GPU Acceleration of Partial Differential Equation Solvers

GPU Acceleration of Partial Differential Equation Solvers

Panagiotis Iosifidis NUMA Engineering Services Ltd.
Phil Weir NUMA Engineering Services Ltd.
Panchacharam Mariappan NUMA Engineering Services Ltd.
Ronan Flanagan NUMA Engineering Services Ltd.
Ivor Spence Queen's University of Belfast
Peter Kilpatrick Queen's University of Belfast
Jim Fitzimons FUSION Intertrade Ireland

Abstract

Differential equations are often directly solvable by analytical means only in their 1-D version. Partial differential equations are generally not solvable by analytical means in 2-D and 3-D, with the exception of few special cases. In all other cases, numerical approximation methods need to be utilized. One of the most popular such methods is the finite element method. Our main areas of focus are the Poisson heat equation and the plate bending equation. The purpose of the current paper is to provide a quick walk through of the various approaches we followed in pursuit of creating optimal solvers, accelerated with the use of graphical processing units, and comparing them in terms of accuracy and time efficiency with existing or self-made non-accelerated solvers.

Keywords: GPU acceleration, finite element, PDE, Poisson, plates, plate bending.

1 Introduction

Graphics processing units (GPUs), originally designed for efficient 3-D computer visualization, contain highly optimised hardware capable of solving large scale mathematical problems, such as those seen in aerospace engineering, mechanical engineering and biomedical research. The utilization of this GPU quality, if properly applied, provides a remarkable improvement in a programs performance in terms of speed. This process is known as GPU acceleration [1].

Our current research was focused on partial differential equations (PDEs). We developed a series of PDE solver codes implementing a solution approximation method

known as the finite element method (FEM).

GPU acceleration requires our existing algorithms to be adapted to the new architecture in an intelligent and considered way. The purpose of the current article is to present the latest developments in the field of GPU accelerated, FEM implementing PDE solvers.

After a short introduction to PDEs and FEM, we provide a description of FEM solvers and their three main parts, data reading, assembly, and solution. We dedicate an additional section to provide the reader with the concept of GPU acceleration.

We then focus on our main contribution, the efficient implementation of the "Cotangent Method" on many-core devices, such as GPUs, for the assembly part of the solver code, and explain in what way it reduces the computation time, while securing precision and boosting memory economy.

Afterwards, we provide a walk through of the various solver codes we have developed, as well as the precision and time-efficiency comparisons we carried out, both between self made accelerated and unaccelerated codes, as well as between our codes and comparable commercial and open source software.

We finalize the document with an overview of the engineering applications and academic benefits of our research as well as our future pursuits on the fields of PDEs and FEM.

All experiments described in the current paper were executed on a computer consisting of an Intel Core i7-3770K CPU @ 3.50GHz x 8 processor mounted with an NVidia GeForce GTX 680 graphics card, consisting of 1536 cores, equiped with 2 GB of global device memory.

2 PDEs and FEM

PDEs are equations involving of an unknown function u, its derivatives. PDEs are normally unsolvable by analytical means, save for a few standard exceptions. Therefore, numerical methods are employed in an attempt to approximate the unknown function. The method of focus in our current research was FEM [2].

The finite element method begins with an unknown function u over a domain Ω whose values are known along the boundary of the domain. The domain is then divided into a number of triangles (in 2-D), or tetrahedra (in 3-D), called elements. The vertices of the elements are termed nodes. The purpose of FEM is to calculate the values of the unknown function at the nodes [3].

There are a variety of FEM solvers on the market, both commercial and open source. The vast majority of them function under the same principles, which reduce the problem into solving a linear system. A generic code consists of three main steps: data reading, assembly and system solver. CPU reads and stores the input data in a set of matrices. In the assembly, the code uses those matrices to form the linear system. In the system solver, the code solves the linear system [4].

Our main contribution was focused on applying a highly parallelizable adaptation on the finite element assembly algorithm [5] to the solution of the Poisson heat equation on the GPU. This assembly method effectively bypasses the traditional shape function methods, thus avoiding complicated, time-consuming integrations.

This method, combined with a comprehensive optimization has provided a huge speed boost to the assembly part of the code, considerably lowering the assembly time from over 8 s to below 10^{-3} s.

3 GPU acceleration

A typical GPU consists of up to 16 streaming multiprocessors. Each streaming multiprocessor can have up to 48 active warps. A warp is a group of 32 threads to perform calculations in SIMD (Single Instruction Multiple Data) fashion. Hence every streaming multiprocessor can perform up to 1536 calculations simultaneously. This architecture gives the GPU the capacity to perform up to 24576 calculations in parallel [6].

Over the past few years, GPU use has become an increasingly important part of engineering mathematics, providing significant and inexpensive speed-ups to industry scale FEM computations. Its relevance to FEM algorithms is thus clear. As explained, FEM relies heavily on efficiency dividing a domain into many smaller pieces and performing a series of computational tasks in every piece separately [1, 3].

