MANCHESTER
1824

The University of Manchester

Implementation of QR Updating Algorithms on the GPU

Andrew, Robert

2012

MIMS EPrint: **2012.80**

Manchester Institute for Mathematical Sciences
School of Mathematics

The University of Manchester

Reports available from: http://eprints.maths.manchester.ac.uk/
And by contacting: The MIMS Secretary
School of Mathematics
The University of Manchester
Manchester, M13 9PL, UK

ISSN 1749-9097

Implementation of QR Updating Algorithms on the GPU

Robert Andrew

2012

A dissertation submitted to the University of Manchester for the degree of MSc Mathematics and Computational Science in the Faculty of Engineering and Physical Sciences

Abstract

The least squares problem is an extremely useful device to represent an approximate solution to overdetermined systems, and a QR factorisation is a common method for solving least squares problems. It is often the case that multiple least squares solutions have to be computed with only minor changes in the underlying data. In this case, knowledge of the difference between the old data set and the new one can be used to update an existing QR factorisation at a reduced computational cost. However, fairly recent developments have introduced the widespread use of massively parallel computational devices known as GPUs. GPUs have allowed QR factorisations, and subsequently, least squares solutions to be calculated in a greatly reduced time. The purpose of this project is to investigate the viability of the implementation of QR updating algorithms on the GPU and attempt gain speedup with a GPU based updating algorithm over both existing sequential QR updating algorithms, and full GPU QR factorisations. The conclusion of the investigation is that GPU based updating algorithms gain speedups over their sequential analogues for almost all problem sizes, whereas the proposed algorithms only gain speedups over the full GPU QR factorisation under certain conditions.

Declaration

No portion of the work referred to in the dissertation has been submitted in support of an application for another degree or qualification of this or any other university or other institute of learning.

Copyright

The author of this dissertation (including any appendices and/or schedules to this dissertation) owns certain copyright or related rights in it (the Copyright) and s/he has given The University of Manchester certain rights to use such Copyright, including for administrative purposes.

Copies of this dissertation, either in full or in extracts and whether in hard or electronic copy, may be made only in accordance with the Copyright, Designs and Patents Act 1988 (as amended) and regulations issued under it or, where appropriate, in accordance with licensing agreements which the University has entered into. This page must form part of any such copies made.

The ownership of certain Copyright, patents, designs, trade marks and other intellectual property (the Intellectual Property) and any reproductions of copyright works in the dissertation, for example graphs and tables (Reproductions), which may be described in this dissertation, may not be owned by the author and may be owned by third parties. Such Intellectual Property and Reproductions cannot and must not be made available for use without the prior written permission of the owner(s) of the relevant Intellectual Property and/or Reproductions.

Further information on the conditions under which disclosure, publication and commercialisation of this dissertation, the Copyright and any Intellectual Property and/or Reproductions described in it may take place is available in the University IP Policy (see http://documents.manchester.ac.uk/display.aspx?DocID=487), in any relevant Dissertation restriction declarations deposited in the University Library, The University Librarys regulations (see http://www.manchester.ac.uk/library/aboutus/regulations) and in The Universitys Guidance for the Presentation of Dissertations.

Acknowledgements

Thanks to the Numerical Analysis Group of the University of Manchester for their comments and ideas for future work.

I would also like to thank Dr Nicholas Dingle for his excellent supervision and support throughout this project.

Contents

1 Introduction **6**

1.1 Notation and Organisation . . . 6

2 Background Material **8**

2.1 QR Factorisations and the Least Squares Problem. . . . 8

2.2 The GPU Programming Model . . . 9

2.2.1 The Programming Model . . . 10

2.2.2 Thread Communication . . . 10

2.2.3 Memory Hierarchy . . . 11

2.2.4 Heterogeneous Programming . . . 12

2.2.5 BLAS subroutines . . . 13

2.3 QR Factorisations . . . 13

2.3.1 Householder Reflections . . . 14

2.3.2 Givens Rotations . . . 14

2.4 QR Updating Algorithms . . . 15

2.4.1 Adding Columns . . . 15

2.4.2 Removing Columns . . . 16

2.4.3 Adding Rows . . . 17

2.4.4 Removing Rows . . . 18

3 Parallelising Householder and Givens **20**

3.1 Parallelising Householder Reflections . . . 20

3.2 Parallelising Givens Rotations . . . 23

4 QR Updating Algorithms **29**

4.1 Adding a Block of Columns . . . 29

4.2 Removing a Block of Columns . . . 32

4.3 Adding a Block of Rows 33

4.4 Removing a Block of Rows 38

5 Results **42**

5.1 Limitations 43

5.2 Choosing the Block Size Parameter 43

5.3 Adding Columns 46

5.3.1 Complexity 46

5.3.2 Adding Columns GPU TUT vs Sequential Update 46

5.3.3 Adding Columns GPU TUS vs CULA Solve 48

5.4 Removing Columns 54

5.4.1 Complexity 54

5.4.2 Removing Columns GPU TUT vs Sequential Update 54

5.4.3 Removing Columns GPU TUS vs CULA Solve 56

5.5 Adding Rows 60

5.5.1 Complexity 60

5.5.2 Adding Rows GPU TUT vs Sequential Update 60

5.5.3 Adding Rows GPU TUS vs CULA Solve 61

5.6 Removing Rows 67

5.6.1 Complexity 67

5.6.2 Removing Rows GPU TUT vs Sequential Update 67

5.6.3 Removing Rows GPU TUS vs CULA Solve 69

5.7 Stability 73

6 Conclusion **75**

6.1 Future Work 78

17,175 words

Chapter 1

Introduction

An overdetermined system is defined as a set of equations containing one or more unknown variables, where the number of equations exceeds the number unknowns. An overdetermined system can often represent observations from some physical system that needs to be modelled. The difficulty with solving overdetermined systems however is that they are often ill posed such that there exists no exact solution to the system. The Least squares method is a common approach to providing an approximate solution to overdetermined systems. There are many methods for solving a least squares problem, however the one that this project is concerned with is the least squares solution via QR factorisation. QR factorisations are computationally expensive procedures but when many QR factorisations need to be calculated with small adjustments in the underlying data, some of this cost can be amortised by QR updating algorithms.

There are 4 general types of updates to the underlying data matrix A, adding a block of columns, removing a block of columns, adding a block of rows, and removing a block of rows. Adding and removing a block of columns from the problem matrix can be interpreted as respectively adding and removing a set of variables from the overdetermined system. Adding and removing rows on the other hand can be interpreted as respectively adding and removing equations from the overdetermined system. In this project I will attempt to accelerate QR updating algorithms on the GPU via a mainstream massively parallel GPU programming language, CUDA. This project investigates the resulting algorithms for their runtime and accuracy against their sequential analogue and a full QR algorithm calculated from scratch.

1.1 Notation and Organisation

Within this project, we denote with A the data matrix undergoing QR factorisation. This matrix has dimensions $n \times m$, n rows and m columns. Matrix 'updates' will entail the addition or removal of contiguous blocks of p columns, or p rows. When a block of columns or rows is added during an update, this block is denoted U. The location of an update within the data matrix A is given as an offset in columns or rows from the top left corner and is denoted k. The products of the factorisation will be denoted Q and R respectively.

All host code (as opposed to kernel code) will be presented in this project as pseudocode. My pseudocode for simplicity will be in a Matlab style, with similar syntax with regards to commenting, indexing, and logical operators. The algorithms however are meant to be implemented in CUDA C so I have not omitted the important memory management statements and other operations specific to massively parallel programming. Note that in future pseudocode I will refer to previously defined functions such as `blockedQRFactorisation()` defined in its own listing in Section 3.1 will be taken as a reference to the listing, any subsequent adjustments to the algorithm will then be clearly stated. Kernel calls within the host pseudocode are identified by the <<<>>> (angle brackets) and are intended to be placeholders, some arguments and grid and block definitions may be omitted for simplicity reasons.

Chapter 2 presents background material. Core concepts are discussed regarding the least squares problem, QR factorisation and updating procedures. An introduction to GPUs and the GPU programming model is also given.

Chapter 3 gives details of the implementation of Givens and Householder algorithms presented in this paper. Documented CUDA kernels are given along with details of the calling code for execution on the host.

Chapter 4 applies the ideas investigated in chapter 3 to the four updating algorithms introduced in chapter 2. Data and computations are structured such that the algorithms will run as efficiently as possible with CUDA on the GPU. A CUDA specific library, CUBLAS, is also employed to build some aspects of the algorithms presented.

Chapter 5 presents the results of the parallelisation of the QR updating algorithms. Runtimes of the implemented algorithms are compared to an existing full QR factorisation provided by CULA and sequential QR update algorithms implemented via CLAPACK subroutines. Analysis is included regarding the major trends in the results with changing size and location of the matrix update, and changing dimensions of the problem being updated. Accuracy in the GPU based updating algorithms is also investigated and commented upon.

Chapter 6 concludes the project and presents ideas for future work.

Chapter 2

Background Material

This chapter presents the background material that this project is based on. First, the least squares problem is introduced, from [12]. This is the main practical application of both QR factorisations and their updating algorithms. Second, the GPU programming model is presented, from [10] and [6]. This is the environment within which all of the algorithms presented in this project will run. Third, both Householder and Givens methods of QR factorisation are discussed, from [3]. QR factorisations share many of the core principles of QR updating and an overdetermined system must obviously be factorized using one of these QR factorisation methods before it may be updated. Finally, the theory behind the four QR updating algorithms themselves is introduced, from [4].

2.1 QR Factorisations and the Least Squares Problem.

This material is also introduced in [12]. In a least squares problem we wish to find a vector x that results in the minimum value of the function:

$$\min_x ||Ax - b||_2 \tag{2.1}$$

where $||\cdot||_2$ is the 2-norm, A is an $n \times m$ matrix and can be interpreted as holding the input coefficients of the variables x within each row. The jth row of A can be interpreted as an instance or a set of the variables that, within the system we are observing, results in the jth element of the right hand side, observation vector b. The objective of the least squares problem is to find the vector x that best fits the equation $Ax = b$ and x can subsequently be used as a linear model to approximate the system we are observing.

A model is normally much more accurate when as many observations as possible are made, so in practice the matrix A is massively overdetermined ($n > m$). In overdetermined systems an exact solution to $Ax = b$ cannot be found and we must therefore seek the closest solution or the solution to the least squares problem (2.1), which gives the minimum residual.

As a stable solution to (2.1) we can factorise the matrix $A = QR$ where Q is orthogonal:

$$Q^T Q = I$$

and R is upper trapezoidal, this is known as a QR factorisation.

With the factorized A we can then adjust the formula (2.1):

$$||Ax - b||_2 = ||Q^T Ax - Q^T b||_2^2$$

then as $A = QR \implies Q^T A = R$ and $d = Q^T b$

$$= ||Rx - d||_2^2$$

$$= \left|\left| \begin{bmatrix} R_1 \\ 0 \end{bmatrix} x - \begin{bmatrix} f \\ g \end{bmatrix} \right|\right|_2^2$$

$$= ||R_1 x - f||_2^2 + ||g||_2^2$$

Where R_1 is upper triangular and $d = \begin{bmatrix} f \\ g \end{bmatrix}$.

Finally $||R_1 x - b||_2^2 = 0$ can be solved exactly by back substitution, leaving $||g||_2^2$ as the minimum residual or error.

2.2 The GPU Programming Model

GPUs are massively parallel computational devices. They gain their performance advantage over CPUs in some algorithms by exploiting independence of operations. An additional requirement for the algorithm is that each independent branch uses the same or similar sequence of elementary operations e.g. add, multiply, load/store etc. This requirement is due to the Single Instruction Multiple Data (SIMD) or Single Instruction Multiple Threads (SIMT) architecture type of the Streaming Multiprocessors (SM) within the GPU.

The GPU programming model is unusual compared to other forms of software interface in that it is closely tied to the underlying hardware with very little abstraction. It is for this reason that I will not consider hardware and software principles of the GPU separately and will instead discuss their combined effect on programming style and performance.

I will consider a popular GPU programming API in this project, the CUDA programming API, which is specific to Nvidia GPUs. Although there are other standards in competition with this model, they are arguably closely related, and therefore share most, if not all, of the principles alluded to in this paper. Throughout this project I will write my programs in C++ and CUDA C.

My chosen testing platform for the algorithms that I will develop during this project is an Nvidia Tesla M2050 GPU, an implementation of the Fermi architecture. Again, as with the CUDA programming API, I will discuss principles of GPU programming using this specific model of GPU as an example. However, the main features I will discuss about this hardware are shared in general with other GPUs, even GPUs from other manufacturers such as AMD. A full description of the CUDA GPU programming model is given in [10].

2.2.1 The Programming Model

The main software interface is organised into kernel calls:

```
MyKernel<<<(dim3)grid dimensions,(dim3)block dimensions>>>(argument1, argument2, ... )
```

Kernels are C functions that are instantiated many times and are run on the GPU in parallel, typically over many SM. Kernel calls are an example of a fork-join design pattern.

Kernel execution is organised within a hierarchy consisting of grids, blocks and threads:

- **Threads** are the lowest level of execution, they each run an instance of the kernel sequentially.
- **Blocks** contain a 1, 2 or 3 dimensional array of threads. Blocks are further broken up into vectors of 32 threads known as warps, these are scheduled to execute on an SM one command at a time in a step-lock fashion so threads are completely synchronised within a warp. Separate warps however are executed according to the warp scheduler and can be executed in any order to hide latencies of memory accesses and pipelined commands, synchronisation and interaction of threads at this level must be managed explicitly by the programmer. A block resides and is executed within a single SM and in the Tesla M2050 GPU there is a limit of 1024 threads per block.
- **Grids** contain many thread blocks and are the highest organisational level within a kernel. There is only one grid per kernel call and blocks within a grid may be scheduled and executed on separate SM. Blocks are assumed to be completely independent of one another and there is no simple way of thread communication at this level. Every block within a grid has the same dimensions.

Each thread is supplied variables such as thread ids, block ids, etc. which are used by the programmer to map a thread to the data it must operate on. The Fermi architecture has 16 SM and each can have several blocks occupying it at any one time.

2.2.2 Thread Communication

By default, each of the many threads in a kernel execution operates independently of all of the other threads. Sometimes however, some co-ordination is required in order to avoid data race hazards within a block, this can be achieved via the `__syncthreads()` statement. Efficient data sharing among all threads within a block can be explicitly managed by the programmer via shared memory.

The `__syncthreads()` statement works by acting as a concurrency barrier which forces any threads that reach it within a block to wait until every thread in that block has reached the statement before continuing the block execution. There is obviously therefore some synchronisation overhead involved in calling `__syncthreads()` and it should only be used when necessary.

Shared memory on the other hand can be allocated statically or dynamically and can be used as low latency memory. Shared memory can be read from, or written to by any thread in a block and can be used to improve memory performance in situations where spatial locality of data is low or when data is reused by separate threads in a block. In a Fermi GPU, shared memory is configurable with L1 cache memory to be 48KB/16KB or 16KB/48KB respectively depending on the requirements of the algorithm.

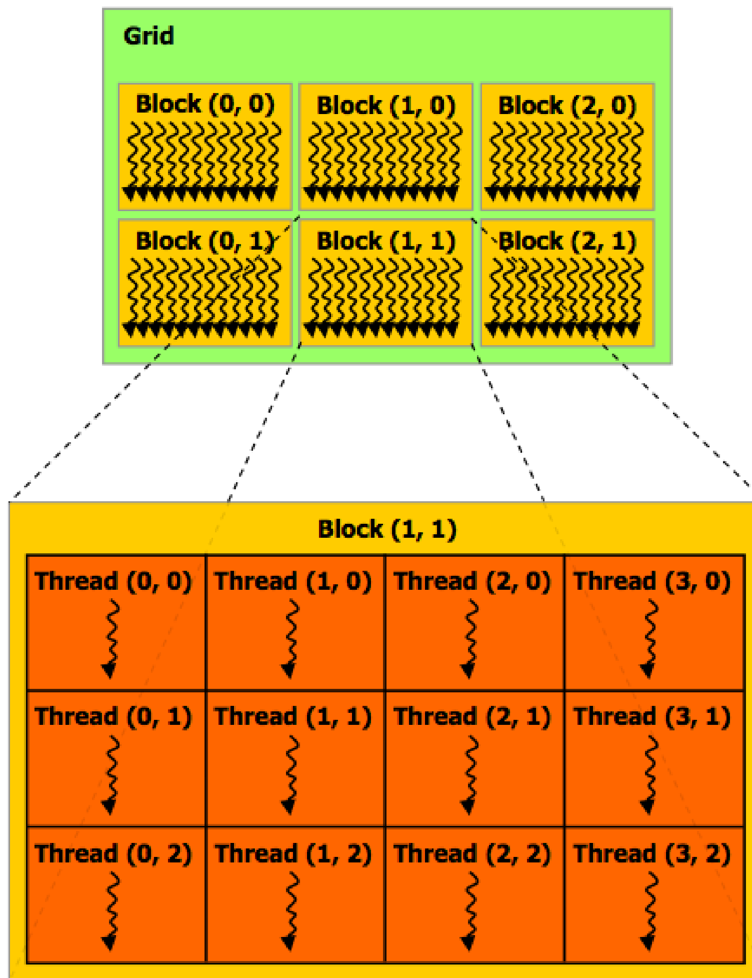

Figure 2.1: A diagram of the GPU software model from [10].

2.2.3 Memory Hierarchy

The increasing requirements placed on GPUs for general computing has meant that modern GPUs have a memory hierarchy to take advantage of spatial and temporal locality of data accesses. With the Tesla M2050 there is 3GB [7] of higher latency DRAM, a smaller L2 cache shared between all 16 SM, and 1 L1 cache per SM (low latency and configurable as previously mentioned). The registers have the lowest access latency of all memory, they also unfortunately have the lowest capacity. Overflow of this capacity can however lead to register spilling to local memory which incurs a performance penalty and should be avoided. The Fermi architecture provides approximately 32K registers per SM shared between a maximum thread occupancy per SM of 1536, which leads to a worst case scenario of approximately 20 registers per thread. Registers are therefore not a plentiful resource.

In general, the presence of a cache hierarchy automatically boosts efficiency of data accesses that exhibit temporal locality. For GPU programming however, due to the nondeterministic nature of thread scheduling, temporal locality is usually handled within a thread or warp execution via assignment to a register or between threads within a block via shared memory. The real benefit from the cache hierarchy with GPU programming comes from spatial locality where, in an architecture such as Fermi, cache lines are transferred between caches in contiguous blocks of 128 bytes. The implication of this is, for example, if memory accesses to 4-bit integers or floating point numbers are aligned in memory

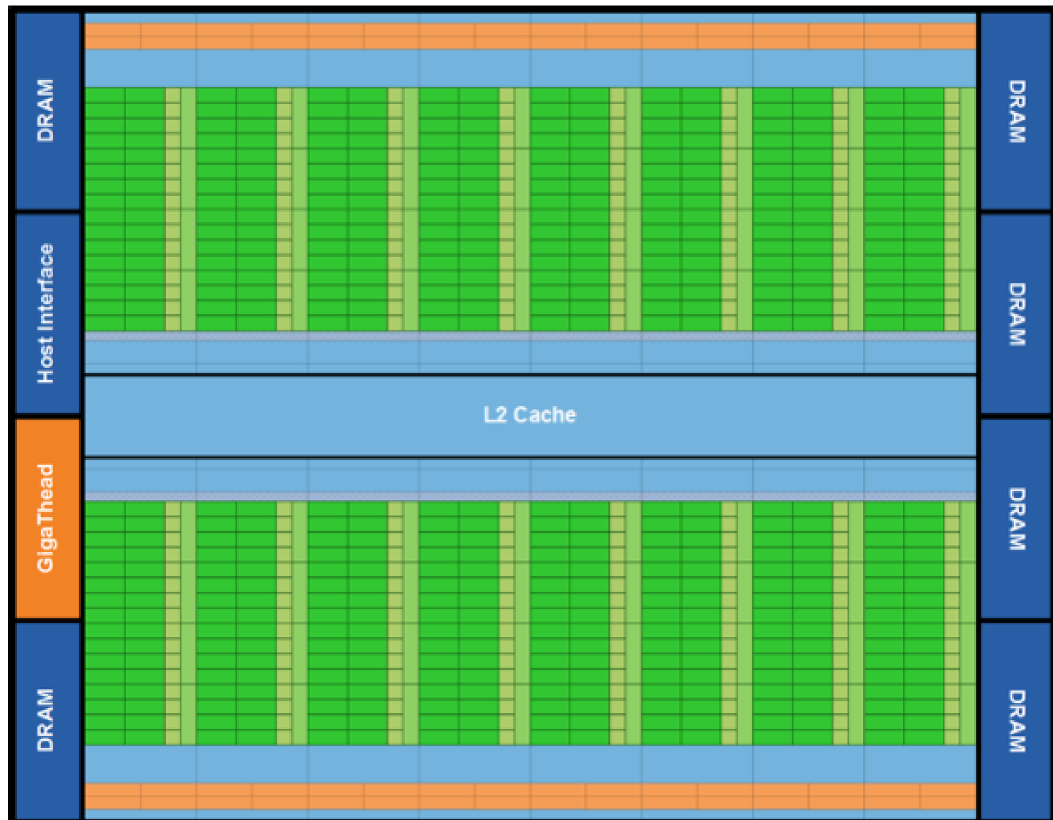

Figure 2.2: A diagram of the Fermi GPU architecture from [6].

and are contiguous for the 32 threads in a warp, all required data can be fetched with one memory access with no wastage. Correct use of coalesced memory alignment can lead to orders of magnitude speedup over similar algorithms with non-coalesced memory accesses.

