Krylov Subspace Solvers in OP2 on GPUs

Lesleis NAGY

Academic year 2010/2011

MSc in High Performance Computing
Edinburgh Parallel Computing Centre
The University of Edinburgh
Year of Presentation: 2011
Abstract

An extensible framework for the execution of Krylov Subspace solvers is presented. The framework implements the Conjugate Gradient method using OP2 and CUSPARSE & CUBLAS libraries. Timings are taken for OP2, CUSPARSE & CUBLAS and serial versions of the implemented Conjugate Gradient method and OP2 is found to be sensitive to the structure of the input matrix for the linear system being solved. A metric called the ‘structure effect factor’ is derived to quantify the performance of OP2 with respect to the structure of the input matrix.
## Contents

1 Introduction ........................................... 1  
   1.1 Project Motivation .................................... 2  
   1.2 Structure and Outline of the Project ................. 2  

2 Design .................................................. 4  
   2.1 Design Architecture ................................. 5  
   2.2 Data Object Types .................................... 7  
      2.2.1 The ‘Vec’ Class ................................. 8  
      2.2.2 The ‘CooMat’ Class .............................. 8  
   2.3 Abstract Krylov Subspace Algorithm Classes ........... 9  
      2.3.1 The ‘Krylov’ Class ............................. 9  
      2.3.2 The ‘ConjugateGradient’ Class ................. 10  
      2.3.3 The ‘ConjugateGradientOP2’ Class ............. 15  
      2.3.4 The ‘ConjugateGradientNotILOP2’ Class ........ 21  
      2.3.5 The ‘ConjugateGradientCUSPARSE’ Class ........ 23  
      2.3.6 The ‘ConjugateGradientSerial’ Class .......... 26  
   2.4 Summary ............................................. 27  

3 Testing and Verification ................................. 28  

4 Matrix and Vector Operations in OP2 ..................... 32  
   4.1 Vector-Vector Operations ............................. 32  
   4.2 Matrix-Vector Operations ............................. 33  
   4.3 Summary ............................................. 36  

5 Benchmarking ........................................... 37  
   5.1 Estimating Storage Requirements ...................... 37  
   5.2 Selection of Matrices ................................ 38  
   5.3 Plotting Axes ....................................... 39  
   5.4 Benchmarking Methodology ............................ 39  
   5.5 Results .............................................. 40  
      5.5.1 Large Scale Features of Matrix-Vector Multiplication 40  
      5.5.2 OP2 Matrix-Vector Multiplication Revisited ....... 42  
      5.5.3 The ‘dotProduct()’ and ‘saxpy()’ Operations ....... 44  
      5.5.4 The ‘setup()’ Procedure ........................ 46
6 Future Work and Conclusions 47
   6.1 Future Work .................................. 47
       6.1.1 Refactoring the KS-Framework .................. 47
       6.1.2 Different Input File Formats .................... 49
       6.1.3 Different Matrix Formats ....................... 50
       6.1.4 Different Kyrlov Subspace Algorithms .......... 52
   6.2 Conclusions .................................. 52

Bibliography 72
List of Tables

1 Basic properties of the selected benchmark matrices. . . . . . . . . . . 58
List of Figures

2.1 UML diagram of the overall object architecture .......................... 6
2.2 Sequence diagram example for ConjugateGradientOP2 .................. 7
2.3 UML class diagram for a dense vector data type .......................... 8
2.4 UML diagram for a sparse coordinate matrix data type ............... 9
2.5 UML diagram for the Krylov Subspace method abstract class .......... 10
2.6 UML diagram for the abstract Conjugate Gradient method class ....... 11
2.7 UML diagram for the OP2 Conjugate Gradient method class .......... 16
2.8 Interleaving of the row and column index arrays for a sparse matrix in OP2 ................................................. 21
2.9 UML diagram for the OP2 Conjugate Gradient method class .......... 22
2.10 UML Diagram for the CUSPARSE/CUBLAS Conjugate Gradient method implementation .............................. 23
2.11 UML Diagram for the Serial Conjugate Gradient method implementation 26

3.1 Verification errors for Conjugate Gradient methods (part 1) .......... 29
3.1 Verification errors for Conjugate Gradient methods (part 2) .......... 30

4.1 Schematic diagram of the algorithm skeleton for saxpy, copy& zero .. 32
4.2 Schematic figure of the dotProduct .......................... 33
4.3 A tridiagonal matrix-vector multiplication ............................. 34
4.4 Mini-partitions for the tridiagonal matrix .............................. 35

5.1 Conjugate Gradient running time divided by number of iterations vs. matrix size ........................................ 40
5.2 Average time for matrix-vector multiplication vs. number of non-zeros . 41
5.3 Estimating thread colouring .................................. 42
5.4 The $S_f$ value vs. number of non-zeros ............................. 44
5.5 The dotProduct operation vs. number of rows .......................... 45
5.6 The saxpy operation vs. number of rows ............................. 45
5.7 Setup time vs. number of matrix non-zero values ..................... 46

6.1 High level UML class diagram for an improved architecture for the KS-Framework ........................................ 49
6.2 Padding the tridiagonal matrix to eliminate block level colouring ..... 50
6.3 Blocking for a sparse matrix .................................. 51
4 Verification errors for test linear system (part 1) .......................... 55
<table>
<thead>
<tr>
<th></th>
<th>Section Description</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>4</td>
<td>Verification errors for test linear system (part 2)</td>
<td>56</td>
</tr>
<tr>
<td>4</td>
<td>Verification errors for test linear system (part 3)</td>
<td>57</td>
</tr>
<tr>
<td>5</td>
<td>Sparsity plots of the selected matrices (part 1)</td>
<td>59</td>
</tr>
<tr>
<td>5</td>
<td>Sparsity plots of the selected matrices (part 2)</td>
<td>60</td>
</tr>
<tr>
<td>5</td>
<td>Sparsity plots of the selected matrices (part 3)</td>
<td>61</td>
</tr>
<tr>
<td>5</td>
<td>Sparsity plots of the selected matrices (part 4)</td>
<td>62</td>
</tr>
<tr>
<td>5</td>
<td>Sparsity plots of the selected matrices (part 5)</td>
<td>63</td>
</tr>
<tr>
<td>6</td>
<td>Example row count histograms</td>
<td>65</td>
</tr>
</tbody>
</table>
Acknowledgements

First and foremost I would like to thank my supervisor Dr. Chris Maynard for his help and support throughout this project. He has always been most generous with his time, and most patient at steering me in the right direction. His comments and advice were welcome and greatly appreciated. I would also like to thank Dr. Mike Giles who was willing to answer my queries and to give me the benefit of his knowledge and experience.

I would also like to thank Thomas Richard and Paul Ross for being consistently sympathetic despite my perpetual state of stress and anxiety!

I would like to dedicate this work to my mother Elizabeth and my grandmother Ann, who encouraged me to return to university and for their never wavering love and support.
Chapter 1

Introduction

Many problems in science and engineering may be posed as differential equations. Unfortunately it is not always possible to find analytical solutions to most differential equations. However finite difference, and finite element methods may be used to find approximate numerical solutions to such problems. Usually these equations reduce to a sparse matrix system of the form $Ax = b$. Krylov Subspace (KS) methods are a class of numerical techniques which can solve such systems by posing them as optimisation problems.

Recent years have also seen the explosion of an affordable high performance computing architecture, namely the GPGPU (General Purpose Graphical Processing Unit). These devices offer a highly parallel and extremely accessible platform, making once intractable problems solvable. Their availability and affordability truly heralds “supercomputing for the masses” [1].

This work implements a framework to allow users to leverage the parallelism of GPGPU devices for the numerical solution of linear systems of the form $Ax = b$. It sets out to create a library which may be used not only to implement Krylov Subspace algorithms (a useful goal in itself) but also to provide a platform which may be used to explore different strategies to gain optimal use of existing hardware. A primary goal of this project has been to create a flexible software platform which allows the abstraction of several key components of Krylov Subspace solvers. To this end, the chosen language is C++ as this allows a modular approach to building a KS solver framework (henceforth referred to as the KS-framework).

The KS-framework has been designed to leverage two technologies which are used to implement the primitive operations required by a Krylov Subspace solver. These are the OP2 open source framework for unstructured grid applications [2] and NVIDIA’s CUSPARSE & CUBLAS [3, 4] libraries. These projects take very different approaches to modelling the matrix-vector computations required by optimisation algorithms. However the presented KS-framework is flexible enough to handle them both, along with the possibility of future extension. This is evidenced by a modular software engineering approach, along with a test driven development model.
The construction of the KS-framework allows the exploration of different strategies to implement the basic computations performed by an optimisation algorithm. To this end the performance of OP2 and CUSPARSE & CUBLAS implementations are compared; with the view that the flexibility of the OP2 model may be used to code efficient implementations of the matrix-vector multiplication operations at the heart of Krylov Subspace algorithms. The paradigm being that if we can exploit the inherent structure of the matrix, we can gain in terms of computational speed.

1.1 Project Motivation

As stated previously, the primary motivation for this project is to construct a flexible framework to execute and explore different implementations of Krylov Subspace algorithms. One of the most fundamental operations of KS methods is matrix-vector multiplication $\mathbf{A}\mathbf{x}$ [9]. This apparently simple task has many variations and is rich in computational possibilities. From this perspective, the proposed design and associated code base not only provides a mechanism to implement different KS algorithms, but also a mechanism which allows potential users to implement matrix-vector operations in the most efficient manner possible for an available platform. Indeed any of the fundamental operations should be amenable to re-implementation by users as new paradigms and approaches are discovered.

The KS-framework implements serial, CUSPARSE & CUBLAS and OP2 versions of the operations underpinning Krylov Subspace methods. The comparison of these for a range of matrices will be presented. The intention of this is to highlight how OP2 performs versus NVIDIA’s own, highly optimised, sparse matrix-vector libraries.

1.2 Structure and Outline of the Project

To begin with, the design details of the KS-framework are discussed. Detailed discussion of CUSPARSE & CUBLAS libraries has been eschewed since they are documented in much greater detail that would be possible here [3, 4]. Furthermore, the way in which an OP2 program is written is not discussed in this work, since this was completed in the project preparation phase of the project. However the project preparation document is available along with Doxygen generated documentation for the KS-framework [5, 6].

The design section details the important aspects of how the KS-framework has been constructed. Briefly the following is discussed:

- the overall design and the template design pattern
- the supported matrix-vector data types
- the different implementations of Krylov Subspace methods, namely the Conjugate Gradient (CG)
The test section outlines the verification of the components which comprise the KS-framework. Unit tests of vector and matrix data types are discussed. The various CG methods are tested against Matlab’s pgc method using random test matrices (also generated using Matlab). The error in the residual is then examined against ‘canonical solutions’.

Following this there is a detailed look at how OP2 performs vector-vector and matrix-vector operations. This section is important as it provides much of the background to how the results of benchmarking are interpreted. Benchmarking itself is then performed using matrices found in the University of Florida Sparse Matrix Collection [7]. The methodology and approach taken is briefly outlined, before an analysis of the results. The analysis focuses particularly on the behaviour of matrix-vector multiplication in OP2 with respect to the structure of the input matrix; since this was the most intriguing aspect of the results and since CUSPARSE & CUBLAS did not show evidence of this type of behaviour.