Unaccelerated programming would have the CPU processor scan through all the elements one by one and perform the appointed tasks on each element separately. Parallel programming assigns a few elements to each thread and the appointed tasks are performed simultaneously. This way the overall workload is accomplished multiple times faster.

Linear system solvers, are also another region where parallel programming can be utilized. Most of the iterative system solvers, such as Gauss Seidel and Conjugate Gradient, require that the same set of calculations is performed for every line of the system. Appointing one line (or one set of lines) to each thread of the GPU can provide a considerable speedup to the solution process [7].

Nonetheless, the speedup is not linear. Various delaying factors such as race conditions and bank conflicts limit the effects of acceleration. These occur when different threads are trying to access the same memory location. In this case, threads are queued, with one accessing the location and the rest waiting for their turn. Such incidents render optimization attempts imperative in order to produce an efficiently performing software [8].

4 Formulation

4.1 Poisson heat equation

The Poisson heat equation is as follows: We have an unknown function $\phi(x, y)$ and a known function $f(x, y)$ defined over a domain Ω, related by the following equation (1) and (2)

$$\frac{\partial^2 \phi}{\partial x^2} + \frac{\partial^2 \phi}{\partial y^2} = f(x, y) \text{ on } \Omega \tag{1}$$

In its three-dimensional version, the equation becomes:

$$\frac{\partial^2 \phi}{\partial x^2} + \frac{\partial^2 \phi}{\partial y^2} + \frac{\partial^2 \phi}{\partial z^2} = f(x, y, z) \text{ on } \Omega \tag{2}$$

The value of ϕ is known throughout the boundary of the domain. While the equation is commonly used to describe thermal transfer processes, its usability extends beyond heat diffusion problems [9].

The aim is to calculate the unknown function ϕ. Apart from a few special cases, the above equation cannot be resolved analytically. In solving the above problem, the finite element method works as follows: The domain Ω is discretized, that is, it is divided into a set of non-overlapping shapes called elements. In the 2-D case these elements are more often triangles or quadrilaterals, while in the 3-D case tetrahedra are often employed. The vertices of the elements are called nodes.

The true solution may then be approximated by a finite-dimensional, piecewise polynomial function, smooth except at edges of the elements. Following the steps of the FEM algorithm, the PDE problem promptly reduces to solving a linear equation system, with the number of equations and the number of unknowns equal to the number of nodes in the domain [3].

Evidently, a more detailed triangulation results in bigger, more complicated final linear system. Thus the system's solution becomes a harder and more time-consuming process. Nonetheless, the solution also becomes more accurate approximation of the unknown function, provided that the solver's precision is adequate.

4.1.1 Solvers and methodology

In storage, the mesh is commonly represented textually as a series of nodes, indexed 2-D or 3-D coordinates, and elements, indexed tuples of node indices.

The first phase of most solver codes consists of reading the mesh file and appropriately storing the data in the CPU memory. For that purpose, in our case, we used a matrix of three (2-D case) or four columns (3-D case). This matrix is called the domain matrix. The number of lines is equal to the number of nodes in the mesh. The elements are also stored in the elements matrix which consists of a number of lines, equal to the number of elements in the mesh, and columns depending on the type of element: four for triangle and five for quadrilateral or tetrahedral.

The second phase of most solver codes consists of reducing the PDE problem to a linear system. The linear system has the form,

$$K\Phi = F, \tag{3}$$

where n is equal to the number of internal (non-boundary) nodes in the mesh, K is an $n \times n$ symmetric square matrix, and Φ and F are column vectors of n entries. The fact that the size of K rises quadratically with the number of nodes, combined with its sparse nature, rendered compact indexing methods imperative for the viability of the code in terms of memory economy.

The third part of most solver codes consists of solving the linear systems. The amount of solver methods, algorithms and libraries available is limitless. Nonetheless, we chose to create our own codes for the purpose of better oversight and to allow the use of data structures suitable for the GPU optimization of the entire FEM algorithm. We experimented with a variety of algorithms, including LU decomposition and the Gauss-Seidel method, and concluded that the preconditioned Conjugate Gradient approach was the most efficient way to address our needs.

4.1.2 The cotangent method

Of the three matrices appearing in Equation 3, Φ is the unknown vector to be calculated and F is provided in the formulation of the problem. For many applications, F may be evaluated be point-wise at nodes and so used with little or no modification. The focus of the assembly process is then the matrix K, known as the global stiffness matrix. Its elements are known as stiffnesses. Hence the number in the $(2, 3)$ position represents the stiffness between nodes 2 and 3, while the number in the $(3, 3)$ position represents the self-stiffness of node 3.

