IMPERIAL COLLEGE LONDON

MSC. ADVANCED COMPUTING ISO 1

Accelerating Unstructured Mesh Computational Fluid Dynamics on the NVidia Tesla GPU Architecture

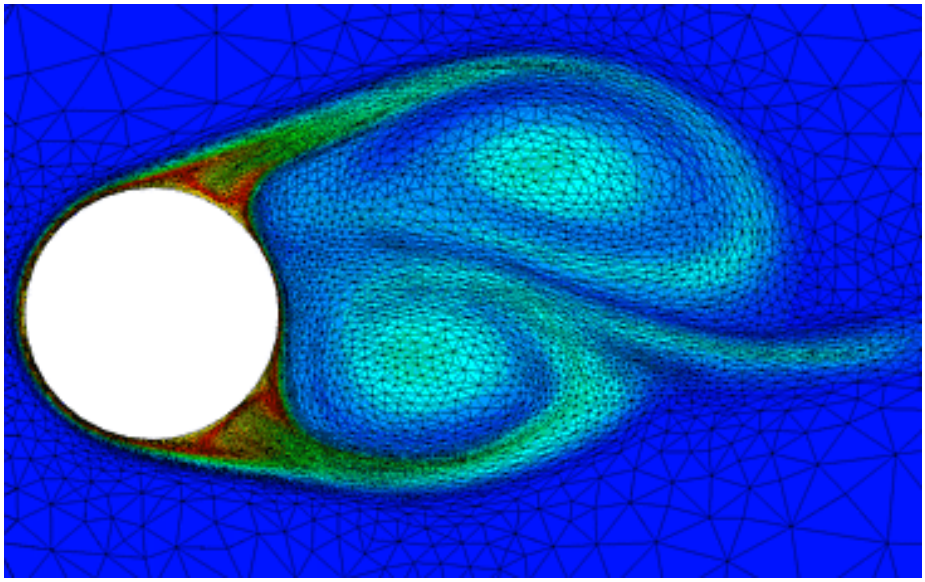


Image: Flow past a heated cylinder, computed using *Fluidity*
Source: http://amcg.ese.ic.ac.uk/

Author:
Graham MARKALL

Supervisor:
Prof. Paul KELLY

Abstract

This report presents steps towards accelerating Fluidity, a general-purpose computational fluid dynamics package. One portion of the code, an iterative solver, is targeted for optimisation by using Graphics Processing Units (GPUs) to perform computations. A literature survey which examines the performance issues of iterative solvers and optimisations which may overcome these issues on classical and vector architectures is presented. Existing iterative solvers which use GPUs for computation are surveyed to identify optimisations which may accelerate our own solver implementation. The results of experimental investigations into improving an iterative solver which uses GPUs developed in a previous work is presented. It is shown that the speed of this solver compares favourably to the solver currently used in Fluidity, being able to solve large systems up to an order of magnitude faster. Numerical accuracy of the solver is shown to be limited, and its utilisation of the GPU solver shows room for improvement. Possible directions for further work which seeks to overcome these limitations is outlined.

Acknowledgements

- I would like to thank Paul Kelly, for his supervision of this ISO, and his support and suggestions throughout.
- Francis Russell, for helping me get starting with using `thehoff`, and with understanding the finite element method.
- David Ham, for his explanations of parts of the test program, which helped me to understand its operation.
- Lee Howes, for helping me out with profiling on `thehoff`.

Contents

Chapter 1

Introduction

This report presents a literature survey and experimental exploration of opportunities to use *Graphics Processing Unit* (GPU) hardware to improve the performance of an unstructured mesh *Computational Fluid Dynamics* (CFD) package. This chapter explains the motivation behind the project, and outlines the specific contributions and conclusions of the study.

1.1 Background

Fluidity (Gorman *et al.*, 2008) is a CFD code developed by the Applied Modelling and Computation Group in the Department of Earth Science and Engineering at Imperial College. The code is general purpose, which allows for a wide range of uses. The include ocean and atmosphere modelling, and simulation of fluidised beds. An interesting application of Fluidity is described in (Shaw *et al.*, 2008).

The code solves the Navier-Stokes equations (Salvi, 2002) on an unstructured, adaptive mesh. In an unstructured mesh, larger elements are used away from the area of interest, or in areas where there is little variation in the solution. Using larger elements reduces the total number of elements in the mesh. The reduction in the number of elements reduces the total amount of computation required.

Despite this optimisation, the simulation of complex models still requires a large amount of computation. A simulation of tidal dynamics in the Baltic Sea running on a dual Intel Xeon 2.8GHz processor required over two days to complete. The performance of Fluidity is important because large simulation times must be minimised.

The main structure of the Fluidity computation is shown in Figure 1.1. This has been adapted from the more detailed code flow chart on page 158 of (Piggott, 2006). The main loop consists of the following phases:

Mesh Generation. It may be necessary to generate a new mesh to maintain the accuracy of the solution. If this is the case, then the new mesh is computed, and a *local stiffness matrix* for each element is computed. The local stiffness matrix specifies the contribution to the solution that an element makes.

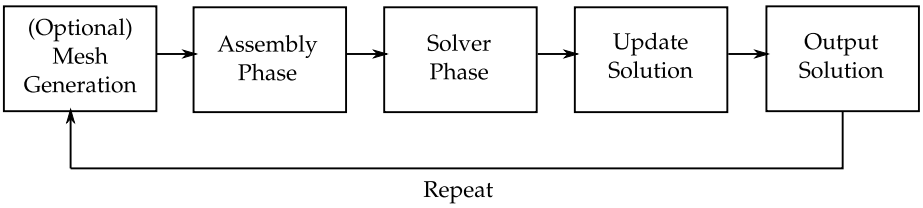


Figure 1.1: Overview of steps in the main loop of Fluidity.

Assembly Phase. In this phase a large system of simultaneous equations is assembled. The system is assembled by performing an `addto` operation for each element in the finite element mesh. The `addto` operation takes as its input a local stiffness matrix which corresponds to a particular element, and the node numbers of the element in question. The `addto` operation adds the values of the local stiffness matrix into the matrix of coefficients at locations which correspond to the node numbers of the element. The `addto` operation may be called several times per element if there are many terms present in the underlying equation, so performance in this area is crucial.

Solver Phase. In this phase, the system of equations which was assembled in the previous phase is solved. As the system of equations is very large and sparse, a direct solution cannot be obtained efficiently. An iterative solver must instead be used. Currently the Fluidity code makes use of PETSc (Balay *et al.*, 2006) to find the solution of the system. This phase makes up a large proportion of the computations in the main loop of Fluidity - therefore, the performance of this phase is also crucial.

Update Solution. As this stage, the change in the solution variables, and a new timestep is calculated. Also, an error measure is calculated at this stage.

Output Solution. The current solution is written out to disk at this stage.

In the Fluidity package a test program is also included, called `test_laplacian`. The test program solves a Laplacian equation over a 1×1 two-dimensional domain. This problem is chosen as it is possible to compute the analytical solution to the problem, which allows the error of the numerical solution to be determined accurately. The main steps of the test program are shown in Figure 1.2.

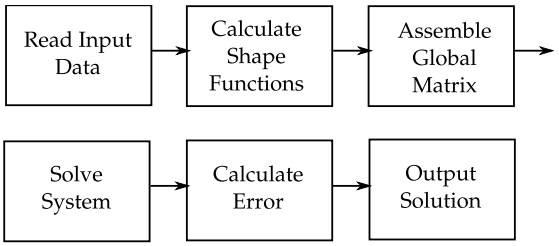


Figure 1.2: Main steps in the execution of `test_laplacian`.

Computationally intensive steps are the assembly of the global matrix, and solving the system. The assembly phase functions very similarly to the assembly phase of the main Fluidity program, building the system by repeatedly calling the `addto` function. The solution phase also makes use of the PETSc solver. However, the test program assembles a matrix which is symmetric and positive definite, which allows the solver to use the Conjugate Gradient algorithm (see Section 2.3). The main Fluidity code assembles matrices which are not symmetric positive definite, which requires the use of the Generalised Minimum Residual (GMRES) algorithm (Barrett *et al.*, 1994) to solve the system.

1.2 Recent Work

Increasing the performance of the assembly and solution phases for the test program by porting the `addto` code and the conjugate gradient solver to a GPU has been investigated (Perryman & Kelly, 2008). Benchmarking the test program using the GPU-based `addto` and solver codes showed that the speed of the `addto` phase is doubled, and that the conjugate gradient solver runs eight times faster than the PETSc conjugate gradient solver.

The benchmarks were executed using a version of the test program which solved a slightly different problem than in the original test program. The original test program produced a problem which required the use of a preconditioner (see Section 5.2), which the GPU conjugate gradient solver did not implement. Part of the work presented in this report describes the necessity and development of a preconditioner for the GPU conjugate gradient solver.

The `addto` phase is not further examined in this report, as the scope of the project has been restricted to improving the GPU conjugate gradient solver.

1.3 Contributions & Report Outline

The contributions of this report are:

- To provide a survey of optimisations used in increasing the performance of iterative solvers for large sparse systems.
- To discuss which of these optimisations may be applied to accelerate portions of Fluidity.
- To present an account of experimental investigations into two of these optimisations: preconditioning, and an alternative matrix storage format.
- To suggest how other applicable optimisations may be further investigated.

The remainder of this report has the following structure: Chapter 2 presents relevant background information. Chapter 3 presents a survey of implementation techniques and optimisations for iterative solvers for classical and vector architectures. Chapter 4 presents a survey of existing solvers which use GPUs. Chapter 5 presents experimental investigations based on optimisations described in the previous two chapters, and outlines how further developments in these areas may proceed. Chapter 6 presents conclusions and a summary of future work.

Chapter 2

Background

2.1 Introduction

This chapter introduces relevant background information. The first part of this chapter discusses the architecture of current GPUs, and a short comparison to other current parallel architectures is made. The second part of the chapter gives an overview of the conjugate gradient method, and the data structure used by Fluidity for storing large sparse matrices.

2.2 Graphics Processing Units

The workload involved in generating realtime 3D graphics involves a large amount of arithmetic operations, performed on large datasets. Due to this requirement, modern GPUs have large amounts of memory bandwidth and many processing cores.

Because GPUs are specialised to perform large amounts of arithmetic, there is no need for many of the architectural optimisations present in modern CPUs, such as branch predictors, cache, and prefetchers, etc. Instead, most of the transistors can be used for units which perform arithmetic. Figure 2.1 shows a very general comparison of the purpose of the transistors in a typical CPU and a typical GPU. In the figure, orange represents transistors devoted to memory, yellow represents transistors devoted to control flow, and green represents transistors which are dedicated to performing arithmetic. Larger areas represent more transistors.

Because of their large memory bandwidth and computational power, GPUs are well-suited to performing computational tasks. To allow programmers to utilise the GPU, manufacturers have released *Software Development Kits* (SDKs) for the GPUs which they manufacture. NVidia has released the CUDA platform (NVidia, 2007b), and AMD provides the Stream toolkit (Advanced Micro Devices, 2008). Unfortunately, code written using one SDK may not readily be compiled using another SDK. In an attempt to standardise development for GPU (and similar) architectures, the OpenCL framework was developed by a consortium of manafacturers (Khronos Group, 2008). Using OpenCL, programmers can write programs which may be compiled to different platforms without modification, including the NVidia and AMD GPUs.

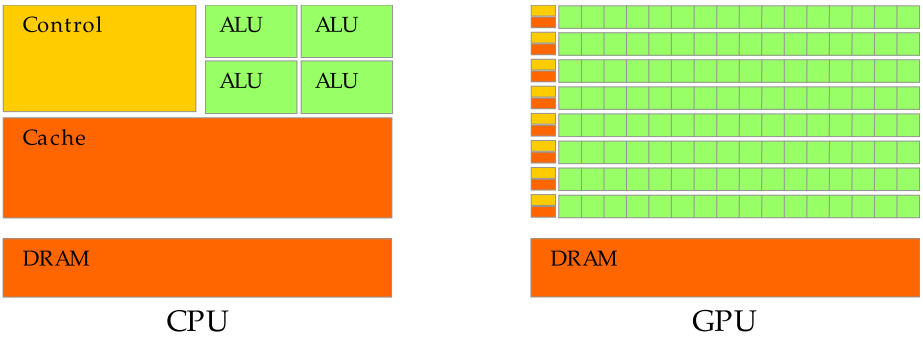


Figure 2.1: Comparison of the architecture of a general purpose CPU with that of a GPU. From (NVidia, 2007b)

OpenCL will also provide support for compilation to similar architectures, which provide high levels of parallelism. These include the Sony, Toshiba and IBM (STI) Cell processor (Gschwind *et al.*, 2006), and forthcoming parallel architectures including the Intel Larrabee architecture (Seiler *et al.*, 2008), and Sun's Rock architecture (Tremblay & Chaudhry, 2008).

Instead of containing a large number of homogeneous cores, the Cell processor includes one scalar processing unit (the Power Processing Element) which acts as a controller for eight RISC processing units (the Synergistic Processing Elements, or SPEs). Each of the SPEs has a local memory store which must be explicitly managed by the programmer.

The Intel Larrabee architecture is a specialised x86 architecture, which is extended with Larrabee-specific instructions. The Larrabee processor will include cores which are based on an embedded Pentium design. Each core will have its own cache, and cache coherency is maintained across all cores.

The Sun Rock architecture is a general purpose CPU which implements the SPARC instruction set. Sixteen cores make up the processor, which are each capable of running two threads. The Rock architecture is targeted at high data workloads, as well has high floating-point workloads.

The code written throughout the duration of this ISO is targeted at the CUDA platform - the previous work (Perryman & Kelly, 2008) towards accelerating Fluidity uding GPUs was written using CUDA, as the available hardware was manufactured by NVidia. At the time of writing, NVidia has announced that it will support OpenCL, but presently only CUDA is supported.

2.3 The Conjugate Gradient Method

The conjugate gradient method was originally presented in (Hestenes & Stiefel, 1952). The derivation of the method is not given here; instead, a brief description of the algorithm will be given. A full explanation of the method may be found in (Shewchuk, 1994). The conjugate gradient method solves systems of the form

$$Ax = b$$

where x is an unknown vector, b is a known vector, and A is a known matrix. The algorithm only succeeds for matrices A which are symmetric, and positive definite. An $n \times n$ matrix M is symmetric if for all $i, j \in 1 \ldots n$, $M_{ij} = M_{ji}$. The matrix M is also positive definite if, for every non-zero vector v, $v^T M v > 0$.