By far the largest issue facing GPU programming as far as memory is concerned is the data transfer latency between the host and device and vice versa. The performance penalty of moving data in this way is often so large that a huge amount of computation within the algorithm and often also large problem sizes are required to gain any performance benefits from using the GPU. This is why in linear algebra on the GPU, we give preference to algorithms rich in matrix operations due to the quadratic or cubic scaling in computational complexity. I use this philosophy when choosing the algorithms to implement throughout this project.

2.2.4 Heterogeneous Programming

All interaction with the CUDA runtime takes place in the host code. The memory address spaces of the GPU and CPU are physically and programmatically considered to be separate and this forces the programmer to explicitly call special CUDA runtime functions to allocate and transfer data to and from GPU global memory. By default, memory transfers are blocking, which means host code waits for the transfer to complete. CUDA does give the option to execute asynchronous memory copies with the use of CUDA streams, but as one of my main goals in my algorithms was to minimise altogether

the number of memory transfers executed, I did not find this necessary.

Kernel calls on the other hand by default are completely asynchronous with respect to host code, and in most modern GPUs occupancy of the SM can even be increased by allowing the in-built block scheduler to execute many kernels in parallel using streams. To ensure that all work on the GPU is complete from within the host code, global device synchronisation statements are available (such as `cudaThreadSynchronize()` in CUDA).

2.2.5 BLAS subroutines

Within the algorithms that I will present in this paper there are a wealth of BLAS specification subroutines. Rather than write my own set of these routines I have utilised the CUBLAS library for BLAS routines based on the GPU, and CLAPACK (containing cBLAS, a C-based Fortran port) for the sequential implementation for comparative purposes. The main difference between the programming with these two libraries is the heterogeneous programming model discussed in the previous section. Data movement must be explicitly managed while programming with CUBLAS and so great care must be taken to minimise high latency communication between the device and the host.
All CUBLAS subroutines utilised in this project operate on single precision floating point numbers, as this offers maximum bandwidth on modern GPUs, therefore the 'S' prefix is used for all CUBLAS calls e.g. `cublasSgemm`, `cublasSgemv`, etc.

A full guide to the CUBLAS library is given in [8]. The main CUBLAS subroutines used in this project are:

- **cublasSgemm**: General matrix-matrix multiply.
- **cublasSgemv**: General matrix-vector multiply.
- **cublasSger**: Rank one update to a general matrix.`ger`$(c, x, y, A) : A := A + cxy^T$ where A is a general matrix, c is a scalar, and both x and y are vectors.
- **cublasSaxpy**: General vector addition. `axpy`$(c, x, y) : y := y + cx$ where c is a scalar, and both x and y are vectors.

2.3 QR Factorisations

Also introduced in [3], QR factorisations can be accomplished in many different ways, but Givens and Householder orthogonal transformations are most widely used. QR factorisation methods can be explained from the point of view of introducing zeros into the matrix being factorised, $A \in \mathbb{R}^{n \times m}$, which becomes the upper-trapezoidal matrix R. The orthogonal matrix $Q \in \mathbb{R}^{n \times n}$ can be formed during the factorisation to store the inverse transformation that has been applied to R such that $A = QR$. Transformations can alternatively be applied directly to the right hand side vector b as defined in (2.1), depending on the requirements of the specific application. Formation of the matrix Q is rarely a practical choice due to its large size, but as explained in Section 2.4, formation of Q is required for some updating algorithms.

2.3.1 Householder Reflections

A Householder reflection is a transformation applied via an orthogonal matrix of the form:

$$P = I - \frac{2}{v^T v} vv^T$$

where P can be chosen for a vector x such that:

$$Px = ||x||e_1$$

where e_1 is a vector of zeros with a 1 as the top element and $||\cdot||$ is the vector 2-norm. This means that if we choose x to be the column beginning at the diagonal of a matrix A we can choose P to zero the sub diagonal of that column. To apply a Householder transformation to zero a sub diagonal in the ith column of a matrix A during a QR factorisation, we embed a Householder matrix P into an $n \times n$ identity matrix, and multiply A from the left:

$$I_P A = A_P, \qquad I_P = \begin{bmatrix} I_{(i-1)} & \\ & P \end{bmatrix}$$

where A_P is the matrix A after the subdiagonal of the ith column has been made zero.

The standard QR factorisation utilising Householder reflections is rich in BLAS level 2 subroutines, this is not ideal for GPU programming as we wish to maximise the work done within a kernel call to fully utilise the excellent instruction bandwidth of the GPU. To solve this issue we can implement a 'blocked' QR factorisation which is rich in BLAS level 3 subroutines, at the expense of a little more computational complexity, which is easily amortised in the presence of larger workloads.

In a blocked QR factorisation, multiple Householder reflections can be compounded into a single transformation matrix:

$$P = P_1 P_2 ... R_r$$

A block of r columns are reduced as if they were a full QR factorisation, then the composite of the blocks Householder reflectors is applied to the remainder of the matrix. By [5], this can further be represented as:

$$P = I + WY^T$$

where W and Y are matrices with the number of rows equal to the length of the longest Householder vector in the block, and number of columns equal to the number of Householder reflectors composited within the block.

2.3.2 Givens Rotations

A Givens rotation is a matrix of the form:

$$G = \begin{bmatrix} c & s \\ -s & c \end{bmatrix}$$

this is applied to:

$$A = \begin{bmatrix} a_1 \\ a_2 \end{bmatrix} \text{ with } c = \frac{a_1}{\sqrt{a_1^2 + a_2^2}}, s = \frac{a_2}{\sqrt{a_1^2 + a_2^2}}$$

it results in:

$$GA = \begin{bmatrix} c & s \\ -s & c \end{bmatrix} \begin{bmatrix} a_1 \\ a_2 \end{bmatrix} = \begin{bmatrix} ca_1 + sa_2 \\ 0 \end{bmatrix}$$

In contrast to the more efficient Householder reflection where zeros are introduced in the entire subdiagonal column per transformation, Givens rotations introduce just one zero each. To apply a Givens transformation to the ith and $(i+1)^{th}$ elements within a column of a matrix A, we must embed a givens matrix within an identity matrix overwriting the ith to $(i+1)^{th}$ elements in the ith to $(i+1)^{th}$ columns, and multiply this matrix through A from the left:

$$I_G A = A_G, \qquad I_G = \begin{bmatrix} I_{(i-1)} & & & \\ & c & s & \\ & -s & c & \\ & & & I_{(n-i-1)} \end{bmatrix}$$

where A_G is the matrix A after a zero has been introduced.

2.4 QR Updating Algorithms

In this section an outline of the theory regarding the QR updating algorithms is presented, the material was initially and more exhaustively presented in [4]. For simplicity, in this project, we make the assumption that all matrices are non-singular, and also overdetermined ($n > m$), before and after they have been updated.

2.4.1 Adding Columns

Adding a block of columns may be necessary for a least squares problem when extra terms need to be added to the linear model. A block of p columns, U, is added to the matrix A from row k such that the new factorisation becomes:

$$\tilde{A} = \tilde{Q}\tilde{R}, \qquad \tilde{A} = \begin{bmatrix} A(1:n, 1:k-1) & U & A(1:n, k:m) \end{bmatrix} \tag{2.2}$$

Multiplying through Q^T from the QR factorisation of A, $A = QR$, shows that:

$$Q^T\tilde{A} = \begin{bmatrix} R(1:n, 1:k-1) & U_Q & R(1:n, k:m) \end{bmatrix}$$

where $U_Q = Q^T U$. Before any work is done, we can see that $R(1:n, 1:k-1) = \tilde{R}(1:n, 1:k-1)$, so we need only consider U_Q and $R(1:n, k:m)$.

At the most general level, we require an orthogonal matrix $O \in \mathbb{R}^{(n-k+1)\times(n-k+1)}$ such that:

$$\begin{bmatrix} I_{k-1} & 0 \\ 0 & O \end{bmatrix} Q^T\tilde{A} = \tilde{R} \tag{2.3}$$

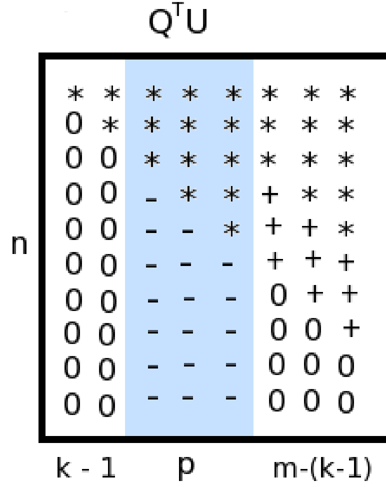

Figure 2.3: A diagram of the matrix R of dimension 10×5, the $p = 3$ columns inserted from column $k = 3$ are shaded in blue. The symbol '-' denotes a non-zero which must be made zero, whereas a '+' denotes a zero which must be made non-zero. '*' denotes a general non-zero element.

If O was made up entirely of Householder reflections, we would gain the desired

$$\begin{bmatrix} I_{k-1} & 0 \\ 0 & O \end{bmatrix} U_Q = \tilde{R}(1:n, k:k+p-1)$$

but

$$\begin{bmatrix} I_{k-1} & 0 \\ 0 & O \end{bmatrix} R(1:n, k:m)$$

would be full. Givens rotations are therefore required to satisfy (2.3). This restriction to Givens rotations can be worked around by use of Householder reflections to reduce the submatrix $U_G(m+1:n, 1:p)$ to an upper trapezoidal form, before the application of less efficient Givens rotations to finish the update. This use of Householder transformations is possible due to the fact that the trailing submatrix $R(m+1:n, k:m)$ is guaranteed to be entirely zero.

2.4.2 Removing Columns

A block of p rows is deleted from A from the column k such that:

$$\tilde{A} = \tilde{Q}\tilde{R}, \qquad \tilde{A} = \begin{bmatrix} A(1:n, 1:k-1) & A(1:n, k+p:m) \end{bmatrix} \tag{2.4}$$

Multiplying through by the orthogonal matrix Q^T we have:

$$Q^T \tilde{A} = \begin{bmatrix} R(1:n, 1:k-1) & R(1:n, k+p:m) \end{bmatrix}$$

Again, we can see that $R(1:n, 1:k-1) = \tilde{R}(1:n, 1:k-1)$. This fact allows us to define just $m-p-k+1$ Householder transforms $H_{k,k+1,\ldots,m-p}$ to reduce the right hand side portion of R.

$$H_{m-p}, \ldots, H_k R(1:n, k+p:m) = \tilde{R}(1:n, k+p:m)$$

so:

$$H_{m-p}, \ldots, H_k R = \tilde{R} \tag{2.5}$$

Thus we can also see that the matrix Q is not required to update the least squares problem.

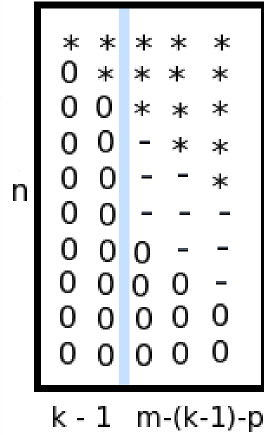

Figure 2.4: A diagram of the matrix R of dimension 10×8, the $p = 3$ columns removed from column $k = 3$ are represented by the blue line. The symbol '-' denotes a non-zero which must be made zero, whereas a '*' denotes a general non-zero element.

2.4.3 Adding Rows

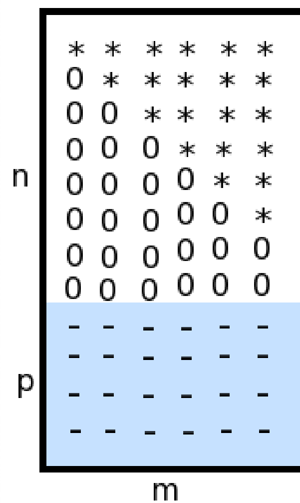

Figure 2.5: A diagram of the matrix R of dimension 8×6, the $p = 4$ rows added are shaded in blue. The symbol '-' denotes a non-zero which must be made zero, whereas a '*' denotes a general non-zero element.

A block of p rows, U, are added to the matrix A from row k such that the new QR factorisation becomes:

$$\tilde{A} = \tilde{Q}\tilde{R}, \qquad \tilde{A} = \begin{bmatrix} A(1:(k-1), 1:m) \\ U \\ A((k):n, 1:m) \end{bmatrix} \tag{2.6}$$

Wherever U is added within the matrix A, we can permute it to the bottom of the matrix such that:

$$P\tilde{A} = \begin{bmatrix} A \\ U \end{bmatrix}$$

where P in this case is a permutation matrix. Multiply both sides by $\begin{bmatrix} Q^T & 0 \\ 0 & I_p \end{bmatrix}$:

$$\begin{bmatrix} Q^T & 0 \\ 0 & I_p \end{bmatrix} P\tilde{A} = \begin{bmatrix} R \\ U \end{bmatrix}$$

With m Householder reflectors $H_{1,2,3,...,m}$ we can reduce:

$$H_m...H_2H_1 \begin{bmatrix} R \\ U \end{bmatrix} = \tilde{R}$$

Again, we can see that Q is not required to update the last squares problem, and as $A = QR$ we have:

$$\tilde{A} = \left(P^T \begin{bmatrix} Q & 0 \\ 0 & I_p \end{bmatrix} H_1...H_m \right) H_m...H_1 \begin{bmatrix} R \\ U \end{bmatrix} = \tilde{Q}\tilde{R}$$

2.4.4 Removing Rows

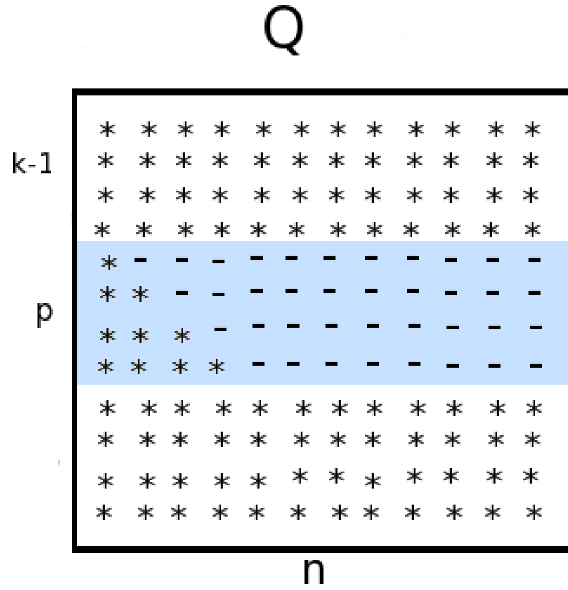

Figure 2.6: A diagram of the matrix Q of dimension 12×12, the $p = 4$ rows removed are shaded in blue. The symbol '-' denotes a non-zero which must be made zero, whereas a '*' denotes a general non-zero element. Transformations used to introduce zeros into this Q matrix are applied to the matrix R to produce $\tilde{R}$

A block of p rows are removed from the matrix A from row k such that the new QR factorisation becomes:

$$\tilde{A} = \tilde{Q}\tilde{R}, \qquad \tilde{A} = \begin{bmatrix} A(1:(k-1), 1:m) \\ A((k+p):n, 1:m) \end{bmatrix} \tag{2.7}$$

In order to show that $\tilde{Q}$ and $\tilde{R}$ in (2.7) can be calculated from just Q and R from the original

factorisation, we must first permute the deleted rows to the top of the matrix A.

$$PA = \begin{bmatrix} A(k:k+p-1, 1:m) \\ \tilde{A} \end{bmatrix}$$

where P in this case is a permutation matrix. A series of Givens matrices represented by the orthogonal matrix G are then employed to introduce zeros directly into a permuted Q, or PQ, to create $PQG = \begin{bmatrix} I & 0 \\ 0 & \tilde{Q} \end{bmatrix}$. These transformations are applied to the matrix R, which is not permuted, along with the fact that $A = QR$ to give the equation:

$$PA = \begin{bmatrix} A(k:k+p-1, 1:m) \\ \tilde{A} \end{bmatrix} = P(QG)(GR) = \begin{bmatrix} I & 0 \\ 0 & \tilde{Q} \end{bmatrix} \begin{bmatrix} S \\ \tilde{R} \end{bmatrix} \Longrightarrow \tilde{A} = \tilde{Q}\tilde{R} \qquad (2.8)$$

The consistency of the original least squares problem can then be preserved by applying the transformations to $d = Q^T b$ from the least squares formula. Note that the more efficient Householder transformations may not be used in this case due to the fact that $\tilde{R}$ would be full following a transformation via a full Householder matrix. Givens transformation matrices are Upper Hessenberg so do not suffer from the same drawbacks.

Chapter 3

Parallelising Householder and Givens

The implementation of the updating algorithms in this project feature heavily both Givens and Householder transformations. Efficiently parallelising these operations for a GPU implementation is therefore a high priority. In this chapter, parallel GPU implementations of both Givens and Householder transformations will be explored and details of the resulting algorithms will be given.

3.1 Parallelising Householder Reflections

QR factorisations via Householder reflections and its efficient implementation on the GPU is detailed in [5]. To create a Householder vector I implemented a CUDA kernel which takes as arguments separately a pointer to the diagonal element of the column we are required to zero, and a pointer to the top element of the sub diagonal column. I do this for efficiency reasons that will be apparent later, specifically when performing an update by adding rows, given in Section 4.3. The output of the kernel is then the raw Householder vector which is normalised such that the leading element is 1. A listing of the Householder kernel is:

```
1  __device__ float s;
2
3  __global__ void houseKernel(float *tau,float *diagVal, float *lcolumn
4                                              , float *v, int n) {
5
6          int tid = blockIdx.x*blockDim.x + threadIdx.x;
7
8          float dotprod = s;
9          if (dotprod != 0 && tid < n) {
10
11                 float vold = *diagVal;
12
13                 float t = sqrtf(dotprod + vold*vold);
14
15                 float vone;
16                 if (*diagVal <= 0) {
17                         vone = vold - t;
18                 }else {
19                         vone = vold + t;
```

```
20                      }
21
22                      float vsq = vone*vone;
23
24                      v[tid] = lcolumn[tid]/vone;
25                      if(tid == 0) {
26                              *tau = (-2*vsq)/((dotprod)+vsq);
27                      }
28              }
29  }
```

In practice a one dimensional grid was used with one dimensional blocks of length 128. This is a reasonable size to provide enough work per block as we are assigning 1 thread per element and the block size is small enough to minimise wastage when the input vectors are smaller. As Householder vectors are applied column by column, it is memory efficient to represent the matrices involved in column major order. By coincidence, CUBLAS also assumes matrices are represented in column major order. In Fermi, cache configuration can be set to provide a larger L1 cache at the expense of shared memory, this option is chosen for `houseKernel<<<>>>` as no shared memory is used.

- line 1: A device declared floating point number s, this is to avoid having to allocate memory every iteration. Before the kernel invocation the dot product of the sub diagonal column is calculated via a CUBLAS routine and placed in this variable.
- line 8: The global floating point number is loaded into a thread specific register for efficiency reasons.
- line 9: `tid < n` handles the edge case so there is no segmentation fault.
- line 13: There is a floating point square root to avoid costly double precision arithmetic and Fermi SM have Special Function Units that handle `sqrtf()` and similar functions in hardware.
- line 15-20: To avoid large errors later involving small `vone` in normalisation, a branch is used to calculate its value. As an entire warp is guaranteed to execute either one statement or the other, there is no expensive warp divergence.
- line 24: Every thread normalises its corresponding element.
- line 25-27: calculation of the single Householder coefficient is done by thread 0 in block 0.

Application of the Householder matrices to the trailing submatrix of the matrix undergoing reduction can be broken down into elementary BLAS subroutines. This means that the highly efficient CUBLAS library may be employed to handle these subroutines. Note that any expensive data transfer during the QR factorisation has been rendered unnecessary due to the fact that all the calculations of a sequential nature were handled within my Householder kernel. The fact that there is no communication of data between the device and host greatly simplifies the use of CUBLAS.

Pseudocode for the blocked QR factorisation is given:

```
1  %A, b, and Q are assumed to be present in GPU memory.
2  blockedQRFactorisation(A,Q,d,blockWidth){
3
4     %allocation of maximum size outside the loop to avoid
5     %redundant reallocation.
6     allocate tau(1:blockWidth);
7     allocate Y(1:n,1:blockWidth);
8     allocate W(1:n,1:blockWidth);
9     %reset V to zeros
10    transfer zeros to Y;
11
12    %for every block.
13    for i = 1:blockWidth:m
14       %reset W to zeros
15       transfer zeros to W;
16       jb = min(blockWidth,m-i);
17
18       endblock = (i+jb-1);
19       for j = i:endblock
20             %column number within the block.
21             index = j-i+1;
22             get address of global s;
23             s = dot(A(j+1:n,j));
24             [tau(index) Y(index:n-j,index)] = housekernel<<<>>>(A(j:n,j));
25
26             %apply the Householder reflector to the current block.
27             %note due the addition due to -1 multiplied to tau within
28             %the kernel.
29             v = Y(index:n-j,index);
30             vA = (v'*A(j:n,j:endblock));
31             A(j:n,j:endblock) =  A(j:n,j:endblock) + tau(index)*v*vA;
32
33             %apply the Householder reflector to W.
34             if  index == 1
35                W(1:n-i,1) = tau(index)*v;
36             else
37                WYv = W(1:n-i,1:index-1)*(Y(1:n-i,1:index-1)'*v);
38                W(1:n-i,index) = tau(index)*v + tau(index)*WYv;
39             end
40
41       end
42
43       %update d
44       d(i:n) = d(i:n) + Y*W'*d(i:n);
45       %update trailing matrix A.
46       if i + jb < m
47             A(i:n,i+jb:m) = A(i:n,i+jb:m) + Y*W'*A(i:n,i+jb:m);
48       end
49       %update Q.
50       Q(1:n,i:n) = Q(1:n,i:n) + Q(1:n,i:n)*W*Y';
51    end
52 }
```

Operations such as that on lines 37-38 or lines 43-50 are broken up into two CUBLAS matrix-matrix or matrix-vector products, making use of an allocated matrix or vector to store the intermediate result. For example:

$$Q := Q + QWY^T \implies Q_{temp} := QW \text{ then } Q := Q + Q_{temp}Y^T$$

is fewer flops than the other multiply order, as the matrices W and Y typically have much fewer columns than rows. The whole operation can be accomplished with just two CUBLAS 'gemm' routines.