The assembly of the global stiffness matrix takes place element by element. The global stiffness matrix begins as an empty matrix. If two nodes (say 2 and 3) belong to a specific element (triangle, quadrilateral, tetrahedron) a number is calculated and added to the respective position (2, 3) of the global stiffness matrix. Positions of the global stiffness matrix that correspond to non-neighbouring nodes (nodes which do not belong to the same element) shall remain zero.

The calculation of the local stiffnesses by traditional algorithms is a difficult and time consuming procedure that requires calculation of complicated integrals. This process takes a toll both on time as well as memory space.

However, when our solution is constrained to be piecewise linear, this is not necessarily the most effective approach. Our main contribution lies in applying a different and more direct method for the assembly process, applicable to piecewise linear approximations over triangles and tetrahedra, and much better suited to many-core architectures. The method is explained in detail in [3] for the 2-D case and [10] for the 3-D case.

By employing cotangent method, we avoid explicit numerical integration, while constructing the local stiffness matrices from shape functions. In this method, each

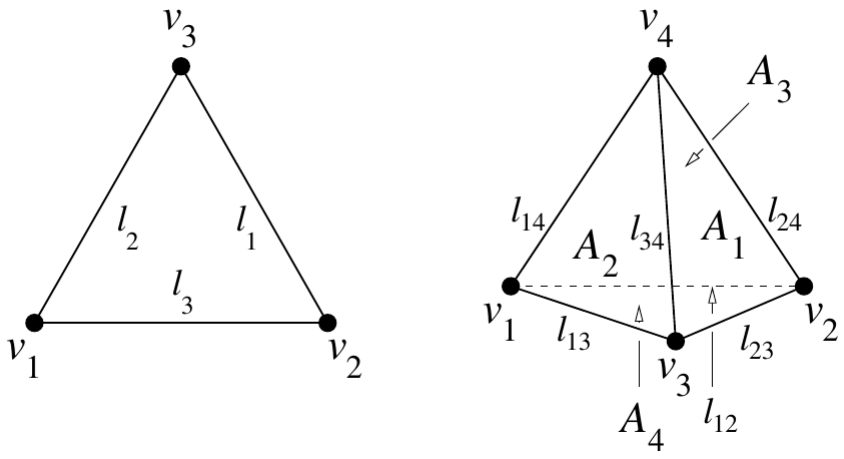

Figure 1: Calculation of the cross stiffness in a triangular and tetrahedral element. In the triangular case, the stiffness between nodes v_1 and v_2 is calculated based on the cotangent of node v_3. In the tetrahedral case, the stiffness between the nodes v_1 and v_2 is calculated as the angle between the planes $A2$ and $A1$, which are the planes that do NOT contain the edge l_{12} [10]

element's contribution to the stiffness between nodes 1 and 2 is easily measured, in 2-D, by calculating the cotangent of the planar angle (θ_3) formed by the two edges not $\vec{12}$, and, in 3-D, the cotangent of the angle formed by the two faces of the tetrahedral element not including an edge $\vec{12}$. In general, θ_i is the angle at vertex v_i of a triangle and θ_{ij} is the dihedral angle at the edges connecting vertices v_i and v_j of a tetrahedron. For that purpose, in this document, this method will be referred to as the *Cotangent Method*. For the 2-D Poisson heat problem, the local (per element) contributions to the global (whole domain) stiffness matrix may be expressed, using the cotangent method, as follows: [3]

$$\begin{pmatrix} \frac{1}{2}(cot\theta_2 + cot\theta_3) & -\frac{1}{2}cot\theta_3 & -\frac{1}{2}cot\theta_2 \\ -\frac{1}{2}cot\theta_3 & \frac{1}{2}(cot\theta_1 + cot\theta_3) & -\frac{1}{2}cot\theta_1 \\ -\frac{1}{2}cot\theta_2 & -\frac{1}{2}cot\theta_1 & \frac{1}{2}(cot\theta_1 + cot\theta_2) \end{pmatrix} \tag{4}$$

For the 3-D Poisson heat problem, the cotangent method provides the following local contribution to the global stiffness matrix, [10] The implementation of the cotangent method, combined with aggressive parallelization of the assembly code, resulted in a significant boost of the time efficiency of the assembly part of the solution process. From that point a variety of approaches has been implemented for the resolution of the linear system.

$$\begin{pmatrix} \sum\limits_{1\neq i<j} l_{ij}cot\theta_{ij} & -l_{34}cot\theta_{34} & -l_{24}cot\theta_{24} & -l_{23}cot\theta_{23} \\ -l_{34}cot\theta_{34} & \sum\limits_{2\neq i<j\neq 2} l_{ij}cot\theta_{ij} & -l_{14}cot\theta_{14} & -l_{13}cot\theta_{13} \\ -l_{24}cot\theta_{24} & -l_{14}cot\theta_{14} & \sum\limits_{3\neq i<j\neq 3} l_{ij}cot\theta_{ij} & -l_{12}cot\theta_{12} \\ -l_{23}cot\theta_{23} & -l_{13}cot\theta_{13} & -l_{12}cot\theta_{12} & \sum\limits_{i<j\neq 4} l_{ij}cot\theta_{ij} \end{pmatrix} \tag{5}$$