The algorithm solves the system by iterating through a loop, each iteration moving closer to the correct solution. When the solution has been found to a given accuracy, the algorithm terminates. Given known inputs A, b, a maximum number of iterations $maxit$, and an error tolerance ϵ, the algorithm is as given in Algorithm 1.

Algorithm 1: Conjugate Gradient Method

$x = 0$;
$r = b - Ax$;
$d = r$;
$\delta_0 = \delta_{new} = r \cdot r$;
while $i < maxit \ \& \ \delta_{new} > \epsilon^2 \times \delta_0$ **do**
  $i = i + 1$;
  $q = Ad$;
  $\alpha = \frac{\delta_{new}}{d \cdot q}$;
  $x = x + \alpha d$;
  $r = r - \alpha q$;
  $\delta_{old} = \delta_{new}$;
  $\delta_{new} = r \cdot r$;
  $\beta = \frac{\delta_{new}}{\delta_{old}}$;
  $d = r + \beta d$;

It can be seen that the method is composed mainly of dot products, vector addition and scalar multiplication, and a matrix-vector multiplication. The first line, $x = 0$, is the initial guess for the solution x. This may be set to something other than the zero vector, for example if an approximation of some or all of the solution is already known. However, it is sufficient to choose the zero vector when no other information is known.

2.3.1 Preconditioning

The *condition number* of a matrix is a measure which can be calculated to determine how "difficult" a system of equations based on the matrix is to solve. Lower condition numbers indicate systems which are easier to solve. The identity matrix has the lowest condition number, a condition number of 1, as a system of equations based on the identity matrix is essentially already solved. There is a relationship between the condition number of a matrix and the number of iterations required to solve the system. Matrices with higher condition numbers will generally require more iterations to solve.

The condition number of a matrix (and consequently the number of iterations and time required to solve the system) can be reduced by applying a preconditioner. The preconditioner transforms the system

$$Ax = b$$

into the system

$$M^{-1}Ax = M^{-1}b \tag{2.1}$$

which has an equivalent solution. The matrix M is chosen to approximate A. The perfect choice of M would be $M = A$, so that $M^{-1}A = I$ - this would solve the system entirely. Unfortunately, to find M in this case would require solving the system $Ax = b$, which is the problem we are trying to solve!

An issue with transforming the system in the manner shown in Equation 2.1, is that the matrix $M^{-1}A$ is generally not symmetric. An alternative transformation which preserves symmetry must be used if the resulting system is to be solved using the conjugate gradient method. A matrix E must be found such that $M = EE^T$. The system $Ax = b$ may then be transformed to:

$$E^{-1}AE^{-T}\hat{x} = E^{-1}b. \tag{2.2}$$

The matrix $E^{-1}AE^{-T}$ is symmetric, so this system may be solved using the conjugate gradient method. Once the solution to this system has been found, the solution $\hat{x}$ can be transformed to find the solution to the original system, x:

$$x = E^{-T}\hat{x} \tag{2.3}$$

The Jacobi Preconditioner

A very simple preconditioner, called the *Jacobi*, or *diagonal preconditioner*, consists of choosing $M = diag(A)$ (Barrett *et al.*, 1994). This preconditioner is straightforward to calculate and apply, but does not reduce the condition number of the matrix A to a great extent. An advantage of this preconditioner is that the preconditioning matrix M need not be explicitly computed and stored.

The Symmetric Successive Over-Relaxation Preconditioner

A more refined preconditioner (which incorporates the diagonal preconditioner) is the *symmetric successive over-relaxation* (SSOR) preconditioner (Barrett *et al.*, 1994). The symmetric matrix A may be decomposed into a sum of matrices:

$$A = D + L + L^T$$

where D is the diagonal, and L and L^T are the lower and upper triangular parts. The matrix M is chosen as:

$$M = \frac{1}{2-\omega}\left(\frac{1}{\omega}D + L\right)\left(\frac{1}{\omega}D\right)^{-1}\left(\frac{1}{\omega}D + L\right)^T$$

where $0 < \omega < 2$ is called the *relaxation parameter*. The SSOR preconditioner may be derived from the matrix A and requires little work to compute.

Other Preconditioners

Throughout the ISO, the performance of only the Jacobi and SSOR preconditioners have been evaluated. Other preconditioners exist which have a greater effect on the condition number of the matrix, though they may require more

computation. As they were not evaluated, other preconditioners are not discussed here, but there are several examples presented in Chapter 10 of (Saad, 2003).

Preconditioning in the Test Program

The test program from Fluidity uses the SSOR preconditioner by default. However, this default choice can be over-ridden. Changing the preconditioner allows the effect on the performance of the solver of each preconditioner to be evaluated. Section 5.2.4 gives an account of changing the default preconditioner to evaluate the performance of different preconditioners.

2.3.2 Other Iterative Methods

There are other iterative methods which overcome the limitations of the conjugate gradient method, and are able to compute the solutions of non-symmetric and indefinite systems. These include the Bi-Conjugate Gradient Stabilised (BCGSTAB) and Generalised Minimum Residual (GMRES) methods. A description of these, and other methods, may be found in (Barrett *et al.*, 1994) and (Saad, 2003). The Fluidity code uses the GMRES method for general systems of equations.

The other iterative methods are built from similar operations to the conjugate gradient method - dot products, vector operations, and matrix-vector multiplication. If the conjugate gradient method can be efficiently implemented on GPUs, then it is very likely that other methods can be implemented using GPUs to obtain similar performance gains. Once the building blocks of the conjugate gradient method have been developed, re-using some of the code will allow the other methods to be developed with relative ease.

2.4 Compressed Row Storage

Fluidity stores matrices with large numbers of non-zero elements using a format which does not store the non-zeroes, to save space. The format used is *Compressed Row Storage* (CRS) (Shahnaz *et al.*, 2005). Instead of storing n^2 elements, only $2nnz + n + 1$ elements are stored, where nnz is the number of non-zero elements and n is the dimension of the matrix. However, the CRS format leads to inefficiency in accessing the elements of the matrix, as it requires an indirection step for every access to a non-zero element in the matrix. The format stores a sparse matrix in three arrays:

- `val` stores the values of the non-zero elements of the matrix in a floating-point format.
- `col_ind` stores the column indices of the corresponding values in `val`.
- `row_ptr` is an integer array which stores the locations in val which start a row.

To assist in conveying how the CRS format stores a sparse matrix, an example will be given. Consider the matrix A:

$$A = \begin{pmatrix} s_{11} & s_{12} & 0 & 0 & 0 & 0 \\ s_{21} & s_{22} & 0 & s_{24} & 0 & 0 \\ 0 & s_{32} & s_{33} & 0 & 0 & 0 \\ 0 & 0 & 0 & s_{44} & s_{45} & 0 \\ 0 & s_{52} & 0 & 0 & s_{55} & 0 \\ 0 & 0 & 0 & 0 & s_{65} & s_{66} \end{pmatrix}$$

where s_{ij} represents a non-zero element. Figure 2.2 shows how the matrix A would be stored using the CRS format.

Figure 2.2: CRS representation of the matrix A.

`val`	s_{11}	s_{12}	s_{21}	s_{22}	s_{24}	s_{32}	s_{33}	s_{44}	s_{45}	s_{52}	s_{55}	s_{65}	s_{66}
`col_ind`	1	2	1	2	4	2	3	4	5	2	5	5	6

`row_ptr`	1	3	6	8	10	12

2.5 Conclusion

This chapter has discussed the architecture of GPUs and given a short comparison to other architectures. In addition, the key points of the conjugate gradient method, preconditioning, and the CRS storage format have been presented. The next chapter continues by discussing performance issues of iterative methods (like the conjugate gradient method) on classical architectures which have been identified in the literature, and gives an overview of strategies which have been implemented in order to mitigate the effects of these performance issues.

Chapter 3

Classical Architectures

3.1 Introduction

This chapter provides a brief examination of the issues involved in producing high performance sparse iterative solvers, and the optimisations which may be made to overcome these issues. The sparse matrix-vector multiplication kernel is introduced, and its performance issues and potential optimisations are discussed. As well as classical architectures, some optimisations for vector architectures are discussed.

3.2 Sparse Matrix-Vector Multiplication

The most important computational kernel in iterative linear solvers is the *Sparse matrix-vector multiplication* (SpMV) kernel (Barrett *et al.*, 1994). The SpMV is the most computationally intensive operation in an iterative solver. Most of the execution time of the main loop of a solver is spent inside the SpMV kernel.

An SpMV kernel which operates on matrix stored in CRS format (stored in the variables `row_ptr`, `col_ind` and `val`) and a vector `x`, the result of which is stored in the vector `y`, is implemented by the following code:

```
for(i=0; i<N; i++)
  for(j=row_ptr[i]; j<row_ptr[i+1]; j++)
    y[i] += val[j]*x[col_ind[j]];
```

About 80% of the time taken by an iterative solver is spent within an SpMV kernel, regardless of the architecture (Buatois *et al.*, 2007). Because of the importance of the SpMV kernel to iterative solvers, the remainder of this section will focus its performance issues and potential optimisations. Other kernels which are used in iterative solvers (e.g. the dot product) make up much less of the execution time of the main loop, and do not suffer from performance issues to the extent which the SpMV kernel does.

3.3 Performance Issues in SpMV Kernels

Traditionally the SpMV kernel shows poor performance - conventional implementations may only run at 10% of the maximum theoretical floating point performance of the machine (Vuduc & Moon, 2005). In general, memory bandwidth was found to be a major limiting factor (Williams *et al.*, 2007). A recent study of the performance of SpMV kernels (Goumas *et al.*, 2008) identified the following specific performance issues:

1. A key feature of the SpMV kernel is that its execution involves a large number of load instructions relative to the number of floating point instructions which are executed. It is observed by (Goumas *et al.*, 2008) that the SpMV kernel performs $O(n^2)$ operations on $O(n^2)$ items of data, in contrast with a matrix-matrix multiply, which performs $O(n^3)$ operations on $O(n^2)$ items of data. This indicates that there is little temporal locality in the SpMV kernel, so optimisations must focus on spatial locality and vectorising memory accesses.

2. The kernel accesses each matrix element exactly once - as a result, there is no temporal locality of accesses into the matrix. However, as each iteration of the loops in the kernel access the arrays which store the matrix sequentially, there is good spatial locality of accesses into the matrix.

3. Indirect memory references - The indices of the non-zero elements (stored in `row_ptr` and `col_ind`) must also be loaded from memory when accessing the elements of the matrix. The value stored in `row_ptr` is used as an offset into the `val` and `col_ind` arrays. This increases the amount of computation and memory bandwidth required to load elements of the matrix.

4. Accesses into the vector `x` (sometimes referred to as the *source vector*) are irregular - there is poor spatial locality of accesses into the source vector. The access into the source vector follows the sparsity pattern of the matrix. When the matrix sparsity pattern has a near-random structure, accesses into the source vector may also exhibit poor temporal locality.

5. Short row lengths - if rows in the matrix have few non-zero elements then little computation is performed in each trip of the inner loop. This negatively affects performance as the ratio of instructions which perform arithmetic to control flow instructions executed is decreased. Simply put, the total efficiency of the computation is decreased.

6. Working set size - If the working set (comprised of the matrix, and the source and destination vectors) is larger than the cache, then performance is negatively impacted as portions of the working set must periodically be flushed from the cache, and subsequently fetched from memory when they are required later on in the execution of the kernel.

3.4 Performance Optimisation of SpMV Kernels

Many optimisations to overcome the performance issues of SpMV kernels have been implemented and tested in recent years. These optimisations can be di-

vided into broad categories:

- Register blocking optimisations
- Cache optimisations
- Matrix reordering optimisations
- Parallel optimisations

Optimisations in each of these areas are identified and examined.

3.4.1 Register Blocking Optimisations

Register blocking optimisations aim to improve performance by reducing the amount of indirection required when accessing elements of the matrix. This can be achieved by storing elements of the matrix which are close to each other together, and using a single indirection to access any element within this group.

A simple optimisation of this type was proposed in (Toledo, 1997). The algorithm finds all the 1×2 blocks of non-zero elements in the matrix. The matrix is represented as the sum of two matrices. One matrix consists of all the 1×2 blocks which were identified, and the other contains the remaining non-zero elements. When the SpMV kernel executes, it can access the elements of the blocked matrix using a single indirection for each 1×2 block of non-zeroes, instead of requiring one indirection per non-zero. In order to better exploit this optimisation, a heuristic algorithm which re-orders the matrix to maximise the number of 1×2 blocks was developed (Pinar & Heath, 1999).

Toledo's blocking algorithm may be generalised to blocks of arbitrary size. The storage format which represents a matrix using these arbitrary blocks is the *Blocked Compressed Row Storage* format (Shahnaz *et al.*, 2005). The format is a modification of the CRS format. Each block is stored as a dense matrix and all its values are stored contiguously. As with CRS, BCRS stores the matrix in three arrays:

- `val` is an array which stores the values of each block in a floating-point format.
- `col_ind` stores the column indices of the $(1,1)$ elements of the blocks stored in `val`
- `row_ptr` stores the indices in `col_ind` and `val` which start a row of blocks.

In a sparse matrix there may be several values which are very close together which do not make a complete block of the required size. If this is the case, then the values may be stored as a single block, padded with zeroes. Computations on this block will then perform some redundant operations such as multiplying by the zero elements. The reduction in indirections usually outweighs the cost of the additional computation. However, too much padding can have a negative effect on performance (Goumas *et al.*, 2008). If this is the case, the amount of padding may be reduced by using a smaller block size, at the cost of increased indirection.

Figure 3.1 shows how the matrix A (see Section 2.4) is stored using the BCRS format with 2×2 blocks.

Figure 3.1: BCRS representation of the matrix A.

val	s_{11}	s_{12}	s_{21}	s_{22};	0	0	0	s_{24};	0	s_{32}	0	0;
	s_{33}	0	s_{44}	0;	0	0	s_{45}	0;	0	s_{52}	0	0;
	s_{55}	0	s_{65}	s_{66};								

col_ind	1	3	1	3	5	1	5	row_ptr	1	3	6

(Pinar & Heath, 1999) also presented an algorithm which maximises the density of the blocks by reordering the matrix. This is also a generalisation of the algorithm which maximises the 1×2 blocking.