3.2 Parallelising Givens Rotations

In order to introduce a zero correctly in the lower of two consecutive column elements in a matrix using Givens rotations there are two main considerations; First, within the rows to the left of the subject elements there be no other non-zero elements, and second, there must be no non-zero elements in the column below the element to be made zero. These restrictions make Givens QR factorisations difficult to parallelise. Algorithms have however been developed for parallelising Givens from the point of view of distributed systems or even multicore CPUs such as in [2].

[2] suggests that each functional unit can be assigned a strip of rows and zeros are introduced:

- First in the leftmost column of the lowest strip.
- Once the leftmost column of the lowest strip has been made zero, the owner of the second lowest strip is notified and it too introduces zeros in its leftmost column. The owner of the lowest strip introduces zeros in its second-to-leftmost column at the same time.
- The algorithm continues in this way until the matrix is upper trapezoidal.

This approach can be adapted to function effectively on the GPU. In contrast to multicore CPUs and distributed systems, GPUs require no data communication between threads due to the unified global memory space. The GPU prefers parallelism of a much finer grain (less work per thread, more threads) compared to CPU based parallelism. In order to achieve a maximum occupancy, or the maximum number of the SM on a GPU active and doing useful work at any one time, a strip width of just two rows maximises the number of zeros being introduced in each parallel step.

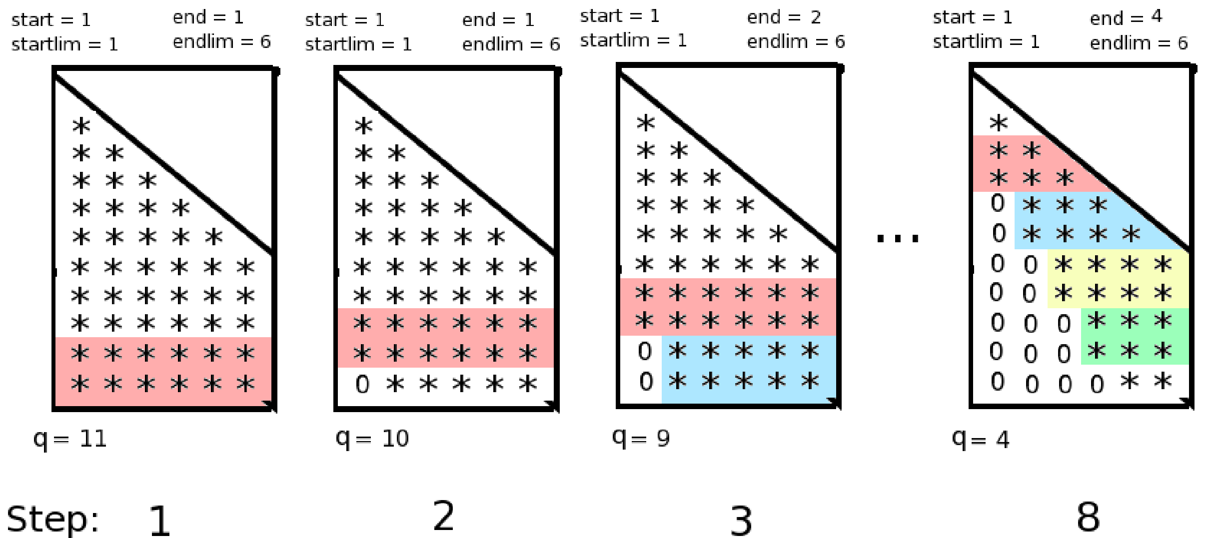

Figure 3.1: A diagram of the application of Givens matrices on a non-zero block on the GPU. As before, an asterisk denotes a non-zero element.

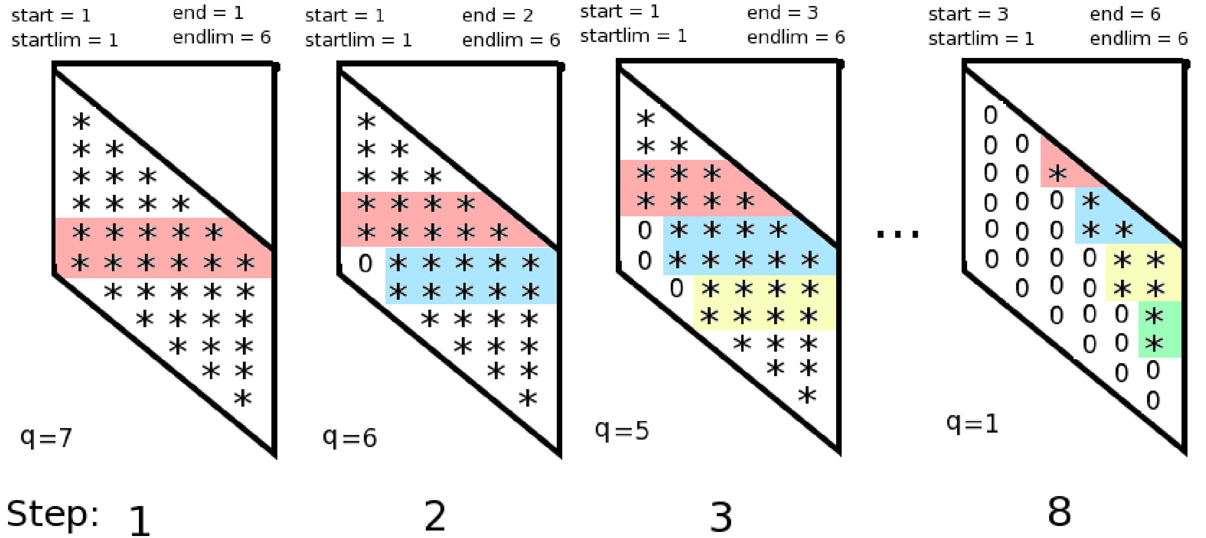

Figure 3.2: A diagram of the application of Givens matrices on a non-zero strip on the GPU. Algorithmically, the only difference between this algorithm and the one depicted in Figure 3.1 is the increment of the variable 'end' once per iteration instead of twice, and the 'q' variable is initialised at (strip height+1) as opposed to n (matrix height).

Figure 3.1 and Figure 3.2 show the GPU Givens approach applied to a matrix. The numerical labels along the bottom denote the kernel or iteration number. The coloured blocks, on the other hand, denote different block assignments within a kernel invocation; red denotes block id y=0 within a 2 dimensional grid, blue denotes y = 1, yellow denotes y = 2, and finally green denotes y = 3. This is extended to arbitrarily large grid dimension y in the obvious way. In addition to distribution of the matrix amongst blocks in the y direction, in this projects' implementation, the x direction is partitioned into blocks of 128 elements.

We can see that it would be beneficial to memory efficiency to represent the matrix in row major order, by the nature of the distribution of matrix elements among threads. As all other methods in this project benefit from matrices in column major order, there is need for an efficient transpose kernel. The transpose kernel implemented in this project is based on the one from [13], and kernel code is given in Figure 3.4. As discussed regarding the Fermi architecture, memory accesses that are coalesced and aligned into 128 byte segments (32 floating point numbers) per warp can be retrieved in one fetch operation. Row dominant matrices provide the coalesced memory condition so memory alignment can be ensured by padding the leading dimension of a matrix to be a multiple of 128 bytes. This matrix padding can be allocated automatically by explicit use of `cudaMallocPitch()`, as opposed to `cudaMalloc()`.

Due to the fact that the matrix is partitioned along its width as well as its height, dependencies are created between thread blocks with regard to the calculation of the Givens coefficients s and c. Calculation of these coefficients for an entire step of Givens rotations must therefore take place before each step, in its own kernel. A diagram of how calculation of Givens coefficients is distributed along the x direction of the 1-dimensional blocks, in the 1-dimensional grid, of the `makeGivens<<<>>>` kernel is given in Figure 3.3. 32 threads are allocated per block, this small number was chosen to give as much power over dynamic parallelism as possible to the block scheduler, as two separate cache lines

will be required per thread. With Fermi, for both `makeGivens<<<>>>` and `applyGivens<<<>>>` the cache configuration is chosen to prefer the L1 cache, as little or no shared memory is used.

```
1  //row major access macro.
2  #define GETR(A,i,j,pitch) A[(i)*(pitch)+(j)]
3  __global__ void makeGivens(int startx, int endx, int starty, float *A
4                         , float *s, float *c, int ldA, int startlim) {
5
6          int i = blockIdx.x*blockDim.x + threadIdx.x;
7          int tozero = starty + 2*i + (startx - startlim);
8          if( i < endx-startx) {
9                  int xindex = i + startx;
10
11                 float a1 = GETR(A,tozero-1,xindex,ldA);
12                 float a2 = GETR(A,tozero,xindex,ldA);
13
14                 float norm = sqrtf(a1*a1 + a2*a2);
15
16                 c[xindex - startx] = a1/norm;
17                 s[xindex - startx] = a2/norm;
18         }
19 }
```

- **line 6-7:** Maps the thread and block index to the row index of the element to be made zero.
- **line 8:** Checks for the edge case, the difference between the arguments `endx-startx` defines how many coefficients are required for this step (see Figure 3.2 and Figure 3.1).
- **line 9:** Maps the thread and block index to the column index of the element to be made zero.
- **line 11-12:** load the relevant elements into registers.
- **line 14:** Explicit multiply as opposed to general power function to represent a square. Floating point square root avoids the possibility of the invocation of expensive double precision arithmetic.
- **line 16-17:** Final calculation and storage of the Givens coefficients.

```
1  //Row major access macro.
2  #define GETR(A,i,j,pitch) A[(i)*(pitch)+(j)]
3  __global__ void applyGivens(int startx, int endx, int starty, float *A
4                            , float *s, float *c, int ldA, int startlim) {
5          __shared__ float cs;
6          __shared__ float ss;
7
8          if(threadIdx.x == 0) {
9                  cs = c[blockIdx.y];
10                 ss = s[blockIdx.y];
11         }
12
13         __syncthreads();
14
15         int i = blockIdx.y;
16         int tozero = starty + 2*i + (startx - startlim);
17         int xindex = blockIdx.x*blockDim.x + threadIdx.x;
18         if(i < endx-startx && xindex < ldA) {
19                 float a1 = GETR(A,tozero-1,xindex,ldA);
20                 float a2 = GETR(A,tozero,xindex,ldA);
21
22                 float tempc = cs;
23                 float temps = ss;
```

```
24                      float tempa = a1;
25                      a1 = tempc*tempa + temps*a2;
26                      a2 = -temps*tempa + tempc*a2;
27
28                      GETR(A,tozero-1,xindex,ldA) = a1;
29                      GETR(A,tozero,xindex,ldA) = a2;
30              }
31  }
```

- **lines 5-6:** Declaration of the Givens coefficients as shared variables, they are common to the entire block so the broadcast mechanic of shared memory is invoked. Shared memory broadcasts are triggered when all threads in a warp read from the same shared memory address, this increases efficiency by providing all threads with a piece of data in one operation.
- **lines 8-13:** Givens coefficients are loaded in once per block, this makes up for the overhead incurred by `__syncthreads()`.
- **line 16:** Maps the thread and block index to the row index of the element to be made zero.
- **line 17:** Maps the thread and block index to the column to be multiplied by the Givens rotation for this specific thread.
- **line 18:** Checks for the edge case, the difference between the arguments `endx-startx` defines how many coefficients are required for this step (see Figure 3.2 and Figure 3.1). The column index must also not exceed the leading dimension, this avoids out-of-bounds errors and segmentation faults.
- **lines 19-20:** load the elements to be operated on into registers.
- **lines 22-23:** Givens coefficients are loaded into thread specific registers from shared memory.
- **lines 24-26:** Apply the Givens rotation.
- **lines 28-29:** Write the new elements back to main memory.

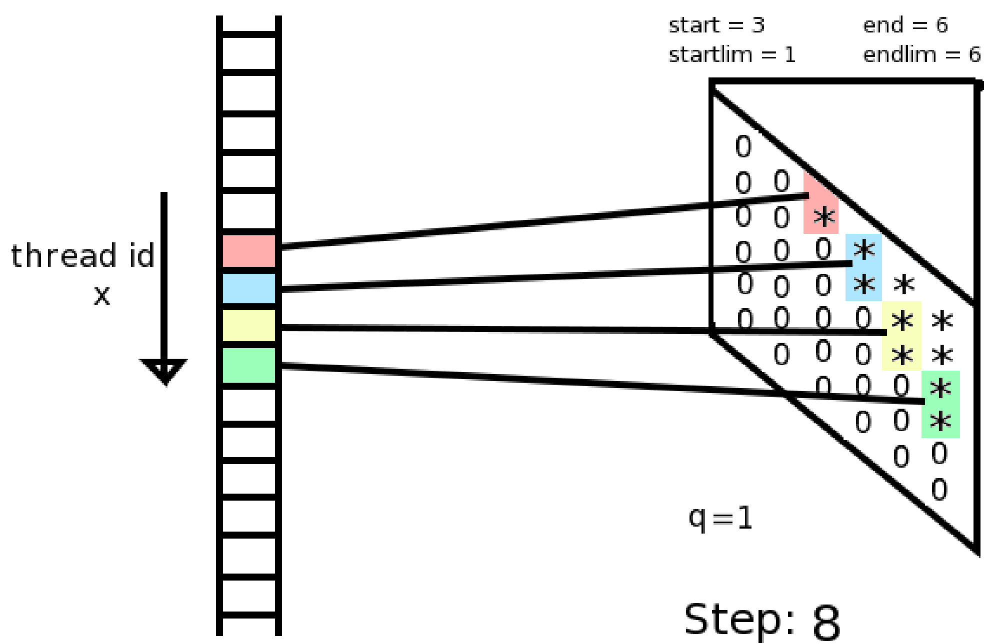

Figure 3.3: A diagram depicting the distribution of the calculation of Givens coefficients among threads in the `makeGivens<<<>>>` kernel. The example shown is the calculation of Givens coefficients prior to their application within the strip in step 8, Figure 3.2.

```
1  #define TILE_DIM 32
2  #define BLOCK_ROWS 4
3  __global__ void transpose(float *idata, int ldi, float *odata
4                                    , int ldo, int width, int height)
5  {
6          __shared__ float tile[TILE_DIM][TILE_DIM];
7
8          //calculate the index of the input matrix.
9          int xIndex = blockIdx.y*TILE_DIM + threadIdx.y;
10         int yIndex = blockIdx.x*TILE_DIM + threadIdx.x;
11         //first is a transpose operation to shared memory.
12         //x thread maps to the y axis so a block
13         //directly maps to the first 4 columns.
14         int index_in = (xIndex)*ldi + yIndex;
15
16         //the stepping must not overflow
17         int inlim = min(TILE_DIM,width-(blockIdx.y*TILE_DIM));
18         int outlim = min(TILE_DIM,height-(blockIdx.x*TILE_DIM));
19
20         //32*4 threads per block and 32*32 values.
21         //conditional code is for edge cases.
22         for (int i=0; i<TILE_DIM; i+=BLOCK_ROWS) {
23            if(threadIdx.x < outlim && threadIdx.y+i < inlim) {
24               tile[threadIdx.x][threadIdx.y+i] = idata[index_in+i*ldi];
25            }
26         }
27         __syncthreads();
28
29         //calculate the index of the output matrix.
30         xIndex = blockIdx.x * TILE_DIM + threadIdx.y;
31         yIndex = blockIdx.y * TILE_DIM + threadIdx.x;
32         int index_out = (xIndex)*ldo + yIndex;
33
34         for (int i=0; i<TILE_DIM; i+=BLOCK_ROWS) {
35            if(threadIdx.x < inlim && threadIdx.y+i < outlim) {
36               odata[index_out+i*ldo] = tile[threadIdx.y+i][threadIdx.x];
37            }
38         }
39 }
```

Figure 3.4: An efficient transpose kernel, based on [13]. Fermi cache configuration is chosen to prefer shared memory, this is due to the fact that multiple blocks scheduled per SM will fill up shared memory fairly quickly ($32 \times 32 \times$ `sizeof(float)` $=$ 4KB per block, 16KB or 48KB shared memory per SM).

Chapter 4

QR Updating Algorithms

I will now consider how each of the 4 standard updates to the QR factorisation, adding a block of columns, removing a block of columns, adding a block of rows, and removing a block of rows, may be implemented on the GPU. My approach will be systematic, introducing first the sequential algorithm, then consideration will be given to ways the algorithm may be effectively parallelised and finally, the GPU based algorithm I implemented will be presented in pseudocode.

4.1 Adding a Block of Columns

The act of adding a block of columns can be defined as considering an initial matrix A $(n \times m)$ where the QR factorisation is $A = QR$, and a block of columns U $(n \times p)$. Providing an index $k, 0 \leq k \leq m+1$, we can define the updated QR factorisation as:

$$[A(1:n, 1:k-1) \quad U \quad A(1:n, k:m)] = \tilde{Q}\tilde{R}$$

which we will calculate given just the original Q, R and the new columns U. A visualisation of an example problem where $m = 5$, $n = 10$, $p = 3$ and $k = 3$ is given in Figure 4.1.

The diagram shows multiple stages in the reduction process, described in detail in [4] and Section 2.4. Step 1 involves the Householder reduction of the lower part of $Q^T U$ and step 2 shows the completion of the update via Givens rotations.

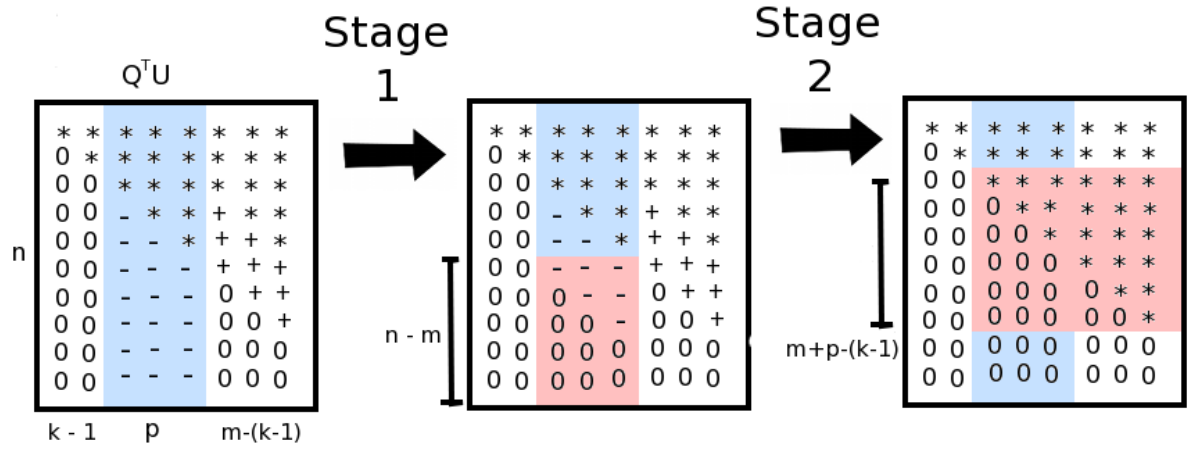

Figure 4.1: A diagram of the update of the matrix R, the '+' symbol shows the non-zeros to be introduced and the '-' symbol shows an existing non-zero that must be made zero. '*' shows a generic non-zero element. The shaded blue shows the added columns, whereas the shaded red shows the 'active' section for that stage in the algorithm, this will be discussed later.