5 Implementations

5.1 2-D LU decomposition solver

The first solver was intended to evaluate the applicability of the cotangent Method. The solution to the linear system, is derived as $\Phi = K^{-1} * F$. In order to verify the solver's validity and precision we solved a series of problems with known answers. To provide a test problem, we began with a known solution,

$$\phi(x, y) = (x - 10)(x - 20)(y - 10)(y - 20) \text{ on } [10, 20] \times [10, 20] \tag{6}$$

In the first solver, the above system was solved through LU decomposition. This method was easy and direct and provided up to 7-digit precision in small domains. Nonetheless this method is not particularly parallelizable, as every step of the process requires input from all the previous steps. Hence every part fo the algorithm requires to wait for the previous parts to finish [11].

Nonetheless, the biggest weakness demonstrated itself with bigger domain sizes. Since each step in the decomposition required input from the previous ones, that meant that inaccuracies derived from small numerical or computational errors would carry on to the next step. As the system gets bigger, more and more errors accrue in the following rows. For big enough domains, the final result is susceptible to become grossly irrelevant to the real solution.

5.2 2-D Gauss Seidel Solver

The second solver consisted of an attempt to fully parallelize the code. Both CUDA and OpenCL were used. While academia generally praises OpenCL's parallel programming capacities, which expand beyond just Nvidia GPUs, scholars are more in conflict when it comes to determining which framework performs better on Nvidia hardware. In our case, CUDA was the base software for most of our work, mainly due to its user-friendly environment.

The code was parallelized on two levels, assembly and solution. The assembly parallelization was a per-element one. The number of GPU threads utilized was set equal to the number of elements in the mesh. It begins by setting all entries of the global

stiffness matrix equal to zero. The global stiffness matrix must have adequate entry spaces for every node's self-stiffness and cross-stiffnesses. Afterwards, each element is studied separatelly and in parallel. For each of the three nodes of the element, the self-stiffness and cross-stiffnesses are calculated according our stated directives [3]. The results are added to the global stiffness matrix K. The addition takes place with the use of atomic functions, in order to ensure that all stiffnesses are properly added and no number is omitted.

The solution parallelization was per node. The number of GPU threads utilized is equal to the number of nodes in the mesh. Each entry in the solution matrix represents the value of the unknown function in the corresponding node. All degrees of freedom are set to zero as initial values. At each iteration, the values are modified according to the equation (7) as explained in [12].

$$x_i^{(n+1)} = \frac{1}{k_{ii}}[b_i - \sum_{i \neq j} k_{ij} x_j^{(n)}] \tag{7}$$

or the variant :

$$x_i^{(n+1)} = \frac{1}{k_{ii}}[b_i - \sum_{i<j} k_{ij} x_j^{(n)} - \sum_{i>j} k_{ij} x_j^{(n+1)}] \tag{8}$$

Both variants update the values of solution x_i, based on the current or previous values of the rest of the solutions. For that purpose, a proper coordination between the GPU threads was necessary. An improper coordination would still managed to bear near results with over 99.999 accuracy for smaller mesh sizes. It must be noted that the accuracy is considerably higher than that of the LU decomposition solver case. The main reason is that, while the LU decomposition algorithm is sensitive to rounding errors, which transfer themselves to each new calculation, the Gauss Seidel algorithm functions function on the principle that errors exist. In fact, the basic principle is that the initial input is iteratively updated toward the solution. This gives the algorithm an important boost to error tolerance.[7].

However, certain issues emerged, stemming from the requirements of the algorithm and the capabilities of the GPU. While the algorithm is not too strict on the updating between the solutions (hence the two variants), it does require a minimum degree of synchronization between the nodes. This prerequisite clashes with the GPU capacities. In large meshes, the number of concurrently active threads is considerably smaller than the number of nodes. Each thread, calculates the solution in a corresponding node. Nonetheless, each active thread carries out all iterations for a particular node, before it can move on the the next node. This means that a specific solution will finish updating, before another can even start. This renders the code useless as it ends up bearing totally irrelevant results.

There have been many solutions to this, all of them with a natural speed burden on the codes efficiency. First and simplest idea was to include one only iteration in each kernel call. This method proved highly efficient in terms of precision, but at a great time cost. The main reason is that each kernel call contains a series of commands that prepare the iteration. These include identifying the neighbourhood and

the corresponding stiffnesses of each node. Since these commands are to be recalled in each kernel call, such a repetition results in above triple solution time. A middle solution resides in practising an average number of 10-20 iterations per kernel call. That way threads have the chance to synchronize as well as update their solutions in a viable manner.