Upon closing there is a brief discussion of conclusions and findings, along with possible directions for future work.
Chapter 2

Design

The overall design goal of the KS-framework is to create a modular platform which can be used to explore sparse matrix encoding strategies along with different implementations of KS algorithms. The strategy presented makes implementation of Krylov Subspace algorithms simple. However it also allows a user to easily select an implementation which is efficient and achieves the best possible performance. Thus the KS-framework should provide a potential user both ease of implementation and good performance - the best of both worlds. Consequently the two major design goals are identified as:

- creation of a flexible framework to allow implementation of Krylov Subspace algorithms
- creation of different matrix and vector types which can be used in a modular manner

The major thrust of the design is to utilise the template design pattern [8]. This allows abstraction of the major algorithmic components of a particular KS algorithm and shifts focus so that the programmer can concentrate on providing efficient implementations for dot products, axpy\(^1\) routines and matrix-vector multiplication.

The scheme presented builds on the strategy employed in ‘Numerical Recipes’ [9]. Succinctly, the abstract ConjugateGradient class is templated for two sets of matrix and vector types - host representations and device representations (discussed further in Section 2.3.2). This then allows the ConjugateGradient class to declare device data as member variables, along with a method to execute the algorithm - which may then refer to these device variables. Child classes then inherit from ConjugateGradient and explicitly provide types which replace the templated parameters. This frees the underlying implementation from repeating the algorithmic skeleton of a particular KS method.

\(^1\) Single precision / double precision / single precision complex / double precision complex \(a \times \text{plus } y\).
The following sections take an overview of each of the main classes within the KS-framework. Additional details (such as detailed descriptions of what functions do) may be found in the Doxygen generated documentation which accompanies this document [6].

2.1 Design Architecture

Figure 2.1 illustrates the core classes of the KS-framework. Overall, the design has been separated into three distinct namespaces, namely: densevector, coomatrix and krylov. The role of each namespace is as follows:

- **densevector** - holds implementations of dense vector data types.
- **coomatrix** - holds implementations of coordinate sparse matrix data types.
- **krylov** - holds implementations of various Krylov Subspace algorithms.

The relationship between the Conjugent Gradient method and its child classes is clear, since it is in fact these classes which implement the core functionality of each KS algorithm. All algorithms inherit from the Krylov class, this is because all algorithms possess the performMethod function which is defined in this class and which actually executes a particular Krylov Subspace method.

The KS algorithm classes have the following naming convention (in camel-case):

```
<METHOD_NAME><UNDERLYING_IMPLEMENTATION>
```

Each part is as follows:

- **METHOD_NAME** - the method which is being implemented, as of writing only the Conjugate Gradient method is supported
- **UNDERLYING_IMPLEMENTATION** - the implementation for the methods which are used for building a given Krylov method, as of writing these are:
  - OP2 the default OP2 implementation using coordinate matrices and an interleaved edge array (covered in detail in section 2.3.3)
  - NotIL0P2 a version of OP2 where the mapping between matrix and vector elements are not interleaved (covered in detail in section 2.3.4)
  - CUSPARSE a version of the Conjugate Gradient algorithm using CUSPARSE and CUBLAS
  - Serial a serial implementation using the overloaded operations of Vec and CooMat, this method is intended for illustrative purposes only and is not benchmarked
Figure 2.2, shows how each of the major components of the KS-framework interact with each other. It illustrates the sequence of function calls between classes, using ConjugateGradient0P2 as an example (all client classes of ConjugentGradient behave in the same manner). The idea behind the template design pattern can clearly be seen; most importantly it is clear how the performMethod function of the ConjugateGradient class delegates to the ConjugateGradient0P2 class. The reader should note that not all calls from the performMethod of the ConjugateGradient class are displayed on the digram for brevity. Furthermore the object, called cg, is illustrated as having two types (ConjugateGradient and ConjugateGradient0P2); this allows the delegation of method calls to be illustrated clearly.
2.2 Data Object Types

The main data object types used in the KS-framework are discussed here. For sparse matrix classes, a simple nomenclature has been employed, which consists of three letters denoting the type of encoding for the matrix followed by ‘Mat’. Thus ‘CooMat’ represents a coordinate encoded sparse matrix.

The basic matrix data types are templated to take any base type, although conjugate gradient methods are implemented for double precision numbers, it is feasible to create complex number templated types for sparse matrices and vectors. Although this would require additional work in OP2, it would be relatively straight forward to achieve in the CUSPARSE version since the CUSPARSE & CUBLAS libraries support complex number variants of $axpy$ methods, inner product and matrix-vector multiplication.
2.2.1 The ‘Vec’ Class

The Vec class is an implementation of a dense vector data structure. It is basically a wrapper for a std::vector and so uses this as its underlying data type. The reason for this is that the C++ standard library vector is a flexible data type which may be templated for different data types.

Currently the KS-framework supports only vectors that are stored in a dense format. The rationale behind this is that although storage for dense matrices scales \(O(N^2)\), storage for vectors scales only \(O(N)\) and so there was little gain for increased complexity. The downside of this decision is that storage is wasted for very low density vectors.

2.2.2 The ‘CooMat’ Class

The CooMat class (figure 2.4) is an implementation of a sparse matrix in using the coordinate storage format [10]. As with the Vec class, std::vector objects are used to store row indices (_rowIdx), column indices (_colIdx) and values (_values) as this allows added flexibility as described previously.

In general the coordinate matrix storage scheme is not the most efficient encoding for sparse matrices - CSR [10] is better. There is significant redundancy in the format - both row and column indices must stored along with matrix values. Furthermore, when performing matrix-vector multiplication on the CPU a double dereference is required to index into the vectors. On GPUs, however, coordinate format matrices become most
useful. This is because the redundancy in the format results in a contiguous memory access pattern among threads. Such an access pattern reduces thread divergence across rows with variable value-densities\(^2\) [11]. Thus coordinate matrices make a good general matrix class for the initial implementations of conjugate gradient methods.

![UML diagram for a sparse coordinate matrix data type](image)

Figure 2.4: UML diagram for a sparse coordinate matrix data type

## 2.3 Abstract Krylov Subspace Algorithm Classes

The abstract algorithm classes define the interfaces for the KS-framework. They provide a contract which implementing classes must adhere to. This means that it is possible to talk about Krylov Subspace classes in terms of the functionality they provide rather than their implementation detail, which is a useful abstraction.

### 2.3.1 The ‘Krylov’ Class

The Krylov class is the parent class of all Krylov Subspace algorithms. It declares only one method, namely `performMethod`. It is this method which is implemented by child KS algorithm classes. The class is templated for the parameters: `TMATRIX`, `TVECTOR` and `T`. The functions of these are:

\(^2\) the ratio of non-zero values to zero values in a matrix, matrix column, matrix row or vector.
• TMATRIX - the matrix type which is used within the KS-framework
• TVECTOR - the vector type which is used within the KS-framework
• T - the underlying type for matrix and vector types: TMATRIX and TVECTOR; this parameter has particular significance when using OP2 (as detailed further in 2.3.2).

The complete class diagram for the Krylov class is show in figure 2.5.

Figure 2.5: UML diagram for the Krylov Subspace method abstract class

### 2.3.2 The ‘ConjugateGradient’ Class

As outlined in section 2.1, all KS algorithm classes are based on the template design pattern [8]. This allows us to isolate a particular Krylov subspace algorithm from the specifics of a particular matrix-vector implementation. Figure 2.6 shows that the ConjugateGradient class is templated for five different types, these are as follows:

• TMATRIX - see section 2.3.1
• TVECTOR - see section 2.3.1
• TDEV_MATRIX - the device representation of the matrix data, since this may be different from the matrix type used within the Krylov subspace algorithm
• TDEV_VECTOR - the device representation of the vector data, again this may be different to the type defined by TVECTOR
• T - the basic underlying type for vectors and matrices. As of writing double types are supported, however additional types can be added (for example complex numbers). It should be noted that creating a template type for the OP2 version of the Conjugate Gradient method is difficult (though not impossible) since OP2 requires C style strings in the declarations of its data structures

As stated previously, the matrix-vector operations necessary for implementation of a conjugate gradient method use C++ templates to define device data type parameters and are declared pure virtual. Thus the ConjugateGradient class cannot itself be instantiated, and child classes are required to provide implementations of the matrix-vector operations. These operations, along with their expected behaviour are outlined below, the actual type parameters are omitted from descriptions for ease of the reader but details can be found in figure 2.6 along with the Doxygen generated documentation [6].
The reader should note that all parameters are passed to the respective methods using reference semantics (i.e. C++ pointers), and that the final parameter of a method holds the result of that method.

```cpp
+ performMethod(matA : &TMA_TRIX, vecb &TVECT_OR) : TVECT_OR
# dotProduct(u : TDEV_VECT_OR*, v : TDEV_VECT_OR*, s : T*) = 0
# saxpy(a : T*, x : TDEV_VECT_OR*, y : TDEV_VECT_OR*, r : TDEV_VECT_OR*) = 0
# mvmult(A : TDEV_MA_TRIX*, x : TDEV_VECT_OR*, y : TDEV_VECT_OR*) = 0
# gaxpy(sAx : T, A : TDEV_MA_TRIX*, x : TDEV_VECT_OR*, sy : T, y : TDEV_VECT_OR, r : TDEV_VECT_OR) = 0
# copy(src : TDEV_VECT_OR*, dst : TDEV_VECT_OR*) = 0
# zero (v : TDEV_VECT_OR*) = 0
# fetch(v : TDEV_VECT_OR*) : TVECT_OR = 0
# setup(matA : &CooMat<double>, vecb : &Vec<double>) = 0
# teardown() = 0
```

Figure 2.6: UML diagram for the abstract Conjugate Gradient method class

The reader should also note that all methods outlined below do not check the correspondence of vector and matrix sizes and do not automatically zero input data. The result of this is that the following methods are simpler to implement. However, the calling methods (i.e. the performMethod function) becomes responsible for checking whether matrices and vectors are of the correct type, and that they are set to zero when required. This also allows faster implementations of methods, since type checking is not implicitly performed each iteration.

The ‘dotProduct’ Method

The dot product function (dotProduct) takes the inner product of two device vectors \( u \), \( v \) and stores the result in a host variable \( s \). It is expected that a compliant implementation is consistent with algorithm 1 (here the symbol \( \gets \) is used to denote accumulation of values).

**Algorithm 1** Dot product

```cpp
dotProduct(u, v, s) {
    for i = 1 to N - 1 do
        s \leftarrow s + u_i \cdot v_i
    end for
}
```
The ‘saxpy’ Method

The saxpy method (saxpy) performs a ‘scalar \(a\) multiplies \(x\) plus \(y\)’ operation on the device vectors \(x\) & \(y\) and stores the result in the device vector \(r\). It is defined in detail in Golub & Van Loan [12], however the version used in the KS-framework is subtly different in that the result is not accumulated in vector \(y\) but in a different vector - \(r\). This uses more memory, but leaves open the possibility for greater parallelism since there is no implied order to the sequence of operations accumulating in the output. It should be semantically equivalent to the operation outlined in algorithm 2 (the reader is invited to contrast this with the definition outlined in Golub & Van Loan).