3.4.2 Cache Optimisations

Cache Blocking Optimisations

Cache blocking optimisations are applicable when the source and destination vectors are too large to fit into the cache (Williams *et al.*, 2007). This scenario occurs for sufficiently large matrices. A cache blocking scheme attempts to keep a portion of the source vector in the cache by splitting the matrix up into tiles, and aims to increase the temporal locality of accesses into the source vector. The values of the result vector may be partially computed for each tile in turn.

Figure 3.2 illustrates an example of a tiling of a matrix into sixteen tiles, along with the source and destination vectors. The number of each tile indicates a possible order in which the SpMV kernel could work on the tiles. When the kernel is operating on tiles 1-4, only the top quarter of the destination vector is accessed. When the kernel operates on tiles 5-8, the next quarter down is is accessed. The temporal locality of accesses into the destination vector is also improved in this case. To achieve optimal performance, the portions of the vector must be able to fit in the cache.

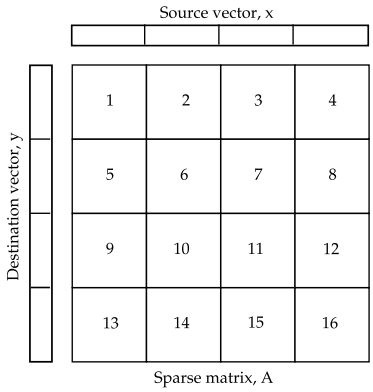


Figure 3.2: A possible tiling of a matrix into 16 tiles, with the source and destination vectors.

However, it can be difficult to determine an optimal tile size (Yelick, 2008). Methods to determine the tile size include profiling different tile sizes, or using a heuristic (Williams *et al.*, 2007).

Cache blocking optimisations are generally less successful than the other types of optimisation described in this section. Matrices which have a random or near-random structure are the most suitable for this optimisation. Additionally, combining cache blocking with other optimisations usually results in lower performance than applying a single optimisation (Yelick, 2008).

Prefetching

Modern CPU microarchitectures implement a prefetcher in their hardware (Williams *et al.*, 2007). The prefetcher detects regular memory access patterns and predicts the locations of future memory accesses. Data is fetched from these locations immediately into the cache. When this data is required by an instruction, the processor will not have to wait for the data to fetched from main memory as it has already been pre-fetched.

As the SpMV kernel accesses the matrix in a regular fashion (working sequentially across the three arrays), the prefetcher is able to detect the simple linear access pattern and prefetch the correct portions of the arrays. However, the access into the source vector follows a random pattern, which the hardware prefetcher cannot correctly predict.

A software prefetcher can be used to prefetch the required portions of the source vector into the cache before they are required, if some information is known about the sparsity structure of the matrix. For example, if the matrix non-zeroes are all close to the main diagonal, then it is possible to infer that when the SpMV kernel is working on rows near the top of the matrix, the entries near the start of the source vector will be required. When the kernel is operating near the bottom of the matrix, entries closer to the end of the source vector will be required.

Investigation into software prefetching found that it provides performance gains on the Intel Clovertown and Sun Niagara2 architectures, even though these architectures implement a hardware prefetcher (Williams *et al.*, 2007). On the Intel architecture, prefetching brings data first to the L2 cache. Another hardware prefetcher fetches data into the L1 cache. Using the software prefetcher provided a mechanism for the required data to be placed directly into the L1 cache. On the Sun Niagara2, the hardware prefetcher only brings data into the L2 cache. Therefore, the software prefetcher was able to provide in increase in speed by bringing the data directly into the L1 cache.

3.4.3 Matrix Reordering Optimisations

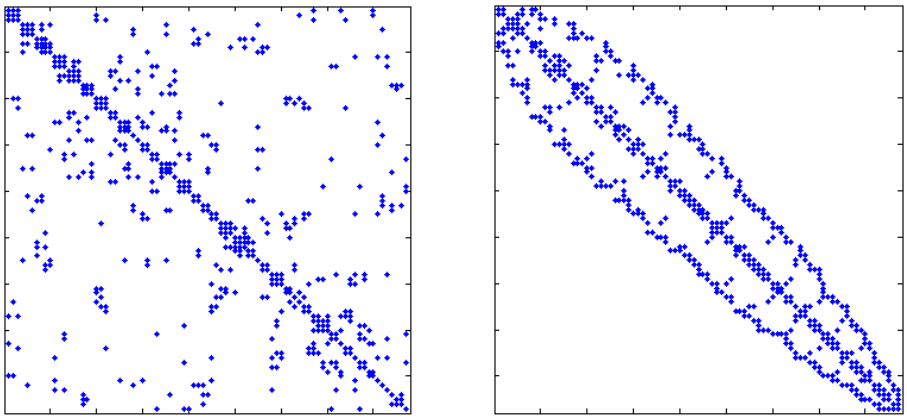
Reordering optimisations improve the spatial and/or temporal locality of the source and/or destination vectors throughout the matrix multiply. A reordering may impose some structure on a matrix which previously was not present.

Commonly-used reordering algorithms are the *Cuthill-McKee* (CM) algorithm, and the *Reverse Cuthill-McKee* (RCM) algorithm (Cuthill & McKee, 1969), which seek to minimise the bandwidth of the matrix. The bandwidth is minimised by reordering the rows and columns so that the non-zeroes of the matrix are as close to the main diagonal as possible.

This ordering increases the spatial and temporal locality of accesses to the source vector. When the dot product of each row of the matrix with the source vector is computed, the entries in the source vector which are accessed will

be much closer together, which increases the spatial locality. When the SpMV kernel operates on subsequent rows, the portion of the source vector which is accessed will overlap with the portion which was accessed when computing the current row. It is more likely that elements of the source vector which were accessed recently will be re-accessed. This increases temporal locality.

An example of the effect of the Reverse Cuthill-McKee ordering is given in Figure 3.3. Figure 3.3(a) shows the sparsity pattern of the original matrix. Figure 3.3(b) shows the sparsity pattern of the matrix after reordering. In this case, the original matrix bandwidth was 80. The reordered matrix had a bandwidth of 18, which is a significant reduction.



(a) Pre-RCM (b) Post-RCM

Figure 3.3: A sparse matrix before and after Reverse Cuthill-McKee reordering.

Another algorithm which may be used is the Column Count algorithm, which sorts the matrix rows by the number of non-zeroes in each row (as used in the ITPACK sparse matrix storage format (Kincaid & Young, 1988)). This algorithm has been shown to be useful on vector machines (D'Avezedo *et al.*, 2005) when the compressed row storage format is used. A performance benefit is provided as a vector machine must work on many rows simultaneously to achieve high performance. If the machine is operating on rows of different lengths, processors which are working on shorter rows will complete their workload first. These processors must then wait for the other processors which are working on longer rows to complete before they can continue with a new row.

It has been suggested that on shared-memory multiprocessors, reorderings such as the Lin-Kernighan algorithm (Lin & Kernighan, 1973), Prim's algorithm and the nearest-neighbour algorithm (Cormen *et al.*, 2001) can improve the spatial and temporal locality of accesses into the source vector (Pichel *et al.*, 2004). These algorithms all reorder the matrix by treating it as the connectivity matrix of a graph, and creating an ordering which minimises the spanning tree of the graph. Benchmarks performed on various shared-memory multiprocessors showed that using the Lin-Kernighan algorithm could improve the performance of the cache whilst the SpMV kernel is executing, reducing the miss rate by up to 20%.

3.4.4 Parallel Optimisations

There are many ways in which parallel processing can speed up the SpMV kernel. Parallel optimisations exploit parallelism present in the SpMV kernel. On shared memory architectures, optimsations such as thread blocking allow multiple cores/processors to work on a portion of the problem. Message passing architectures require the problem to be partitioned, and portions of the matrix distributed across multiple nodes.

Thread Blocking

The workload may be divided between several threads, an optimisation called *Thread Blocking*. The matrix may be partitioned row-wise or column-wise, and each thread computes the result for one partition. It is preferable to partition the matrix row-wise as it is more straightforward to implement (Williams *et al.*, 2007). In row-wise partitioning, only one thread works on a particular element of the result vector; when the matrix is partitioned column-wise, partial results for each element of the destination vector are computed by each thread and must be reduced. Rather than allowing each thread to work on an equal number of rows, a more optimal partitioning can be determined by allowing each thread to work on an equal number of non-zeroes (Williams *et al.*, 2007).

Thread blocking is a simple example of a scheme exploiting parallelism in the SpMV kernel. Modifications to the basic thread blocking scheme consist of alternative partitionings of the matrix, or alternative ways to distribute the computation.

Partitioning

On message passing architectures, it is necessary to split the matrix into partitions which are stored on separate nodes. Each node computes the portion of the result which depends upon the portion of the matrix which it owns. As with thread blocking, partitioning the matrix by rows is an option. However, other partitioning schemes can reduce the storage requirements for each node.

The Reverse Cuthill-McKee algorithm may be applied before partitioning takes place to decrease the storage requirements (Yelick, 2008). Because the elements of the matrix are near the main diagonal after application of the algorithm, only a portion of the source vector is required by each individual node. As an example, consider the matrix after reordering shown in Figure 3.3(b). One possible partitioning could be to divide the matrix between two nodes, the first operating on the top half of matrix, and the second working on the bottom half of the matrix. Before the RCM algorithm is applied, each node would need to store the whole source vector, as the non-zero elements in the top half of the matrix span the entire width of the matrix. After the algorithm is applied, the first node only requires just over half of the source vector to be stored. Similarly, the second node needs to store just over half of the total source vector, although it requires the part of the vector towards the end.

Vector Architectures

As well as classical architectures, vector architectures are used for solving large systems. On vector architectures, the *Jagged Diagonal Storage* (JAD) format

(Shahnaz *et al.*, 2005) is often used as it allows the machine to perform the same operation on many elements of the matrix at once, which is required for high performance (Tiyyagura *et al.*, 2006).

Performance optimisation on vector machines can be obtained by modifying the SpMV kernel so that the computation is performed on more than one diagonal of the JAD format. This exposes parallelism in the kernel, which helps to improve overall performance as the utilisation of processors in the vector machine is increased. Benchmarking showed that kernels operating on five diagonals simultaneously achieved an increase in floating-point operations per second (FLOPs) of approximately 3.35 times.

Additionally, speedups were obtained by storing partial results of the computation in *vector registers*. Vector registers are small areas of storage very close the the processing units, which can be accessed more quickly than main memory. Using the registers can also increase the performance of the SpMV kernel.

3.4.5 Other Optimisations

Other optimisations which do not fit into these broad categories have also been suggested in the literature. These optimisations include:

- (Goumas *et al.*, 2008) propose that the pressure on the memory can be alleviated by using the smallest data types possible, thereby reducing the total size of the dataset. For example, if double-precision accuracy is not required for all parts of the computation, using single-precision values can reduce the storage required by each value from 8 bytes to 4 bytes. Additionally, if the number of non-zeroes and degree of the matrix does not exceed 65535, it is possible to use the `short int` type for the `row_ptr` and `col_ind` arrays in the CSR format. This reduces the storage required for these arrays from 4 bytes per element to 2 bytes per element.

- The SpMV kernel consists of two loops: the outer loop iterates over `row_ptr`, and the inner loop iterates over `val` and `col_ind`. The kernel may be rewritten so that the outer loop iterates through the non-zero values of the matrix. Although a nested loop is still used, this optimisation often results in higher performance for a serial SpMV kernel (Williams *et al.*, 2007).

3.5 Summary of Classical SpMV Optimisations

The optimisations presented will be reviewed. Additionally, their suitablility to the GPU architecture must be examined in order to determine the direction of future development of the GPU-based SpMV kernel and iterative solver.

Register Blocking. Memory bandwidth utilisation is maximised on NVidia GPUs when data is accessed sequentially (NVidia, 2007b). As the BCRS format stores the elements of each block in contiguous memory, it is possible that the memory access pattern throughout the execution of the SpMV kernel will be more favourable when BCRS is used than when CRS is used. In the CRS format, the elements which would make up a block are not stored contiguously.

Matrix Reordering. As the Cuthill-McKee algorithm pushes all the non-zero elements towards the main diagonal, it may increase the number of elements in the source vector which are accessed contiguously throughout the execution of the SpMV kernel. Therefore, the performance of the SpMV kernel (and the solver in general) when using this reordering of the matrix should be evaluated, and compared to the performance when no reordering is implemented.

Cache Optimisations. As there is no hardware-controlled cache on NVidia GPUs, there is little utility for optimisations which seek to increase the cache hit rate. However, as each multiprocessor on the GPU has a 16KB shared memory which can be explicitly controlled by the programmer (NVidia, 2007b), it is possible that a prefetcher which brings data into shared memory may provide some performance benefit. As the size of the shared memory is very limited, neither the matrix nor the source vector will fit completely in the shared memory. Because access to the matrix follows a regular pattern, the GPU memory performance will not be impacted, so the matrix is not a good candidate for being prefetched. However, access to the source vector follows a random pattern, which will impact upon the memory performance of the kernel on the GPU. Although the source vector cannot fit into the shared memory, the portion of the source vector required for each row of the matrix can be reduced by applying the Cuthill-McKee algorithm to the matrix.

Parallel Optimisations. Thread blocking is already implemented by the prototype GPU SpMV kernel (Perryman & Kelly, 2008). Partitioning the matrix is currently unnecessary as the code has only been developed to utilise a single GPU. For extremely large problems, multiple GPUs will eventually be required, as the whole working set will not fit inside the memory of a single GPU. At this stage, partitioning should not yet be investigated, but will be essential in the future if the GPU-based kernel is to utilise multiple GPUs. The JAD format may be investigated, but its investigation should not be regarded as a high priority for developing a fast GPU-based SpMV kernel and iterative solver. Implementing the JAD format would require extra development effort, as it is not the matrix format natively supported by Fluidity. Also, implementing the JAD format in Fluidity may introduce inefficiency to other areas of the computation.

3.6 Conclusion

This chapter has presented a review of common optimisations found in the literature for optimising SpMV kernels on classical and vector architectures. Techniques which may be amenable to improving the performance of a GPU-based SpMV kernel and iterative solver have been identified. The next chapter continues to identify possible optimisations, by studying recent implementations of iterative solvers which use GPUs for computation.

Chapter 4

Existing Implementations

4.1 Introduction

In this chapter, recent implementations of solvers which use GPUs for computation are surveyed and evaluated. The optimisations implemented are examined and evaluated, in order to identify the optimisations which will bring the highest performance gains to a GPU-based solver. The CuBLAS library is also examined, as it provides kernels which could potentially be used in the development of an iterative solver. Additionally, an implementation of the multifrontal method which utilises GPUs is examined - however, it is shown that the multifrontal method should not currently be pursued. This chapter concludes by giving a summary of the optimisations used in these implementations, and identifies which should be investigated further.