5.3 3-D Gauss Seidel solver

Having met an adequate level of success with the 2-D Gauss Seidel Solver we moved on to the 3-D case. The main issue encountered with the 3-D case was that there was not adequate academic material providing an efficient and parallelizable assembly algorithm. Fortunately, the research of Jonathan Richard Shewchuk [10] provided us with the 3-D counterpart of the cotangent method. This method provided with an efficient and easily calculatable assembly for the stiffness matrix.

Furthermore, we focused on making a code that would emulate and compete the behaviour of its open source and commercial counterparts. For that purpose, we implemented a subroutine that reads a common .msh file. That is the file type that professional and open source software such as FEniCS and VTK . File reading cannot be performed in parallel by the GPU. Nonetheless, attempts to implement multi-core CPU parallelization brought an up to four times speed-up in the file reading process.

Concerning the assembly, a more challenging indexing method had to be implemented, in order to account for the bigger size of data. In 2-D each node has on average 6-10 neighbours. In 3-D this number rises up to 27-30 neighbours. An indexing that would guarantee maximal memory economy was therefore imperative. We managed to create such an indexing patterns of progressive quality and tested the code for the first time versus existing competitive software. Our software of choise was FENICS. Its wide usability combined with the positive feedback made the challenge appealing. The results of the comparison were highly encouraging.

In a domain of over half million nodes and over 2.8 million elements, our code brought results in under 30 seconds, while the unaccelerated FEniCS solver required 8 minutes and 37 seconds in order to bring results under the same precision rating for the same solver algorithm (Gauss Seidel).

5.4 Conjugate gradient solver

The 3-D Gauss Seidel solver provided the first evidence of the performance benefits of parallel programming but the contrast was not especially marked. The third-party reference solver was configured to use the same linear solver as ours and so its performance was not necessarily reflective of the maximum attainable through a suitably chosen configuration. What we needed was a code, that would compete and outperform available commercial and open source software at its maximum capacity.

Given that the linear system solution part was the most time consuming part of the code, we decided to pursue more efficient system solvers. The conjugate method,

as an efficient, parallelizable, iterative Krylov solver was appealing. The standard algorithm may be stated [5] as follows:

$$
\begin{aligned}
d_{(0)} &= r_{(0)} = F \\
\alpha_{(i)} &= \frac{r_{(i)}^T r_{(i)}}{d_{(i)}^T K d_{(i)}} \\
x_{(i+1)} &= x_{(i)} + \alpha_{(i)} d(i) \\
r_{(i+1)} &= r_{(i)} - \alpha_{(i)} K d(i) \\
\beta_{(i+1)} &= \frac{r_{(i+1)}^T r_{(i+1)}}{r_{(i)}^T r d_{(i)}} \\
d_{(i+1)} &= r_{(i+1)} + \beta_{(i+1)} d(i)
\end{aligned}
$$

with K, F and x being the stiffness matrix, the source term and the unknown function, respectively. The column matrices d and r are generated for the needs of the algorithm.

The advantage of this algorithm over the Gauss-Seidel approach is that it is divided into distinct steps that require the completion of the previous, before the next can be initiated. This way, perfect thread synchronization is accomplished. Race conditions still exists, but these do not hamper the quality of the output.

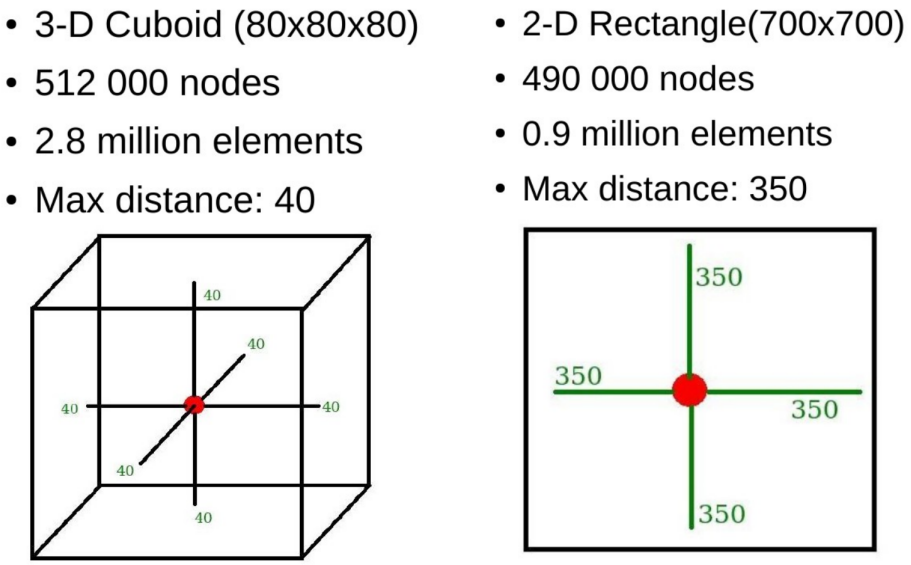

Figure 2: The above sketch illustrates the impact of dimensionality on distance to boundary for similar number of degrees of freedom

The results of that code were highly encouraging. In a domain of over $500,000$ nodes and over $2,800,000$ elements, the code managed to bring results in under $5\,\mathrm{s}$.