**Algorithm 2** saxpy operation

```c
saxpy(a, x, y, r) {
    for i = 1 to N − 1 do
        \(r_i \leftarrow a \cdot x_i + y_i\)
    end do
}
```

The ‘mvmult()’ Method

The matrix-vector multiplication method (mvmult) takes the product of a device matrix \(A\) and a device vector \(x\), and stores the result in the device vector \(y\). It should be equivalent to the definition outlined in algorithm 3.

**Algorithm 3** Matrix vector multiplication

```c
mvmult(A, x, y) {
    for each row \(A_i\) in \(A\) do
        \(y_i \leftarrow A_i \cdot x\)
    end for
}
```

The ‘gaxpy()’ Method

The gaxpy method (gaxpy) performs the ‘generalised \(a\) multiplied by \(x\) plus \(y\)’ operation. This operation is outlined in detail in Golub & Van Loan. However the result of the operation is stored in \(r\) instead of \(y\) (for reasons similar to those outlined in the definition of saxpy). It should be equivalent to the operation outlined in algorithm 4.
**Algorithm 4** gaxpy operation

```
gaxpy(sAx, A, x, sy, y, r) {
    for each row \( A_i \) in A do
        \( z_i \leftarrow A_i \cdot x \)
    end for
    \( r_i \leftarrow (sAx \cdot z) + (sy \cdot y) \)
}
```

The ‘copy()’ Method

The copy method (copy) copies one vector into another element by element. It is equivalent to the operation outlined in 5.

**Algorithm 5** Copy operation

```
copy(src, dst) {
    for \( i = 0 \) to \( N - 1 \) do
        \( src_i \leftarrow dst_i \)
    end for
}
```

The ‘zero()’ Method

The vector zeroing method (zero) sets all values of an element to zero. It is equivalent to the operation outlined in 6.

**Algorithm 6** Zero operation

```
zero(v) {
    for \( i = 0 \) to \( N - 1 \) do
        \( v_i \leftarrow 0 \)
    end for
}
```

The ‘fetch()’ Method

The fetch method (fetch) retrieves a device vector type and packs it into a host vector type. This method may be used to get the final result vector from the device.
The ‘setup()’ Method

The initialisation method (setup) is used to perform initialisation steps prior to executing the conjugate gradient method. Referring to algorithm 7, it is the first step which is performed. This method should perform the following actions:

- copy data from device to host - performing any necessary conversions.
- initialise the library subsystem (if required).

The ‘teardown()’ Method

This method is the last method called. It should:

- clear up memory allocated on device and any additional structures used on the host.
- shut down any library subsystem.

The ‘performMethod()’ Method

The performMethod function contains the implementation of a conjugate gradient method. This method operates on device data, and so the class contains protected data member pointers to the device data.

The basic outline of the code implemented in performMethod is shown in algorithm 7 (appendix A shows how algorithm 7 corresponds with the C++ implementation).
Algorithm 7 The Conjugate Gradient method (reproduced from [10])

Set $v_k$ to zero (done on host)
Compute $r_k = b - Av_k$
Set $p_k = r_k$

while ($k \leq \text{maxiter}$)

\[ \alpha \leftarrow r_k^T \cdot r_k / p_k^T \cdot Ap_k \]

\[ v_{k+1} \leftarrow v_k + \alpha p_k \]
\[ r_{k+1} \leftarrow r_k - \alpha Ap_k \]

\[ ||r_{k+1}||_2 \leftarrow \sqrt{b^T \cdot b} \]
\[ ||b||_2 \leftarrow \sqrt{b^T \cdot b} \]

if ($||r_{k+1}||_2 / ||b||_2 < \text{tol}$) break

\[ \beta \leftarrow r_{k+1}^T \cdot r_k / r_k^T \cdot r_k \]

\[ p_{k+1} \leftarrow r_{k+1} + \beta p_k \]

$k \leftarrow k + 1$
end while

2.3.3 The ‘ConjugateGradientOP2’ Class

Figure 2.7 outlines the ConjugateGradientOP2 class. This is the default OP2 implementation and makes use of CooMat coordinate matrices.

The device types (TDEV_MATRIX and TDEV_VECTOR) are explicitly of type op_dat. Within the implementation of the class, there is a base type pointer (beginning with the prefix loc_) for each of the TDEV_VECTOR types declared in the ConjugateGradient class. This is a requirement of OP2, since device data must be bound to data on the host.
The OP2 implementation introduces three important concepts, namely VectorSet, ValueSet and EdgeSet. These are variables of type op_set (for VectorSet and ValueSet) and op_map (for EdgeSet). VectorSet represents a collection of elements stored in a vector and ValueSet represents non-zero matrix values. The EdgeSet contains the mappings between vector elements and the non-zeros of the matrix.

The ‘dotProduct()’ Method

Implementing a dot product in OP2 is relatively straightforward. Fundamentally the relationship between the elements of the two input vectors is the identity mapping, thus dot products do not require indirect datasets [13], this is a consequence of the definition found in algorithm 1. The code snippet in listing 1 illustrates how the dot product kernel (dotProdKernel) is called from OP2.

**Listing 1** Execution of the dot product kernel

```c
dotProduct(op_dat u, op_dat v, double* s) {
    op_par_loop(dotProdKernel, "dotProdKernel", VectorSet,
                op_arg_dat(u, -1, OP_ID, 1, "double", OP_READ),
                op_arg_dat(v, -1, OP_ID, 1, "double", OP_READ),
                op_arg_gbl(s, 1, "double", OP_INC));
}
```

Each element of a VectorSet is assigned to a single thread which performs the operation outlined in listing 2. The resulting products are then added together using using a sum reduction operation (indicated by the OP_INC in the example).
The dot product kernel

```
inline void dotprodKernel(double *u, double *v, double *dp) {
  *dp += (*u) * (*v);
}
```

The ‘saxpy(), ‘copy() and ‘zero()’ Methods

The saxpy, copy and zero methods are executed in a manner similar to the OP2 skeleton outlined in listing 3. This is because each of these operations has the same algorithmic structure. Basically these operations loop over some elements of a vector (or vectors v1 to vn), perform some operation on elements with the same index and then store the result at the corresponding index in the output. The constants, c1 to cn may also be used to scale some of the input vectors.

```
func(double <c1>, ..., double <cn>,
     op_dat <v1>, ..., op_dat <vn>,
     op_dat <output>) {
  op_par_loop(<operation>, "<operation>", VectorSet,
               op_arg_gbl(&<c1>, 1, "double", OP_READ),
               ...
               op_arg_gbl(&<cn>, 1, "double", OP_READ),
               ...
               op_arg_dat(<v1>, -1, OP_ID, 1, "double", OP_READ),
               ...
               op_arg_dat(<vn>, -1, OP_ID, 1, "double", OP_READ),
               op_arg_dat(<output>, -1, OP_ID, 1, "double", OP_WRITE));
}
```

The kernels for the saxpy, copy and zero methods closely resemble each other. This is hinted at by the fact that they are called in the same fashion (as described previously). They are shown in listings 4, 5 and 6. This similarity is an artefact of their definition, since algorithms 2, 5 and 6 have similar structures.

```
inline void saxpyKernel(const double *a, double *x, double *y,
                        double *r) {
  *r = (*a) * (*x) + (*y);
}
```
The 'mvmult()' Method

The matrix-vector multiplication method makes use of OP2 indirection [13]. This is highlighted by the appearance of EdgeMap in the OP2 call to the mvmultKernel (as illustrated in listing 7). The EdgeMap array contains the associations between the non-zero values in the input matrix $A$ and the input vector $x$. The result of this operation is stored in the vector $y$. The EdgeMap array is constructed by interleaving the rowIdx and colIdx arrays of the input matrix.

Listing 7 Matrix-vector multiplication in OP2

```
mvmult(op_dat A, op_dat x, op_dat y) {
    op_par_loop(mvmultKernel, "mvmultKernel", ValueSet,
    op_arg_dat(A, -1, OP_ID, 1, "double", OP_READ),
    op_arg_dat(x, 1, EdgeMap, 1, "double", OP_READ),
    op_arg_dat(y, 0, EdgeMap, 1, "double", OP_INC));
}
```

The kernel mvmultKernel is outlined in listing 8. The code is relatively simple, with OP2 performing the work of partitioning the data over the ValueSet.

The ‘vadd()’ and ‘vscale()’ Methods

The add and scale methods are useful building blocks for the gaxpy operation. The vscale and vadd methods scale a vector by a number, and add one vector to another.
However they are not defined in the `ConjugateGradient` class. They are declared as private methods and as such are only useable by the OP2 implementation of the Conjugate Gradient method. The reason for this is that the original `ConjugateGradient` class only defines those methods necessary for the execution of a CG method. The calling scheme is the same as that outlined by the skeleton in listing 3, since they both use the identity mapping between vector elements. The code for the kernels are shown in listings 9 and 10.

Listing 9 OP2 vector addition kernel
```c
inline void addKernel(double *y, double *r) {
    *r = (*y) + (*r);
}
```

Listing 10 OP2 vector scale kernel
```c
inline void scaleKernel(const double *alpha, double *r) {
    *r = (*alpha) * (*r);
}
```

The ‘gaxpy()' Method

The implementation of the gaxpy method is very simple since it is constructed from several operations which have been defined previously. The code for the method is outlined in listing 11.

Listing 11 OP2 gaxpy method
```c
gaxpy(double sAx, op_dat A, op_dat x, double sy, op_dat y, op_dat r) {
    double div = sAx / sy;

    zero(r);
    mvmult(A, x, r);
    vscale(div, r);
    vadd(y, r);
    vscale(sy, r);
}
```

In order to see why listing 11 is correct, it is important to understand that a gaxpy operation is defined as \( r \leftarrow sAx \times Ax + sy \times y \). Where \( sAx \) and \( sy \) are scalars, \( A \) is a matrix and \( x \) and \( y \) are vectors. This operation may then be broken down in the following manner.
\( r \leftarrow 0 \)
\( r \leftarrow Ax \)
\( r \leftarrow (sAx/sy) \times Ax \)
\( r \leftarrow (sAx/sy) \times Ax + y \)
\( r \leftarrow sAx \times Ax + sy \times y \)

This is by no means the best or most efficient way to implement this operation. However it is quite simple. Furthermore, since this method is only called once, the inefficiency and the numerical rounding error arising from the division and then the re-multiplication of the scalars \( sAx \) and \( sy \) is limited.

**The ‘setup()’ Method**

The default Conjugate Gradient implementation in OP2 uses the coordinate matrix data type. This means that it is simple to map the data to the graph model employed by OP2 for the matrix-vector multiplication operation [5]. This mapping occurs in the `setup` method of the `ConjugateGradientOP2` class and it is the most intensive section of this method. The mapping between matrix and vector values is simply an interleaving of row and column index arrays. For example, the edge mapping for the sparse system in figure 2.8a is achieved by interleaving row and matrix index arrays, resulting in an edge mapping array of size \( 2 \times nNz \) (see figure 2.8b). This means that even numbered elements of the array refer to row indices, and odd numbered elements refer to column indices.