The QR update via adding columns in pseudocode is detailed below. Inputs to the update function are the original Q and R from the factorisation $A = QR$, the block of columns U being added, and the right hand side vector from the least squares problem $d = Q^T b$.

```
1  QRUpdateAddColumns(R,Q,d,U,k,p) {
2          allocate Q(1:n,1:n);
3          allocate d(1:n);
4          allocate U(1:n,1:p);
5          allocate R2(1:n,1:m+p);
6
7          transfer HOST:Q to GPU:Q;
8          transfer HOST:d to GPU:d;
9          transfer HOST:U to GPU:U;
10         transfer HOST:R(1:n,1:k-1) to GPU:R2(1:n,1:k-1);
11         transfer HOST:R(1:n,k:m) to GPU:R2(1:n,k+p:m+p);
12
13 %PHASE 0: initialise U.
14
15         R2(1:n,k:k+p-1) = Q' * U;
16
17 %PHASE 1: Householder reflections.
18
19         blockedQRFactorisation(R2(m+1:n,k:k+p-1),Q,d);
20
21         %if the columns were added on the right hand side the
22         %algorithm is complete.
23         if k == m
24                 return;
25         end
26
27
28 %PHASE 2: Givens rotations.
29         %transpose a section into row major form.
30         allocate givensA(1:m+p-(k-1),1:m+p-(k-1));
31         transpose<<<>>> R2(k:m+p,k:m+p) to givensA;
32
33         startlim = 1;
```

```
34          endlim = pnumadded;
35          start = startlim;
36          ending = start+1;
37          stripwidth = m-p-(k-1);
38          q = stripwidth;
39
40          %allocate an arbitrarily large buffer for c and s
41          allocate s(1:stripwidth);
42          allocate c(1:stripwidth);
43
44          while ending > start
45                  sync;
46
47                  [s c] = makeGivens<<<>>>(givensA);
48
49                  sync;
50
51                  stream 0: applyGivens<<<>>>(givensA,s,c);
52                  stream 1: applyGivens<<<>>>(Q(1:n,k:n),s,c);
53                  stream 2: applyGivens<<<>>>(d(k:n),s,c);
54
55                  %if the first givens rotation is mapped to zero within
56                  %the triangle that needs to remain nonzero within the
57                  %matrix, we increment the starting barrier.
58                  %alternatively and initially, we decrement n.
59                  if (n+(start-startlim) <= start)
60                          start++;
61                  else
62                          q--;
63                  end
64                  %every iteration, as we are making zeros only within a
65                  %diagonal, non-zero strip, we increment the end barrier
66                  %until we reach the end of the matrix.
67                  if (ending < endlim)
68                          ending++;
69                  end
70
71          end
72
73          transpose<<<>>> givensA to R2(k:m+p,k:m+p);
74
75          sync;
76  }
```

- **lines 2-11:** Memory management on the GPU. Space is allocated in `R2` to accommodate the new columns `U` between the $(k-1)^{th}$ and the kth columns of R.
- **line 15:** Apply Q to the added columns U. Save the result in the allocated space within `R2`.
- **line 19:** QR factorise the lower block of $Q^T U$ as described in Section 3.1 and step 1 in Figure 4.1, the section shaded in red.
- **lines 23-25:** If the columns are added to the right of R, then the update is complete and the function exits.
- **lines 30-31:** Transposes the section shown in red in step 2 of Figure 4.1 to the newly allocated `givensA`, such that it is in row major order. `givensA` is buffered by `cudaMallocPitch()`.
- **lines 44-71:** Givens rotations are applied to a strip of non-zeros in `givensA` as described in Figure 3.2. The application of the Givens coefficients is dependent on their calculation, so the

calculation of Givens coefficients is surrounded by synchronisation statements. The application of Givens rotations to `givensA`, Q and d however are completely independent, so a separate CUDA stream can be called upon for each. Streams are then scheduled and executed simultaneously on the GPU. Givens rotations are applied to Q^T so transposing Q is not necessary. Also the leading dimension of d is trivially 1 as it is a vector, so there is no need to transpose d either.

- **line 73:** Transposes `givensA` back to the section shown in red in step 2 of Figure 4.1.

4.2 Removing a Block of Columns

The act of removing a block of columns can be defined as considering an initial matrix A $(n \times m)$ where the QR factorisation is $A = QR$. Providing an index $k, 0 \leq k \leq m - p + 1$, and a block width p, we can define the updated QR factorisation as:

$$[A(1:n, 1:k-1) \quad A(1:n, k+p:m)] = \tilde{Q}\tilde{R}$$

Which we will calculate given just the original R. Unlike adding rows in the previous section, Q is not required for the update. A visualisation of an example problem where $m = 8$, $n = 10$, $p = 3$ and $k = 3$ is given in Figure 4.2.

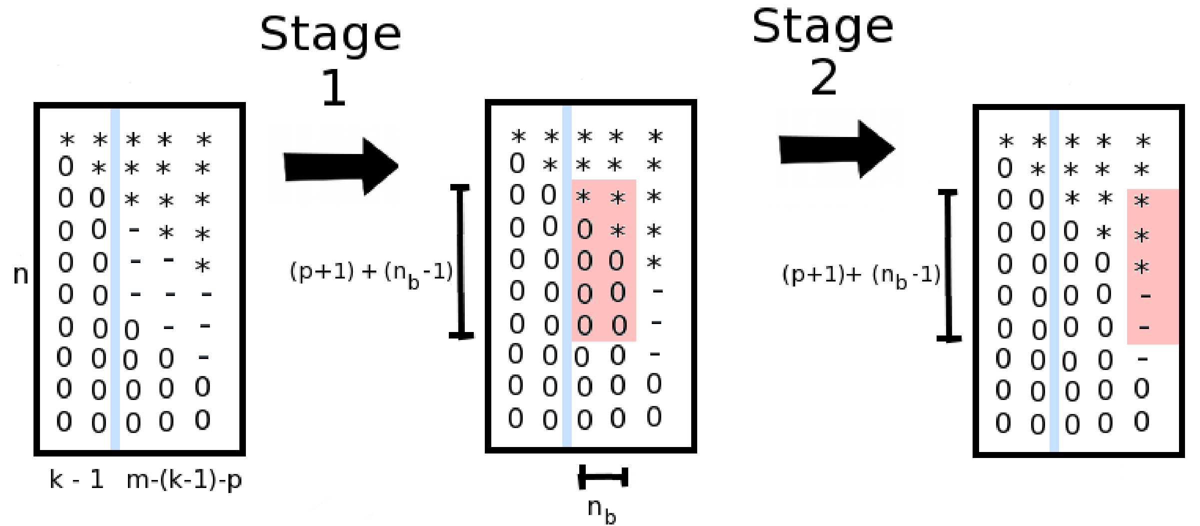

Figure 4.2: A diagram of the reduction of one block within the blocked update of the matrix R, the '-' symbol shows an existing non-zero that must be made zero and '*' shows a generic non-zero element. 'n_b' = 2 in this example and represents the block size. The shaded blue line shows the point that the removed columns used to occupy, whereas the shaded red shows the 'active' section for that stage in the algorithm, this will be discussed later.

The QR update via removing columns in pseudocode is detailed below. Inputs to the update function are the original R from the factorisation $A = QR$, and the right hand side vector from the least squares problem $d = Q^T b$.

```
1  QRUpdateRemoveColumns(R,d,k,p) {
2          allocate d(1:n);
3          allocate R2(1:n,1:m-p);
4
5          transfer HOST:d to GPU:d;
6          transfer HOST:R(1:n,1:k-1) to GPU:R2(1:n,1:k-1);
7          transfer HOST:R(1:n,k+p:m) to GPU:R2(1:n,k:m-p);
8
9          if kstart == m-p+1
10                 return;
11         end
12
13         blockedQRFactorisation(R2(k:n,k:m-p),d);
14
15         sync;
16 }
```

- **lines 2-7:** Memory management on the GPU. Space is allocated in `R2` for the remaining $m - p$ columns. This allows columns $R(1 : n, k : k + p - 1)$ to be deleted by the act of transferring the data to the GPU.
- **lines 9-11:** If the right hand side p columns were deleted, the update is complete, and the function returns.
- **line 13:** Blocked QR factorisation much like the one described in Section 3.1, applied to the submatrix to the right of the removed rows as pictured in Figure 4.2. The adjustments to the blocked QR algorithm that was presented in Section 3.1 are for efficiency reasons, avoiding unnecessary addition and multiplication by zero. First, Householder vectors can be limited to being just $p + 1$ elements long. Second, W and Y matrices need only be of dimension $(p + 1) + (n_b - 1) \times n_b$ where n_b is the block size in columns. These block dimensions are again pictured in Figure 4.2.

4.3 Adding a Block of Rows

The act of adding a block of rows can be defined as considering an initial matrix A $(n \times m)$ where the QR factorisation is $A = QR$, and a block of rows U $(p \times m)$. Providing an index $k, 0 \leq k \leq n + 1$, we can define the updated QR factorisation as:

$$\begin{bmatrix} A(1 : k - 1, 1 : m) \\ U \\ A(k : n, 1 : m) \end{bmatrix} = \tilde{Q}\tilde{R}$$

Which we will calculate given just the original R, by Section 2.4, and the new rows U. A visualisation of the reduction of a single block within a blocked update where $m = 6$, $n = 8$, $p = 4$ is given in Figure 4.3. Note that the value of k does not affect the algorithm method as the added rows can be trivially permuted to the bottom of the matrix A before the update procedure begins. As adding rows also adds elements to the length of the right hand side vector b from the least squares problem, we denote these added elements as e.

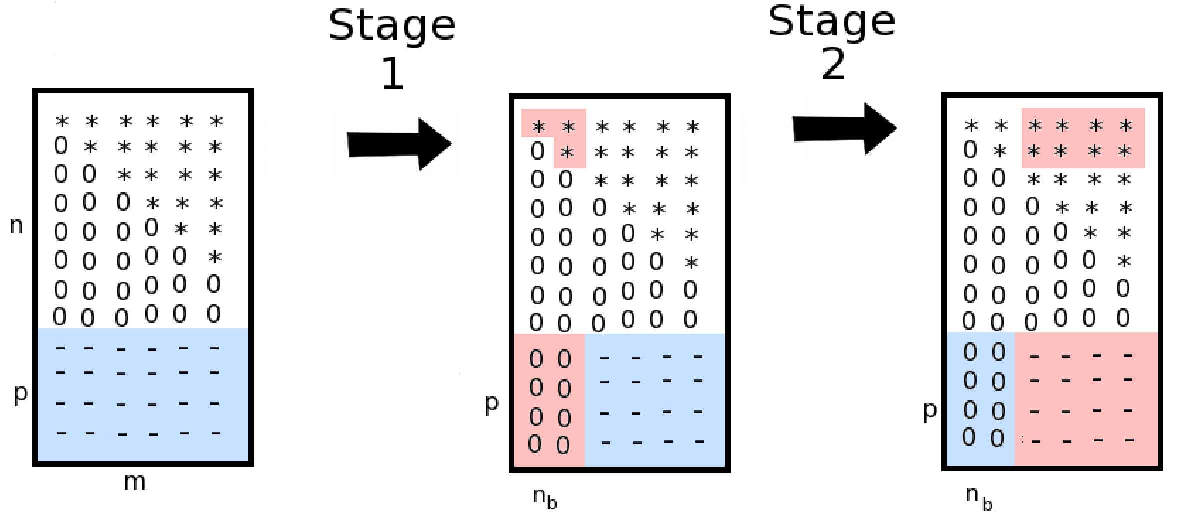

Figure 4.3: A diagram of the reduction of one block within the update of the matrix R, the '-' symbol shows the non-zeros that must be made zero during the update procedure. '*' shows a generic non-zero element. The shaded blue shows the added rows. The elements shaded in red in the central matrix are the elements involved in reduction of a block via Householder reflections, note that the intermediate zeros are excluded from the calculation. Block size in this example is $n_b = 2$. The elements in the red shaded area in the matrix to the right are multiplied by the matrices produced in the previous step.

As can be seen in Figure 4.3, a transformation can be applied to a small section of the non-zero triangular part of R separately to U to avoid wasteful arithmetic involving zero. Use of the W and Y method method of Householder blocking, introduced in Section 3.1 from [5], would involve both the full W and Y matrices being applied to both R and U.

An alternative yet similar approach to WY Householder blocking is proposed and explained fully in [4] involving an upper triangular, square matrix T and a matrix V containing Householder vectors. Given a set of Householder matrices, the product can be linked to V and T matrices by:

$$H_{n_b}...H_2H_1 = I - VT^TV^T$$

where n_b represents block size. V is made up of Householder matrices within its columns, $V = [v_1 \quad v_2 \quad v_3 \quad ... \quad v_{n_b}]$. By the nature of the problem pictured in Figure 4.3:

$$V = \begin{bmatrix} I_{n_b} \\ 0 \\ v_{n+1:n+p} \end{bmatrix}$$

As the top of V is the identity, and the middle is zero, only the lowest p rows must be stored increasing efficiency. The matrix T on the other hand is defined recursively as:

$$T_1 = \tau_1, \qquad T_i = \begin{bmatrix} T_{i-1} & -\tau_i T_{i-1} V(1:p, 1:i-1)^T v_i \\ 0 & \tau_i \end{bmatrix}, \qquad i = 2:n_b$$

where τ_i represents the Householder coefficient corresponding to the ith Householder vector in the

i^{th} column of V. Assignment of individual τ_i elements within T in a loop is inefficient on the GPU. Fortunately, as their assignment along the diagonal of T is independent of the remainder of the formula for T, the entire diagonal can be assigned in parallel beforehand within a simple kernel.

```
1  //Column major access macro.
2  #define GET(A,i,j,n) A[(j)*(n)+(i)]
3  __global__ void setTau(float *tau, float *T, int limit, int ldT) {
4          int i = blockIdx.x*blockDim.x + threadIdx.x;
5          if(i < limit) {
6             GET(T,i,i,ldT) = tau[i];
7          }
8  }
```

Pseudocode for the construction of T is presented as the function `buildT`. It takes as parameters an allocated square matrix T of dimension $n_b \times n_b$, and a vector of Householder coefficients `tau`.

```
1  buildT(tau,T) {
2          setTau<<<>>>(tau,T);
3
4          for k = 2:n_b
5             T(1:k-1,k) = -tau(k)*T(1:k-1,1:k-1)*V(1:p,1:k-1)'*V(1:p,k);
6          end
7  }
```

Note that as all `tau` Householder coefficients are created via `houseKernel<<<>>>` presented in Section 3.1, the unary minus operator has been pre-applied to allow passing to CUBLAS calls by reference with no unnecessary repetition of calculation. Line 5 can be executed via 2 'gemv' CUBLAS calls.

Another kernel used within the following `QRUpdateAddRows()` pseudocode is the `doubleSum<<<>>>` kernel. Its purpose is to perform an elementwise sum of the entire matrices `A` and `B` of dimension `Ctile2` $\times$ `Ctile3` to a tile within the matrix `C` or:

$$C(\texttt{Ctile0} : (\texttt{Ctile0} + \texttt{Ctile2}), \texttt{Ctile1} : (\texttt{Ctile1} + \texttt{Ctile3})) \mathrel{+}= A + B$$

The tile within the matrix `C` is also of dimension `Ctile2` $\times$ `Ctile3` but its top left corner is at element (`Ctile0`, `Ctile1`) within `C`. The matrix tiling and allocation of thread blocks is the same as that described in the efficient matrix transpose kernel shown in Figure 3.4 and given in [13].

```
1  //Column major access macro.
2  #define GET(A,i,j,n) A[(j)*(n)+(i)]
3  __global__ void doubleSum(float *A,float *B,float *C,int Ctile0
4                      ,int Ctile1,int Ctile2,int Ctile3,int Csize0,int step) {
5          int x = blockIdx.y*WARP_DIM +threadIdx.y;
6          int y = blockIdx.x*WARP_DIM +threadIdx.x;
7
8          int xbase = Ctile1;
9          int ybase = Ctile0;
10         int ldC = Csize0;
11         int ceily = Ctile2;
12         float *baseptr = &(GET(C,ybase,xbase,ldR));
13         int xlim = min((blockIdx.y+1)*WARP_DIM,Rtile3);
14         if(y < ceily) {
15            for(int i = x; i < xlim;i+=step) {
16               GET(baseptr,y,i,ldR) += GET(A,y,i,ceily) + GET(B,y,i,ceily);
17            }
18         }
19 }
```

According to [4], the matrices V and T formed during reduction of a block can be applied to the trailing submatrix of $\begin{bmatrix} R \\ U \end{bmatrix}$ by:

$$[I - VT^TV^T]\begin{bmatrix} R \\ U \end{bmatrix} = \left[I_{n+p-k_b} - \begin{bmatrix} I_{n_b} \\ 0 \\ V \end{bmatrix} T^T \begin{bmatrix} I_{n_b} & 0 & V^T \end{bmatrix} \right] \begin{bmatrix} R(k_b : k_b + n_b - 1, k_b + n_b : m) \\ R(k_b + n_b : n, k_b + n_b : m) \\ U(1 : p, k_b + n_b : m) \end{bmatrix}$$

$$= \begin{bmatrix} (I_{n_b} - T^T)R(k_b : k_b + n_b - 1, k_b + n_b : m) - T^TV^TU(1 : p, k_b + n_b : m) \\ R(k_b + n_b : n, k_b + n_b : m) \\ -VT^TR(k_b : k_b + n_b - 1, k_b + n_b : m) + (I - VT^TV^T)U(1 : p, k_b + n_b : m) \end{bmatrix}$$

and applied to $\begin{bmatrix} d \\ e \end{bmatrix}$ by:

$$[I - VT^TV^T]\begin{bmatrix} d \\ e \end{bmatrix} = \begin{bmatrix} d(1 : k_b - 1) \\ (I_{n_b} - T^T)d(k_b : k_b + n_b - 1) - T^TV^Te \\ d(k_b + n_b : n) \\ -VT^Td(k_b : k_b + n_b - 1) + (I - VT^TV^T)e \end{bmatrix}$$

where k_b is the column index in the blocked update where the recently reduced block began, and n_b is the block size in columns.

The QR update via adding rows in pseudocode is detailed below. Inputs to the update function are the original R from the factorisation $A = QR$, the block of columns U being added. The right hand side vector from the least squares problem is $d = Q^Tb$ and the added elements to the right hand side vector corresponding to the rows of U are provided in e. n_b is the block size parameter.

```
1  QRUpdateAddRows(R,d,U,e,p,n_b) {
2        allocate d(1:n);
3        allocate R2(1:n+p,1:m);
4
5        transfer HOST:d to GPU:d;
6        transfer HOST:R(1:n,1:m) to GPU:R2(1:n,1:m);
7        transfer HOST:U to GPU:U;
8
9        allocate V(1:p,1:n_b);
10       allocate T(1:n_b,1:n_b);
11       allocate tau(1:n_b);
12
13       int blockcounter = 0;
14       for j = 1:n_b:m
15
16          %PHASE 1: REDUCE BLOCK
17          jb = min(n_b,m-j);
18
19          endblock = (i+jb-1);
20          for k = j:endblock
21             index = k-j+1;
22             %form a Householder vector.
23             get address of global s;
24
25             s = dot(U(1:p,k));
26            [tau(index) V(1:p,index)]=housekernel<<<>>>(U(1:p,k),R2(k,k));
27
```

```
28              allocate VTU(1:jb-index+1);
29
30              [VTU] = gemv(U(1:p,k:endblock)',V(1:p,index);
31
32              allocate RJ(1:jb-index+1);
33
34              copy R2(k,k:endblock) to RJ;
35
36              %two rank one updates to U.
37
38              ger(tau(index),V(1:p,index),RJ,U(1:p,k:endblock))
39              ger(tau(index),V(1:p,index),VTU,U(1:p,k:endblock))
40
41              %add two vectors to the row in R2.
42
43              axpy(tau[index],RJ,RJ);
44              axpy(tau[index],VTU,RJ);
45              copy RJ to R2(k,k:endblock);
46
47          end
48
49          %PHASE 2: BUILD T
50
51          buildT(tau,T)
52
53          %PHASE 3: UPDATE REMAINING R2, U, b AND e
54
55          allocate TV(1:jb,1:p)
56
57          [TV] = gemm(T',V');
58
59          %form temporary Te and Td.
60
61          allocate Te(1:jb);
62
63          [Te] = gemv(TV,e);
64
65          allocate Td(1:jb);
66
67          [Td] = gemv(T',b(j:endblock));
68
69          %Update e.
70
71          e = e + gemv(V(1:p,1:jb),Td);
72          e = e + gemv(V(1:p,1:jb),Te);
73
74          %Update b.
75
76          axpy(1,Td, d(j:endblock));
77          axpy(1,Te, d(j:endblock));
78
79
80          if  j + jb < m
81
82              %update trailing U and R2.
83              allocate TU(1:jb,1:m-endblock);
84              allocate TR(1:jb,1:m-endblock);
85
86              [TU] = gemm(TV,U(1:p,endblock+1:m));
87              [TR] = gemm(T,R2(j:endblock,endblock+1:m));
88
89              doubleSum<<<>>>(TR,TU,R2(j:endblock,endblock+1:m));
90
```

```
91                U(1:p,endblock+1:m) = U(1:p,endblock+1:m) + gemm(V,TR);
92                U(1:p,endblock+1:m) = U(1:p,endblock+1:m) + gemm(V,TU);
93
94            end
95        end
96    }
```

We can see that the add rows update is almost entirely made up of CUBLAS routines 'gemm', 'gemv', 'ger', 'axpy', and 'copy'.

4.4 Removing a Block of Rows

The act of removing a block of rows can be defined as considering an initial matrix A $(n \times m)$ where the QR factorisation is $A = QR$. Providing an index $k, 0 \leq k \leq n - p + 1$, and a block width p, we can define the updated QR factorisation as:

$$\begin{bmatrix} A(1:k-1, 1:m) \\ A(k+p:n, 1:m) \end{bmatrix} = \tilde{Q}\tilde{R}$$

Which we will calculate given just the original Q and R. A visualisation of an example problem where $m = 5$, $n = 12$, $p = 4$ and $k = 5$ is given in Figure 4.4.

To update Q and R, zeros must be introduced into Q. As Givens transformations to this end are applied to Q^T, we can keep Q in column major format and apply the Givens algorithm as detailed in Figure 3.1. A strip of rows Z is assigned within Q corresponding to the removed rows from A, $Z := Q(k : k + p - 1, 1 : n)$, as shaded in blue in Figure 4.4. This strip is where the Givens coefficient calculation kernel is applied. For efficiency, the number of kernels being spawned per step can be greatly reduced by allocating a large contiguous block of memory to house the matrices Q and R along with the vector d as pictured in Figure 4.5. An `applyGivens<<<>>>` kernel can then be applied once to the entire composite matrix as opposed to applying a kernel per transformed matrix. This also presents the opportunity to align memory for all accesses by allocating the composite matrix via `cudaMallocPitch()`. The thread blocks for the `applyGivens<<<>>>` kernel can be increased in size from $x = 128$ to $x = 256$ for better time efficiency due to the large size of the composite matrix. This reduces the number of blocks, and therefore scheduling overhead, as well as increases the performance gain from the use of shared memory for Givens coefficients, as discussed in Section 3.2.