4.2 cuBLAS (NVidia, 2007a)

NVidia has produced an implementation of a subset of the BLAS (Lawson *et al.*, 1979) functions which has been implemented using CUDA. BLAS is an *Application Programming Interface* for libraries which perform linear algebra functions, or *kernels*. The BLAS routines are split into three levels:

Level 1 contains operations which are performed on vectors. These operations include vector addition, multiplication of a vector by a scalar, and dot products, amongst many others. An example of a Level 1 kernel is the `daxpy` kernel, which computes the operation

$$y \leftarrow \alpha x + y$$

using double precision arithmetic, where α is a scalar, and x and y are vectors.

Level 2 contains *matrix-vector* operations, which operate on a matrix and one or more vectors. This level includes matrix-vector multiplication, and solving $Ax = b$ where A is a triangular matrix. An example of a Level 2 kernel is the `dgemv` kernel, which computes the operation

$$y \leftarrow \alpha Ax + \beta y$$

using double precision arithmetic, where α and β are scalars, A is a matrix, and x and y are vectors.

Level 3 contains *matrix-matrix* operations, which operate on more than one matrix. This includes matrix-matrix multiplication amongst other operations. An example of a Level 3 kernel is the `dgemm` kernel, which computes the operation

$$C \leftarrow \alpha AB + \beta C$$

using double precision arithmetic, where α and β are scalars, and A, B and C are matrices.

This library has been developed to allow the GPU to be used to perform linear algebra computations without the programmer having to interact with the CUDA environment. Because the BLAS kernels are standard, the programmer may take an existing code which uses the BLAS kernels and modify it to perform linear algebra computations using the GPU with minimal effort.

Because linear iterative solvers (such as the conjugate gradient method) are made up of linear algebra operations, it would be expected that it is possible to replace the calls to BLAS functions in the solver to calls to CuBLAS. This would allow experimentation with using a GPU to accelerate the solver very easily.

Unfortunately, CuBLAS only supports dense matrix formats. Therefore, it cannot be used to implement a solver which is suitable for the systems of equations which arise in Fluidity - these systems are very large and sparse, and must be stored in a sparse matrix format.

4.3 Concurrent Number Cruncher: A GPU Implementation of a General Sparse Linear Solver (Buatois *et al.*, 2007)

This paper describes the implementation of a Jacobi-preconditioned conjugate gradient solver which uses the GPU for computation. This solver is implemented for the CPU, and AMD-ATI and NVidia GPUs. Floating point computations are performed in single precision.

The solver supports both the CRS and BCRS storage formats. When the BCRS format is used, a choice is available between 2×2 blocks and 4×4 blocks. The benchmarks presented in the paper show that optimal performance is achieved on nVidia cards when the 4×4 block size is used. However, for more sparse matrices, the filling ratio of the 4×4 blocks may drop to such a low level that a 2×2 block size may offer better performance. The example matrices presented do not show very high filling ratios; The 2×2 block size showed a filling ratio of 50%, and the 4×4 block size showed a filling ratio of 29%.

The implementation exploits the capability of nVidia GPUs to fetch up to four single precision floating point values at once, by using the `float4` data

type (NVidia, 2007b). When the 2×2 block size is used, each block is stored as a single `float4` vector. When the 4×4 block size is used, the blocks are stored as four 2×2 sub-blocks each of the `float4` type. This maximises the amount of data which can be retrieved in a single fetch.

The 2×2 block size is optimal for reading the coefficients of the matrix, allowing four values to be fetched simultaneously. However, it is sub-optimal for reading and writing the source and destination vectors as only two corresponding elements may be read/written in a single operation. The 4×4 block size is optimal for reading and writing the vectors, as it allows a full four elements to be read/written in a single operation. This may explain why the best performance was achieved using the 4×4 block size.

The CNC implements the Reverse Cuthill-McKee reordering, but testing showed that this had no effect upon the performance of the solver. The reason for this is unknown - however, it is shown that implementing the reordering does not significanly affect the fill rate of blocks when using the BCRS format.

The dot product between two vectors is computed in parallel in this implementation. Each thread multiplies a pair of elements from the source vectors and stores the result. This leads to a number of partial results being stored in contiguous memory locations. Each thread of the reduction kernel is able to reduce four values with a single fetch by treating intermediate results as a single `float4` vector.

The `saxpy` (Similar to the `daxpy` operation, but with single-precision arithmetic, see Section 4.2) operation is parallelised in this implementation by each thread computing a single element of the result, y. It is stated that in order for this operation to be efficient, there must be an order of magnitude more threads than there are processing units. Because there is a one-to-one correspondence between elements and threads, there must be an order of magnitude more elements than there are processing units. As an example, on an NVidia 280GTX GPU there are 240 processing units, so for maximum efficiency the length of the result vector must be an order of magnitude greater than 240.

The results presented show that the preconditioned conjugate gradient method executes up to six times faster on a GPU than on a CPU. The sparse matrix-vector multiplication executes approximately four times faster on a GPU than on a CPU. The SpMV operation shows less speedup because its floating-point throughput is restricted by the low levels of spatial and temporal locality when accessing elements of a matrix stored in a sparse format.

A comparison of the results for the GPU solver using CRS and BCRS shows a large performance improvement when using BCRS compared to CRS. BCRS 4×4 was shown to be approximately 50% faster than BCRS 2×2 on NVidia and ATI GPUs. Additionally, BCRS 2×2 was shown to be 300% faster than CRS on NVidia GPUs.

Future work based on CNC includes modifying the solver to support parallelism across multiple GPUs, and clusters of multiple PCs with GPUs. Other methods are also to be implemented using the CNC framework, such as the GMRES method. Implementing these methods will increase the applicability of CNC, as they allow systems to be solved which are not symmetric or positive definite.

4.4 Implementing the Conjugate Gradient Algorithm on Multi-core Systems (Wiggers *et al.*, 2007)

A parallel implementation of the conjugate gradient algorithm for multicore CPUs and for GPUs is described. The motivation for this implementation is to speed up the processing of *Digital Optical Tomography* (DOT) data, which requires a large set of linear equations to be solved.

This implementation uses a matrix stored in a symmetric CRS format. This is similar to the CRS format, but saves space when storing a symmetric matrix by only storing the upper or lower triangle of the matrix.

As the sparse matrix-vector multiplication operation of the solver is the most time consuming part of the conjugate gradient solver, the authors chose to focus on finding an efficient implementation of this algorithm for the CPU and GPU. Their attempts are focused on increasing the spatial and temporal locality of data in the algorithm.

Several methods of splitting the matrix up into parts to be processed separately are discussed. Figure 4.1 gives a graphical representation of the splitting mathods discussed.

- The even-odd split divides the direct neighbours of the matrix into two different sets. This splitting requires the results of two sets to be combined, and results in a large numerical error. Due to the large numerical error, the number of iterations required for convergence doubles. Because of this poor performance, the even-odd split is not discussed further.
- The block row split divides the matrix into two parts, an upper and a lower part. The symmetry of the matrix is lost when using this splitting, so twice the amount of storage is required to store the two sub-matrices in CRS format.
- The third splitting discussed is the submatrix split, which splits the matrix into several square submatrices. This splitting retains the symmetry of the system of equations, and does not introduce large numerical errors. This splitting was found to be the most efficient for the CPU implementation.

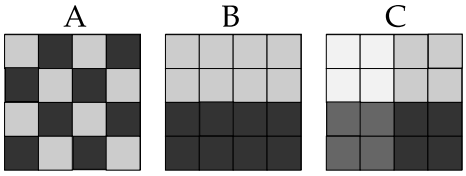


Figure 4.1: Various ways to split a matrix, from (Wiggers *et al.*, 2007). A: Even-odd split. B: Block row split. C: Submatrix split.

The Reverse Cuthill-McKee algorithm is used to re-order the matrix when it is implemented on a CPU. By moving most of the elements close to the diagonal, the algorithm was expected to improve spatial locaility. However, there is no comparison of the performance of the solver between the re-ordered matrix and the original matrix ordering.

It is inefficient to distribute the processing of parts of a single row across multiple threads on the GPU. Instead, the GPU implementation uses one thread for each row of the matrix in the SpMV kernel. Instead of using the Reverse Cuthill-McKee reordering on the GPU, the matrix is re-ordered using the column count algorithm. This improves performance because threads which execute on the same multiprocessor must all execute the same instruction (Lindholm *et al.*, 2008). If individual processors reach the end of a row before other processors on the same multiprocessor, then they must stay idle until all processors have reached the end of the row. When the rows are ordered by the number of non-zeroes, all the processors in one multiprocessor will complete at the same time or very close together.

The performance of the solver was tested using a single system of equations with matrix dimensions of $138,324 \times 138,324$, of which approximately 2.5×10^6 elements were non-zero. The GPU implementation of the solver was found to run approximately 2.56 times faster than the CPU implementation. However, it is stated that the relative error of the GPU implementation is higher than the CPU - this is likely to be due to the single precision arithmetic used by the GPU. In order to obtain a similar relative error to the CPU implementation, the time required by the GPU implementation is similar to the time required by the CPU implementation.

The speed of the GPU implementation was limited by the memory bandwidth. It is stated that using the "software controlled caches" (shared memory) of the GPU will relieve the pressure on the memory bandwidth, thus increasing the performance. However, this would require a change to the conjugate gradient algorithm which was not implemented. The required changes to the algorithm are not discussed, but it is thought that this could mean reordering the matrix using an alternative algorithm. Future work may involve implementing these changes.

4.5 Sparse Matrix Solvers on the GPU: Conjugate Gradients and Multigrid (Bolz *et al.*, 2005)

This paper describes the implementation of a conjugate gradient solver on the GPU to solve the Poisson equation ($\nabla^2 u = f$), with Neumann boundary conditions. This is similar to the problem solved by the test program.

A short description of the approach to the implementation is given, which views the GPU as a stream processor (Khailany *et al.*, 2001). The gather operation of a stream processor is implemented by fetching random-access data into the shared memory of a multiprocessor, which is referred to as a "saved stream". This hides the high latency of true random memory access on the GPU. There is also mention of loading four elements simultaneously using wide data types. It is likely that the `float4` data type is used to achieve this, as in the Concurrent Number Cruncher. An efficient parallel sum-reduction algorithm is presented, which, similarly to other implementations, reduces four elements per thread per iteration.

The matrix is stored in a modified CRS format in this implementation. The diagonal terms will always be non-zero for problems of Poisson's equation, so the diagonal terms are stored sequentially in a vector for fast access. The off-

diagonal non-zero terms are stored in a CRS format. When a diagonal term is read, the vector is accessed, and when an off-diagonal term is read, the CRS data structure is consulted. The rows of the matrix are sorted by the number of non-zeros in this implementation, for a similar reason to the one given in (Wiggers *et al.*, 2007).

An efficient implementation of and SpMV kernel is presented, which exploits the ability to access the diagonal terms directly, without using the indirection of the CRS structure. Since the destination vector is read from and written to, it must be ordered in memory so that a trade-off between efficient read operations and efficient write operations is achieved.

Performance results state that the implementation of the conjugate gradient solver is able to perform approximately 110 iterations per second using an NVidia Geforce FX card. Performance is thought to be limited by the SpMV product causing cache thrashing, due to its random pattern of memory access.

4.6 Implementing a GPU-enhanced cluster for Large-Scale simulations (Lucas *et al.*, 2007)

This paper describes the implementation of the multifrontal method using the GPU. Unlike the other implementations examined, this implementation does not execute the majority of the solver on the GPU, but instead uses the GPU as a co-processor to execute the most computationally-intensive operations. The multifrontal method is an important solver in Mechanical Computer Aided Engineering (MCAE) applications, as it is more reliable than iterative solvers, which may not converge for the systems which arise in this area. The multifrontal method (Duff & Reid, 1983) is an algorithm to factorise sparse symmetric matrices by decomposing the process into factorisations of smaller matrices. The algorithm exploits the fact that there are no dependencies between certain parts of the solution.

To solve the system it is sufficient to perform all computations using single precision arithmetic. The format which the matrix is stored in is not discussed - however, it is likely that some form of compressed storage is required, as the test problem consisted of 235,692 equations. Storing a sparse matrix of this size using a dense format would have wasted a large amount of memory.

It is stated that only the larger frontal matrices are factored on the GPU. Because of the overhead of transferring the matrix to the GPU and invoking a kernel, and then transferring the result back to main memory, it is faster to factor small matrices on the CPU. As the larger frontal matrices represent a significant proportion of the workload, it was possible for the GPU to take on 65% of the total factor operations whilst only factoring the 60 largest matrices.

The criteria for executing the factorisation of a matrix on the GPU are that it has a size greater than 127, or if its leading dimension (size plus degree) was greater than 1023. These criteria were determined by measuring the time taken to factor matrices of varying sizes on the CPU and GPU. It was found that as the sparsity of the matrix increases, the speedup offered by the GPU in factoring the matrix decreased.

Since the frontal matrices are dense, the CuBLAS routine `sgemm` was used to factorise them. This routine was chosen as it was able to achieve over 50%

of the peak floating-point performance of the GPU. It is unlikely that writing a routine from scratch to perform the same operation would outperform the CuBLAS routine without requiring a lot of work to tune the routine.

An overall speedup of approximately 1.97 was obtained for the CPU-GPU solver compared to using the CPU alone. Although this is a relatively small performance improvement compared to the speedup presented in other papers surveyed, it is a significant speedup which halves the execution time of the solver.

The multifrontal method is a very different approach to the iterative methods used in Fluidity and the other papers surveyed, and it is not considered to be worthwhile investigating. The multifrontal method is (like the conjugate gradient method) restricted to symmetric matrices. Because of this restriction it would be suitable to solve the systems which arise in the test program, but would not be able to solve the non-symmetric systems which arise in the main Fluidity package. Additionally, no steps have been taken towards using the multifrontal method in Fluidity, so further work would be required to extend Fluidity to use the multifrontal solver. This contrasts with iterative methods, which are already used in Fluidity, so it is straightforward to use one iterative method in place of another.