The `setup` method is also responsible for initialisation of OP2, and declaring the arrays which are then subsequently bound to the equivalent device data variables in the `ConjugateGradient` class (figure 2.6).
Figure 2.8: Interleaving of the row and column index arrays for a sparse matrix in OP2

2.3.4 The ‘ConjugateGradientNotILOP2’ Class

The implementation of the ConjugateGradientNotILOP2 class is fundamentally the same as the ConjugateGradientOP2 class. Its UML class diagram is shown in figure 2.9. In contrast to the UML diagram for ConjugateGradientOP2, this class contains two edge mapping arrays of type op_map. These are EdgeMapRow and EdgeMapCol, and are associated with the rowIdx and colIdx of the input CooMat matrix. The main difference between this class and ConjugateGradientOP2 is that in this implementation the edge information which holds the mappings between matrix and vector elements is no longer interleaved - since there are now separate EdgeMapRow and EdgeMapCol structures.
All OP2 kernels are implemented in the same manner as in ConjugateGradientOP2 and are described in detail in section 2.3.3. However mvmult and setup methods are different, since they no longer reference a single EdgeMap.

The ‘mvmult()’ Method

The kernel for the matrix vector multiplication operation is the same as that found in the ConjugateGradientOP2 class. However calling the kernel is slightly different as shown in listing 12. The reader should note that in this instance EdgeMapCol and EdgeMapRow are referenced.

Listing 12 OP2 matrix-vector multiplication (non interleaved version)

```c
mvmult(op_dat A, op_dat x, op_dat y) {
    op_par_loop(mvmultKernel, "mvmultKernel", ValueSet,
        op_arg_dat(A, -1, OP_ID, 1, "double", OP_READ),
        op_arg_dat(x, 0, EdgeMapCol, 1, "double", OP_READ),
        op_arg_dat(y, 0, EdgeMapRow, 1, "double", OP_INC));
}
```

It is important to understand that even though different index arrays are used, the relationships between matrix and vector elements is unchanged. It is only the representation used when executing op_par_loop that is different. This certainly has an effect on the startup method, but not necessarily on the execution time of a matrix vector multiplication.
2.3.5 The ‘ConjugateGradientCUSPARSE’ Class

Figure 2.10 illustrates the ConjugateGradientCUSPARSE class. This is the CUSPARSE & CUBLAS [3, 4] variant of the CG algorithm. Like the OP2 version, this class uses the coordinate matrix implementation primarily. However CUSPARSE uses CSR [10] formatted matrices to perform its operations. Consequently the setup procedure must first perform a conversion from COO format to CSR format. It should also be noted, that CUSPARSE and CUBLAS function calls are asynchronous since CUDA 4.0. As of writing this is an undocumented feature of these libraries. Consequently cudaThreadSynchronize calls are used throughout the code for accurate timings (though these are omitted from the listings below).

```
# dotProduct(u : double*, v : double*, s : double*)
# saxpy(a : double*, x : double*, y : double*, r : double*)
# mvmult(A : double*, x : double*, y : double*)
# gaxpy(sAx : double, A : double*, x : double*, double : sy, y : void*, r : double*)
# copy(src : double*, dst : double*)
# set (a : double, v : double*)
# fetch(v : TDEV_VECTOR*) : Vec<double>
# setup(matA : CooMat<double>&, vecb : Vec<double>&)
# teardown()
```

Figure 2.10: UML Diagram for the CUSPARSE/CUBLAS Conjugate Gradient method implementation

The ‘dotProduct’ Method

The implementation of the dotProduct method is extremely simple using CUBLAS and is shown in listing 13. This is basically a wrapper to the CUBLAS call to cublasDdot and is detailed in the CUBLAS user manual [3]. This function will bring back the result of the double precision dot product to the host device.

```
Listing 13 CUSPARSE/CUBLAS dot product operation

dotProduct(double *u, double *v, double *s) {
    *s = cublasDdot(n, u, 1, v, 1);
}
```

The ‘saxpy’ Method

The saxpy method implementation is also simple using CUBLAS. This operation works by accumulating the result in the y value - semantically it is similar to how the saxpy
operation is defined in Golub and Van Loan, consequently the input y value is first written to the r vector. Then the saxpy operation is applied. The reader should note that the output value is kept on the device after this operation completes.

Listing 14 CUSPARSE/CUBLAS saxpy operation

```c
saxpy(double a, double *x, double *y, double *r) {
    cublasDcopy(n, y, 1, r, 1);
    cublasDaxpy(n, a, x, 1, r, 1);
}
```

The ‘mvmult’ Method

The matrix-vector multiplication operation is shown in listing 15. This is actually the CUSPARSE gaxpy operation, but ignoring the final vector addition, which is achieved by passing 0.0 for the 9th parameter (which indicates that the y does not have to be a valid input parameter). The full details of the cusparseDcsrmv are detailed in the CUSPARSE reference manual [4]. For completeness, the different parameters to the function are:

- nRow the number of rows of the matrix
- nCol the number of columns of the matrix
- cuDescra the matrix descriptor
- A the matrix non-zero values (see [4])
- rowStart the array of row starts in the CSR matrix
- AColIdx the column indices of the input coordinate matrix
- y the output

The CUSPARSE cusparseDcsrmv makes use of matrices in CSR format [10]. This conversion occurs in the setup routine and occurs only once.

Listing 15 CUSPARSE/CUBLAS matrix-vector multiplication

```c
mvmult(double *A, double *x, double *y) {
    cusparseDcsrmv(cuHandle, CUSPARSE_OPERATION_NON_TRANSPOSE,
    nRow, nCol, 1.0, cuDescra, A, rowStart, AColIdx, x, 0.0, y);
}
```
The ‘gaxpy’ Method

Implementation of the gaxpy method is almost exactly the same as the mvmult implementation, except that there is a preceding copy operation before delegating to theCUSPARSE cusparseDcsrmv call. This is because the sy scalar value is passed to the function (2nd to last parameter) and so results are accumulated in r.

Listing 16 CUSPARSE/CUBLAS gaxpy operation

```c
#define CUBLAS_CUSPARSE_API

// CUSPARSE/CUBLAS gaxpy operation

// gaxpy(dA, dA, dx, dy, dr) {
// cublasDcopy(n, y, 1, r, 1);
// cusparseDcsrmv(cuHandle, CUSPARSE_OPERATION_NON_TRANSPOSE,
// nRow, nCol,
// dA, cuDescra,
// dx, dy, dr);
// }
```

The ‘copy’ and ‘zero’ Methods

Implementation of copy method is simply a wrapper to the appropriate CUBLAS call. However the zero method is implemented by multiplying each element of the input vector by zero. They are reproduced in listings 17 and 18 for completeness.

Listing 17 CUBLAS/CUSPARSE copy operation

```c
// CUBLAS/CUSPARSE copy operation

copy(double *src, double *dst) {
    cublasDcopy(n, src, 1, dst, 1);
}
```

Listing 18 CUBLAS/CUSPARSE zero operation

```c
// CUBLAS/CUSPARSE zero operation

zero(double *v) {
    cublasDscal(n, 0.0, v, 1);
}
```

The ‘setup()’ Method

The CUSPARSE library makes use of matrices in CSR format rather than the COO format. Consequently it is necessary to perform a conversion of the input matrix. CUSPARSE provides the method cusparseXcoo2csr which performs this operation, and
the output is stored in the rowStart array. The other tasks of the setup routine are to allocate data on the device, copy over input matrix and vector values and to initialise the CUSPARSE and CUBLAS libraries.

### 2.3.6 The ‘ConjugateGradientSerial’ Class

The UML diagram for the ConjugateGradientSerial class is shown in figure 2.11. This implementation makes use of the overloaded operators defined in the CooMat and Vec implementations. It is intended for illustrative purposes only, since the default operations require a large number of memory swaps. For example, the saxpy operation is shown in listing 19. Each of the operations (plus and multiply) results in the creation of a new Vec object consequently this operation requires a large amount of memory and lots of memory swapping. It does, however, demonstrate the flexibility of the design by interpreting the ‘device’ and the ‘host’ as being equivalent - this results in there being no additional data types declared for the class.

```c
# dotProduct(a : double*, v : double*, s : double*)
# saxpy(a : double*, x : double*, y : double*, r : double*)
# mvmult(A : double*, x : double*, y : double*)
# gaxpy(sAx : double, A : double*, x : double*, double : sy, y : void*, r : double*)
# copy(src : double*, dst : double*)
# set(a : double, v : double*)
# fetch(v : TDEV_VECTOR*) : Vec<double>
# setup(matA : CooMat<double>&, vecb : Vec<double>&)
# teardown()
```

Figure 2.11: UML Diagram for the Serial Conjugate Gradient method implementation

#### Listing 19 Serial saxpy operation

```cpp
saxpy(T a, densevector::Vec<T> * x, densevector::Vec<T> * y, densevector::Vec<T> * r) {
    (*r) = ((*x) * a) + (*y);
}
```
2.4 Summary

The design presented for the KS-framework demonstrates the simplicity of implementing a new version of a Conjugate Gradient method. It makes use of a simple design pattern and leverages it to gain significant code reuse and modularity. It is significant how little code is actually required for each implementations presented above. Indeed the listings presented in sections 2.3.3 and 2.3.5 comprise the majority of code required for the OP2 and CUSPARSE/CUBLAS implementations respectively. This is important since fewer lines of code result in fewer programming errors.
Chapter 3

Testing and Verification

Testing and verification of the KS-framework is performed on two levels. Firstly there are standard unit tests for the sparse matrix type and vector type (Vec and CooMat). These unit tests verify that the standard operations defined in these classes behave correctly, and have been implemented using the GoogleTest C++ testing framework [14]. The global functions which load data are also tested using a series of test input files.

Verification of CG algorithms is more involved. This requires first generating random test systems, i.e. random matrices $A$ and associated vectors $b$. This is achieved using Matlab’s sprandsym function which generates symmetric random sparse matrices. The eigenvalues of the system are then computed using Matlab’s `eig` function and all values are confirmed to be positive; this then indicates that the generated matrix is both symmetric and positive definite.

Once a test matrix has been generated, the Matlab pge method is used to compute the solution, $x$, of the system to within a tolerance of $10^{-15}$ for the target residual. This solution is then the ‘canonical solution’ to any given randomly generated system. The matrices and associated vectors are then read in, and solved by the relevant CG algorithm in the KS-framework - by varying tolerances for the target residual, from $10^{-1}$ down to $10^{-15}$. The Euclidean norm (show in equation 3.1) is then taken for the KS-framework solution versus Matlab’s canonical solution.

$$||u - v||_2 = \sqrt{(u_1 - v_1)^2 + \cdots + (u_n - v_n)^2}$$ \hspace{1cm} (3.1)

It is expected that the order of error reduces as the order of the tolerance is reduced, i.e. the plot of $\log_{10}$ of the error versus $\log_{10}$ of the input tolerance behaves linearly. This is indeed the case as illustrated by figure 3.1, which demonstrates the behaviour of the Euclidean distance versus the input tolerance for a $100 \times 100$ test system (a complete gallery of how the error behaves for different sizes of matrices may be found in appendix B).
Figure 3.1: Verification errors for Conjugate Gradient methods (part 1)
As can be seen from the graphs in figure 3.1 (and as is evidenced from the graphs in appendix B), the error of the different methods is almost the same, with very little variation. The graphs in figure 3.1 also show that there is a tail off as the input tolerance reaches $10^{-15}$. At this point the executing algorithms are encountering machine precision and the improvement in the accuracy of the solution becomes smaller. This may also be observed when examining the graphs in appendix B. Indeed as the size of the matrix is reduced, the tail off effect becomes less significant, since the smaller matrices converge in fewer iterations and so accumulate less errors due to loss of precision. Figure 4c in
appendix B shows an ‘out-tick’ which is exactly horizontal between an input tolerance of $1^{-15}$ and $1^{-14}$, since at $1^{-14}$ the algorithm ‘overshoots’ the stopping criteria value by so much that executing the method to the next order of magnitude does not result in an improvement.