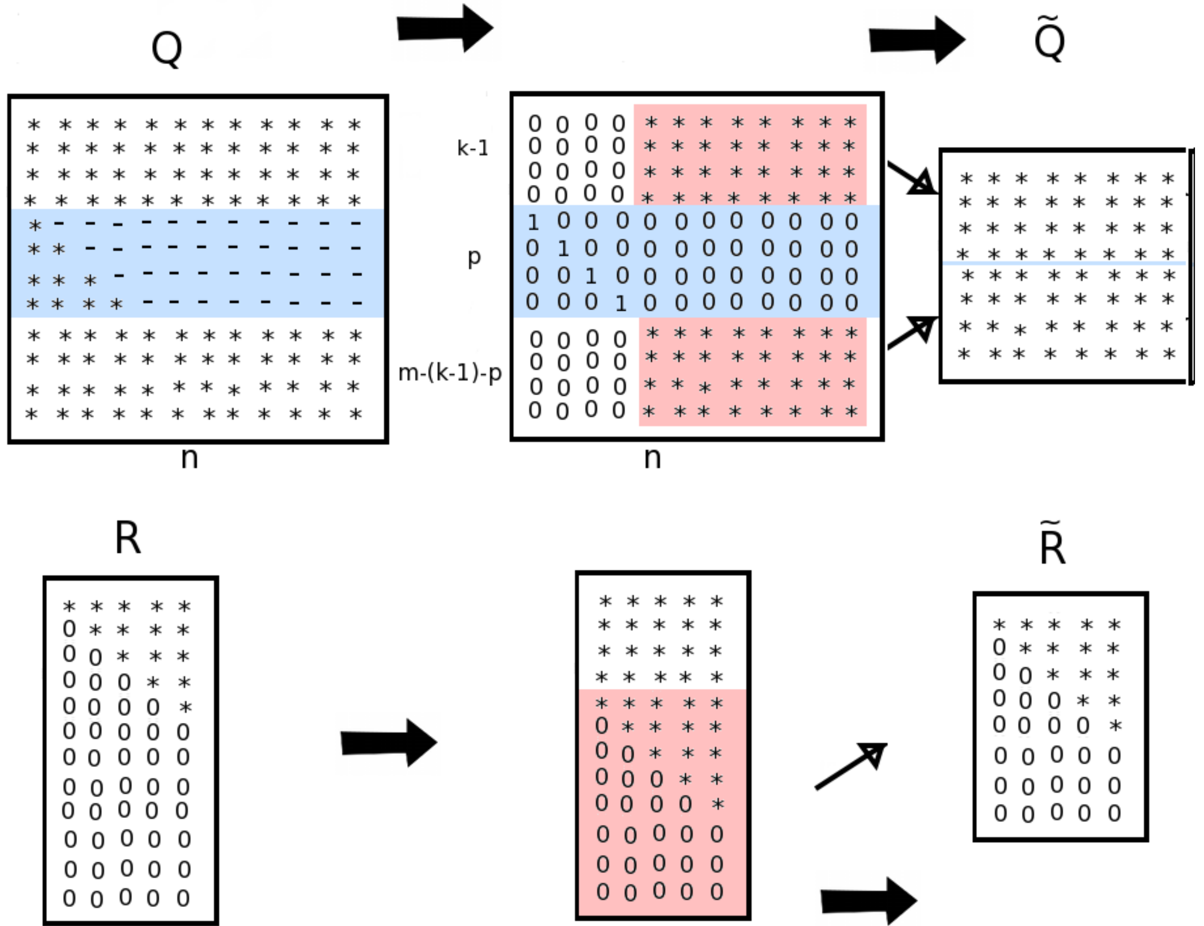

Figure 4.4: A diagram of the update of the matrix R, the '-' symbol shows the non-zeros are to be explicitly made zero during the update procedure. '*' shows a generic non-zero element. The shaded blue shows the rows in Q corresponding to the rows removed in A. The elements shaded in red are the elements to be copied to form $\tilde{Q}$ and $\tilde{R}$. In contrast to the other updating algorithms presented in this project, the matrix Q is the subject of the calculated Givens rotations as opposed to R.

The QR update via removing rows in pseudocode is detailed below. Inputs to the update function are the original Q and R from the factorisation $A = QR$, and the right hand side vector from the least squares problem $d = Q^T b$.

```
1  QRUpdateRemoveRows(R,Q,d,k,p) {
2          allocate R(1:n,1:m);
3          %allocate a composite matrix to contain, Q, R', and b.
4          allocate C(1:n+m+1,1:n);
5
6          transfer HOST:R to GPU:R;
7
8          %transfer Q and d to their positions in the composite matrix.
9          transfer HOST:Q to GPU:C(1:n,1:n);
10         transfer HOST:d to GPU:C(n+m+1,1:n);
11
12         %assign the strip in Q that corresponds to the removed rows.
13         Z = C(k:k+p-1,n);
14
15         %transpose R into the composite matrix.
16         transpose<<<>>> R to C(n+1:n+m,1:n);
17
18         startlim = 1;
19         endlim = pnumadded;
```

```
20          start = startlim;
21          ending = start+1;
22          q = n-1;
23
24          %allocate an arbitrarily large buffer for c and s
25          allocate s(1:endlim-startlim+1);
26          allocate c(1:endlim-startlim+1);
27
28          counter = 0;
29          while ending > start
30                  sync;
31
32                  [s c] = makeGivens<<<>>>(Z);
33
34                  sync;
35                  applyGivens<<<>>>(C,s,c);
36
37                  %if the first givens rotation is mapped to zero
38                  %within the triangle that needs to remain
39                  %nonzero within the matrix, we increment the
40                  %starting barrier. alternatively and initially, we
41                  %decrement n.
42                  if (n+(start-startlim) <= start)
43                          start++;
44                  else
45                          q--;
46                  end
47                  %every 2 iterations we increment the number of
48                  %Givens rotations performed in a step as we are
49                  %making zeros in a block. We increment the end
50                  %barrier until we reach the end of the matrix.
51                  counter++;
52                  if (mod(counter,2) == 0 && ending < endlim)
53                          ending++;
54                  end
55
56          end
57
58          allocate R(1:n-p,1:m);
59          transpose<<<>>> C(n+1:n+m,p+1:n) to R;
60
61          %prepare the updated Q and d to output.
62          allocate Q(1:n-p,1:n-p);
63          allocate d(1:n-p);
64
65          copy C(1:k-1,p+1:n) to Q(1:k-1,1:n-p);
66          copy C(k+p:n,p+1:n) to Q(k:n-p,1:n-p);
67
68          copy C(n+m+1,p+1:n) to d;
69
70          sync;
71  }
```

- **lines 2-10:** Allocate memory on the GPU along with the large composite matrix C. Q and d are transferred to their places in C consistent with Figure 4.5.
- **line 13:** Assign the variable Z to the strip to be made zero by Givens transformations, Z is shown shaded in blue in Figure 4.4.
- **line 16:** Transpose R to its place in the composite matrix C. The transpose procedure is as described in 3.4.

- **lines 18-56:** The block Givens transform algorithm described in Figure 3.1. In the process of the transformation the Q matrix is reduced to the form shown in the centre of the top of Figure 4.4, with zeros in the first p columns and in Z, with the identity matrix embedded in the first p columns of Z. This form is guaranteed after the transformation as Q is orthogonal.
- **lines 58-68:** Prepare the outputs of the update, $\tilde{Q}$, $\tilde{R}$ and the new d.

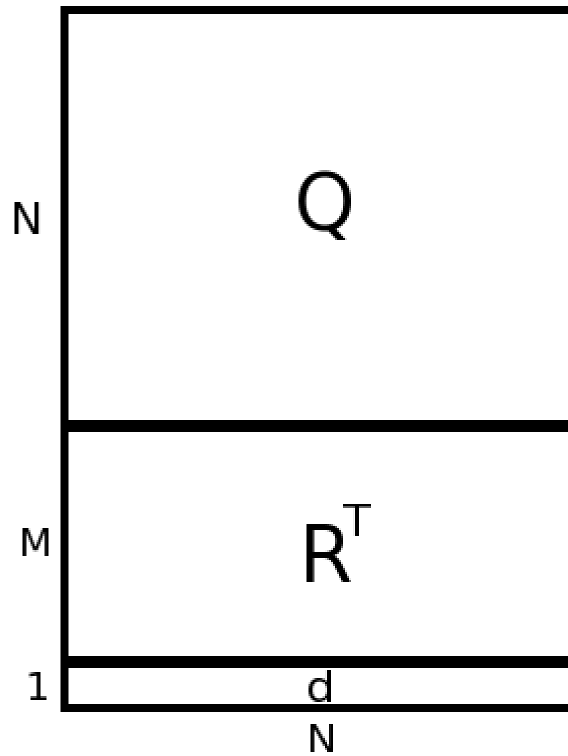

Figure 4.5: A diagram of the $n + m + 1 \times n$ dimension composite matrix used to efficiently apply Givens rotations in the update algorithm for removing a block of rows.

Chapter 5

Results

To test the effectiveness of the ideas presented in this project, plots of runtime are presented for all four of the subject updating algorithms: adding columns, removing columns, adding rows, and removing rows. All tests will be run via an Nvidia Tesla M2050 (Fermi) GPU from an Intel Xeon E5649 CPU, 12 core, 2.53GHz as host.

Runtimes of the GPU based algorithms presented in this report are first compared to a sequential implementation of each algorithm implemented entirely on a single CPU core. All kernel invocations are replaced by sequential loops and any CUBLAS calls substituted by their corresponding calls provided by the CLAPACK library. All sequential loops are organised in such a way that they are cache efficient. In order to make the comparison between the sequential and GPU updating algorithms as direct and informative as possible, input and output data from the timed algorithm will exist in host memory. In the GPU updating implementations this requires us to time the transfer of inputs to the CPU as well as timing the transfer of outputs back to the host following the update. From now on I will refer to this process as Transfer-Update-Transfer or **TUT**. For sequential implementations, no initial or final copying of data is recorded, unless it is relevant to the course of the algorithm.

The second runtime comparison presented is against a full QR factorisation based least squares solve from an existing popular commercial linear algebra library, CULA version R14. Specifically, the runtime of the CULA '`culaDeviceSgels`', a QR factorisation least squares routine, along with the necessary allocation of memory and transfer to the device is measured. This full QR factorisation begins with the updated matrix $\tilde{A}$, and right hand side vector $\tilde{b}$, in host memory and ends with the solution to the updated least squares problem:

$$\min_x ||\tilde{A}x - \tilde{b}||_2 \tag{5.1}$$

residing in memory on the GPU. As a direct and practically useful comparison, the algorithms developed in this project will be timed from the point that they receive a Q, R, and $d = Q^T b$, from a previously executed QR factorisation. The inputs are subsequently transferred to the GPU, and updated via the appropriate QR updating algorithm. The time measurement stops after the resulting $\tilde{R}$, and $\tilde{d}$ are used to solve (5.1) via a CUBLAS back substitution routine, `cublasStrsm()`. This timed procedure will be referred to from now on as Transfer-Update-Solve or **TUS**. The Q matrix is not

required for updates involving the addition of rows or the removal of columns, therefore Q is omitted from these algorithms.

The overdetermined systems that the algorithms presented in this paper are applied to are composed of uniformly random generated numbers on the interval $(-1, 1)$, this is assumed to ensure that they are always non-singular. All runtimes are measured in seconds and all values presented are average runtimes over 5 executions.

5.1 Limitations

A limitation of note with regard to the tests versus the sequential CLAPACK implementation is that the CLAPACK library itself is a Fortran port [1] and is not entirely optimised for the CPU architecture it is run on compared to a more platform, and language specific implementation of BLAS such as ATLAS [14]. Another limitation is with regard to the use of the CULA library as a GPU based comparison. CULA is a successful, mature, and highly optimised library that has been developed over many years (since as early as 2004), so any direct comparison between the runtime of the CULA algorithm and the algorithms presented in this project would be unfair. However, the relative general trend in runtimes between timed programs will be relevant.

5.2 Choosing the Block Size Parameter

For the updating algorithms that involve Householder blocking, namely adding columns, removing columns, and adding rows, each must have a constant block size n_b to use during all other tests. It is not practical in this case to try all possible other parameter values in conjunction with n_b to attempt to find an absolute optimum value. Instead we will approximate an optimum blocking value for each algorithm, based on a model problem size, as a proportion of the problem width.

The results for the block size test for GPU TUT algorithms are shown in Figure 5.1, and runtime values are given in Table 5.2. Problem parameters were chosen such that each algorithm applied 1000 Householder vectors to zero 1000 columns. The optimum block size can be seen to lie between 50 and 100 columns per block. For simplicity I will choose the optimum block size for all further tests to be $\frac{100}{1000} = 0.1$ multiplied by the number of columns to zero, or ten blocks per factorisation. For simplicity I will also assume this is the optimum block size for the GPU TUS algorithms.

Runtimes for sequential algorithms on the other hand were monotonically increasing as a function of block size, as shown in Table 5.1. For sequential tests a block size of 1 will be used, or a non-blocked approach to applying Householder reflections.

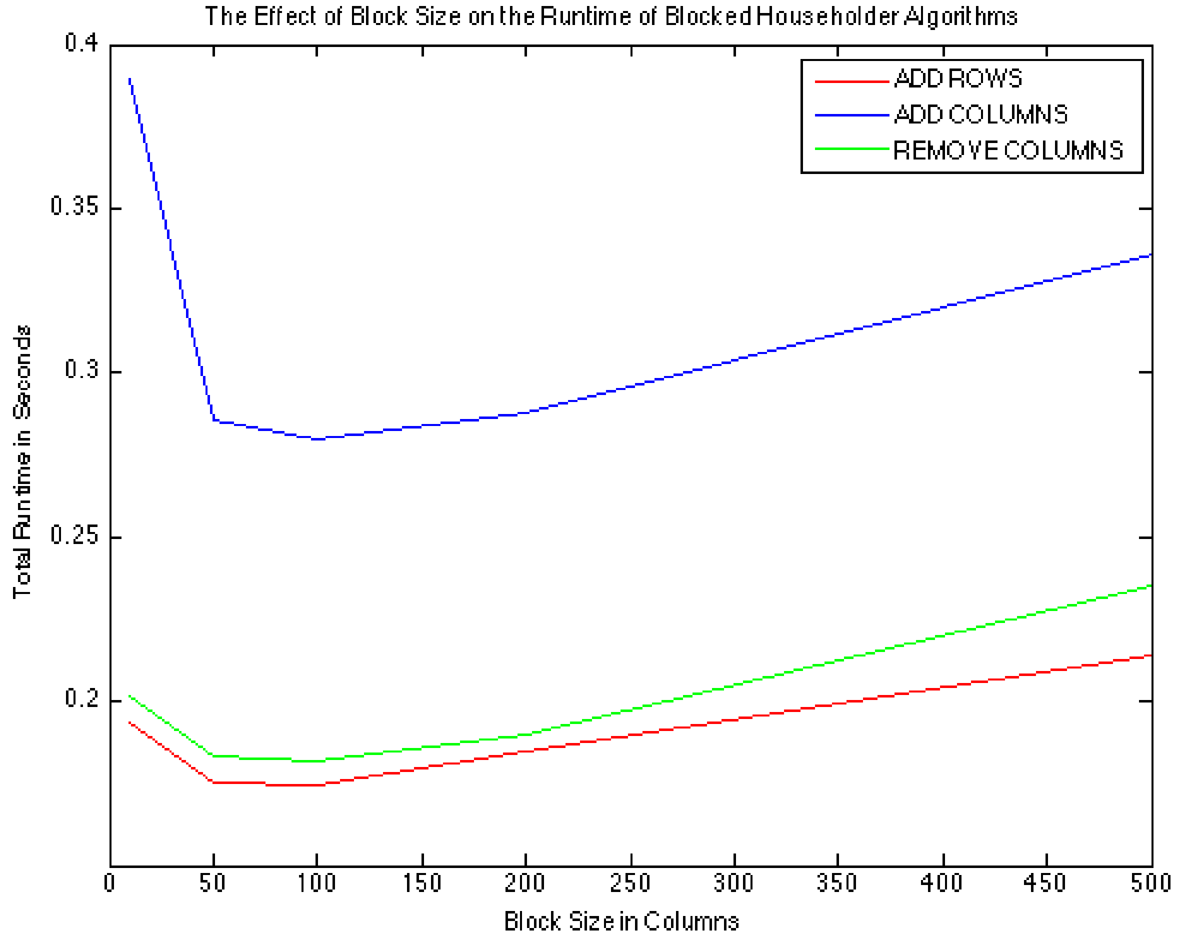

Figure 5.1: Plot of Runtimes of GPU TUT algorithms against block size for adding columns, removing columns, and adding rows. Values are given in Table 5.2.

n_b	Adding Rows $n = 4000, m = 1000$ $k = 250, p = 200$	Adding Columns $n = 4000, m = 1000$ $k = 250, p = 1000$	Removing Columns $n = 4000, m = 1200$ $k = 0, p = 200$
1	0.368	8.506	0.232
10	0.370	8.526	0.241
50	0.382	8.638	0.292
100	0.435	8.763	0.353
200	0.514	8.999	0.476
500	0.777	9.693	0.713

Table 5.1: Runtimes in seconds for sequential updating algorithms involving 1000 Householder reflections applied via different block sizes n_b. Removing Rows is omitted as is does not involve Householder Reflections.

n_b	Adding Rows $n = 4000, m = 1000$ $k = 250, p = 200$	Adding Columns $n = 4000, m = 1000$ $k = 250, p = 1000$	Removing Columns $n = 4000, m = 1200$ $k = 0, p = 200$
10	0.194	0.389	0.201
50	0.175	0.285	0.183
100	0.175	0.280	0.182
200	0.185	0.288	0.190
500	0.214	0.336	0.235

Table 5.2: Runtimes in seconds for GPU TUT algorithms involving 1000 Householder reflections applied via different block sizes n_b. Removing Rows is omitted as is does not involve Householder Reflections.

5.3 Adding Columns

Updating a QR factorisation by adding a block of columns requires the information contained in the orthogonal matrix Q to calculate $\tilde{R}$. This requirement dictates that, to test the algorithms practical viability, we must output $\tilde{Q}$ as well as $\tilde{R}$ to enable subsequent updates. The orthogonal matrix Q is a large $n \times n$ matrix that is a by-product of a full QR factorisation of A by definition, $A = QR$. Q is however, for efficiency and memory capacity reasons, not often formed in practice. For example, CULA linear solve does not require the formation of Q, this is a large performance advantage over the update algorithm. In this case therefore, the direct comparison is forfeited in favour of the practical one as the full formation of Q is rarely necessary.

5.3.1 Complexity

The complexity of a column update is:

$$O(nmp + (n-m)p^2 + (n-m)np + (m+p-k)^2 p + (m+p-k)pn) \tag{5.2}$$

where the expression could be simplified by collecting like terms, but as it is more useful for our purposes to identify the complexities of separate parts of the algorithm, this was not done. The first term comes from the matrix multiplication $Q^T U$, whereas second and third terms come from the QR factorisation of the lower part of U detailed in Figure 4.1, and the subsequent application of the transforms to the Q matrix. Finally the last two terms come from the application of Givens matrices to R and Q to finish off the update, also detailed in Figure 4.1. The full factorisation becomes, following the addition of p columns:

$$O(n(m+p)^2) \tag{5.3}$$

so the update is in theory only less complex for certain values of the constituent variables, n, m, k, and p.

k represents the location within the matrix A at which the columns U are added. When k gets smaller, the fifth term in (5.2) becomes dominant, and complexity of (5.2) approaches (5.3).

5.3.2 Adding Columns GPU TUT vs Sequential Update

The GPU TUT adding columns algorithm performed very well compared to the sequential update. Due to the large, polynomial complexity of the update, and the involvement of the large matrix Q, the GPU implementation TUT was 10s to 100s of times faster than the sequential version for fairly low values of n and m. This is presented in Table 5.3, and plotted in Figure 5.2. Runtime values for GPU TUT vs sequential update with varying p is given in Table 5.7, it can be obviously seen that there is increasing speedup of GPU TUT over the sequential update with increased p.