4.7 Conclusions

There are a number of areas which have been investigated in the literature which have so far not been explored with the Fluidity GPU solver. These areas are:

Preconditioning. (Buatois *et al.*, 2007) implemented a preconditioned conjugate gradient method. Presently, there is no preconditioning implemented in the GPU solver of Fluidity. The Jacobi preconditioner (as used in the paper) would be straightforward to implement, as it requires no calculation and a minimal amount of extra memory. Other preconditioners, such as the SSOR preconditioner should also be evaluated. However, other preconditioners require more storage space and computation to generate the preconditioner. A good preconditioner will strike a balance between the computation and storage requirements of the preconditioner, and the reduction in the number of iterations to convergence.

Matrix Storage. CNC (Buatois *et al.*, 2007) uses both the CRS and BCRS formats to store the matrix, and showed the best performance using the largest (4×4) block size. (Wiggers *et al.*, 2007) used a symmetric CRS format, which requires just under half of the storage space of the CRS format for a symmetric matrix, and may potentially increase perfomance by allowing a larger portion of the matrix to fit in a cache memory due to its reduced size. In (Bolz *et al.*, 2005) a modified CRS format is used, which exploits the structure of the matrices generated by the finite element method for Poisson problems. In Fludity, only a standard CRS format is used. As large performance gains were seen with the BCRS format in CNC, it is important to investigate the format to determine if it could also lead to performance gains for the Fluidity solver. The modified CRS format which stores elements of the diagonal contiguously may lead to a

small increase in performance in Fluidity, as it allows for the diagonal elements to be accessed without the indirection. As the performance gain is likely to be small, this should only be investigated if BCRS offers no speedup, as it would be difficult to implement in conjunction with BCRS.

Matrix Reordering. (Wiggers *et al.*, 2007) used the column count (sorting by the number of non-zero entries) algorithm for the GPU implementation of the solver, and the reverse Cuthill-McKee algorithm for the CPU solver. At present the Fluidity GPU solver does not reorder the matrix before solving. These re-orderings may be implemented in the solver in order to determine their effect upon the performance of the solver. Although the reverse Cuthill-McKee algorithm did not affect the fill rate of the blocks in BCRS, moving the non-zeroes as close to the diagonal as possible may improve the spatial and temporal locality of values in an SpMV product operation. Alternatively, the column count reordering may allow threads which are close to one another to take similar branches (due to having similar numbers of elements to work on), which would improve performance by avoiding processors idling.

Single-precision arithmetic. (Perryman & Kelly, 2008) states that the convergence of the conjugate gradient solver in Fluidity is poor. However, the papers reviewed have achieved convergence using single-precision arithmetic. This does not necessarily mean that there is a problem with the GPU-based solver - it is possible that the systems generated by the test problem are difficult to solve, whereas the authors of the papers would have picked systems which are easier to solve, in order to demonstrate the effectiveness of their solvers (see the next chapter for more information on the convergence of the GPU-based solver). As current (and possibly future) GPUs only have a small number of double-precision processors compared to the number of single-precision processors, it is important to explore the possibility of executing as much computation in single-precision as possible, to increase the amount of parallelism in the execution of the solver.

Vectorising. (Buatois *et al.*, 2007) and (Bolz *et al.*, 2005) used the `float4` data type in order to read four single-precision values from memory concurrently. This is currently not possible to implement in the Fluidity solver, as it operates in double-precision. However, if the Fluidity solver could be improved to provide better convergence with single-precision arithmetic, then vectorising would be very likely to increase the performance of the solver by using the available memory bandwidth more effectively.

Preconditioning and the BCRS storage format were chosen for experimental investigation. These two optimisations were both chosen as it was expected that they would provide high performance gains whilst requiring a relatively small amount of effort to investigate. The following chapter provides details of these investigations. Chapter 6 presents an outline of how the other optimisations may be investigated further.

Chapter 5

Experimental Investigations

5.1 Introduction

In this chapter, the results of experimental investigations into two of the previously identified areas are described. First, the reasons why the implementation of a preconditioner in the GPU-based solver is necessary is given. The design and implementation of the preconditioner are subsequently explained. An analysis of the performance and accuracy of the GPU-based solver with preconditioning is presented, and an examination of the limitations of the preconditioner and necessary further developments are given. In the latter section of this chapter, an investigation into using the BCRS storage format is described, which evaluates the potential of the BCRS format to bring about an improvement in performance of the GPU-based solver.

5.2 Preconditioning

5.2.1 Background

The prototype GPU solver showed promising results - Figure 5.1 shows a comparison of the time taken for the PETSc solver to solve problems in the test program and the time taken by the GPU-based solver to solve the same problems. However, there are two issues with the original GPU-based solver:

1. The GPU-based solver fails to converge on a solution for test matrices which are larger than approxmately $130,000 \times 130,000$ elements.

2. The test program which incorporates the GPU-based solver computes a solution which varies across the domain in a similar way to the original, but the absolute values differ to the original solution.

The first issue does not necessarily indicate an error in GPU-based solver. It was noted in (Perryman & Kelly, 2008) that the PETSc-based solver also does not appear to converge well on the test problem.

The second issue can be illustrated by examining the solutions produced by the original test program and the test program which uses the GPU-based solver. Figure 5.2 illustrates the difference in the solutions. Figure 5.2(a) gives

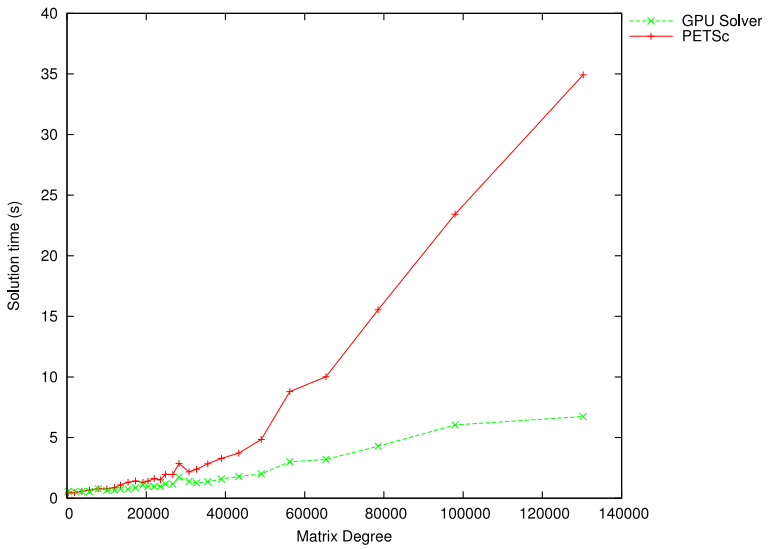
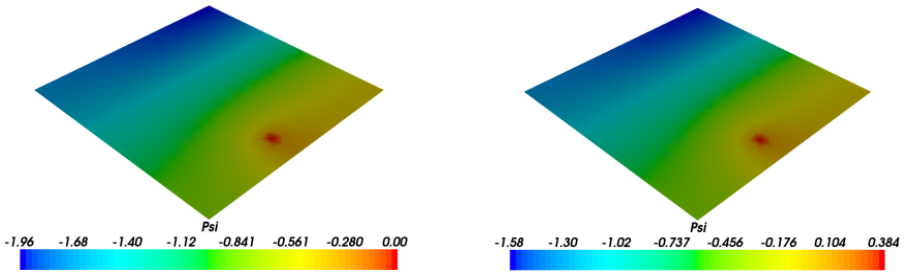


Figure 5.1: Time taken to solve systems assembled in the test program using the PETSc solver and the original GPU-based solver. (Data sourced from (Perryman & Kelly, 2008))

a graphical representation of the solution produced by the original program, and Figure 5.2(b) gives the same for the modified test program. It can be seen from the figure that the solutions differ by a constant.



(a) Original (b) Modified

Figure 5.2: Solutions produced by the original test program and a modified version which uses the GPU solver.

This issue has arisen due to a modification made to the test problem which affected the solution to the system. In order to explain the modification, it is necessary to provide some background information about the implementation of the test program.

In the original test program, the value of the solution at node 1 was fixed at 0 in order to guarantee that the solution to the system of equations has a

unique solution. This is normally achieved by setting $A_{11} = 1$, $A(1, n) = 0$ and $A(m, 1) = 0$ for all $n, m > 1$, and setting $b_1 = 0$. However, setting an entire column to zero in a matrix stored using the CRS format is difficult to achieve (Ham, 2008). This issue is avoided in the test program by setting $A_{11} = 10^{292}$ and $b_1 = 0$. The effect of this modification is that A_{11} is so large that the other entries in row 1 and column 1 are insignificant in comparison. This effectively makes these other elements vanish.

It may be expected that inserting a large number into the matrix would cause an overflow in the solver. However, this is not an issue in practice due to the operation of the preconditioner. Recall that the Jacobi preconditioner builds a preconditioning matrix based on the entries of the main diagonal of the matrix A. This preconditioning matrix is then inverted, before being multiplied with the matrix A. The net result of these operations is that each row and column of the matrix A is divided by the value of its diagonal entry. This cancels out the very large value at A_{11} and reduces it down to exactly 1. Additionally, the entries on the first row and column of A are divided by this very large number, so they are set to be very close to zero. The mechanism just described provides an efficient method to zero the first row and column of the matrix which requires little effort to implement.

Other preconditioners (such as the SSOR preconditioner, Section 2.3.1) also incorporate the diagonal preconditioner, so they have a similar effect in cancelling the large entry down to 1 and other entries in the row and column down to close to zero. However, the large value presents a major problem for an iterative method which does not use a preconditioner.

The GPU solver used by the modified test program did not implement a preconditioner, which meant that the large entry of A caused overflow errors in the calculations. In order to overcome this problem, a change was made to the modified test program, which, instead of setting $A_{11} = 10^{292}$, set $A_{11} = 1$. The result of this change is that the modified program assembled a system which does not have a unique solution, and the matrix A is referred to as *singular*. When the matrix is singular, the conjugate gradient method may converge on any of an infinite number of possible solutions (Shewchuk, 1994). It is also possible that the method may not converge, as the system of equations is not positive definite. This also explains why the PETSc solver did not appear to converge well when trying to solve the test problem.

This modification provides an explanation for both of the issues encountered in the GPU-based conjugate gradient solver. The first issue arises because systems which are not positive definite are constructed, which causes the conjugate gradient method to fail to converge. The second issue arises because the conjugate gradient method iterates towards any one of an infinite number of possible solutions when the matrix is singular, as there is no unique solution for the method to iterate towards.

Because these two issues arise when the problem is modified, it can be concluded that the modified version of the problem is not a suitable problem for comparing implementations of the conjugate gradient method. Additionally, because the unpreconditioned GPU based conjugate gradient solver cannot handle the unmodified test problem, it cannot be fairly evaluated against the PETSc solver. It is therefore necessary, before further development and evaluation of the GPU-based conjugate gradient solver can proceed, to modify the solver so that it implements a preconditioner.

5.2.2 Implementation

Because the conjugate gradient solver requires the matrix to be symmetric, the symmetry-preserving method of applying the preconditioner must be used (Equation 2.2). The Jacobi preconditioning matrix M is a diagonal matrix, so E can be computed by taking the square root of all the entries of the diagonal. Then, $E = M^{\frac{1}{2}}$ which (because M is diagonal) satisfies the equation $M = EE^T$ as required. Additionally, $E = E^T$, so $E^{-1} = E^{-T}$. Therefore, it is only necessary to compute and store E^{-1}. Because E is a diagonal matrix, E^{-1} can be computed by dividing 1 by each element of E.

Algorithm 2 is the resulting algorithm for calculating the preconditioning matrix and applying it to transform the system of equations. Once the system is solved, the solution must be transformed to find the solution to the original system.

Algorithm 2: Jacobi Preconditioning

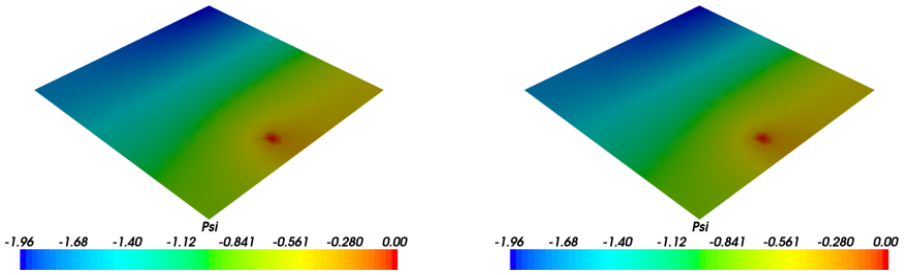
$E^{-1} = E^{-T} =$ An $n \times n$ zero matrix;
for $i = 1 \ldots n$ **do** `/* Calculate` E^{-1} `and` E^{-T} `*/`
  $E_{ii}^{-1} = E_{ii}^{-T} = \frac{1}{\sqrt{A_{ii}}}$;
for $i = 1 \ldots n$ **do** `/* Left Preconditioning` A `and` b `*/`
  $b_i = E_{ii}^{-1} \times b_i$;
  for $j = 1 \ldots n$ **do**
    $A_{ij} = E_{ii}^{-1} \times A_{ij}$;
for $i = 1 \ldots n$ **do** `/* Right Preconditioning` A `*/`
  for $j = 1 \ldots n$ **do**
    $A_{ji} = E_{ii}^{-T} \times A_{ji}$;
Solve the resulting system $Ax = b$ *using the CG method*;
for $i = 1 \ldots n$ **do** `/* Transform the solution */`
  $x_i = E_{ii}^{-T} \times x_i$;

An implementation of this algorithm was added to the GPU-based conjugate gradient solver. Parallelism in the algorithm was exploited by dividing the workload of the loop between many threads. This modified GPU-based conjugate gradient solver was integrated into the original test program.

Upon initial testing, it was found that setting the value $A_{11} = 10^{292}$ caused an overflow error. Although the GPU-based solver performs computation in double precision, the matrix values are stored in single-precision to minimise storage requirements and relieve pressure on the memory bandwidth. A consequence of this design decision is that setting a value in the matrix to a very large number results in an overflow when the value is converted to single-precision. In order to remedy this issue, the value 10^{38} is used, which does not overflow single precision arithmetic, but still has the desired effect of making all other entries on the row and column effectively equal to zero.

5.2.3 Testing

The preconditioned GPU-based solver was tested to ensure that it produced solutions which were similar to those produced by the unmodified test program. Testing was performed by generating problems of various sizes using the `triangle` program (see Section 5.3.1 for more description of the `triangle` program). The original and modified test programs were executed to compute solutions to the problem. The solutions produced by the two programs were then compared. The modified program produced solutions which differed from the original program by a maximum of 10^{-2}. Further analysis of the error in the GPU-based solver's solution follows. A graphical representation of the solutions found by the original and preconditioned GPU solver is given in Figure 5.3. It can be seen from the legends that the numeric value of the solutions produced by both solvers is approximately equal. This is in contrast to the solutions found by the unpreconditioned GPU solver with the modified test program, which did not agree.