Although each graph presented in figure 3.1, and the gallery in appendix B looks almost the same, there are some minor differences observed in the Euclidean distance from the canonical solution. These differences may be accounted for because, OP2, CUSPARSE and serial code executes arithmetic operations in different orders and for finite precision such arithmetic operations are not associative [10].
Chapter 4

Matrix and Vector Operations in OP2

Prior to discussing the methodology and results of benchmarking it is important to take a brief look at how OP2 performs matrix-vector, and vector-vector operations in detail. This will give an appreciation of what OP2 is doing ‘under the hood’ and will provide a sufficient foundation from which to interpret the data that follows.

4.1 Vector-Vector Operations

As has been outlined in section 2.3.3, the operations dotProduct, saxpy, copy and zero are examples of vector-vector operations. That is they combine two or more vectors in some way and produce an output which is either a scalar (in the case of dotProduct) or a vector (in the case of saxpy, copy and zero). In the parlance of OP2 these are referred to as direct sets since the mapping between the elements of any two vectors is the identity set [13]. The OP2 skeleton for calling direct set operations is outlined in listing 3 in section 2.3.3 and is schematically illustrated in figure 4.1.

![Figure 4.1: Schematic diagram of the algorithm skeleton for saxpy, copy and zero methods](image)

The figure defines an abstract operation that takes as input a set of scalars (c1 to c2) and a set of vectors (v1 to vn). In the case of saxpy there will be three input vectors and
one scalar, in the case of copy there would be two input vectors and in the case of zero there would be one input vector which is zeroed. The operation is applied to the vectors and scalars by passing over each index (illustrated by subscripts in the schematic), performing the computation and then storing the result in the output at the corresponding location. On the GPGPU each operation may be assigned to a thread and each thread may be run independently, as in this case there are no write conflicts. Since vectors are stored contiguously, this access pattern also results in utilising memory coalescing on the GPGPU [15]. Consequently, direct operations are very efficient on the GPGPU, and are expected to be efficient in OP2 since each each set element (i.e. a vector) contains only one data element (a double precision number) [13].

The implementation of the dotProduct is slightly different. This operation consists of first multiplying the input vectors together, element by element. These products must then be added together to acquire the final dot product. Schematically the algorithm is illustrated in figure 4.2. The important thing to note is that synchronisation is required between threads prior to the sum reduction operation. OP2 provides a reduction operation via the OP_INC parameter to the op_arg_dat function. The actual implementation of the reduction is based on the NVIDIA reduce API example [13, 16].

Figure 4.2: Schematic figure of the dotProduct

### 4.2 Matrix-Vector Operations

Matrix-vector multiplications comprise the most computationally intensive part of the Conjugate Gradient method. As such it is important to consider in some detail how they are performed in OP2. Figure 4.3 illustrates an example matrix-vector multiplication which will be used to highlight how this operation is executed in OP2. In the figure, the matrix $A$ is multiplied by the vector $x$ to give the resulting vector $y$. Values of $A$, $x$ and $y$ are shown component-wise.
The vectors in the above example are referred to by OP2 as indirect sets [13]. Considering the calling procedure in listing 7 of section 2.3.3 the set of non-zero values (ValueSet) is iterated over element by element. Each one of these non-zero values \(a_{i,j}\) indexes an element \(x_j\) by which it is multiplied and an element \(y_i\), to which the product \(a_{i,j} \times x_j\) is written. This information is stored in the EdgeMap.

Upon the first execution of algorithm 7, OP2 constructs a plan. This plan is stored on the GPGPU, and reused upon subsequent calls. The plan construction phase breaks up the indirect sets (the vectors) into chunks which can reside in the shared memory of a Streaming Multiprocessor (SM) on the GPGPU. Figure 4.4 gives an example of how non-zero values will be read up over 6 mini-partitions.

If there are \(M_{SM}\) Bytes of memory available for an SM, and \(N\) is the worst case mini-partition size, i.e. the maximum number of elements of an indirect set which may be stored. Then each mini-partition will require \(2N\) elements, one for \(x\) and one for \(y\) [17]. Since double precision numbers are used, the size of an indirect set element is 8 Bytes, thus the maximum number of elements per indirect set may be computed using \(M_{SM} = 2N \times 8 = 16N\).

As an example, if 48 Bytes of shared memory are assumed then the worst case block size, \(N\), may be calculated as \(2N \times 8 = 48\), i.e. \(N = 3\). Thus three elements of two indirect sets may be stored on an SM (i.e. the 6 elements which will fill the SM). This is a worst case estimate for the size of the block since there is the potential for reuse of \(x\) and \(y\) elements. OP2 will then read in \(N\) non-zero values since the worst case calculation has guaranteed that there will be a vector element to which it will eventually belong. To do this it makes use of the mapping information provided to work out which vector elements to store on the GPGPU [17, 13]. In the example, a worst case block size of 3 has been calculated, however, only a block size of 2 is required in practice (when reading in \(a_{0,0}\), \(a_{0,1}\) and \(a_{1,0}\) since \(y_0\) is reused. This reuse is important because it affects thread colouring. This reuse is also important, since it is strongly dependent on the structure of the matrix.
Figures 4.3 and 4.4 also illustrate the ideas of thread colouring and block colouring. In the example, mini-partition 0 requires 2 thread colours to avoid write conflicts. This is because $a_{0,0} \times x_0$ writes to $y_0$ and $a_{1,0} \times x_0$ writes to $y_1$, so they may have the same colour as they do not conflict. However $a_{0,1} \times x_1$ also needs to write to $y_0$ and must therefore have a different colour. This is the same for all mini-partitions except the last which only needs one thread colour. From figure 4.4, it can also be seen that a total of 2 block level colours are required. This is because any two mini-partitions share a value, for example mini-partition 0 and mini-partition 1 share the $y_1$ value and writing to this element must also be synchronised.
4.3 Summary

From the brief look at how OP2 handles vector-vector and matrix-vector operations, some important hypotheses can be made. Firstly, it would be expected that the operations involving direct sets would be at least comparable to the CUSPARSE and CUBLAS implementation. However from the brief look at how indirect sets are handled it can be surmised that a large number of thread or block colours result in a high degree of synchronisation and thus thread divergence, which could potentially slow down the computation. The number of thread colours may also be observed to be related to the maximum number non-zero values within an inter-block row; since all non-zeros in a row will write to the same $y_i$ value. This then suggests that the model employed by OP2 could be sensitive to the structure of the matrix.
Chapter 5

Benchmarking

In order to benchmark the Conjugate Gradient implementation a selection of matrices from the University of Florida (UFL) Sparse Matrix Collection [7] are used. The following, is a brief outline of the testing procedure which is explained in further detail below.

Each matrix is assumed to fit on to a single GPGPU, since NVIDIA’s CUSPARSE & CUBLAS libraries currently only target a single CUDA device. Consequently a simple estimate is established for the amount of space matrices will occupy on a CUDA device. Once matrices have been selected, they must be converted from their native Matrix Market format to a COO text file format (hence referred to as COO-format). A simple program, called convert, is supplied with the KS-framework project which allows conversion of Matrix Market (MM) formatted matrices to the much simpler COO-format supported by the KS-framework (this format is described in chapter 2).

The selected matrices are symmetric and positive definite, consequently they are guaranteed to converge using the Conjugate Gradient method. The chosen target residual is selected to be $10^{-6}$, this is an arbitrary value since the total running time for the conjugate gradient method is divided by the number of iterations to convergence.

The analysis concentrates primarily on the behaviour of the OP2 implementation of the Conjugate Gradient method, with particular focus on matrix-vector multiplication. This is primarily because matrix-vector multiplication is the most expensive part of the CG method and because the behaviour of OP2 differed so radically from the CUSPARSE & CUBLAS implementation.

5.1 Estimating Storage Requirements

The KS-framework code has been benchmarked on the Tesla C2050 GPGPU, which contains 14 SM cores (this may be established using the cudaGetDeviceProperties function [18]). Each core contains a total of 64KB memory which may be split as
16K/48K shared shared memory/level 1 cache (or vice versa) [19, 15]. Benchmarking assumes that SMs are configured to use 48K for shared memory, thus giving a raw maximum of \((48 \times 14)\)KB = 672KB shared memory. The Tesla C2050 also contains a total of 2GB global memory (established using cudaGetDeviceProperties).

The OP2 library is able to exploit more than one GPU on the system. However in order to compare like with like matrices are chosen so that they completely fit on to one GPU. OP2 and CUSPARSE/CUBLAS make use of the shared memory on the GPGPU and so it is interesting to see how crossing the 672KB threshold affects computation time, if it has any effect at all. In order to estimate the storage requirements for a sparse matrix, the formula in equation 5.1 is used.

\[
S_{\text{dbl}} \cdot nNz + 2 \cdot S_{\text{int}} \cdot nNz + 8 \cdot S_{\text{double}} \cdot \text{rows} = 16(nNz + \text{rows})
\] (5.1)

Here \(nNz\), represents the number of non-zero values in a matrix, \(\text{rows}\) represents the number of rows of the matrix (or columns since square matrices are used). \(S_{\text{int}}\) represents the size in bytes of an integer and \(S_{\text{dbl}}\) represents the size in bytes of a double value (it is assumed that \(S_{\text{int}} = 4\) and \(S_{\text{dbl}} = 8\)). The formula assumes eight arrays of size \(\text{rows}\), since this is the number of intermediate vectors which are required by the CG algorithm. It also assumes that there will be \(nNz\) lots of doubles and \(2 \times nNz\) lots of integers for the sparse matrix values and row and column index arrays respectively.

For large matrices with a 'reasonable' value-density (i.e. \(nNz \gg \text{rows}\)), this formula is a good estimate, since under these conditions GPU memory requirements will grow \(O(nNz)\) with only a small contribution from the size of vectors.

### 5.2 Selection of Matrices

As mentioned previously, the University of Florida Sparse Matrix Collection has been used as the source for benchmarking matrices. This is primarily because the collection contains a wide sample of 'real world' problems ranging from geomechanical models of the earths crust to circuit simulation. The selected matrices are all positive definite and symmetrical, since these are sufficient conditions for convergence - though not necessary since non positive definite systems may also converge (although this is not guaranteed).

Details for the benchmark matrices are depicted in appendix C. Table 1 in appendix C shows some of the basic properties of these matrices, such as the number of non-zero values and number of rows/columns and estimated storage size on the GPGPU.

One restriction on the choice of matrices is that the system that they form converges with the OP2 and CUSPARSE implementations of the CG algorithm. An example of a matrix which forms a system that does not converge is \(m_{t1}\) (figure 5r in appendix
D). This matrix had a condition number $\sim 10^{11}$, with a minimum value $\sim 10^{-3}$ and a maximum value $\sim 10^{11}$. In this case double precision arithmetic ran out of precision and the implementations of the Conjugate Gradient method did not converge.