The code was compared against the third-party code, FEniCS [13] and proved superior. FEniCS required over 27 seconds to bring results of equal precision.

However, as the number of internal degrees of freedom increases, the maximum number of edges that must be traversed to reach a Dirichlet boundary increases and the performance degrades significantly. This was particularly noticeable moving from a 3-D domain to a 2-D domain of a similar number of elements.

The reasoning behind this lies in the solution update after every iteration under the Conjugate Gradient algorithm. The nodes begin updating their solutions from the edges of the domain , with the influence of the boundary conditions on degrees of freedom progressing inward on successive iterations. Consequently, the bigger the distance from the boundaries , the more iterations required for the inner nodes to converge(Figure2).

The standard approach to improving linear solver performance in such situations is the use of suitable preconditioning. We select a preconditioner matrix M. Of the suggested preconditioners, the simplest one to use was the diagonal matrix formed of the diagonal entries of the global stiffness matrix, K, especially well-suited for our highly parallelized settings. We then follow the Conjugate Gradient algorithm, modified as follows:

$$
\begin{aligned}
\text{Let } r_{(0)} =& F \text{ and } d_{(0)} = M^{-1}r(0) \\
\alpha_{(i)} =& \frac{r_{(i)}^T M^{-1} r_{(i)}}{d_{(i)}^T K d_{(i)}} \\
x_{(i+1)} =& x_{(i)} + \alpha_{(i)} d(i) \\
r_{(i+1)} =& r_{(i)} - \alpha_{(i)} K d(i) \\
\beta_{(i+1)} =& \frac{r_{(i+1)}^T M^{-1} r_{(i+1)}}{r_{(i)}^T M^{-1} r d_{(i)}} \\
d_{(i+1)} =& M^{-1} r_{(i+1)} + \beta_{(i+1)} d(i)
\end{aligned}
$$

The preconditioning provided an impressive speed boost. The code was tested on a domain of over 1,000,000 nodes and over 2,000,000 elements and brought results within $18\,\mathrm{s}$. The FEniCS software, when run serially, required over $170\,\mathrm{s}$ to bring comparable results.

6 Plate bending problem

To expand this work in a more mathematically involved and engineering applicable context, we considered the standard plate bending problem. This involves the calculation of point-wise displacement and rotation of a flat, horizontal plate under the effect of vertical forces and having boundary constraints on displacement and certain of its derivatives. Research into the benefits of parallel programming in the field of plate

bending is available, but while certain researchers are active in both plate bending and GPU acceleration patterns [14], publications combining those two fields are limited.

To derive a suitable plate bending element we combined the approaches of a number of authors [15, 16, 17, 18, 19]. These sources suggested a more efficient means of attaining the global stiffness matrix than numerical integration. For triangular meshes, we take shape functions,

$$N(x,y) = a_1 + a_2x + a_3y + a_4x^2 + a_5xy + a_6y^2 + a_7x^3 + a_8(x^2y + xy^2) + a_9y^3,$$

where the a_i are unknown degrees of freedom. In the quadrilateral case, the shape functions are chosen to be

$$\begin{aligned} N(x,y) = a_1 + a_2x + a_3y + a_4x^2 + a_5xy + a_6y^2 + a_7x^3 + \\ a_8x^2y + a_9xy^2 + a_{10}y^3 + a_{11}x^3y + a_{12}xy^3. \end{aligned}$$

For our chosen shape functions and degrees of freedom, the mapping from degrees of freedom to nodal displacements and their derivatives gives rise to local matrices A_{tri}, for triangular cells, and A_{quad}, for quadrilaterals, as follows,