(a) Original (b) GPU, with preconditioner

Figure 5.3: Solutions produced by the original test program and the test program using the GPU solver which includes preconditioning.

Performance Analysis

The performance of the GPU-based solver and the PETSc solver were also compared. In making the comparison, the time taken for uploading the problem to the GPU and downloading the solution back from the GPU was included, to give a more accurate impression of the performance of the GPU-based solver. Additionally, the PETSc solver was configured to use the Jacobi preconditioner with the conjugate gradient method, to produce a like-for-like comparison. The hardware which the solvers were tested on had the following specification:

- One core of an Intel Core 2 Duo E8400 processor running at 3GHz with 6MB of L2 cache.
- 2GB of main memory.
- One NVidia Geforce GTX 280 GPU.

Figure 5.4 shows the time taken for each solver to solve systems of various sizes. It can be seen from the graph that the modified GPU-based solver is capable of solving systems of equations which consist of 2.5 million simultaneous

equations. This is a great improvement on the original GPU solver/modified test program, which did not converge for systems of equations containing more than approximately 130,000 equations. The upper bound on the number of equations which can be solved is not known, as the `triangle` program runs out of memory on the test machine when generating meshes for larger problems.

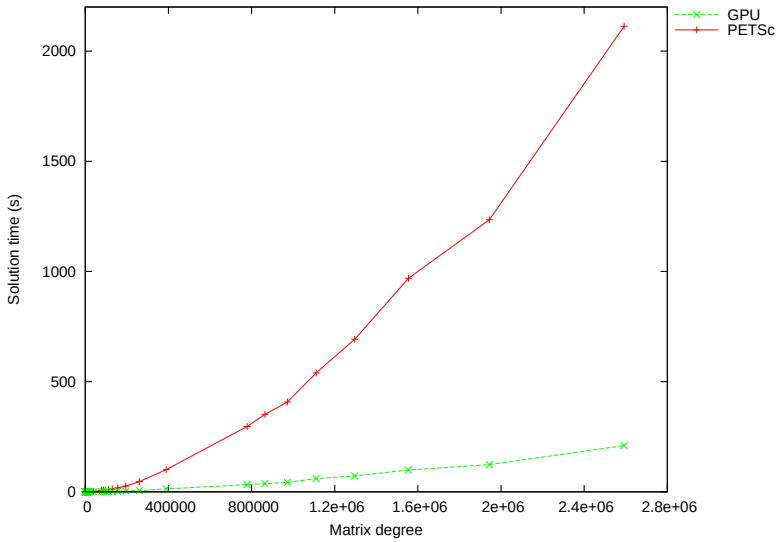


Figure 5.4: Time taken to solve systems assembled in the test program using the PETSc solver and the preconditioned GPU solver.

It can be seen that the performance benefit of the GPU-based solver increases with the size of the problem. For the largest problem tested, of 2,592,182 equations, the GPU solver required 210 seconds to solve. The PETSc solver running on the CPU required 2112 seconds to solve the system of equations, an order of magnitude slower.

The performance of the solver may also be analysed by calculating the total number of floating point operations (FLOPs) executed per second. A method of calculating the number of FLOPs per iteration of the conjugate gradient solver, and the amount of data it uses in loading and storing data to perform calculations is required to estimate the performance.

First, the algorithm must be examined to identify its main operations. By inspection, it was determined that the main loop of the conjugate gradient algorithm contains the following operations:

- One matrix-vector multiplication.
- Two vector dot products.
- Three additions of a vector to another vector multiplied by a scalar (referred to as AXPY).

The number of FLOPs which is required to compute the result of each of these operations was determined as follows:

Matrix-vector Product. In general a matrix-vector product would require n^2 multiplications for an $n \times n$ matrix. However, when using the CRS format, it is possible to avoid performing computations with the zero elements of the matrix. The matrices generated in the test program contain approximately 6.8 non-zeroes per row (see Section 5.3 for examples of matrices which are assembled in the test program). An estimate of the number of multiplications required is obtained by multiplying the number of rows in the matrix, n, by 6.8. Additionally, elements of the result vector will require 6.8 additions each, as the sum of each of the multiplications on one row is computed. Therefore, the matrix-vector multiplication requires $2 \times 6.8n$ FLOPs, or $13.6n$ FLOPs.

Vector Dot Product. Elements from each vector are multiplied together, so n multiplications are required for a vector of length n. The sum of all the multiplications is also computed, so n additions are required. The dot product required a total of $2n$ FLOPs.

AXPY. Multiplying each element of one of the vectors by a scalar requires n multiplications for a vector of length n. Each element of the result is added to a corresponding element in another vector, so an additional n additions are required. The AXPY operation requires a total of $2n$ FLOPs.

The total number of FLOPs required for one iteration of the main loop can be calculated by summing the FLOPs required by the operations which make up the main loop. The total number of FLOPs required for one iteration is:

$$\begin{aligned} & 13.6n && \text{(Matrix-vector product)} \\ + \; & 2 \times 2n && \text{(Two dot products)} \\ + \; & 3 \times 2n && \text{(Three AXPYs)} \\ = \; & 23.6n \end{aligned}$$

The amount of data transferred which is used by computations inside the main loop of the conjugate gradient solver may be calculated in a similar manner. The number of bytes transferred throughout one iteration of the main loop is derived as follows:

Matrix-vector Product. In calculating the matrix-vector product, each element of the matrix is loaded exactly once. As there are approximately 6.8 non-zeroes per row, $6.8n$ single precision floating point values must be loaded to retrieve the entire matrix. As a single precision value requires four bytes of storage, this means that $27.2n$ bytes are loaded. Approximately 6.8 entries are loaded from the source vector per row. However, the source vector is stored in double precision format, so twice as many ($54.4n$) must be loaded for the source vector. Finally, the destination vector (also stored in double precision format) is loaded and stored 6.8 times per row, as the results of multiplying elements from the source vector and matrix row are repeatedly added to the element in the destination

vector. This requires another $54.4n$ bytes to be loaded and $54.4n$ bytes to be stored. The total data transfer for the matrix-vector product is $190.4n$ bytes.

Vector Dot Product. When computing the dot product, each element of two vectors of length n stored in double precision format are loaded. This requires $16n$ bytes to be loaded. Additionally, the sum of all these results is computed, which must be stored once and loaded once for each intermediate computation. This requires another $8n$ bytes to be loaded and $8n$ bytes to be stored. The total number of bytes transferred in executing the dot product is $32n$ bytes.

AXPY. The AXPY also required two vectors of length n stored in double precision format to be read. This requires $16n$ bytes to be loaded. Additionally, the result is stored back to a vector, which requires $8n$ bytes to be stored. The total number of bytes transferred in executing the AXPY operation is $24n$ bytes.

The total amount of data transferred during one iteration of the main loop is calculated as follows:

$$\begin{aligned} & 190.4n && \text{(Matrix-vector product)} \\ + \; & 2 \times 32n && \text{(Two dot products)} \\ + \; & 3 \times 24n && \text{(Three AXPYs)} \\ = \; & 326.4n \end{aligned}$$

The number of iterations required for convergence and the degree of the matrix were recorded for each problem size when the GPU-based solver was tested. Using this information, it is possible to estimate the total number of FLOPs executed during the execution of the main loop of the conjugate gradient method using the following equation:

$$FLOPs = Degree \times Iterations \times 23.6$$

It is also possible to estimate the total number of bytes transferred throughout the execution of the main loop:

$$Bytes = Degree \times Iterations \times 326.4$$

Using information about the time spent in the main loop of the solver, it was possible to calculate an estimate of the number of GigaFLOPs (FLOPs $\times 10^9$) per second achieved by the solver in the main loop. Additionally, an estimate of the transfer rate of values which are used in the main loop of the conjugate gradient solver is calculated. These estimates are shown in Figures 5.5 (GFLOPs per second) and 5.6 (Gigabytes transferred per second).

It appears from inspecting these graphs that the FLOPs per second achieved and the GB transferred per second are in exact proportion. However, one should not necessarily conclude that one of these results depends on the other to a great extent. Although there is likely to be some dependence, these two measures are very closely correlated because they are estimates which have

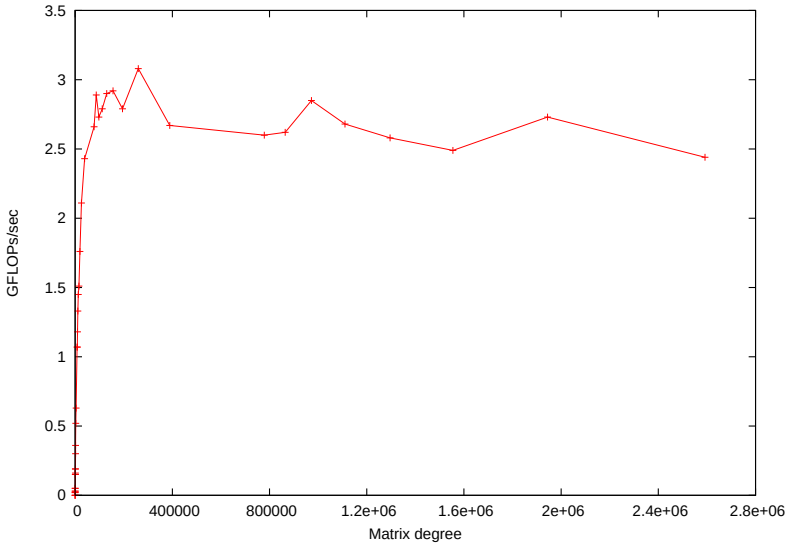


Figure 5.5: Estimated GigaFLOPs per second achieved when executing the GPU-based solver for varying problem sizes.

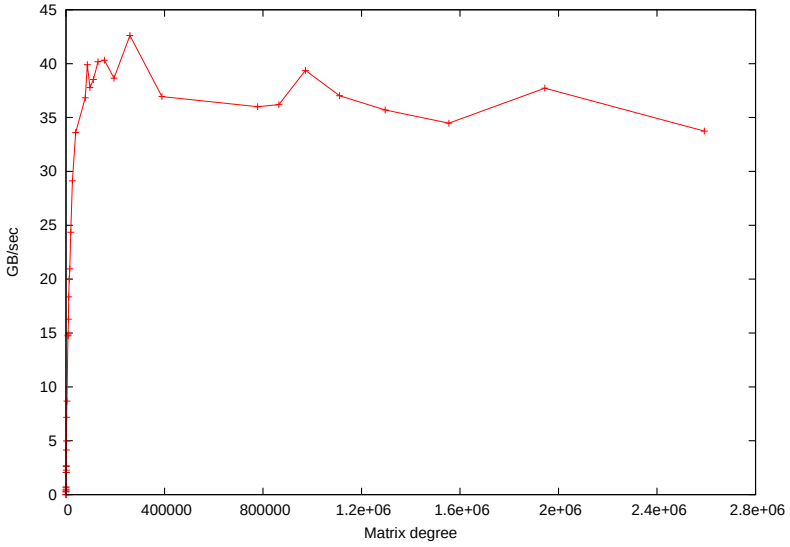


Figure 5.6: Estimated Gigabytes transferred per second when executing the GPU-based solver for varying problem sizes.

been generated based on exactly the same information. Also, when interpreting the graphs, the very low values to the left of the graph are discounted, as they represent the throughput for very small problems, for which overheads represent a disproportionate amount of the execution time.

The maximum double precision floating point performance of the NVidia 280 GTX GPU is 90 GFLOPs per second (Heise, 2008). The maximum estimated performance of the solver is just over 3 GFLOPs per second, which is very low utilisation of the maximum performance. This level of performance is greater than that reported in (Buatois *et al.*, 2007). However, older hardware was used in benchmarking the CNC.

The maximum memory bandwidth of the NVidia 280 GTX GPU is approximately 141GB per second (Heise, 2008). The utilisation of the maximum memory bandwidth is between approximately 25% and 30% of the maximum. It is thought that the low utilisation is caused by accesses into the source vector, which follow a random pattern. High transfer rates are only achieved on the NVidia GPU when accesses follow a sequential pattern.

Because the memory bandwidth is limited by the memory access pattern, it is thought that the low level of GFLOPs/sec is the result of processors having to wait for data to arrive before they can perform computation. Additionally, the indirection required to access elements of the matrix requires extra bandwidth and computations. It is likely that optimisations which increase the spatial locality of accesses into the source vector will result in improved bandwidth utilisation and GFLOPs per second, as fetching adjacent elements allows the utilisation of memory bandwidth on the GPU to be maximised.

Error Analysis

An analysis of the error in the final solution provided by the GPU-based solver was also analysed. A comparison between the error produced by the GPU-based solver and the PETSc solver is shown in Figure 5.7.

It appears from this graph that the PETSc error is zero for almost all solutions - however, the PETSc error is approximately two orders of magnitude smaller than the GPU-based solver error, which makes it appear to be almost zero on the graph. Experimentation with decreasing the value of ϵ (the error tolerance) in the conjugate gradient method did not decrease the error produced by the GPU-based solver.

It is thought that the larger error is introduced by converting the matrix entries to be stored in single precision format. When the GPU-based solver was originally designed, it was felt that this would not introduce a large error, as the matrix values are not re-computed throughout the execution of the solver, so any truncation error would not be compounded (Perryman & Kelly, 2008). However, it appears that storing the matrix entries in double-precision floating-point format should be investigated, in order to improve the accuracy of the GPU-based solver.

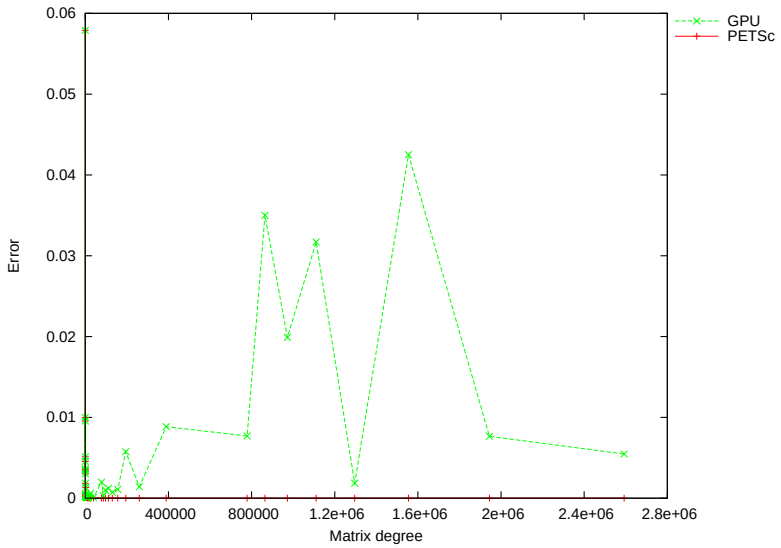


Figure 5.7: Error in the solution of systems assembled in the test program using the PETSc solver and the preconditioned GPU solver.