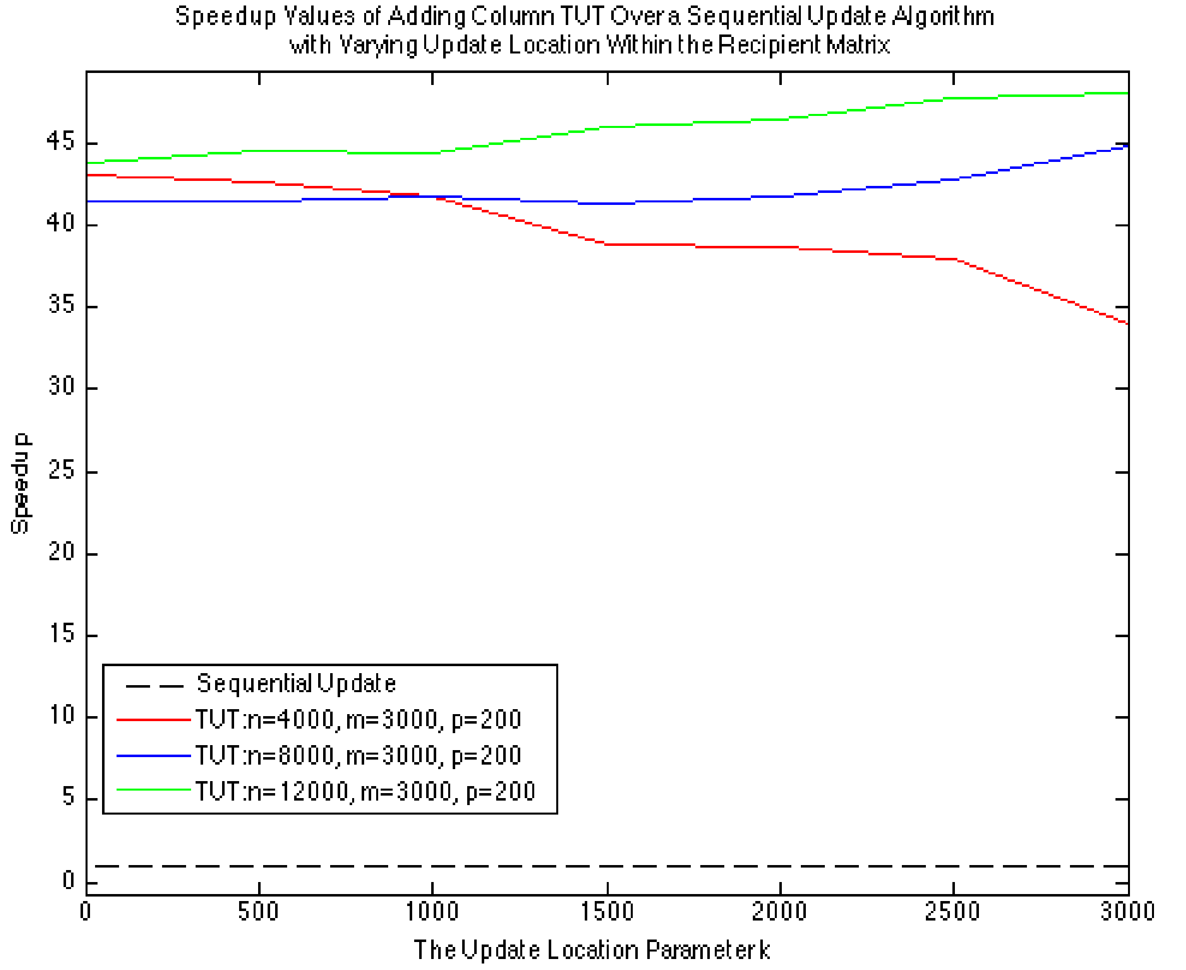

Figure 5.2: Plot of speedup of the GPU TUT by adding columns against a sequential update, with varying update location parameter k. Values for the plot are given in Table 5.3.

	n=4000		n=8000		n=12000	
k	Sequential Update	TUT	Sequential Update	TUT	Sequential Update	TUT
0	102.347	2.380	163.806	3.949	259.746	5.929
500	81.432	1.912	137.077	3.311	226.277	5.079
1000	61.654	1.478	112.467	2.697	192.782	4.348
1500	42.123	1.086	89.610	2.167	163.276	3.551
2000	28.622	0.739	68.869	1.649	135.380	2.917
2500	16.647	0.439	49.776	1.162	108.846	2.280
3000	5.846	0.172	32.932	0.735	83.457	1.737

Table 5.3: Runtimes in seconds for GPU TUT by adding columns against a sequential update, with parameters: $m = 3000, p = 200$. The plot for these values is Figure 5.2.

5.3.3 Adding Columns GPU TUS vs CULA Solve

Now, we consider our second comparison between GPU TUS and the CULA solve. The theoretical inverse relationship between k and runtime is investigated in Figure 5.3 and Figure 5.4. Runtimes are shown as speedup values with respect to the CULA least squares algorithm. The trend observed with both plots Figure 5.3 and Figure 5.4 is indeed the inverse relationship between k and runtime of the GPU update algorithm, with only the highest values of k showing observed speedup over the CULA algorithm. Different lines on both plots show data from different values of the matrix height n. The lower values of n in this case exhibit larger speedup values in the update over the full factorisation, this is possibly due to the added complexity in the updating algorithm with the involvement of the matrix Q.

Figure 5.4 shows the results from speed tests with higher m values than Figure 5.3. The complexity of a full QR factorisation is increased with higher values of m whereas complexity in the update is increased with the difference between m and k. This means that, as observed, the GPU updating algorithm performs better relative to the CULA algorithm with increased m, and k close to m.

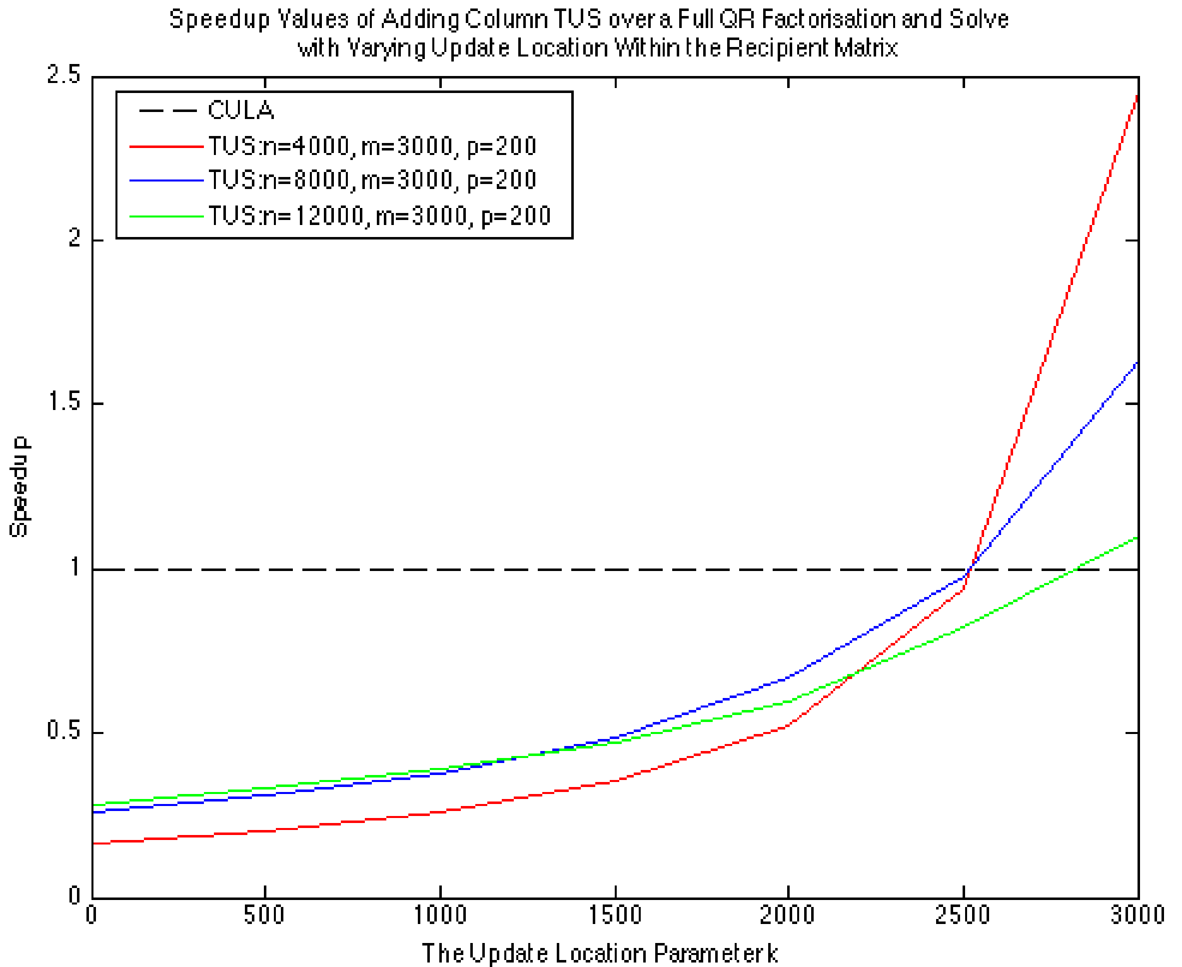

Figure 5.3: Plot of speedup of the GPU implementation of the adding columns updating algorithm against a CULA least squares solve, with varying update location parameter k. Values for the plot are given in Table 5.4.

	n=4000		n=8000		n=12000	
k	CULA Solve	TUS	CULA Solve	TUS	CULA Solve	TUS
0	0.380	2.357	1.000	3.867	1.594	5.670
500		1.887		3.219		4.825
1000		1.464		2.636		4.048
1500		1.075		2.046		3.360
2000		0.725		1.499		2.667
2500		0.404		1.021		1.941
3000		0.155		0.613		1.445

Table 5.4: Runtimes in seconds for GPU TUS by adding columns vs CULA Solve, with parameters: $m = 3000, p = 200$. The plot for these values is Figure 5.3. Note that the runtime for CULA solve is constant, this is due to that fact that varying k does not change the problem size.

	n=8000		n=12000		n=16000	
k	CULA Solve	TUS	CULA Solve	TUS	CULA Solve	TUS
0	1.797	8.585	3.003	11.385	4.241	14.458
500		7.670		10.280		13.184
1000		6.730		9.162		11.930
1500		5.951		8.152		10.642
2000		5.112		7.249		9.639
2500		4.393		6.307		8.551
3000		3.720		5.423		7.512
3500		3.068		4.562		6.534
4000		2.458		3.865		5.425
4500		1.876		3.055		4.660
5000		1.388		2.414		3.717
5500		0.911		1.675		2.953
6000		0.503		1.177		2.187

Table 5.5: Runtimes in seconds for GPU TUS by adding columns vs CULA Solve, with parameters: $m = 6000, p = 200$. The plot for these values is Figure 5.4. Note that the runtime for CULA solve is constant, this is due to that fact that varying k does not change the problem size.

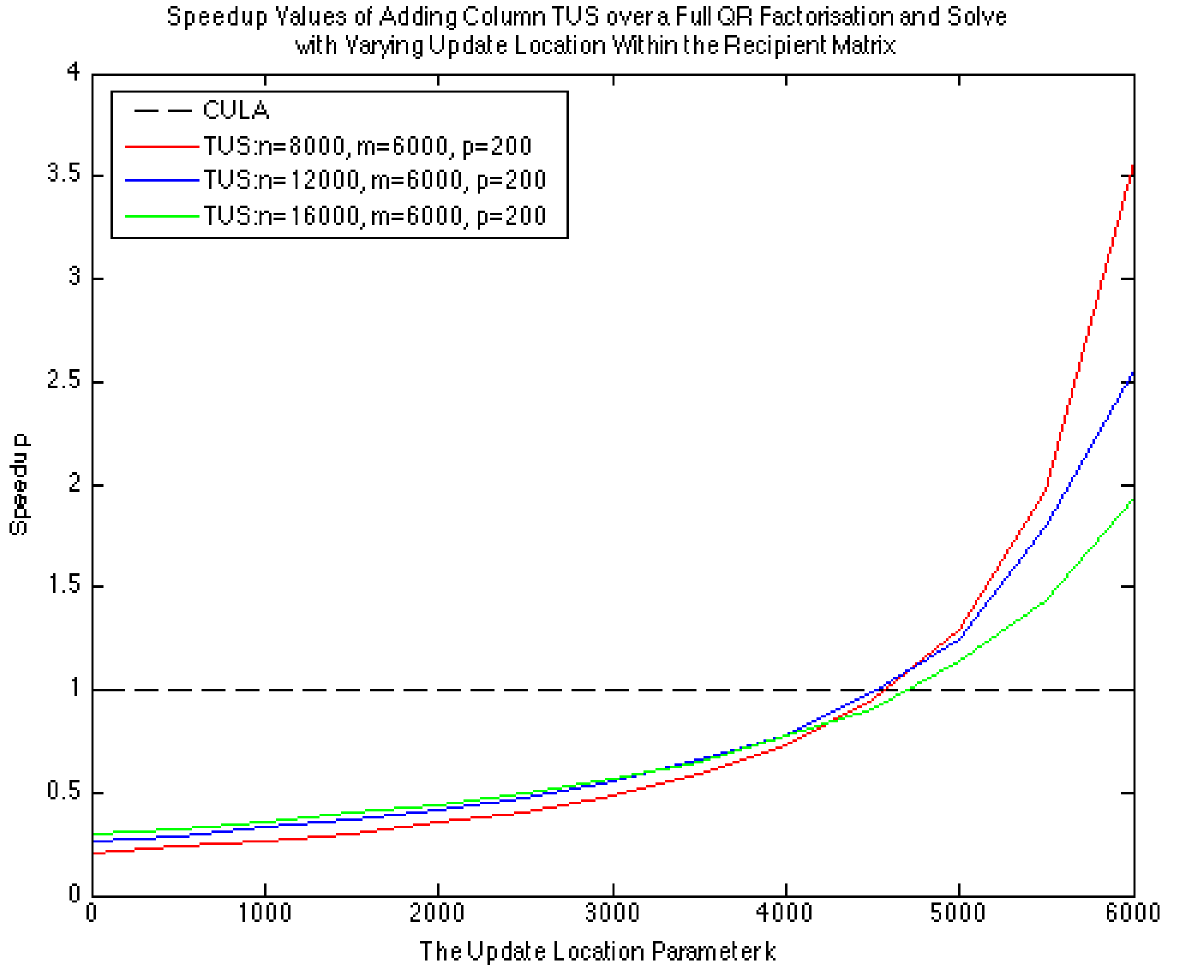

Figure 5.4: Plot of speedup of the GPU implementation of the adding columns updating algorithm against a CULA least squares solve, with varying update location parameter k. Values for this plot are shown in Table 5.5.

This trend regarding the difference between k and m is as expected as, referring the the diagram of the algorithm in Figure 4.1, the shaded red tile involved in the Givens Rotation (stage 2) of the update gets larger with $m - k$. This observation is shown to be relevant in Figure 5.5, with runtime values in Table 5.6. Figure 5.5 presents a pie chart showing the composition of the runtime of the GPU updating algorithm. The top chart presents the running of the algorithm with k close to 0. We can see that the runtime is dominated by the Givens rotation stage, which corresponds to the last two terms of (5.2), so is as expected. This conclusion is reinforced by the second chart which pictures the composition of the runtime of the same algorithm, but this time with k close to m. The runtimes of all algorithm stages other than the Givens stage in fact stay fairly constant as can be seen in Table 5.6, while the proportion contributed by the Givens stage diminishes considerably.

The origin of the poor performance of the Givens rotation stage of the algorithm may be attributed to the $O((m - k) + (m - k))$ number of kernels invoked during the procedure incurring a large overhead. This effect can be observed in Figure 5.5 as the relatively small runtime of the comparable complexity, single kernel $Q^T U$ matrix multiplication or the transpose procedure, have small runtimes in comparison to the Givens stage. A possible method to reduce the number of kernel calls required during the application of Givens rotations is presented in Figure 4.5 and investigated in Table 5.20

and Figure 5.17. The extremely small contribution overall runtime given by the transpose procedure also justifies its inclusion in the algorithm to make the Givens stage more cache efficient as discussed in Section 3.2.

The complexity of (5.2) also increases quadratically with p in the second term, corresponding to the Householder transformation procedure. This trend is shown in Figure 5.6, with values given in Table 5.7. TUS runs faster than the full GPU QR factorisation only for the smaller values of p, this is to be expected as with $p \to m$ the complexity of the update approaches the full QR factorisation. This extra complexity is due to the fact that the Householder reflections calculated to reduce the lower part of the added columns U, as pictured in the central matrix in Figure 4.1, also have to be applied to the large matrix Q. For the experiment shown in Figure 5.6, a high constant value of k was chosen. This was due to the findings presented in Figure 5.3 and Figure 5.4 that the best case performance can be gained from a small $m - k$ value. However, $k = m$ was not chosen as the Givens stage of the update would not be executed as the update would be complete following the Householder reflections stage, see Figure 4.1. This would not be an accurate portrayal of the full update algorithm due to the large contribution of the Givens stage to total runtime as shown in Figure 5.5.

Algorithm Stage	k = 500	k = 5500
Memory Transfer (HOST→GPU)	0.4913	0.5102
$Q^T U$	0.1922	0.1922
Householder Reflections	1.3950	1.3936
Transpose Procedures	0.0291	0.0006
Givens Rotations	11.0144	0.7949

Table 5.6: Runtimes in seconds corresponding to the stages pictured in Figure 5.5.

	n = 5000				n = 8000			
	Experiment 1		Experiment 2		Experiment 1		Experiment 2	
p	CULA Solve	TUS	Seq. Update	TUT	CULA Solve	TUS	Seq. Update	TUT
100	0.500	0.180	5.410	0.216	0.870	0.386	19.225	0.498
300	0.528	0.323	15.837	0.348	0.919	0.732	57.924	0.866
500	0.555	0.428	26.001	0.448	0.973	0.967	83.385	1.060
700	0.564	0.526	35.988	0.586	1.028	1.198	133.165	1.326
900	0.603	0.643	45.503	0.685	1.085	1.392	172.120	1.523
1100	0.620	0.731	54.726	0.767	1.149	1.606	205.736	1.763
1300	0.648	0.848	63.853	0.888	1.221	1.885	246.458	2.001
1500	0.683	0.954	72.324	0.974	1.230	2.027	277.459	2.181

Table 5.7: Runtimes in seconds for $k = 2980, m = 3000$. The plot for the values corresponding to the comparison between GPU TUS and the CULA solve is shown in Figure 5.6. The values corresponding to the comparison between GPU TUT and the sequential update are not plotted.

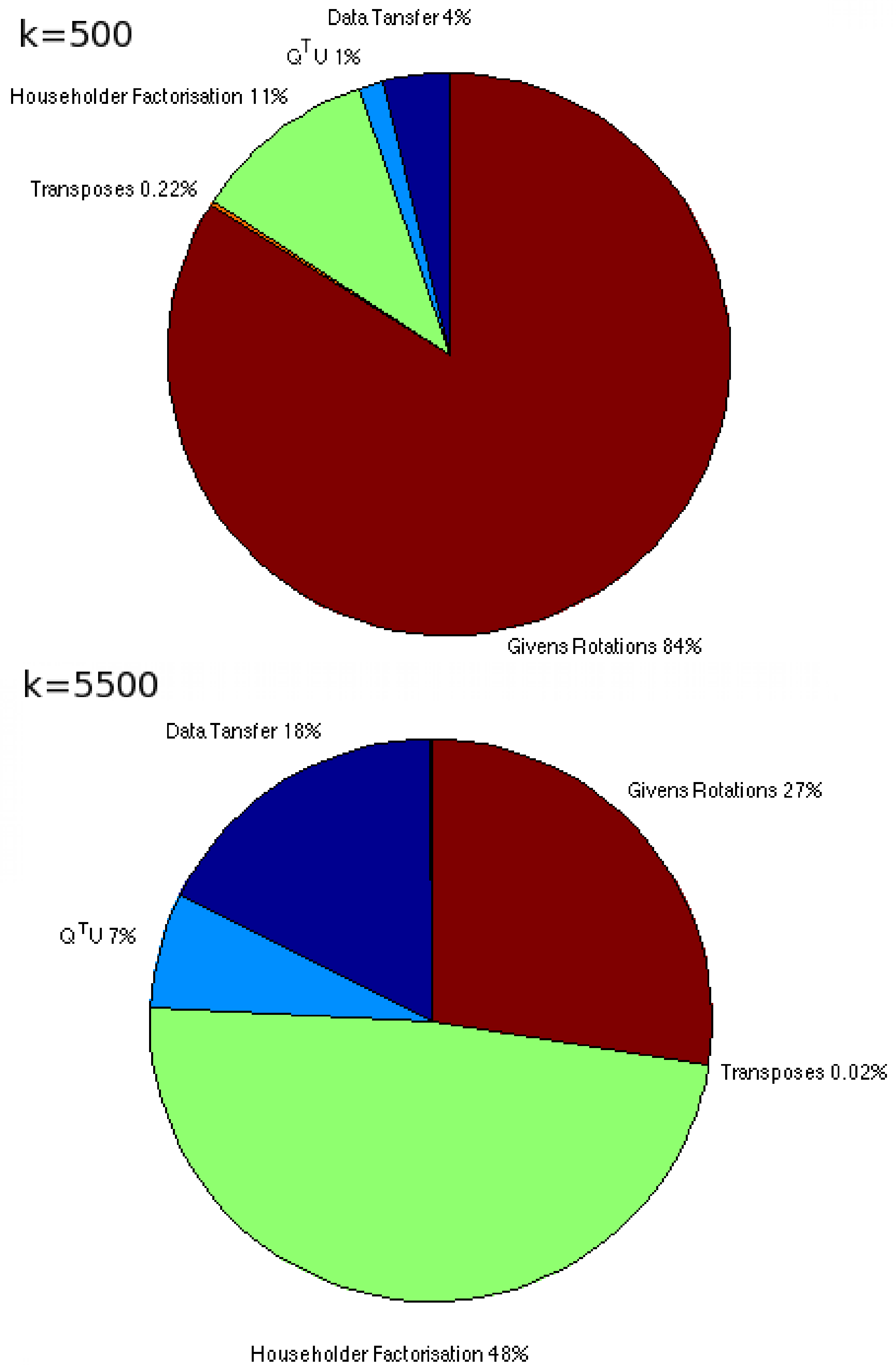

Figure 5.5: Pie charts showing the respective contributions to the runtime of the GPU update algorithm by the various components that compose it. The top execution was run via the parameters $n = 16000$, $m = 6000$, $k = 500$, and $p = 200$, and the bottom via $n = 16000$, $m = 6000$, $k = 5500$,and $p = 200$.