### 5.3 Plotting Axes

In order to effectively compare the different Conjugate Gradient implementations against each other it is important to select the correct axes against which to plot the data. The running time for one iteration of the Conjugate Gradient loop is plotted against the number on non-zeros in the matrix. This metric was chosen because each system converges in a different number of iterations and so dividing the total running time of the CG loop by the number of iterations to convergence - allows comparison between systems formed from matrices that have different structures. The number of non-zeros was selected as the x-axis because, in terms of storage, the number of non-zeros will dominate the amount of information that may be placed on the GPGPU; the number of non-zeros will also dominate the total number of operations which are carried out by the CG implementation, since matrix-vector multiplication is the most expensive step. Furthermore, it has been shown in section 4 that the number of non-zero values may have an effect effect on the way in which vector elements are distributed to the Streaming Multiprocessors. In a sense using the number of non-zeros is misleading, since it is not explicitly affecting the running time, but providing a ‘best ordering’ for the data points of the graph.

When plotting the results of individual operations, the x-axis selected for the matrix-vector multiplication routine `mvmult` is once again the number of non-zero values, for the same reasons as before. However for dot products, and saxpy operations the x-axis is selected to be the number of rows of the matrix. This is due to the fact that these operations work only on the vectors and as such these vector values are all equivalent in size to the number of rows of the input matrix. In the case of these operations, the size of the vector directly affects the running time.

### 5.4 Benchmarking Methodology

The actual benchmarking was run on the fermi1 machine which contains 4 Tesla C2050 GPGPUs, 47GB of RAM and 4 six core Intel Xeon X5650 processors running at 2.67GHz. The operating system for this machine is Scientific Linux SL release 5.5, which is based on Redhat Enterprise Linux.

Benchmarks were run by reading in matrix and vector files in the COO-format described previously. The benchmark residual was selected to be $10^{-6}$. This is an arbitrary value since the overall running time for the CG method is normalised by the number of iterations to convergence. The boundary condition vector $b$, for each of the linear systems
\((Ax = b)\) being solved by the Conjugate Gradient implementations was selected to be the unit vector (i.e. all elements set to 1). Each benchmark was run three times, and the results averaged over runs. Fine timings, are the timings for each individual operation (mvmult, saxpy and dotProduct). These timings were calculated by running the benchmark and then averaging over the total number of each of the operations executed. Again three complete runs were executed, and an average taken of overall running times.

Benchmarks were run for the OP2, OP2 not-interleaved (OP2 - NIL), OP2 - serial and CUSPARSE & CUBLAS (CUSPARSE) versions of CG. The OP2 - serial version is the serial generated code from the standard OP2 implementation of ConjugateGradientOP2; it was selected for serial benchmarking since it was the fastest serial implementation (as opposed to ConjugateGradientSerial).

## 5.5 Results

### 5.5.1 Large Scale Features of Matrix-Vector Multiplication

Figure 5.1 shows the running time for the Conjugate Gradient routine divided by the number of iterations to convergence (i.e. the running time for one iteration) against the number of non-zero values of each of the selected benchmark matrices. The graphs presented are annotated with the letter for each of the matrices in appendix C and D.

![Figure 5.1: Conjugate Gradient running time divided by number of iterations vs. matrix size](image-url)
This graph shows several interesting features. Firstly, it is clear that the serial implementation is worse for large matrix sizes, since the running time increases sharply after point \( e \)\. The second thing to note is that the difference between interleaved and non-interleaved OP2 implementations is virtually indistinguishable. Consequently, using two separate arrays for the edge mapping makes no discernible difference to the running time of the Conjugate Gradient implementation. In general the CUSPARSE implementation shows evidence of low linear growth in running time after point \( h \), with small ‘blips’ on the order of \( 7 \times 10^{-4} \) and \( 3 \times 10^{-4} \) seconds around points \( k \) and \( l \) respectively. However the most striking feature of the graph is the ‘OP2-bump’ which occurs between points \( h \) and \( l \); and the ‘OP2-spike’ which occurs after point \( o \).

Figure 5.2 shows the running time for a single iteration of a matrix-vector multiplication. It can be clearly seen that the main features identified in figure 5.1 also appear in this graph. Since the shapes of the two graphs are so similar and since it is known that matrix-vector multiplication is the major contributor to the running time of the CG method it supports the view that the overall running time should be divided by the number of iterations when comparing different systems. Figure 5.2 also shows how the CUSPARSE implementation behaves as expected, with only a small deviation between points \( m \) and \( o \). It also should be noted that, overall, running times in figure 5.2 are slightly higher than those in figure 5.1. This can be explained by realising that timings are taken every time an operation is executed in figure 5.2, however the timings in figure 5.1 are only taken at the start and end of the CG algorithm and then normalised by total number of iterations.

Figure 5.2: Average time for matrix-vector multiplication vs. number of non-zeros

So far, the most striking features of the two graphs presented, the OP2-bump and OP2-spike have been unaccounted for. They are clearly not dependent purely on the number
of non-zeros, since the timing graph for OP2 dips back to near the running time for 
CUSPARSE at greater non-zero values (points l to o). Consequently the running time 
must be affected by the structure of the sparse matrix.

### 5.5.2 OP2 Matrix-Vector Multiplication Revisited

The clue to why the OP2-bump and OP2-spikes occur is to be found in the way in which 
vectors are distributed to shared memory on a Streaming Multiprocessor. From chapter 
4 it was ascertained that the non-zeros will be split in to blocks of size $M_{SM} = 2N \times 8$. 
In the case of the GPGPU, each SM has a maximum of 48K shared memory, this results 
in $49152 = 2^N \times 8$, or $N = 3072$. Consequently OP2 will assign 3072 double values 
for each of the the two partial vectors in the worst case. However in practice, as non- 
zeros are read in, some of the vector elements will be reused. The example in figure 5.3 
assumes that $N = 6$ (i.e each SM can hold a maximum of $6 \times 2 \times 8 = 96$ Bytes). In 
reality 48 Bytes will be allocated for blocks 0 - since $y_0$ is used twice, $y_1$ is used three 
times and $y_2$ is used once (this is also the case for block 1); and 32 Bytes for block 3 
(since $y_4$ and $y_5$ are both used twice).

![Figure 5.3: Estimating thread colouring - (block 0 is black, block 1 is red, block 2 is blue)](image)

Now the maximum reuse of each block determines the number of thread colours which 
will be required within that block. The higher the number of thread colours within a 
block the greater the time taken to execute the sub matrix-vector multiplication within 
that block due to warp divergence since each thread within a colour must wait for its 
neighbours to complete writing (followed by a `$cudaThreadSynchronize` call) [13]. 

The number of thread colours per block may be estimated by splitting the number of 
non-zero values into buckets of size 3072 and counting across rows to find the maxi- 
mum row size per block. This is not as simple as just taking the maximum row size
since, as figure 5.3 shows rows themselves may be split across blocks (in practice it was found that a row would only be split across two blocks). Once the maximum thread colouring of each block is taken, it is possible to work out the average number of thread colours per block by taking the average of the block maximums. For the example in figure 5.3, the average number of thread colours is 2.66. Based on this idea, the Matlab script in appendix E may be used to estimate the average number of thread colours needed per OP2 block, this value is then defined as \( B_{tc} \). It is worth emphasising that OP2 produces this ‘average thread colouring’ statistic when diagnostic information is enabled. However OP2 is a framework designed for unstructured grid applications and so doesn’t know about matrices - it calculates this value by examining the connectivity of the input mesh. The Matlab script found in appendix E, calculates ‘average thread colouring’ directly from the matrix.

Calculating the average number of thread colours per block is only part of the picture. In order to gain a full understanding of why the OP2 algorithm has the general shape found in figures 5.1 and 5.2 it is also necessary to account for the number of blocks which will run on the GPGPU. This may be achieved by dividing the total number of non-zeros by the number of non-zeros which will be processed by an SM. This is outlined in equation 5.2, where \( B_n \) is the number of blocks and \( nNz \) is the number of non-zeros.

\[
B_n = \left\lceil \frac{nNz}{3072} \right\rceil \tag{5.2}
\]

Since there are 14 SMs per GPGPU, 14 blocks may be run at any one time, and so it would be expected that that matrices with high inter-block thread averages, but block sizes of less than 14, would not be affected by thread colouring to the degree that larger matrices (occupying larger numbers of blocks) would. In order to demonstrate this, a ‘structure-effect factor’ \( (S_f) \) is calculated. This is a metric which should give a rule of thumb guide as to how well the matrix vector operation will perform using OP2. It is defined in equation 5.3, which assumes that 14 blocks will run without collisions - a fair assumption for the data since the number of block colours observed in practice was always 2.

\[
S_f = \frac{B_n}{14} \times B_{tc} \tag{5.3}
\]

Plotting \( S_f \) against the number of non-zeros results in the graph presented in figure 5.4. It is immediately obvious that the structure effect factor does not exactly match the results found in 5.2, for example, \( S_f \) is concave points \( h \) and \( k \) rather than convex. This can be accounted for by examining the sparsity graphs 5h, 5i, 5j and 5k (found in appendix D). The \( S_f \) value is based only on a crude average and does not take account of any gaps on a line. These gaps would not be read in by an SM and so the indexing for
the vector values would not be contiguous. It is clear that 5h, 5i and 5k have large gaps between small clusters of row values. Figure 5j is deceptive since it is not apparent that there are gaps within a row. However this is the case since, for example, in row 1 there is a gap between the 5th and 1013th columns. The combination of these large gaps and small clusters of values would have an adverse effect when reading and writing vector values between SMs and global memory. However $S_f$ does capture the main features of how matrix-vector multiplication behaves. It can be seen that if $S_f$ is still relatively large in the OP2-bump region and it estimates the relationship between the first and last points. It also shows the steep drop after the OP2-bump region as in 5.2, before rising again in the spike region.

![Graph showing $S_f$ value vs. number of non-zeros](image)

**Figure 5.4:** The $S_f$ value vs. number of non-zeros

### 5.5.3 The ‘dotProduct()’ and ‘saxpy()’ Operations

Graphing the performance of row based operations is much simpler than depicting matrix-vector multiplication since these operations work only on vectors. These vectors are dense, contiguous and directly relate to the running time. The vertical black line depicts the maximum number of row values which can fit into the total of 672K of shared memory (this corresponds to 86016 row elements).
Examining figures 5.5 and 5.6 it can be observed that for smaller vector sizes the CUBLAS library is better than OP2 by a small amount. Up to the 672K boundary both OP2 and CUBLAS are relatively flat hinting that the utilisation of shared memory could be responsible for the performance increase. From the small timing values, and the shape of the graph it is clear that dot product and and saxpy operations contribute only a small amount to the overall running time for a Conjugate Gradient implementa-
tion. It is also interesting to note that for figure 5.6 the OP2 implementations of saxpy perform better than CUBLAS after the 672K boundary.