5.2.4 Further Work

Eliminating explicit computation of $E^{-1}AE^{-T}$

The algorithm implemented explicitly computes $E^{-1}AE^{-T}$. Although this is not an issue when using the Jacobi preconditioner because it is straightforward to apply, using other preconditioners (such as SSOR) may make $E^{-1}AE^{-T}$ difficult to compute explicitly, as they will affect the sparsity pattern of the matrix. Rebuilding the CSR representation of a new matrix is a costly operation, and should be avoided if at all possible. An alternative is to use a variant of the conjugate gradient method, called the *Preconditioned Conjugate Gradient Method* (Shewchuk, 1994). This method does not require $E^{-1}AE^{-T}$ or $M^{-1}A$ to be computed. Instead, the method is modified so that the matrix M^{-1} is applied to certain vectors during the iteration.

Implementing other preconditioners

Because the test program uses the conjugate gradient method with an SSOR preconditioner by default, it is expected that the SSOR preconditioner gives higher performance than the Jacobi preconditioner. To investigate this hypothesis, the original test program was executed using the Jacobi preconditioner and using the SSOR preconditioner for the same problem sizes. The performance of the solver using these two preconditioners was then compared. Figure 5.8 shows the performance of the PETSc solver when using each of these preconditioners.

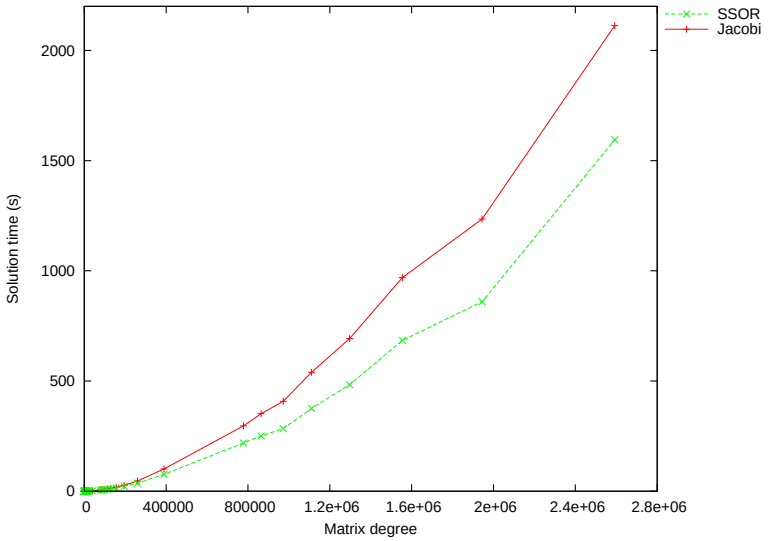


Figure 5.8: Time taken for PETSc to solve various sizes of system using the Jacobi and SSOR preconditioners.

It is clear that the SSOR preconditioner results in better performance than the Jacobi preconditioner. For larger problems, using the SSOR preconditioner decreases the execution time by up to 30%. This presents a strong case for developing and testing other preconditioners in the GPU-based solver code, as the development effort required is small, and will clearly provide large performance gains.

5.3 The BCRS Matrix Format

Using the BCRS storage format has been shown to be effective at improving the performance of iterative solvers, and the SpMV kernel in general. Implementing the BCRS format in Fluidity and the GPU-based solver will require a significant amount of effort. In order to explore the potential of the BCRS format to improve the performance of the solver without having to spend time on its implementation, the effect of BCRS storage of matrices assembled in the test program has been examined.

5.3.1 Test Matrices

Three test matrices (A, B and C) are used for evaluation in this section. The test matrices were generated using the test program and the `triangle` program. The `triangle` program creates a finite element mesh over a given domain built from triangular elements. The *Maximum Triangle Area* (MTA) is passed to the `triangle` program as a parameter. This parameter constrains the size of the

elements. When a smaller MTA is given, a finer mesh is produced, which leads to a larger number of finite elements. The test program inputs this mesh then assembles the matrix. Once the matrix is assembled, it is output to disk. Table 5.1 shows the degree, MTA, number of non-zero matrix elements, and the density of these matrices.

Matrix	MTA	Degree	Non-zeroes	Density	Sparsity Pattern
A	0.01	87	567	7.49%	Figure 5.9
B	0.001	820	5632	0.83%	Figure 5.10(a)
C	0.0005	1620	11104	0.42%	Figure 5.10(b)

Table 5.1: Characteristics of the test matrices

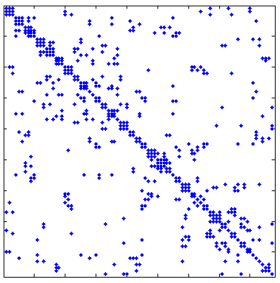
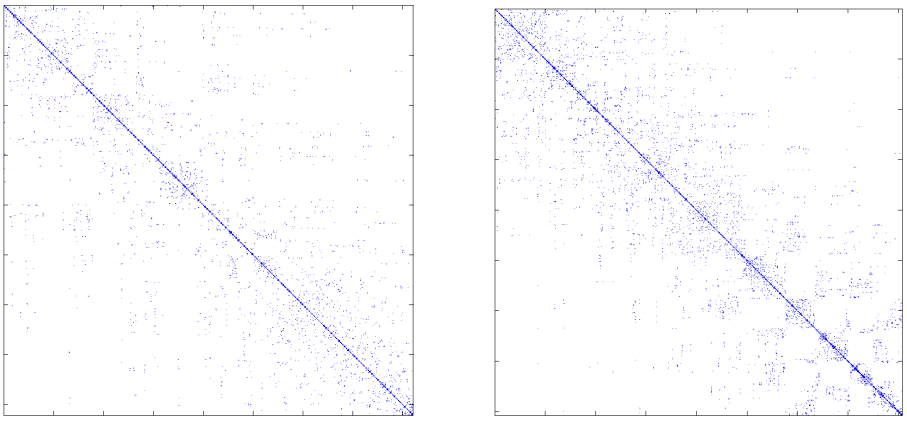


Figure 5.9: Sparsity pattern of test matrix A.



(a) Test matrix B (b) Test matrix C

Figure 5.10: Sparsity pattern of test matrices B and C.

5.3.2 Exploring the BCRS format

The potential of the BCRS format to improve the performance of the solver has been assessed by examinining what the block fill ratio for the test matrices is, and comparing it to the fill ratio of the example matrices given in (Buatois *et al.*,

2007). The fill ratio is calculated for block sizes between 2×2 and 10×10, in order to ensure that enough block sizes have been examined to decide whether the performance of a BCRS implementation is worth the development effort.

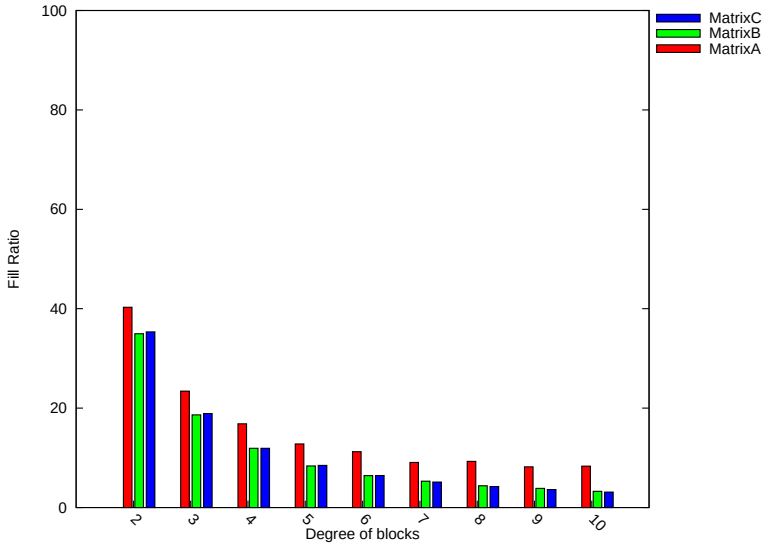


Figure 5.11: Fill Ratio of blocks in the BCRS storage format for varying block sizes.

Figure 5.11 shows the fill ratio of each block size for each of the test matrices. As can be expected, the smaller block sizes have a greater filling ratio. The filling ratio of the matrices appears to be poor - for the 2×2 block size the average is under 40%. The 4×4 block size is even worse, with a fill ratio under 15%.

It can be noticed that the matrix A has a higher fill ratio than the matrices B and C for each block size. Because the matrix A is quite small in comparison to the other matrices, it is possible that it does not provide an accurate representation of the fill ratio of matrices which may arise in the test program. Despite the matrix C having approximately twice the dimension of matrix B, the matrices B and C show similar fill ratios for each block size.

The example matrix used in (Buatois *et al.*, 2007) for smoothing showed a fill ratio of approximately 38% with the 4×4 block size - using this block size gave the best performance. However, in the case of the test matrices, the only block size with a fill ratio close to 40% for the larger matrices (B and C) is the 2×2 block size. It can be concluded that the block size most likely to give a performance improvement is the 2×2 block size. Larger block sizes are likely to introduce too much padding, which will lead to an excessive amount of redundant computation, and unnecessarily use memory bandwidth.

Because only the 2×2 block size is likely to generate any performance improvement, the development of a solver which uses the BCRS format should not be regarded as a high priority. If the amount of padding required out-

weighs the reduction in indirection, then there is no smaller block size which can be used, so the implementation of the BCRS format will have been a waste of effort. However, when another solver (e.g. GMRES) which can solve problems which arise in the main Fluidity codebase is developed, the BCRS storage format should be revisited. Matrices which are assembled in solving these problems may exhibit a higher fill ratio when stored in the BCRS format than the matrices generated by the test problem.

5.4 Conclusion

We have seen that the development of a preconditioner for the GPU-based solver has made an improvement which allows the unmodified test problem to be solved, and that this has produced results which are numerically similar to the PETSc solver. Additionally, the GPU-based solver can solve systems up to an order of magnitude faster than the PETSc solver, and can solve systems of over 2.5 million equations. Although the solver shows promising performance results, analysis has shown that the utilisation of processing units and memory bandwidth is poor. This indicates that with further development and optimisation, there is potential to increase the speed of the solver much further. The accuracy of the solver has been seen to be limited, so further development is required to increase the accuracy of the solution. Additionally, the case for extending the solver to implement other, better-performing preconditioners have been seen.

Investigations into the BCRS storage format have revealed that it is unlikely to give a significant performance increase to the solver for the test problem. However, the main Fluidity program may be more amenable to being accelerated by using BCRS as a storage format. When a general GPU-based solver is developed to solve systems in Fluidity, the BCRS storage format should be revisited.

Chapter 6

Conclusions and Further Work

6.1 Conclusions

A survey and evaluation of the literature on iterative solvers for classical and GPU architectures has been presented. Experimental evaluations of potential optimisations have been undertaken.

The main component of iterative solvers, the SpMV kernel, has been discussed. Its performance issues have been identified, and commonly-used optimisations for classical and vector architectures have been discussed. Many of these optimisations may be applied to SpMV kernels for GPUs.

Existing implementations of GPU-based solvers described in the literature have been reviewed to identify optimisations which have shown performance benefits. Two of these optimisations were selected for experimental investigation. Other potentially beneficial optimisations are described in the next section.

The GPU-based conjugate gradient code has been extended in order to allow it to solve the original problem for the test program, by implementing preconditioning. The solver is now capable of solving very large systems of equations, consisting of over 2.5 million equations. The performance of the solver has been compared to the PETSc conjugate gradient solver, and it has been seen that the GPU-based solver converges up to an order of magnitude more quickly than the PETSc solver.

Analysis of the performance of the GPU-based solver in terms of its computational and memory throughput have shown that there is room for optimisations which make better use of the GPU hardware. Presently its utilisation of the available processing units and memory bandwidth is low. Optimisations which may improve the performance of the solver are part of further work.

The numerical performance of the solver, whilst stable, is not as accurate as the PETSc solver. Frequently the PETSc solver produces solutions with an error which is two orders of magnitude smaller than the error from the GPU-based solver. Further work is required to reduce this error.

The BCRS storage format has been examined to determine whether its implementation is likely to improve the performance of the GPU-based solver. It

is concluded that matrices assembled by the test problem are too sparse for the solver to benefit from an implementation of the BCRS format.

Further work involves investigation in a variety of different directions, and is described in the following section.

6.2 Further Work

Further work is divided into two categories: that which should proceed in order to discover how to further accelerate the solver for the test problem, and that which should proceed once the full Fluidity package is to be accelerated by implementing an alternative iterative solver. Work to accelerate the test problem should proceed first, as it is more straightforward to work with - it is expected the knowledge and experience gained from this development will ease the development of more advanced methods for the full Fluidity package. Further work to accelerate the test problem involves:

Implicit Preconditioning. It has been seen in Section 5.2.4 that explicitly calculating the matrix of coefficients may present a problem for preconditioners other than the Jacobi preconditioner. This issue can be avoided by modifying the GPU-based solver to implement the preconditioned conjugate gradient method.

Other Preconditioners. In Section 5.2.4 it was also shown that alternative preconditioners can decrease the time taken by the solver by up to 30%. Once the preconditioned conjugate gradient method is implemented, alternative preconditioners, such as the SSOR preconditioner should be implemented. Additionally, to facilitate development of other preconditioners, an interface for developing preconditioners, and a mechanism to select the preconditioner at runtime should be implemented.

Reordering Optimisations. The Cuthill-McKee and Column Count algorithms have yet to be experimentally tested. The Cuthill-McKee algorithm may be evaluated with little implementation effort by applying the algorithm to the mesh output from `triangle` before it is used as input to the test program. Implementations of the algorithm which operate on `triangle` output are freely available, such as (Stahel, 2008).

Software Prefetching. An area of the solver which is most likely to benefit from prefetching is the SpMV kernel. As accesses into the source vector follow a random pattern, a prefetcher should fetch a portion of the source vector into the shared memory, where it can be accessed very quickly. Additionally, combining this optimisation with the RCM re-ordering algorithm may minimise the portion of the source vector required for each element of the result.