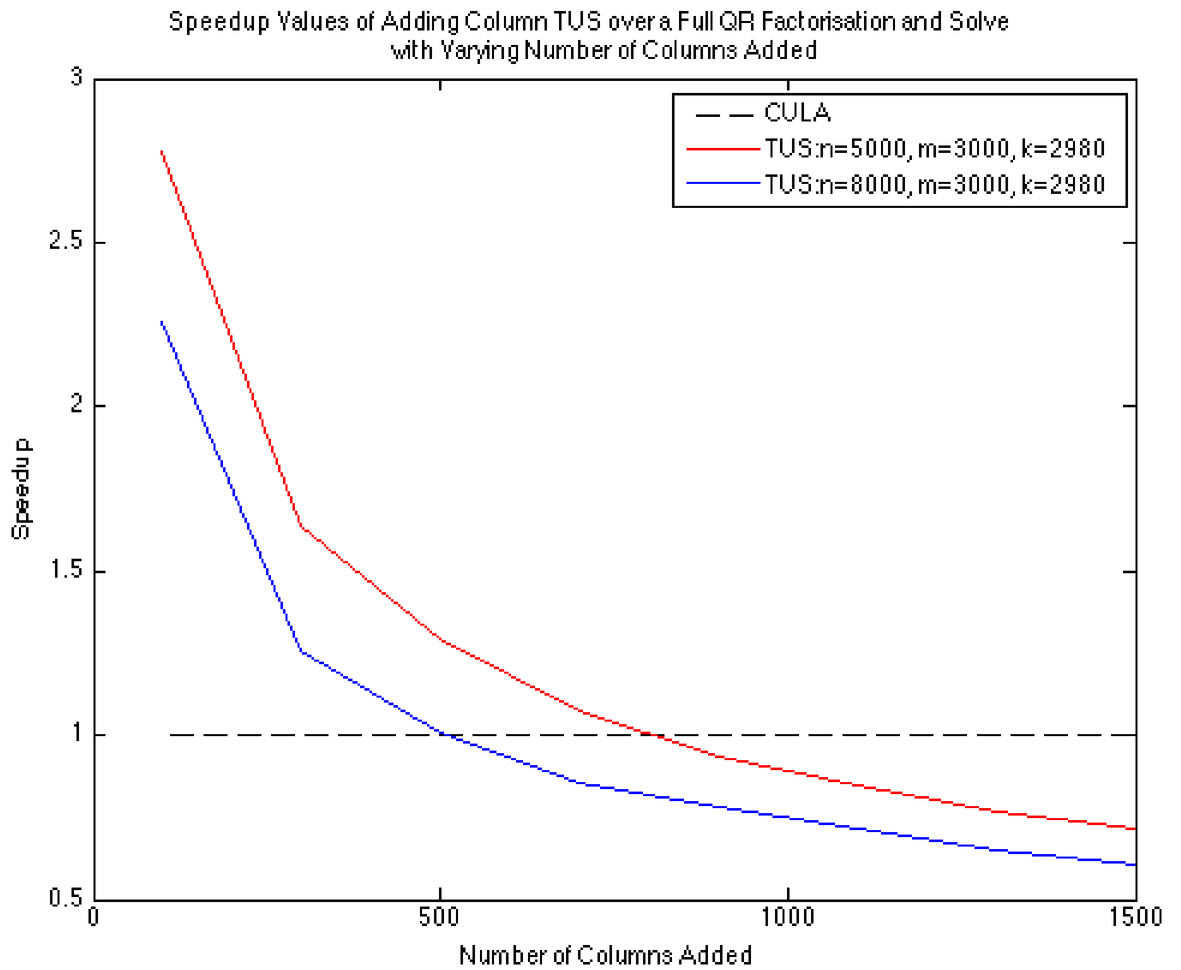

Figure 5.6: Plot of speedup of the GPU implementation of the adding columns updating algorithm against a CULA least squares solve, with varying update size parameter p. $k = 2980$ as if $k = 3000$ the Givens rotation stage of the updating algorithm would not take place.

5.4 Removing Columns

Unlike the update by adding columns discussed in the previous section, the update by removing columns can be implemented without the involvement of the orthogonal matrix Q.

5.4.1 Complexity

Without the involvement of Q, a decreased complexity is expected compared to algorithms that do involve Q. The complexity of a removing columns update procedure is:

$$O((m-k-p)^2 p) \tag{5.4}$$

compared to the complexity of a full QR factorisation of an $n \times m$ matrix with p columns removed:

$$O(n(m-p)^2) \tag{5.5}$$

An obvious first observation given (5.4) and (5.5) is that the complexity of an update via removing columns is not dependent on the matrix height n, whereas the complexity of a full QR factorisation is. The implication of this fact is that, given a sufficiently large n, GPU TUS will outperform a full GPU QR factorisation. The trend with n is shown in Figure 5.9 with a linear increase in speedup of the GPU updating algorithm over the full QR factorisation. This is a feature of the underlying algorithms as in Table 5.11 the runtime of the updating algorithm stays fairly constant with increased n and runtime for the full QR factorisation increases linearly, any minor increase in the runtime for GPU TUS with increased n can be attributed to increased data transfer overhead. Another observation of note with (5.4) is that the complexity of an update by removing columns is, as with adding columns, heavily dependent on the value $m-k$. This is in fact demonstrated for removing columns GPU TUT vs the sequential implementation in Figure 5.7, and GPU TUS vs a full QR factorisation in Figure 5.8.

5.4.2 Removing Columns GPU TUT vs Sequential Update

GPU TUT gains a speedup over the sequential update for large values of $m-k$ and p as larger values of these variables increases the number of flops involved in the update by (5.4). Higher p values increase the length of the Householder vectors applied to R, and in turn increases the size of the matrix blocks in the blocked GPU Householder implementation. Increased complexity in these BLAS level 3 routines better utilises the instruction bandwidth available on the GPU, and therefore increases throughput and overall algorithm efficiency over a sequential implementation. A 'Householder block' is pictured as shaded in red in the centre of Figure 4.2. Larger $m-k$ values increases the size of the trailing submatrix to be multiplied as well as increasing Householder block size in this case, which increases instruction throughput on the GPU in the same way as with p.

The 'trailing submatrix' referred to in this case is pictured as shaded in red to the right in Figure 4.2. For small values of $m-k$ there is observed slowdown of GPU TUT against the sequential implementation, this is due to the small overall algorithmic complexity and therefore runtime being exceeded by the various overheads of GPU execution. These overheads are namely the cost of kernel calls and, more importantly, initial and final transfers of data to and from the GPU respectively. Even

though this discussed slowdown is shown in Figure 5.7, the actual runtime values shown in Table 5.8, Table 5.9, and Table 5.10 corresponding to these cases are practically negligible for both GPU TUT and the sequential update.

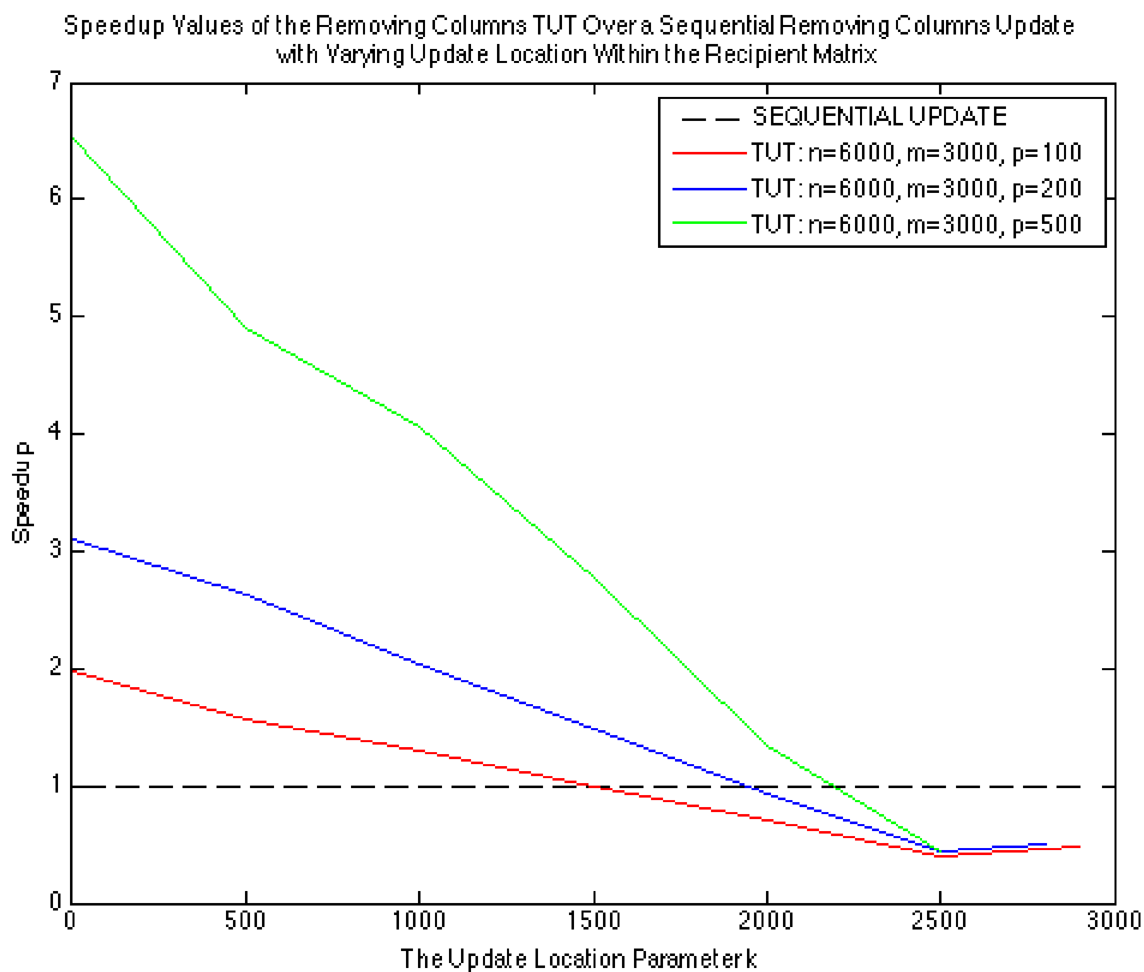

Figure 5.7: A plot of the speedup of GPU TUT for removing columns against a sequential update via removing columns. Each line exhibits a different p value and each line shows the trend of speedup with varying k.

5.4.3 Removing Columns GPU TUS vs CULA Solve

GPU TUS gains speedup over the full CULA QR for all values except the smallest k, as discussed previously however, simply increasing n would mean GPU TUS speedup over full QR for all k. For comparison purposes however, modest values of n are used throughout this section. As can be easily seen from Figure 4.2, $m - k - p + 1$ Householder reflectors are required per update. The smaller the value of $m - k$ therefore, the less work there is to do to complete an update. Increased values of p also decreases the number of the $m - k - p + 1$ Householder reflectors required to carry out an update, however increased p also increases the amount of overall complexity for GPU TUS by (5.4) and decreases the overall complexity for a full QR factorisation by (5.5), due to the smaller matrix size after column removal, this theoretically should close the gap in runtime. However, as the GPU has exceptional instruction throughput and performance is therefore less sensitive to increased work within a kernel and is more sensitive to an increased number of kernel calls, the overall effect of increased p is to increase speedup of GPU TUS over full GPU QR.

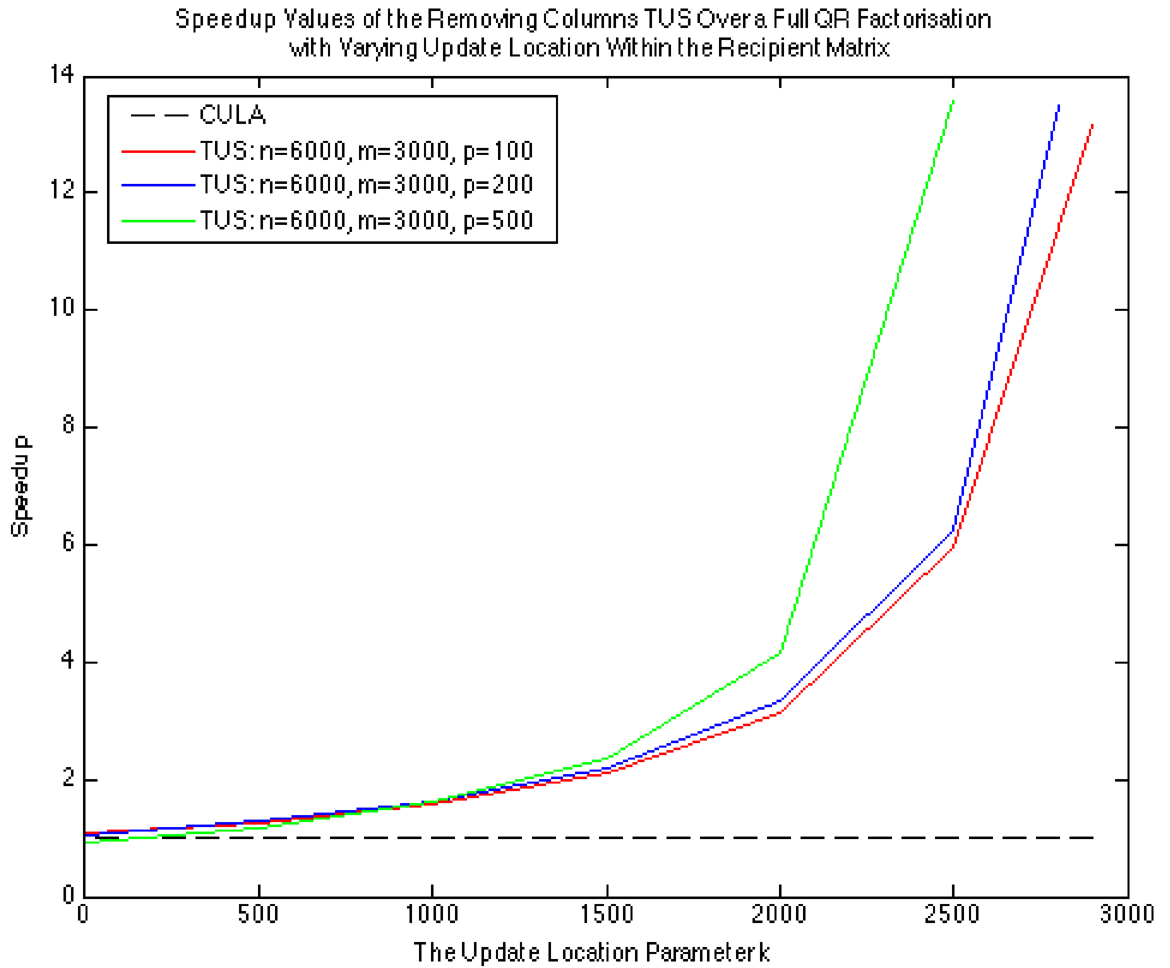

Figure 5.8: A plot of the speedup of GPU TUS for removing columns against a full QR factorisation in CULA. Each line exhibits a different p value and each line shows the trend of speedup with varying k.

	Experiment 1		Experiment 2	
k	CULA Solve	TUS	Sequential Update	TUT
0	0.662	0.601	1.260	0.640
500		0.526	0.811	0.520
1000		0.417	0.557	0.426
1500		0.312	0.315	0.318
2000		0.212	0.145	0.208
2500		0.111	0.049	0.124
2900		0.050	0.026	0.054

Table 5.8: Runtimes in seconds for **p=100**, $m = 3000, n = 6000$.

	Experiment 1		Experiment 2	
k	CULA Solve	TUS	Sequential Update	TUT
0	0.631	0.605	1.886	0.609
500		0.482	1.282	0.489
1000		0.387	0.796	0.392
1500		0.287	0.429	0.289
2000		0.189	0.178	0.190
2500		0.101	0.046	0.106
2800		0.047	0.024	0.048

Table 5.9: Runtimes in seconds for **p=200**, $m = 3000, n = 6000$.

	Experiment 1		Experiment 2	
k	CULA Solve	TUS	Sequential Update	TUT
0	0.565	0.609	3.987	0.611
500		0.479	2.358	0.481
1000		0.353	1.455	0.359
1500		0.239	0.664	0.239
2000		0.136	0.186	0.139
2500		0.042	0.025	0.055

Table 5.10: Runtimes in seconds for **p=500**, $m = 3000, n = 6000$.

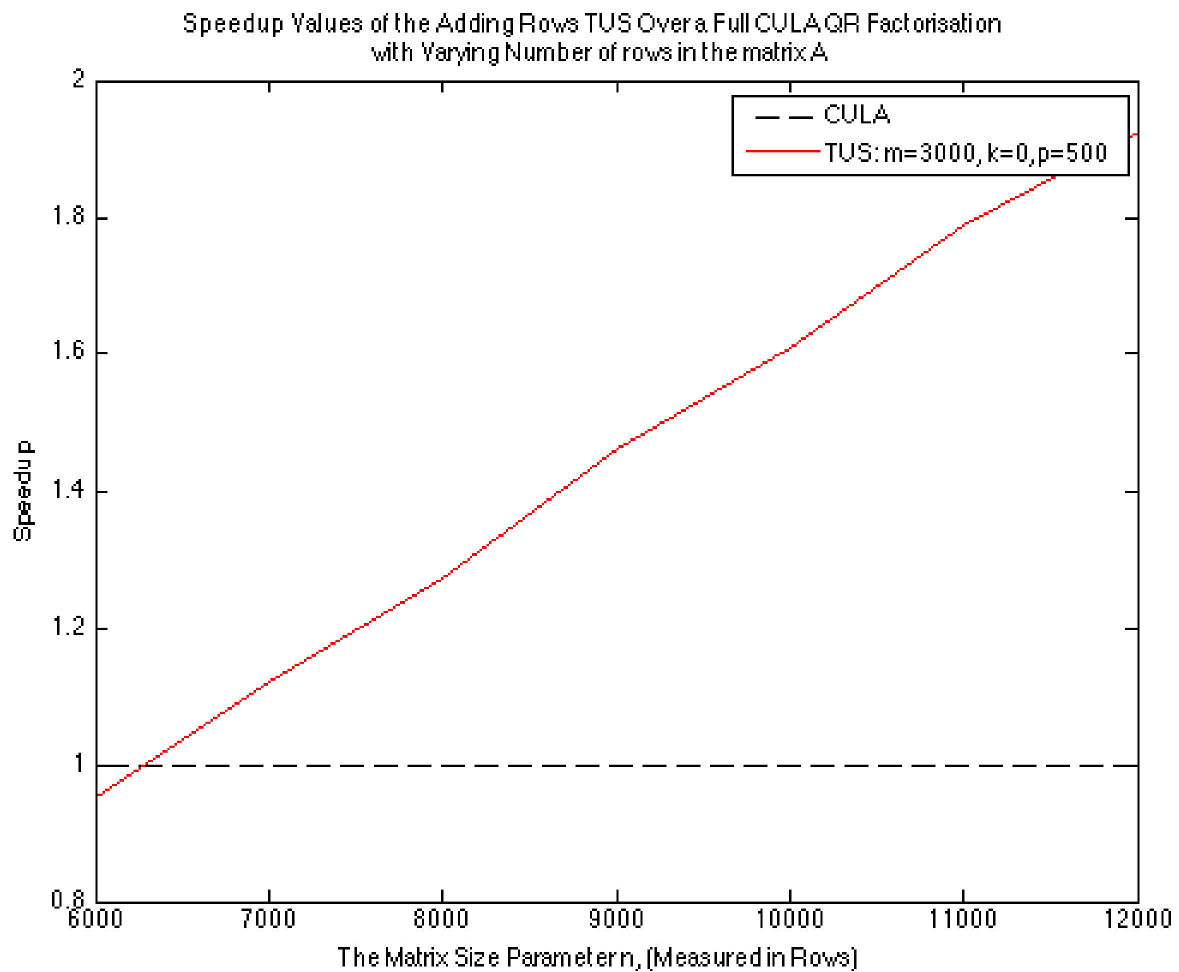

Figure 5.9: A plot of the speedup of GPU TUS for removing columns against a full QR factorisation in CULA. The plot shows the trend of speedup with varying n. Values are given In Table 5.11.

n	CULA Solve	TUS
6000	0.562	0.591
7000	0.672	0.600
8000	0.753	0.592
9000	0.871	0.596
10000	0.959	0.597
11000	1.101	0.616
12000	1.193	0.620

Table 5.11: Runtimes in seconds for removing columns GPU TUS against a full QR factorisation for parameters $m = 3000, k = 0, p = 500$. The plot corresponding to these values is shown in Figure 5.9.

5.5 Adding Rows

Unlike the other updating algorithms, the block of rows U added to the matrix A can be permuted to the bottom of the matrix without altering the algorithm method as shown in Section 2.4, and Figure 4.3. Given this fact we can neglect the variation of the update location parameter k for this section and set it to 0 for all tests involving the update by adding rows. Like the update via removing columns, the update via adding rows does not require the orthogonal matrix Q as an input.

5.5.1 Complexity

The complexity of the update by adding rows is:

$$O(m^2p + m^2) \tag{5.6}$$

compared to the complexity of a full QR factorisation of an $n \times m$ matrix with p rows added:

$$O((n + p)m^2) \tag{5.7}$$

Once again, the matrix height n does not appear within the (5.6) complexity formula. This is due to the fact that, as shown in Figure 4.3, each Householder transformation needs only to be applied to the added rows matrix U of dimension $p \times m$, and a single row within the non-zero triangular part of R. This again means that the runtime of the QR update will remain constant with increasing n while the runtime of the full QR factorisation will increase linearly, as shown in Figure 5.13 and Table 5.16.

5.5.2 Adding Rows GPU TUT vs Sequential Update

The trend in speedup of GPU TUT over the sequential update is given in Figure 5.10, for varying update size p. As expected we can see increased speedup over the sequential version for increased p as this increases the amount of computation to be done per Householder transform, and therefore in the GPU TUT implementation, more computation per kernel invocation. As previously discussed, more work per kernel invocation increases efficiency on the GPU due to its large instruction bandwidth. Increasing the matrix width parameter m also increases GPU TUT speedup over the sequential algorithm. This can be attributed to the increased size of the trailing submatrix that the blocked Householder matrix is applied to, pictured on the right of Figure 4.3 shaded in red. The application of a blocked Householder matrix to another matrix is a BLAS level 3 operation, so as large BLAS level 3 operations achieve high instruction throughput on the GPU, the efficiency and speedup is increased.