$$A_{\text{tri}} = \begin{pmatrix} 1 & x_1 & y_1 & x_1^2 & x_1y_1 & y_1^2 & x_1^3 & x_1^2y_1 + x_1y_1^2 & y_1^3 \\ 0 & 1 & 0 & 2x_1 & y_1 & 0 & 3x_1^2 & 2x_1y_1 + y_1^2 & 0 \\ 0 & 0 & 1 & 0 & x_1 & 2y_1 & 0 & x_1^2 + 2x_1y_1 & 3y_1^2 \\ 1 & x_2 & y_2 & x_2^2 & x_2y_2 & y_2^2 & x_2^3 & x_2^2y_2 + x_2y_2^2 & y_2^3 \\ 0 & 1 & 0 & 2x_2 & y_2 & 0 & 3x_2^2 & 2x_2y_2 + y_2^2 & 0 \\ 0 & 0 & 1 & 0 & x_2 & 2y_2 & 0 & x_2^2 + 2x_2y_2 & 3y_2^2 \\ 1 & x_3 & y_3 & x_3^2 & x_3y_3 & y_3^2 & x_3^3 & x_3^2y_3 + x_3y_3^2 & y_3^3 \\ 0 & 1 & 0 & 2x_3 & y_3 & 0 & 3x_3^2 & 2x_3y_3 + y_3^2 & 0 \\ 0 & 0 & 1 & 0 & x_3 & 2y_3 & 0 & x_3^2 + 2x_3y_3 & 3y_3^2 \end{pmatrix}$$

$A_{quad} =$

$$\begin{pmatrix} 1 & x_1 & y_1 & x_1^2 & x_1y_1 & y_1^2 & x_1^3 & x_1^2y_1 & x_1y_1^2 & y_1^3 & x_1^3y_1 & x_1y_1^3 \\ 0 & 0 & 1 & 0 & x_1 & 2y_1 & 0 & x_1^2 & 2x_1y_1 & 3y_1^2 & x_1^3 & 3x_1y_1^2 \\ 0 & -1 & 0 & -2x_1 & -y_1 & 0 & -3x_1^2 & -2x_1y_1 & -y_1^2 & 0 & -3x_1^2y_1 & -y_1^3 \\ 1 & x_2 & y_2 & x_2^2 & x_2y_2 & y_2^2 & x_2^3 & x_2^2y_2 & x_2y_2^2 & y_2^3 & x_2^3y_2 & x_2y_2^3 \\ 0 & 0 & 1 & 0 & x_2 & 2y_2 & 0 & x_2^2 & 2x_2y_2 & 3y_2^2 & x_2^3 & 3x_2y_2^2 \\ 0 & -1 & 0 & -2x_2 & -y_2 & 0 & -3x_2^2 & -2x_2y_2 & -y_2^2 & 0 & -3x_2^2y_2 & -y_2^3 \\ 1 & x_3 & y_3 & x_3^2 & x_3y_3 & y_3^2 & x_3^3 & x_3^2y_3 & x_3y_3^2 & y_3^3 & x_3^3y_3 & x_3y_3^3 \\ 0 & 0 & 1 & 0 & x_3 & 2y_3 & 0 & x_3^2 & 2x_3y_3 & 3y_3^2 & x_3^3 & 3x_3y_3^2 \\ 0 & -1 & 0 & -2x_3 & -y_3 & 0 & -3x_3^2 & -2x_3y_3 & -y_3^2 & 0 & -3x_3^2y_3 & -y_3^3 \\ 1 & x_4 & y_4 & x_4^2 & x_4y_4 & y_4^2 & x_4^3 & x_4^2y_4 & x_4y_4^2 & y_4^3 & x_4^3y_4 & x_4y_4^3 \\ 0 & 0 & 1 & 0 & x_4 & 2y_4 & 0 & x_4^2 & 2x_4y_4 & 3y_4^2 & x_4^3 & 3x_4y_4^2 \\ 0 & -1 & 0 & -2x_4 & -y_4 & 0 & -3x_4^2 & -2x_4y_4 & -y_4^2 & 0 & -3x_4^2y_4 & -y_4^3 \end{pmatrix}$$

To obtain the strains from the displacements and derivatives, we require a transformation. For the quadrilateral case, this is as follows,

$$H_{\text{quad}} = \begin{pmatrix} 0 & 0 & 0 & -2 & 0 & 0 & -6x & -2y & 0 & 0 & -6xy & 0 \\ 0 & 0 & 0 & 0 & 0 & -2 & 0 & 0 & -2x & -6y & 0 & -6xy \\ 0 & 0 & 0 & 0 & -2 & 0 & 0 & -4x & -4y & 0 & -6x^2 & -6y^2 \end{pmatrix}$$

The matrix $B = H \times A^{-1}$ may then be applied to a degree of freedom vector to give local strains. Our elasticity tensor E, which converts those strains to the point-wise stresses, is defined to be (in the case of plane strain),

$$E = \begin{pmatrix} 1-\nu & \nu & 0 \\ \nu & 1-\nu & 0 \\ 0 & 0 & 0.5-\nu \end{pmatrix}$$

where ν is Poisson's ratio for elastic deformation. The local stiffness matrix may then be defined as,

$$K = \iint B^T E B dx dy \tag{9}$$

For triangular elements, we performed this integration analytically by classifying triangles in the xy plane into three types: those with two edges joining a vertical edge to their right, those with two edges joining a vertical edge to their left, and all other triangles. In the case of quadrilateral elements, we included a restriction to rectilinear cells, allowing us to again analytically integrate. This provided us with local contributions to build the global stiffness matrix.