Matrix Storage. The preconditioned GPU-based solver produces solutions which are correct but have larger error terms than the PETSc solver. This is likely to be the result of the matrix being stored in single-precision format. Storing the matrix in double precision format should be tested to determine if this increases the accuracy of the solution produced by the GPU-based solver. A potential drawback of using double precision for the matrix is

that texture memory, where the matrix is stored, does not support double precision values (NVidia, 2007b). However, a possible workaround could be developed by using an `int2` type (which is supported) and casting it to a double after fetching:

```
static __inline__ __device__ double fetch_double
                             (texture<int2,1> val, int elem)
{
  int2 v = tex1Dfetch(val, elem);
  return __hiloint2double(v.y, v.x);
}
```

Single Precision Arithmetic. It has been shown that other implementations of GPU-based solvers perform computation using single precision arithmetic, and achieve convergence. However, it appears likely that the test problem requires double precision to find an accurate solution. Because there are eight times more single precision processors than there are for double precision an NVidia 280GTX GPU, it is likely that performance would be increased if computations could be performed in single precision. Efforts have been made to develop solvers which obtain high speed whilst maintaining accuracy by computing using a mixture of single and double precision (Göddeke *et al.*, 2005), (Buttari *et al.*, 2008). Development of the GPU-based solver to use mixed precision may follow techniques outlined in the existing literature in this area.

Vectorising. Should a mixed-precision GPU-based solver be implemented, its performance may be further increased by using vector data types, such as the `float4` data type. This will allow greater utilisation of available memory bandwidth, leading to an overall increase in performance.

Other areas may be investigated in the longer-term - presently, their development will not have a great impact on the solution to the test problem, but may have benefits when solving systems assembled by the main Fluidity package:

Other Methods. In particular the GMRES method will eventually have to be implemented using the GPU to solve the systems which are assembled in Fluidity. Once the conjugate gradient solver has been thoroughly investigated and tested, the development of a GPU-based GMRES solver should not require a large effort. Kernels which make up the majority of the method (including the SpMV) kernel may be re-used from the conjugate gradient solver.

Multi-GPU Solvers. In order to solve huge problems, it will be necessary eventually to develop a solver which partitions the problem and distributes computation across multiple GPUs. An early version of an implementation of a conjugate gradient solver which uses multiple GPUs is presented in (Okuda & Georgescu, 2008). Large performance gains are demonstrated when using multiple GPUs, which provides an indication that a multiple GPU solver is feasible.

BCRS Storage. Although the BCRS format shows little benefit for the conjugate gradient solver when solving the test problem, this does not imply that solving systems of equations generated by Fluidity will not benefit from storing the matrix using the BCRS format. When a GPU-based solver for Fluidity is in development, matrices assembled in Fluidity should be tested to determine the fill ratio of blocks in the BCRS format. A large fill ratio will indicate that the solver may show increased performance using BCRS over the standard CRS format.

Finally, two optimisations have been identified which are unlikely to provide a significant performance increase relative to the effort of their implementation. It is unlikely that further investigation of these optimisations on GPUs is worthwhile.

JAD Format. Although the JAD format may increase performance, it is very different to the CRS storage format used in Fluidity. As high performance has been obtained from the GPU-based solver when using the CRS format, there is little need to explore the JAD format. This may appear to conflict with the suggestion that the BCRS format should be investigated - however, it is straightforward to assess the potential of the BCRS format to improve performance. Additionally, implementing the JAD format in Fluidity will require much more effort than implementing the BCRS format, as it is very dissimilar to the CRS format.

Cache Blocking Optimisations. Cache blocking optimisations seek to increase the hit rate of the cache throughout the execution of an SpMV kernel. However, GPUs have no cache, so there is no need to further investigate these optimisations. Instead, the GPU will be more likely to benefit from prefetching data into shared memory, which acts as a software controlled cache to some extent.

References

Advanced Micro Devices, Inc. 2008. *ATI Stream SDK User Guide*.

Balay, Satish, Buschelman, Kris, Eijkhout, Victor, Gropp, William D., Kaushik, Dinesh, Knepley, Matthew G., McInnes, Lois Curfman, Smith, Barry F., & Zhang, Hong. 2006 (Sept.). *PETSc Users Manual*. Tech. rept. ANL-95/11 - Revision 2.3.2. Argonne National Laboratory. see `http://www.mcs.anl.gov/petsc`.

Barrett, R., Berry, M., Chan, T. F., Demmel, J., Donato, J., Dongarra, J., Eijkhout, V., Pozo, R., Romine, C., & der Vorst, H. Van. 1994. *Templates for the Solution of Linear Systems: Building Blocks for Iterative Methods, 2nd Edition*. Philadelphia, PA: SIAM.

Bolz, Jeff, Farmer, Ian, Grinspun, Eitan, & Schröder, Peter. 2005. Sparse matrix solvers on the GPU: conjugate gradients and multigrid. *Page 171 of: SIGGRAPH '05: ACM SIGGRAPH 2005 Courses*. New York, NY, USA: ACM.

Buatois, Luc, Caumon, Guillaume, & Lvy, Bruno. 2007. Concurrent Number Cruncher: An Efficient Sparse Linear Solver on the GPU. *In: High Performance Computation Conference (HPCC), Springer Lecture Notes in Computer Sciences*. Award: Second best student paper.

Buttari, Alfredo, Dongarra, Jack, Kurzak, Jakub, Luszczek, Piotr, & Tomov, Stanimir. 2008. Using Mixed Precision for Sparse Matrix Computations to Enhance the Performance while Achieving 64-bit Accuracy. *ACM Trans. Math. Softw.*, **34**(4), 1–22.

Cormen, Thomas H., Leiserson, Charles E., Rivest, Ronald L., & Stein, Clifford. 2001. *Introduction to Algorithms*. Cambridge, MA, USA: The MIT Press.

Cuthill, E., & McKee, J. 1969. Reducing the bandwidth of sparse symmetric matrices. *Pages 157–172 of: Proceedings of the 1969 24th national conference*. New York, NY, USA: ACM.

D'Avezedo, E. F., Fahey, M. R., & Mills, R. T. 2005. Vectorized Sparse Matrix Multiply for Compressed Row Storage Format. *In: Proceedings of the 5th International Conference on Computational Science*. Springer-Verlag.

Duff, I. S., & Reid, J. K. 1983. The Multifrontal Solution of Indefinite Sparse Symmetric Linear. *ACM Trans. Math. Softw.*, **9**(3), 302–325.

Göddeke, D., Strzodka, R., & Turek, S. 2005. Accelerating Double Precision FEM Simulations with GPUs. *Pages 139–144 of:* Hülsemann, F., Kowarschik, M., & Rüde, U. (eds), *Simulationstechnique 18th Symposium in Erlangen, September 2005*, vol. Frontiers in Simulation. SCS Publishing House e.V. ASIM 2005.

Gorman, Gerard, Piggot, Matthew, & Farrell, Patrick. 2008. *About Fluidity*. http://amcg.ese.ic.ac.uk/index.php?title=FLUIDITY.

Goumas, Georgios, Kourtis, Kornilios, Anastopoulos, Nikos, Karakasis, Vasileios, & Koziris, Nectarios. 2008. Understanding the Performance of Sparse Matrix-Vector Multiplication. *Pages 283–292 of: PDP '08: Proceedings of the 16th Euromicro Conference on Parallel, Distributed and Network-Based Processing (PDP 2008)*. Washington, DC, USA: IEEE Computer Society.

Gschwind, Michael, Hofstee, H. Peter, Flachs, Brian, Hopkins, Martin, Watanabe, Yukio, & Yamazaki, Takeshi. 2006. Synergistic Processing in Cell's Multicore Architecture. *IEEE Micro*, **26**(2), 10–24.

Ham, David. 2008. *Regarding the 1,1 matrix entry in test_laplacian*. Email message sent 11th Dec 2008.

Heise. 2008. *Nvidia kratzt mit neuen Grafik- und Compute-Prozessoren an der TFLOPS-Schallmauer*. `http://www.heise.de/newsticker/Nvidia-kratzt-mit-neuen-Grafik-und-Compute-Prozessoren-an-der-TFLOPS-Schallmauer--/meldung/109495`.

Hestenes, Magnus, & Stiefel, Eduard. 1952. Methods of Conjugate Gradients for Solving Linear Systems. *Journal of Research of the National Bureau of Standards*, **49**(6), 409–436.

Khailany, Brucek, Dally, William J., Kapasi, Ujval J., Mattson, Peter, Namkoong, Jinyung, Owens, John D., Towles, Brian, Chang, Andrew, & Rixner, Scott. 2001. Imagine: Media Processing with Streams. *IEEE Micro*, **21**(2), 35–46.

Khronos Group, The. 2008. *OpenCL 1.0 Working Specification*.

Kincaid, D. R., & Young, D. M. 1988. A brief review of the ITPACK project. *J. Comput. Appl. Math.*, **24**(1-2), 121–127.

Lawson, C. L., Hanson, R. J., Kincaid, D. R., & Krogh, F. T. 1979. Basic Linear Algebra Subprograms for Fortran Usage. *ACM Trans. Math. Softw.*, **5**(3), 308–323.

Lin, S., & Kernighan, B. W. 1973. An Effective Heuristic Algorithm for the Traveling-Salesman Problem. *Operations Research*, **21**(2), 498–516.

Lindholm, Erik, Nickolls, John, Oberman, Stuart, & Montrym, John. 2008. NVIDIA Tesla: A Unified Graphics and Computing Architecture. *IEEE Micro*, **28**(2), 39–55.

Lucas, Robert F., Wagenbreth, Gene, & Davis, Dan M. 2007. Implementing a GPU Enhanced Cluster for Large-Scale Simulations. *In: Proceedings of The Interservice/Industry Training, Simulation & Education Conference (I/ITSEC)*.

NVidia. 2007a. *The CuBLAS Library*.

NVidia. 2007b. *NVIDIA CUDA Compute Unified Device Architecture - Programming Guide*.

Okuda, Hiroshi, & Georgescu, Serban. 2008. *Conjugate Gradients on a Multi-GPU System*. http://nkl.cc.u-tokyo.ac.jp/seminars/0810-WS/abstracts/okuda-a.pdf.

Perryman, Tristan, & Kelly, Paul H. J. 2008. *Accelerating Fluidity Using the GPU*. UROP Report. Imperial College London.

Pichel, J.C., Heras, D.B., Cabaleiro, J.C., & Rivera, F.F. 2004. Improving the locality of the sparse matrix-vector product on shared memory multiprocessors. *Parallel, Distributed and Network-Based Processing, 2004. Proceedings. 12th Euromicro Conference on*, Feb., 66–71.

Piggott, M. D. 2006. *Fluidity/ICOM Manual*.

Pinar, Ali, & Heath, Michael T. 1999. Improving Performance of Sparse Matrix-Vector Multiplication. *SC Conference*, **0**, 30.

Saad, Y. 2003. *Iterative Methods for Sparse Linear Systems, 2nd edition*. Philadelpha, PA: SIAM.

Salvi, Rodolfo. 2002. *The Navier-Stokes Equations: Theory and Numerical Methods*. Marcel Dekker.

Seiler, Larry, Carmean, Doug, Sprangle, Eric, Forsyth, Tom, Abrash, Michael, Dubey, Pradeep, Junkins, Stephen, Lake, Adam, Sugerman, Jeremy, Cavin, Robert, Espasa, Roger, Grochowski, Ed, Juan, Toni, & Hanrahan, Pat. 2008. Larrabee: a many-core x86 architecture for visual computing. *ACM Trans. Graph.*, **27**(3), 1–15.

Shahnaz, Rukhsana, Usman, Anila, & Chughtai, Imran R. 2005. Review of Storage Techniques for Sparse Matrices. *9th International Multitopic Conference, IEEE INMIC 2005*, Dec., 1–7.

Shaw, B., Ambraseys, N. N., England, P. C., Floyd, M. A., Gorman, G. J., Higham, T. F. G., Jackson, J. A., Nocquet, J. M. Pain, C. C., & Piggott, M. D. 2008. Eastern Mediterranean tectonics and tsunami hazard inferred from the AD 365 earthquake. *Nature Geoscience*, **1**, 268–276.

Shewchuk, Jonathan R. 1994. *An Introduction to the Conjugate Gradient Method Without the Agonizing Pain*. Tech. rept. Pittsburgh, PA, USA.

Stahel, Andreas. 2008. *An implementation of the Cuthill-McKee algorithm*. https://staff.hti.bfh.ch/sha1/pwf/fem/CuthillMcKee/.

Tiyyagura, Sunil R., Küster, Uwe, & Borowski, Stefan. 2006. Performance Improvement of Sparse Matrix Vector Product on Vector Machines. *Computational Science - ICCS 2006*, **3991**, 196–203.

Toledo, Sivan. 1997. Improving the memory-system performance of sparse-matrix vector multiplication. *In: IBM Journal of Research and Development*.

Tremblay, M., & Chaudhry, S. 2008. A Third-Generation 65nm 16-Core 32-Thread Plus 32-Scout-Thread CMT SPARC Processor. *Solid-State Circuits Conference, 2008. ISSCC 2008. Digest of Technical Papers. IEEE International*, Feb., 82–83.

Vuduc, Richard, & Moon, Hyun-Jin. 2005 (September). Fast sparse matrix vector multiplication by exploiting variable block structure. *In: Proceedings of the International Conference on High-Performance Computing and Communications*. LNCS 3726.

Wiggers, W. A., Bakker, V., Kokkeler, A. B. J., & Smit, G. J. M. 2007. Implementing the conjugate gradient algorithm on multi-core systems. *Pages 11–14 of:* Nurmi, J., Takala, J., & Vainio, O. (eds), *Proceedings of the International Symposium on System-on-Chip (SoC 2007), Tampere*. Piscataway, NJ: IEEE.

Williams, Samuel, Oliker, Leonid, Vuduc, Richard, Shalf, John, Yelick, Katherine, & Demmel, James. 2007. Optimization of sparse matrix-vector multiplication on emerging multicore platforms. *Pages 1–12 of: SC '07: Proceedings of the 2007 ACM/IEEE conference on Supercomputing*. New York, NY, USA: ACM.

Yelick, Kathy. 2008 (Dec.). *CS 267:Applications of Parallel Computers, Lecture 17*. http://www.cs.berkeley.edu/ yelick/cs267.