### 5.5.4 The ‘setup()’ Procedure

The setup method is plotted against the estimated storage size using equation 5.1. This is because this value accounts for the intermediate vectors required for running the Conjugate Gradient algorithm and so gives a more accurate indication of the amount of data transferred from the host to the device. Points have been labelled with respect to the matrix which is part of the system being solved. The serial graph stops at $5 \times 10^4$ KB since benchmarks were not run for the serial version beyond this size.

![Figure 5.7: Setup time vs. number of matrix non-zero values](image)

It can be observed from figure 5.7 that the non-interleaved version (OP2-Nil) of the OP2 CG implementation is much better than than the interleaved version (OP2). This is because of the interleaving step, which is behaves $O(nNz)$ since the row and column index arrays must be traversed in order to construct an interleaved edge array. The CUSPARSE-CUBLAS version is worse than the OP2 and OP2-Nil versions for small matrices because there is still a conversion step which must convert the input coordinate format matrix to CSR, however this cost becomes less significant as the size of the system increases due to the bandwidth limitation between device and host.
Chapter 6

Future Work and Conclusions

Some possible future directions for the work presented are outlined, followed by a brief summary of conclusions.

6.1 Future Work

Given the short amount of time and the inevitable stumbling blocks encountered, there was not enough time to pursue several interesting branches of inquiry. However some possibilities for future work are outlined below.

6.1.1 Refactoring the KS-Framework

As it stands, the organisation of the KS-framework is not optimal. An alternative design is proposed in figure 6.1. Here the matrix-vector operations have been factored out of the implementation into classes of their own by making use of the strategy design pattern [8]. The advantage of this design is that vector-vector and matrix-vector functionality may be reused more easily in other KS implementations. This is also useful, since the OP2 preprocessor will only need to operate on a single class, instead of multiple classes. Furthermore, due to the current implementation, OP2 header files may only be included once in a program since the model assumes a monolithic application structure. Using the strategy pattern should also help to alleviate this problem.

In the proposed design shown in figure 6.1, the MatrixVectorOps class defines the basic vector-vector and matrix-vector operations which are currently defined in the ConjugateGradient child classes. The classes which inherit from MatrixVectorOps are then responsible for providing specific implementations, such as OP2 and CUSPARSE/CUBLAS. In the new design the abstract Krylov class, holds a reference to an abstract MatrixVectorOps object which is then available to all implementing classes. Actual concrete implementations such as ConjugateGradientOp2 then responsible for
creating the correct type of MatrixVectorOps. The dotted line in the diagram indicates that there is a relationship between the concrete MatrixVectorOps implementations provider and the corresponding Krylov class. For example the ConjugateGradientOP2 class is responsible for creating a MatrixVectorOpsOP2 class to perform its operations.

This design is an evolution of what has been implemented as part of the KS-framework. The only real difference is that some functionality has been encapsulated elsewhere. The ConjugateGradient class still makes use of the template design pattern, with delegation occurring as before. As an additional detail, each ConjugateGradient implementation will still be responsible for handling device and host address spaces.

It is the opinion of the author that this change should be fairly straightforward to implement, and would have advantages in terms of code organisation and reusability. It is also expected build times would be lower, since there would be only single file to be parsed by the OP2 preprocessor.
6.1.2 Different Input File Formats

The code presented works with only a simple input file format, which must contain all the the sparse matrix information. This is clearly wasteful in the case of symmetrical
matrices, as only half of the information needs to be stored. Consequently it would be useful to add functionality to read in different file formats. Adding support for the Matrix Market format is particularly straightforward as the host web site contains a selection of utility functions which could be used [20].

6.1.3 Different Matrix Formats

As has been demonstrated, OP2 is dependent on the structure of the matrix. Consequently it may be possible to gain speed increases by approaching the matrix-vector multiplication problem differently. Two examples of different approaches are shown below.

Tridiagonal Matrices Revisited

Figure 5.3 in section 5 suggests an interesting possibility for optimisation of matrix-vector multiplication. This figure shows that two blocks contend to write to \( y_0 \) and \( y_4 \) in the output vector when each SM takes 6 blocks. Figure 6.2 shows that if the start of the matrix is padded with a zero, block level colouring is no longer required. However there is an effect on the maximum number of thread colours within a block. It would be interesting to explore whether the increase in thread colours is offset by the decrease in block colouring.

![Figure 6.2: Padding the tridiagonal matrix to eliminate block level colouring](image)

Block Based Matrices

Coding matrices as blocks is one other strategy which may be employed to improve the matrix-vector multiplication operation. Figure 6.3 illustrates how system consisting of
a $9 \times 9$ matrix and two vectors of length 9, is reduced to smaller blocks. In the example, the larger matrix is split up into $3 \times 3$ blocks; any block containing a non-zero value is padded with zeros. The vectors are also ‘aggregated’ in this manner. This results in a conceptually smaller system, shown at the bottom of the diagram. The system is not in reality smaller, it is larger - since the additional zeros add more computation. However the cost of additional computations can be compensated for by the structural changes to the matrix, and the fact that arithmetic operations are relatively cheap. Furthermore, the indexing of elements is simplified since each element is larger.

The main difficulty with this method arises for determining an optimum BDIM value. Usually the way in which the problem is constructed should suggest an optimum value BDIM value, since in practice it is always known how a the linear system is derived. Implementing block matrix-vector multiplication in OP2 is not too difficult and a proof of concept is provided in appendix F. The main difficulty is with converting a large matrix, usually supplied in a coordinate matrix format. This process could itself be done on the GPGPU but is not trivial.
6.1.4 Different Krylov Subspace Algorithms

The original project proposal envisaged implementing several different types of Krylov Subspace algorithm for the KS-framework. However, in the end only the Conjugate Gradient method was chosen due to time constraints. It should be relatively simple to implement other KS solvers using the framework, for example BiCGStab should be fairly simple to implement. This would certainly be the case if the code refactoring outlined previously is adopted.

6.2 Conclusions

In the course of this project several important goals have been realised. The primary purpose of the work presented here was to build a framework to allow future users to make use of Krylov Subspace methods in a simple and elegant manner. The framework described is modular and may be extended easily. Wrapping the OP2 framework was not a trivial task. There were many obstacles to overcome, particularly in reconciling OP2’s implementation with the object orientated design paradigm. However, it has been demonstrated that this is a fruitful endeavour, since the approach taken also allows other libraries such as CUSPARSE and CUBLAS to be incorporated in to the design. This also opens up the possibility of targeting different platforms and other libraries.

The OP2 framework has also been benchmarked and compared against the CUSPARSE implementation, and found to diverge drastically in execution time for certain types of matrix. The sensitivity of OP2 to the structure of the input matrix when solving linear systems is an interesting result, which can be traced back to the approach taken to solving unstructured mesh computations. This was not a trivial piece of analysis since the reason for OP2’s behaviour was only revealed after normalising timing information for a complete run of a CG implementation with matrices of varying structures; along with detailed understanding of how OP2 works. Graphing the running time divided by the number of iterations against the number of non-zeros in the input matrix was the ‘best’ approach, since it was not possible to fully disentangle all of the factors at work when interpreting the timing results.

It would have been useful to have available a method of creating a matrix, the size of which could be scaled, yet remain positive definite and symmetric (e.g. a code which could generate matrices encoding a PDE at finer and finer meshes). This would have lead to the non-zero values having a much more explicit meaning when plotting time graphs. As it is, the author tried to extract the most out of the the data available. The calculation of the ‘structure effect factor’ demonstrates why OP2 performs badly in some cases; it also opens up the possibility that re-ordering the rows of the matrix may improve OP2 matrix-vector multiplication, by optimising for such a metric.
Appendices
Algorithm 7 Implementation of the Conjugate Gradient Method vs. Pseudocode

setup(matA, vecb);

T num, den, alpha, beta;
T r_kp1_norm, b_norm;
int k = 1;

gaxpy(-1, A, v_k, 1, b, r_k);
gaxpy(-1, A, v_k, 1, b, p_k);

while (k <= MAX_ITER) {
mvmult(A, p_k, mvec);
alpha = num = den = 0.0;
dotProduct(r_k, r_k, &num);
dotProduct(p_k, mvec, &den);
alpha = num/den;
saxpy(alpha, p_k, v_k, v_kp1);
saxpy(-alpha, mvec, r_k, r_kp1);

r_kp1_norm = b_norm = 0.0;
dotProduct(r_kp1, r_kp1, &r_kp1_norm);
dotProduct(b, b, &b_norm);
r_kp1_norm = sqrt(r_kp1_norm);
b_norm = sqrt(b_norm);

if ( r_kp1_norm / b_norm < TOL) {
    break;
}

num = den = 0.0;
dotProduct(r_kp1, r_kp1, &num);
dotProduct(r_k, r_k, &den);
beta = num/den;
saxpy(beta, p_k, r_kp1, p_kp1);

copy(v_kp1, v_k);
copy(r_kp1, r_k);
copy(p_kp1, p_k);
zero(mvec);
k++;
}

if (||r_{k+1}||_2/||b||_2 < tol) {
    break
}

endwhile

Set v_k to zero (done on host)
Compute r_k ← b − Av_k
Set p_k ← r_k

while (k ≤ maxiter)
α ← r_k\text{T} · r_k/p_k\text{T} · Ap_k

v_{k+1} ← v_k + αp_k
r_{k+1} ← r_k − αAp_k

||r_{k+1}||_2/||b||_2 < tol

β ← r_{k+1}\text{T} · r_k/r_k\text{T} · r_k

p_{k+1} ← r_{k+1} + βp_k

k ← k + 1

endwhile
Appendix B - Graphs of Verification Errors

Behaviour of verification error for six randomly generated test systems of the form $Ax = b$. Each graph depicts the errors for the Serial, Serial OP2, OP2 and CUSPARSE implementations overlaid. However in most cases differences in error as so small as to be indistinguishable.

![Graph of Verification Errors](image)

(a) Verification errors 100x100 test matrix

![Graph of Verification Errors](image)

(b) Verification errors 90x90 test matrix

Figure 4: Verification errors for test linear system (part 1)
Figure 4: Verification errors for test linear system (part 2)
Figure 4: Verification errors for test linear system (part 3)
Appendix C - Statistics for Benchmark Matrices

The following table illustrates the basic properties of the selected benchmark matrices. There is only one entry for rows/columns since all matrices are square. The ‘nNz’ column describes the number of non-zero entries in the matrix and the KB and MB columns show the estimated matrix size. The matrix m_t1 is included for completeness but is not used for benchmarking.