Testing the code with the appropriate force vector at the RHS, and solving the code by implementing the Preconditioned Conjugate Gradient Algorithm, the code quickly bore reliable results. The assembly time was reduced to below 0.001 second outperforming the available professional and open-source software, Visual FEA and Nastran.

7 Conclusion

Our 12-month research provided us with useful material both for application, as well as educational purposes. We provided an overview of the PDE Problems and demonstrated the Finite Element Method as a viable approach to approximate the solution.

We conducted a thorough research on the Poisson Equation and explored new, more efficient ways to implement the finite element method. We utilized the Cotangent Method and attained a monumental speed-up in the assembly process. We formulated a series of solvers both in 2-D and 3-D and effectively managed to outperform their professional and open-source counterparts.

With this success at hand, we moved forward to produce a series of GPU accelerated solvers for the plate bending problem. By implementing an assembly method that partially bypasses the shape function formation, we managed to create an accurate, time efficient and memory efficient assembly algorithm. Once again, the accelerated code managed to over perform the competitive software, especially in the assembly part of the code.

With this success at hand, we can conclude that GPU acceleration is a very efficient way to improve the performance and memory economy of current and future solvers. Having accumulated a series of functional and efficient solvers, we can now move

on and prepare a solver for the membrane stress problem for the ultimate purpose of combining membrane stress and plate bending in one coherent shell element.

References

[1] P. Harish and P. Narayanan, "Accelerating large graph algorithms on the GPU using CUDA", High performance computing–HiPC 2007, Springer, 2007.

[2] C. Johnson, "Numerical solution of partial differential equations by the finite element method", Courier Dover Publications, 2012.

[3] , P.J. Olver and C. Shakiban, "A First Course in Applied Mathematics", 1st edition, Prentice Hall, Inc, 2009.

[4] M. Köster, D. Göddeke, H. Wobker and S. Turek. et al., "How to gain speedups of 1000 on single processors with fast FEM solvers Benchmarking numerical and computational efficiency", Fakultät für Mathematik, Technische Universität Dortmund, Ergebnisberichte des Instituts fr Angewandte Mathematik, Nummer 382, Oct. 2008.

[5] ,J.R. Schewchuk, "An introduction to the conjugate gradient method without the agonizing pain", http://www.cs.cmu.edu/ jrs/, 1994.

[6] J. Luitjens and S. Rennich, "CUDA warps and occupancy", GPU Computing Webinar, 7:12, 2011.

[7] J. Burgerscentrum, "Iterative solution methods", Applied Numerical Mathematics, 51(4): 437-450, 2011.

[8] C. Cecka, A. J. Lew and E. Darve, "Assembly of finite element methods on graphics processors", International journal for numerical methods in engineering, 85(5): 640-669, 2011.

[9] T.R. Burmleve and R.P. Buck, "Numerical solution of the Nernst-Planck and Poisson equation system with applications to membrane electrochemistry and solid state physics", Journal of Electroanalytical Chemistry and Interfacial Electrochemistry, 90(1): 1-31, 1978.

[10] R. Shewchuk, "What is a good linear finite element? Interpolation, conditioning, anisotropy, and quality measures", http://www.cs.cmu.edu/ jrs/jrspapers.html, 2002.

[11] ,T. Young and M.J. Mohlenkamp, "Introduction to Numerical Methods and Matlab Programming for Engineers", Department of Mathematics, Ohio University, August 10, 2012.

[12] R.L. Burden and J.D. Faires, "Numerical analysis, 7th edition, Prindle Weber and Schmidt, Boston, 2001.

[13] A. Logg, K. A. Mardal, G. N. Wells, et al., "Automated Solution of Differential Equations by the Finite Element Method", Springer, 2012.

[14] S. Huang, J. Xiao, Y. Hu and T. Wang, "GPU-accelerated Boundary Element Method for Burton-Miller Equation in Acoustics", Chinese Journal of Computational Physics,4:001, 2011.

[15] E. Ventsel and T. Krauthammer, “Thin plates and shells: theory: analysis, and applications”, CRC press, 2001.
[16] B. Vanan, M. Rajyalakshm and R. Inala, “Static analysis of an isotropic rectangular plate using finite element analysis (FEA)”, Journal of Mechanical Engineering Research, 4(4): 148-162, 2012.
[17] O.C. Zienkiewicz and R.L. Taylor, “The finite element method for solid and structural mechanics”, Butterworth-heinemann, 2005.
[18] L. Prášil and J. Mackerle, “Finite element analyses and simulations of gears and gear drives: A bibliography 1997-2006”, Engineering computations, 25(3), 196-219, 2008.
[19] R.W. Clough and J.L. Tocher, “Finite element stiffness matrices for analysis of plates in bending”, Proceedings of conference on matrix methods in structural analysis, 515-545, 1965.