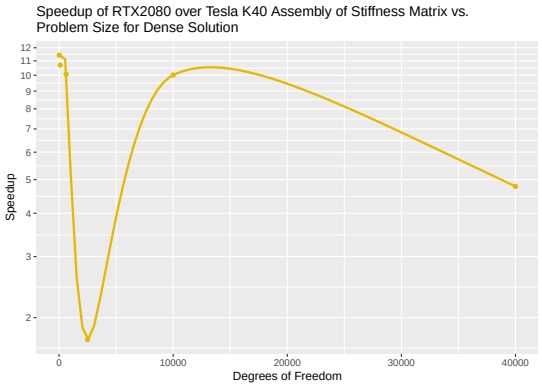
Figure 5.39: Speed-ups RTX 2080 over Tesla K40 of solver times versus problem size for three main approaches.



(a) Generation of element matrices time speed-up of RTX 2080 over Tesla K40.



(b) Assembly of global system time speed-up of RTX 2080 over Tesla K40 for sparse solution.



(c) Assembly of global system time speed-up of RTX 2080 over Tesla K40 for dense solution.

Figure 5.40: Speed-ups RTX 2080 over Tesla K40 of different kernel times versus problem size for three main approaches.

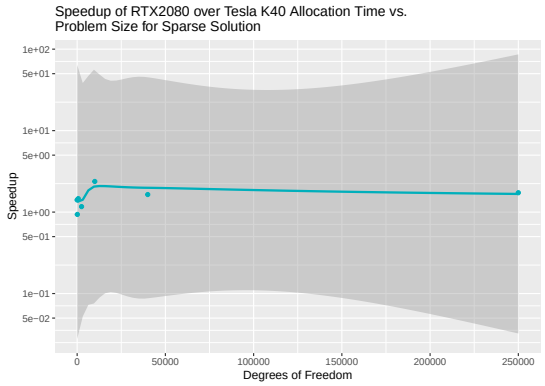
these may bring. However, the FEMSES approach has most of its code purpose written for this report and coded almost entirely on the Tesla and so, expectedly, performs quite well on this by comparison.

Generation of Element Matrices & Assembly of Global System

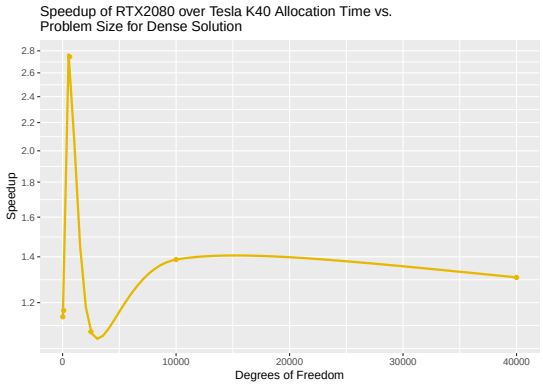
The speed-ups of time taken for the generation of element matrices and the assembly of both sparse and dense global linear systems are shown in Figure 5.5. Both the generation of the element matrices and the assembly of the sparse solution achieve relatively stable speed-ups of about 5×. Positive results in both instances. The dense assembly speed-up is a little harder to make inferences from. Given that the sample size for these tests is much smaller (evident from the large confidence intervals in the plots), there appears to be an outlier for problem sizes of 1000. Besides this, although the trend is negative, the dense assembly also showed positive speed-ups for various the number of unknowns tested.

Allocations & Transfers

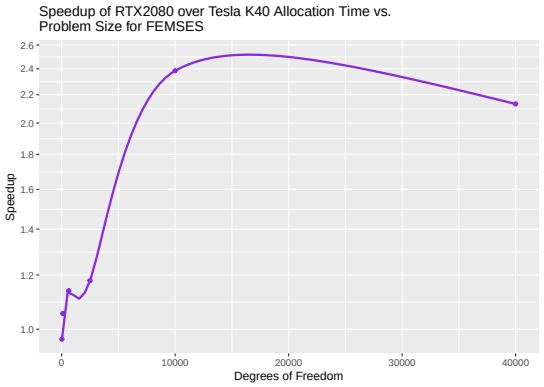
Figure 5.41 shows the speed-ups in allocation time for the three different approaches. The speed-up see for allocating the sparse matrices, again shows a nice stable speed-up of about



(a) Allocation time speed-up of RTX 2080 over Tesla K40 for sparse solution.

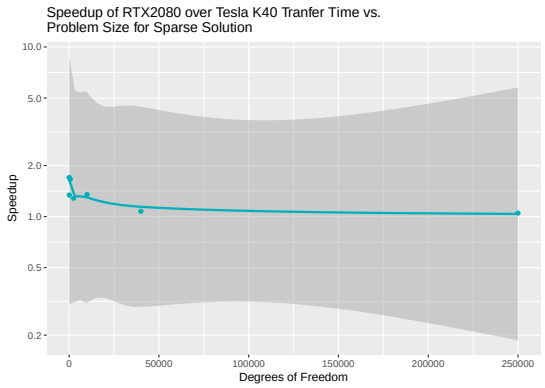


(b) Allocation time speed-up of RTX 2080 over Tesla K40 for dense solution.

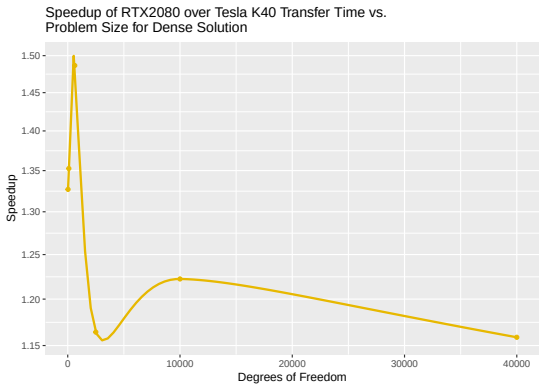


(c) Allocation time speed-up of RTX 2080 over Tesla K40 for FEMSES.

Figure 5.41: Speed-ups RTX 2080 over Tesla K40 of allocation times versus problem size for three main approaches.



(a) Transfer time speed-up of RTX 2080 over Tesla K40 for sparse solution.



(b) Transfer time speed-up of RTX 2080 over Tesla K40 for dense solution/FEMSES.

Figure 5.42: Speed-ups RTX 2080 over Tesla K40 of transfer times versus problem size for sparse and dense solutions.

2× as fast as the old card. The dense and FEMSES case both have more unusual trends. The dense case, ignoring a massive spike outlier, appears to also be quite stable at 1.4×, but there is an outlier nonetheless. For the FEMSES case the graph demonstrates what appears to be logarithmic scaling at a glance. Difficult to deduce why this is the case other than on the bad side of variance with clock speeds on the card while testing.

Again, similar to the allocation times, the transfer times in Figure 5.42, appeared quite stable for the CSR cases and then quite erratic for the dense case. This is also an unusual result and presumably down to the same issues seen in the other cases. This would make sense as the problem size causing the spikes appears to be the same for all the outliers. Besides this, however, again positive speed-ups are obeserved.

All considered then, it's clear that the newer architecture, overall, outperformed the Tesla. It would be interesting to investigate further the potential speed-ups that could be achieved with this new card given more time for testing.

Chapter 6

Conclusion

6.1 Final Remarks

6.2 Future Work

6.3 Recommendations & Limitations

Appendix A

Appendix

...