<table>
<thead>
<tr>
<th>Ref</th>
<th>UFL ID</th>
<th>Name</th>
<th>Cnd. no.</th>
<th>Rows/Cols</th>
<th>nNz</th>
<th>KB</th>
</tr>
</thead>
<tbody>
<tr>
<td>a</td>
<td>27</td>
<td>bcsstk05</td>
<td>3.53E+04</td>
<td>153</td>
<td>2423</td>
<td>40.25</td>
</tr>
<tr>
<td>b</td>
<td>24</td>
<td>bcsstk02</td>
<td>1.29E+04</td>
<td>66</td>
<td>4356</td>
<td>69.09</td>
</tr>
<tr>
<td>c</td>
<td>62</td>
<td>bcsstm07</td>
<td>1.25E+04</td>
<td>420</td>
<td>7252</td>
<td>119.88</td>
</tr>
<tr>
<td>d</td>
<td>31</td>
<td>bcsstk09</td>
<td>3.10E+04</td>
<td>1083</td>
<td>18437</td>
<td>305.00</td>
</tr>
<tr>
<td>e</td>
<td>339</td>
<td>bcsstk34</td>
<td>3.64E+04</td>
<td>588</td>
<td>21418</td>
<td>343.84</td>
</tr>
<tr>
<td>f</td>
<td>49</td>
<td>bcsstk27</td>
<td>7.71E+04</td>
<td>1224</td>
<td>56126</td>
<td>896.09</td>
</tr>
<tr>
<td>g</td>
<td>758</td>
<td>nasa2146</td>
<td>4.36E+03</td>
<td>2146</td>
<td>72250</td>
<td>1162.44</td>
</tr>
<tr>
<td>h</td>
<td>1330</td>
<td>Kuu</td>
<td>3.26E+04</td>
<td>7102</td>
<td>340200</td>
<td>5426.59</td>
</tr>
<tr>
<td>i</td>
<td>752</td>
<td>finan512</td>
<td>9.84E+01</td>
<td>74752</td>
<td>596992</td>
<td>10496.00</td>
</tr>
<tr>
<td>j</td>
<td>845</td>
<td>qa8fm</td>
<td>1.10E+02</td>
<td>66127</td>
<td>1660579</td>
<td>26979.78</td>
</tr>
<tr>
<td>k</td>
<td>1849</td>
<td>Dubcova3</td>
<td>1.15E+04</td>
<td>146689</td>
<td>3636643</td>
<td>59114.56</td>
</tr>
<tr>
<td>l</td>
<td>1853</td>
<td>parabolic_fem</td>
<td>2.11E+05</td>
<td>525825</td>
<td>3674625</td>
<td>65632.03</td>
</tr>
<tr>
<td>m</td>
<td>1423</td>
<td>apache2</td>
<td>5.32E+06</td>
<td>715176</td>
<td>4817870</td>
<td>86453.84</td>
</tr>
<tr>
<td>n</td>
<td>1883</td>
<td>ecology2</td>
<td>6.66E+07</td>
<td>999999</td>
<td>4995991</td>
<td>93687.34</td>
</tr>
<tr>
<td>o</td>
<td>1421</td>
<td>G3_circuit</td>
<td>2.24E+07</td>
<td>1585478</td>
<td>7660826</td>
<td>144473.50</td>
</tr>
<tr>
<td>p</td>
<td>1403</td>
<td>thermal2</td>
<td>7.48E+06</td>
<td>1228045</td>
<td>8580313</td>
<td>153255.59</td>
</tr>
<tr>
<td>q</td>
<td>942</td>
<td>af_shell3</td>
<td>1.37E+06</td>
<td>504855</td>
<td>17562051</td>
<td>282295.41</td>
</tr>
<tr>
<td>r</td>
<td>1269</td>
<td>m_t1</td>
<td>3.17E+11</td>
<td>97578</td>
<td>9753570</td>
<td>153924.19</td>
</tr>
</tbody>
</table>

Table 1: Basic properties of the selected benchmark matrices.
Appendix D - Sparsity Plots of Benchmark Matrices

Sparsity plots of selected benchmark matrices. The reader should note that the matrix m_t1 in figure 5r is not used for benchmarking since it demonstrates non-convergent behaviour.

Figure 5: Sparsity plots of the selected matrices (part 1)
Figure 5: Sparsity plots of the selected matrices (part 2)
Figure 5: Sparsity plots of the selected matrices (part 3)
Figure 5: Sparsity plots of the selected matrices (part 4)
Figure 5: Sparsity plots of the selected matrices (part 5)
Appendix E - Thread Colours in an OP2 Mini-Partition

The code in listing 22 is a Matlab script which outlines how average thread colouring may be calculated.

**Listing 22** Estimating block size

```matlab
function colour_est = threadColourEstFun( A )

% Estimates the average thread colouring.
% Input:
% A - a sparse matrix
% Output:
% colour_est - the estimate of the average number
% of thread colours per block.

[rowIdx, colIdx, values] = find(A);
M = [rowIdx, colIdx, values];

% Make sure that data is in row major order
Ms = sortrows(M,1);

nNz = size(M,1);

itemsPerBlock = 3072;
blocks = ceil(nNz/itemsPerBlock);

values = [];

% Loop in buckets of size 'itemsPerBlock'
for i=1:blocks
    from = (i-1)*itemsPerBlock + 1;
    if (i==blocks)
        to = nNz;
    else
        to = i*itemsPerBlock;
    end

    % Calculate the number of non-zero rows with the same row index
    chunk = Ms(from:to,1);
    [a,b] = hist(chunk,unique(chunk));

    % Add the maximum to the list of block maximums
    values=[values max(a)];
end

% Take the block average
colour_est = mean(values);
end
```
The call to \texttt{hist} will construct a histogram for each block containing the number of non-zeros per row, as shown in figure 6. These histograms are based on the example presented in figure 5.3, chapter 5.

Figure 6: Example row count histograms for each block in figure 5.3
Appendix F - Block Matrix Vector Multiplication

Listing 23 presents a proof of concept for block matrix vector multiplication.

Listing 23 Block matrix vector multiplication

/ *
 * standard headers
 */

#include <stdlib.h>
#include <stdio.h>
#include <string.h>
#include <math.h>
#include <iostream>
using namespace std;

/ *
 * OP header file
 */

#include "op_seq.h"

#include "mvBlockMultKernel.h"

/**
 * Compute a matrix vector multiplication.
 */
void mvmult(op_dat *A, op_dat *b, op_dat *y);

op_set MatrixBlockSet;
op_set VectorBlockSet;
op_map Edge;
Listing 23 Block matrix vector multiplication, continued...

```c
/**
 * Main function
 */
int main(int argc, char **argv)
{
    double b_host[15] = {
        3.18487665148575, // block 0
        3.46838173324202,
        3.68216227705289,
        2.58044780697623, // block 1
        2.65264503886913,
        2.69836760977502,
        2.92523943100291, // block 2
        2.4895006633681,
        2.98799928568279,
        3.3326297594396, // block 3
        3.3073497587169,
        2.607086642594,
        2.60832699927523, // block 4
        2.70863471158222,
        3.2248693653952
    };

    // sets all elements to zero (in c++)
    double y_host[15] = {0.0};

    /**
     * A Matrix which looks like this:
     *          B0 B1 ** ** **
     *          B2 B3 ** ** **
     *          ** ** B4 ** **
     *          ** ** ** B5 B6
     *          ** ** ** B7 B8
     */
    double A_host[9*9] = {
        // block 0 (goes with vector block 0)
        3.500359369953650, 1.329321196633150, 0.843850710771907,
        1.329321196633150, 3.243647801471900, 0.693594738867917,
        0.843850710771907, 0.693594738867917, 3.29345993863470,
    }
}
```
Listing 23  Block matrix vector multiplication, continued...

// block 1 (goes with vector block 1)
1.897902116880710, 1.776807412885210, 1.092185887680080,
1.776807412885210, 1.792206916066710, 1.073366825378520,
1.092185887680080, 1.073366825378520, 1.036585864704890,

// block 2 (goes with vector block 0)
1.897902116880710, 1.776807412885210, 1.092185887680080,
1.776807412885210, 1.792206916066710, 1.073366825378520,
1.092185887680080, 1.073366825378520, 1.036585864704890,

// block 3 (goes with vector block 1)
3.978214033843590, 1.455481576818770, 1.752047085186140,
1.455481576818770, 3.594639288928470, 1.361738830212260,
1.752047085186140, 1.361738830212260, 3.587097381110110,

// block 4 (goes with vector block 2)
2.612798181448170, 0.345201608313854, 0.222742420152518,
0.345201608313854, 2.576200748115930, 0.263725510613803,
0.222742420152518, 0.263725510613803, 2.689649175272360,

// block 5 (goes with vector block 3)
3.486260837659900, 0.525624989859949, 0.961630920140873,
0.525624989859949, 2.339270001771510, 0.446561347000641,
0.961630920140873, 0.446561347000641, 3.253280206039300,

// block 6 (goes with vector block 4)
0.856136732684869, 0.862098207511404, 0.729894881581938,
0.862098207511404, 1.148940294351920, 1.021435830120340,
0.729894881581938, 1.021435830120340, 0.917591700155570,

// block 7 (goes with vector block 3)
0.856136732684869, 0.862098207511404, 0.729894881581938,
0.862098207511404, 1.148940294351920, 1.021435830120340,
0.729894881581938, 1.021435830120340, 0.917591700155570,

// block 8 (goes with vector block 4)
3.183635642432450, 0.546983676951976, 0.702118497273915,
0.546983676951976, 2.508492567513740, 0.518258757445250,
0.702118497273915, 0.518258757445250, 2.885468250458620);
Listing 23 Block matrix vector multiplication, continued...

```cpp
int pedge[9*2] = {
    0, 0,
    0, 1,
    1, 0,
    1, 1,
    2, 2,
    3, 3,
    3, 4,
    4, 3,
    4, 4};

// There are nine blocks in a matrix
for (int i = 0; i < 9; i++) {
    cout << pedge[i*2] << " , " << pedge[i*2+1] << endl;
}

cout << "pedge[10]:" << pedge[10] << endl;

op_dat A; // Matrix A
op_dat b; // Vector b
op_dat y; // Vector y (result of A.b)

// There are 9 blocks in a matrix
// i.e. 9 set elements in a MatrixBlockSet
MatrixBlockSet = op_decl_set(9, "double");

// There are 5 blocks in a vector
// i.e. 5 set elements in a VectorBlockSet
VectorBlockSet = op_decl_set(5 , "double");

// Map MatrixBlockSets to VectorBlockSets
Edge = op_decl_map(MatrixBlockSet,
                    VectorBlockSet, 2,
                    &pedge[0], "Edge");

// There are 9 elements in a matrix block
// i.e. 9 items per MatrixBlockSet element
A = op_decl_dat(MatrixBlockSet, 9, "double", A_host, "A");

// There are 3 elements in a vector block
// i.e. 3 items per VectorBlockSet element
b = op_decl_dat(VectorBlockSet, 3, "double", b_host, "b");

// There are 3 elements in a vector block
// i.e. 3 items per VectorBlockSet element
y = op_decl_dat(VectorBlockSet, 3, "double", y_host, "y");
```
Listing 23 Block matrix vector multiplication, continued...

```c
// Call the matrix vector function
mvmult(&A, &b, &y);

// Retrieve data to host
op_fetch_data(y);
op_fetch_data(b);

// Display data
for (int i = 0; i < 15; i++) {
    std::cout << y_host[i] << std::endl;
}
std::cout << std::endl;
for (int i = 0; i < 15; i++) {
    cout << b_host[i] << endl;
}
op_exit();
}
```

The kernel function for mvBlockMultKernel is shown in listing 24. The variable BDIM is defined at compile time and is the dimension of a (square) block.

Listing 24 Block matrix vector multiplication kernel

```c
inline void mvBlockMultKernel(double *A, double *b, double *r)
{
    for (int i = 0; i < BDIM; i++) {
        double r_loc = 0;
        for (int j = 0; j < BDIM; j++) {
            r_loc += A[BDIM*i + j] * b[j];
        }
        r[i] += r_loc;
    }
}
```
Bibliography