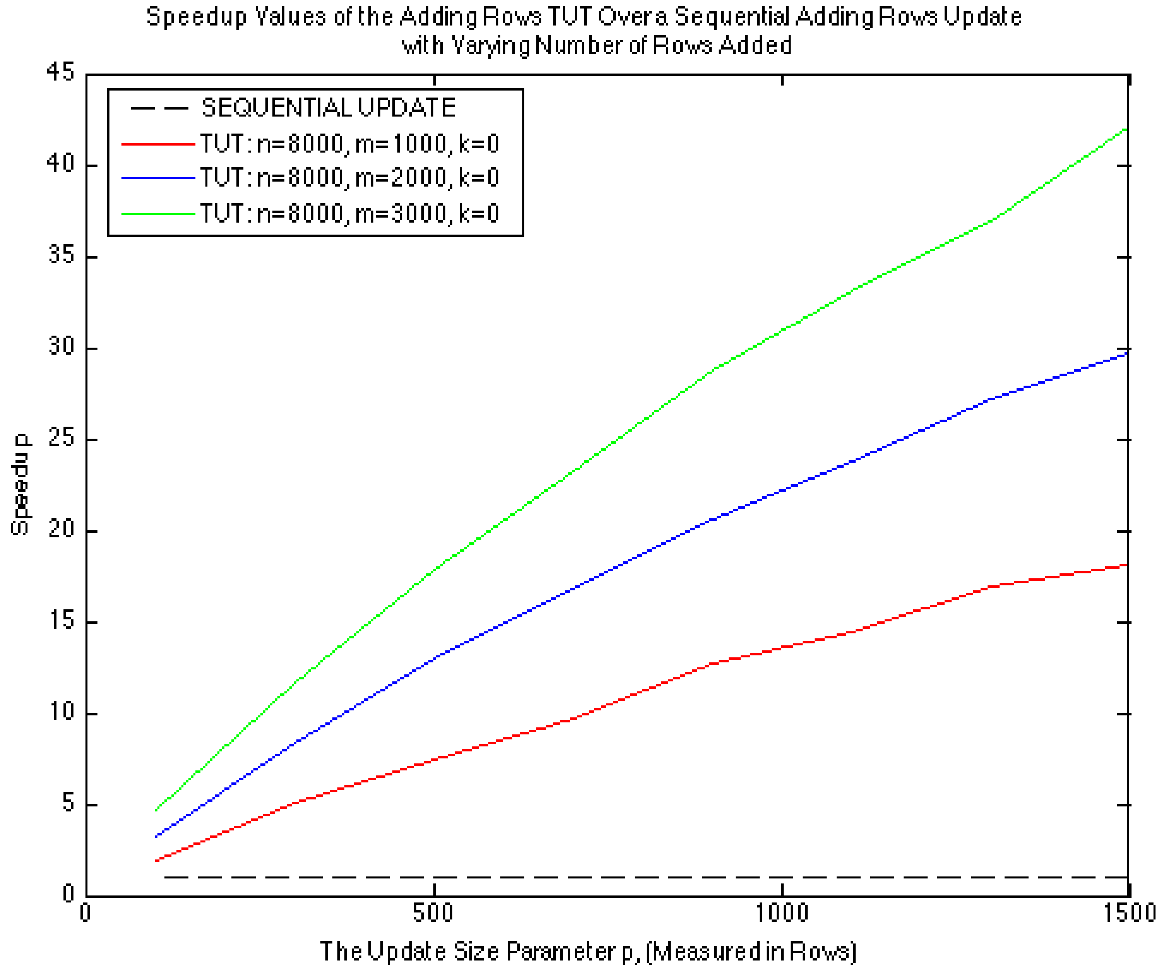

Figure 5.10: A plot of the speedup of GPU TUT for adding rows against a sequential update via adding rows. Each line exhibits a different m value and each line shows the trend of speedup with varying p. Values are given in Table 5.12, Table 5.13, and Table 5.14.

5.5.3 Adding Rows GPU TUS vs CULA Solve

Now, considering Figure 5.11 involving speedup of GPU TUS over CULA Solve. We observe the opposite trend in speedup compared to the observations in Figure 5.10 against the sequential update, with speedups decreasing with increased p and m. The trend involving p however is not surprising as the relative complexities $\frac{(5.6)}{(5.7)} \to 1$ as $p \to \infty$, this however suggests that it should be the case that $speedup \to 1$ as $p \to \infty$ as well, which is not the case as observed by the crossover on Figure 5.12 for $p = 1500$, at $m = 7000$. This can be explained in part by the fact that, in the process of avoiding multiplication by zero, pictured in Figure 4.3, the application of Householder transformations was split into twice the number of CUBLAS calls than is required in an efficient blocked full QR factorisation such as the one presented in [5]. This cost seems to be amortised by the reduced amount of complexity in the update for low p values, for higher values of p however, this is not the case.

The main purpose of Figure 5.12 is to fully illustrate the trend of speedup of GPU TUS over CULA Solve with varying matrix width m. Efficiency of GPU TUS decreases with increasing m due to the fact that, for each of the m Householder reflections, multiple CUBLAS calls take place sequentially as described in Section 4.3. This is not as efficient as implementing just one kernel per Householder reflection at the expense of a longer development time. It can be observed by Table 5.15 that runtimes

of both GPU TUS and CULA Solve increase super linearly with increased m, as expected from (5.6) and (5.7).

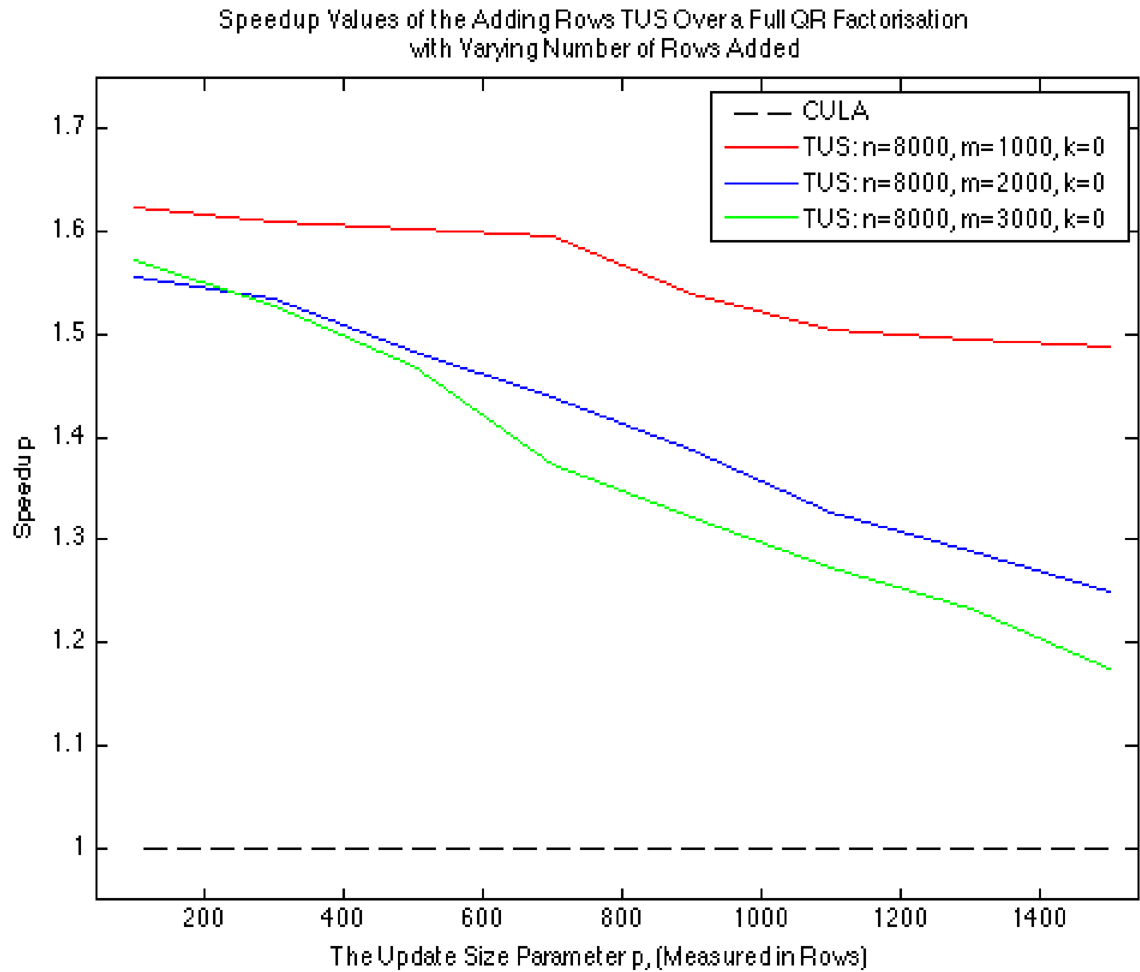

Figure 5.11: A plot of the speedup of GPU TUS for adding rows against a full GPU CULA QR factorisation. Each line exhibits a different m value and each line shows the trend of speedup with varying p. Values are given in Table 5.12, Table 5.13, and Table 5.14.

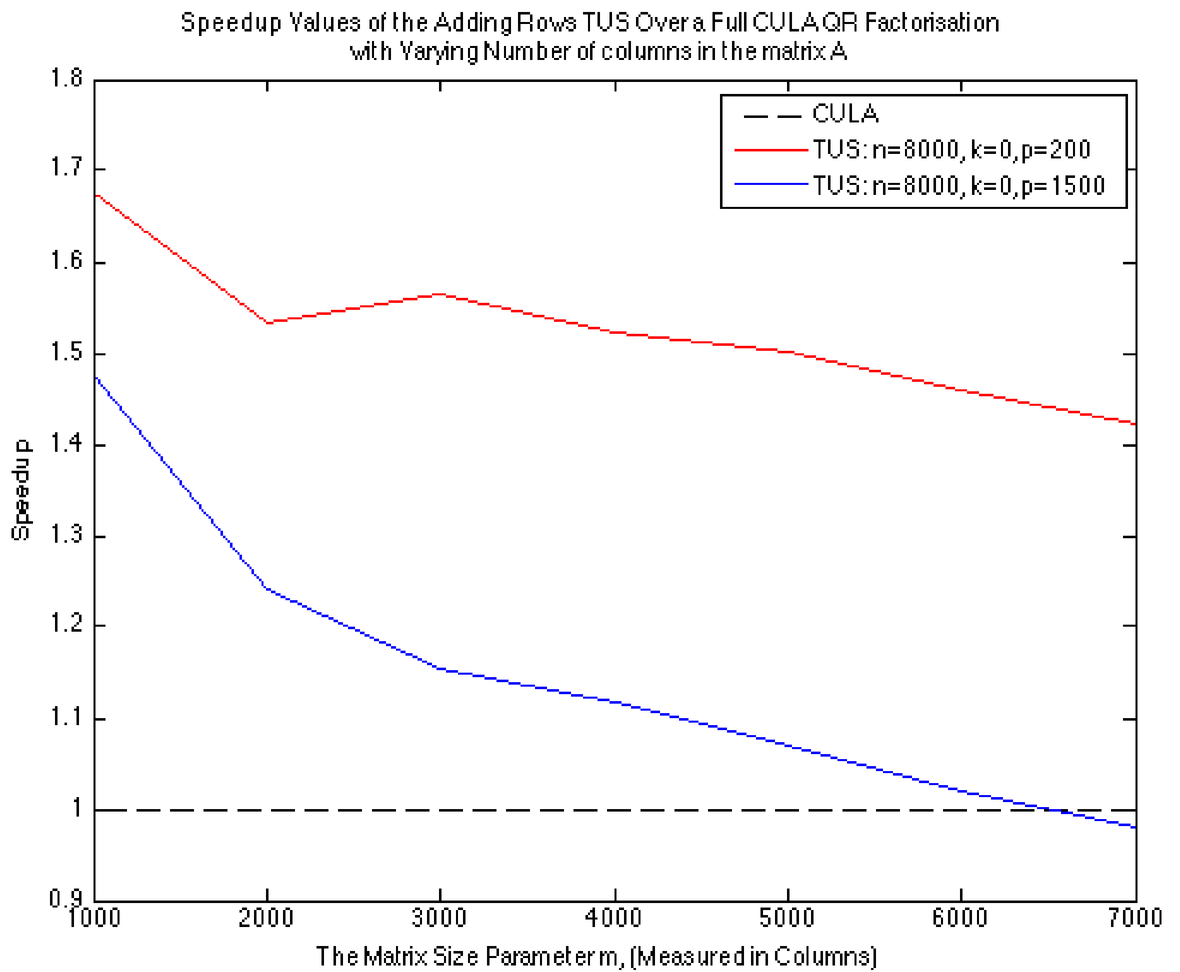

Figure 5.12: A plot of the speedup of GPU TUS for adding rows against a full GPU CULA QR factorisation. Each line exhibits a different p value and each line shows the trend of speedup with varying m. Values are given in Table 5.15.

	Experiment 1		Experiment 2	
p	CULA Solve	TUS	Sequential Update	TUT
100	0.316	0.194	0.375	0.203
300	0.322	0.200	1.020	0.203
500	0.328	0.205	1.622	0.218
700	0.341	0.214	2.181	0.227
900	0.343	0.223	2.953	0.233
1100	0.359	0.239	3.501	0.244
1300	0.367	0.245	4.240	0.251
1500	0.370	0.249	4.840	0.267

Table 5.12: Runtimes in seconds for **m=1000**, $k = 0, n = 8000$.

	Experiment 1		Experiment 2	
p	CULA Solve	TUS	Sequential Update	TUT
100	0.611	0.393	1.296	0.408
300	0.627	0.409	3.506	0.422
500	0.641	0.433	5.727	0.443
700	0.658	0.458	7.960	0.476
900	0.673	0.485	10.255	0.499
1100	0.687	0.518	12.493	0.526
1300	0.690	0.536	14.839	0.545
1500	0.717	0.574	17.578	0.591

Table 5.13: Runtimes in seconds for **m=2000**, $k = 0, n = 8000$.

	Experiment 1		Experiment 2	
p	CULA Solve	TUS	Sequential Update	TUT
100	0.958	0.609	2.904	0.629
300	1.008	0.660	7.894	0.674
500	1.025	0.698	12.909	0.723
700	1.039	0.757	18.179	0.781
900	1.089	0.824	23.764	0.827
1100	1.115	0.876	28.877	0.872
1300	1.140	0.925	34.577	0.935
1500	1.136	0.968	41.810	0.995

Table 5.14: Runtimes in seconds for **m=3000**, $k = 0, n = 8000$.

	p=200		p=1500	
m	CULA Solve	TUS	CULA Solve	TUS
1000	0.321	0.192	0.373	0.253
2000	0.611	0.398	0.730	0.589
3000	0.981	0.627	1.137	0.986
4000	1.383	0.908	1.654	1.482
5000	1.802	1.200	2.219	2.075
6000	2.265	1.551	2.807	2.754
7000	2.746	1.931	3.414	3.485

Table 5.15: Runtimes in seconds for GPU TUS for adding rows against a full GPU CULA QR factorisation at parameter values: $n = 8000, k = 0$. Speedups of GPU TUS over CULA QR factorisation are plotted in Figure 5.12.

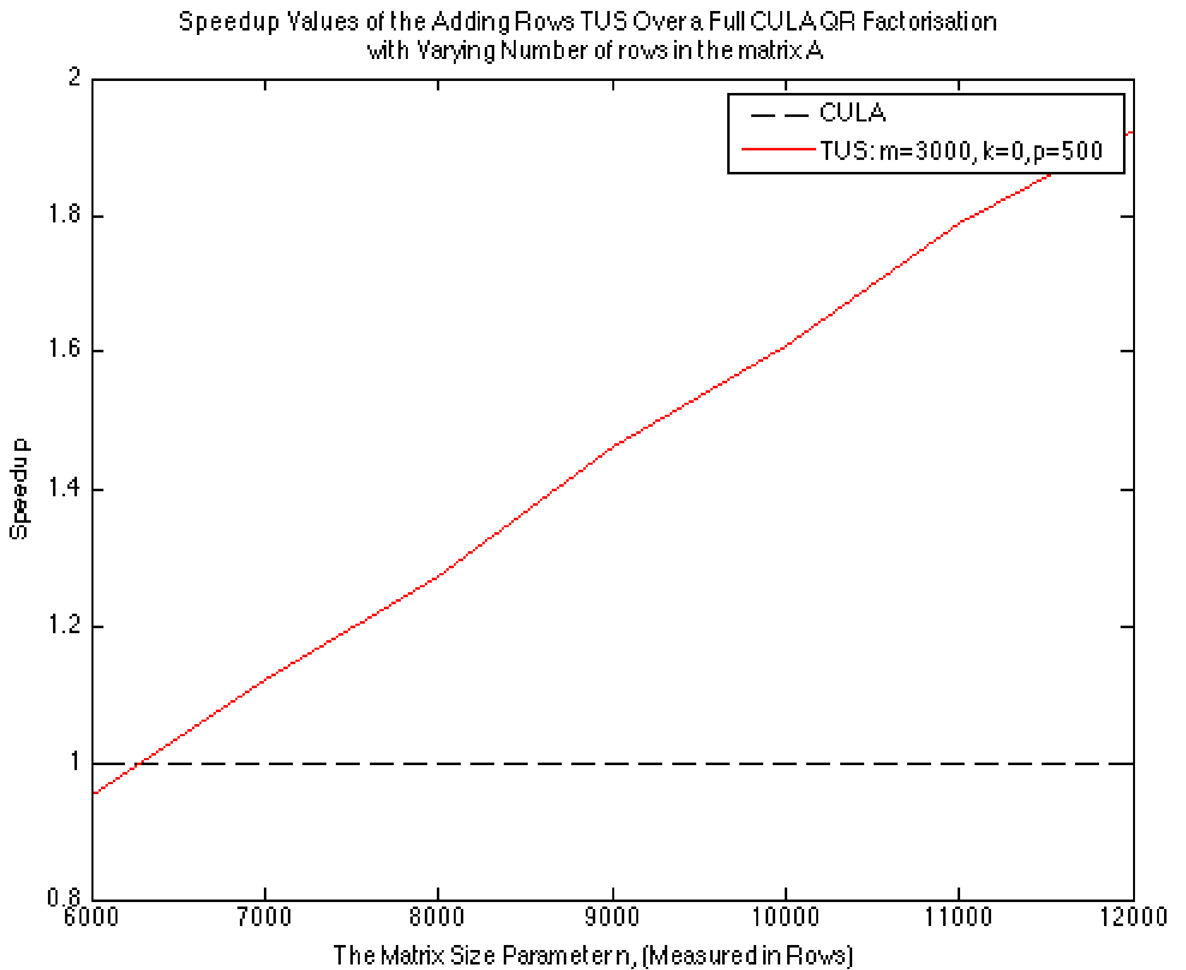

Figure 5.13: A plot of the speedup of GPU TUS for adding rows against a full QR factorisation in CULA. The plot shows the trend of speedup with varying n. Values are given In Table 5.16.

n	CULA Solve	TUS
8000	1.174	1.006
9000	1.300	0.991
10000	1.440	1.021
11000	1.583	1.015
12000	1.717	1.019
13000	1.845	0.999
14000	2.002	1.040

Table 5.16: Runtimes in seconds for GPU TUS for adding rows against a full QR factorisation in CULA for parameters $m = 3000, k = 0, p = 500$. The plot corresponding to these values is shown in Figure 5.13.

5.6 Removing Rows

As in the update for adding columns, updating by removing rows requires the orthogonal matrix Q as an input. First, Figure 5.17 and Table 5.20 show the improvement in runtime of the removing rows TUS using the composite matrix, as proposed in Section 4.4 and illustrated in Figure 4.5, against the application of Givens rotation kernels to the matrices Q, R, and the vector d separately using streams. The method involving streams was used previously for the adding columns update algorithm. An approximate 45% speedup is observed by using the composite matrix method and reducing 3 lower complexity kernel calls to apply the Givens rotations to one, higher complexity kernel invocation. This further demonstrates the importance of reducing the number of kernel calls to increase performance. Regardless of this performance improvement, GPU TUS still performed poorly overall compared to the full QR factorisation.

5.6.1 Complexity

The updating algorithm by removing rows is entirely implemented using Givens rotations. The fact that updating by removing rows requires Q, coupled with the fact that Givens transformations must be applied to introduce a strip of zeros of p width directly into the matrix Q, make an update by removing rows the most computationally complex of all four updates. The complexity formula is:

$$O(pn(n+m+1)) \tag{5.8}$$

compared to the complexity of a full QR factorisation of an $n \times m$ matrix with p rows removed:

$$O((n-p)m^2) \tag{5.9}$$

As can be seen from (5.8), the complexity of the removing rows update is not dependent on the update location parameter k. This is due to the fact that all Givens transformations must be applied to the entire length n of the matrix Q and the entire width m of the matrix R, regardless of the value of k, this is discussed in Section 4.4.

5.6.2 Removing Rows GPU TUT vs Sequential Update

First of all, the GPU TUT by removing rows gains extremely large speedups over the sequential update. This is largely due to the high complexity of the algorithm causing runtimes of the sequential update to become extremely large for fairly small problem sizes. We can see from Table 5.17 that the runtime of the sequential update increases linearly with p as expected from (5.8), but at a much higher rate than the GPU implementation which is evidence that the algorithm parallelises well. However with increased n, even though both algorithms experience a quadratic increase in runtime and speedup of GPU TUT over the sequential algorithm tends to increase with n, we see that speedup slows its increase with higher n values by Figure 5.14. This could be due to the fact that the number of kernels invoked per GPU update increases with n and the maximum number of Givens rotations applied per kernel is capped by p, as shown in Figure 3.1. No further tests were done involving sequential

updates due to the large execution times and poor performance for all parameters compared to the GPU implementation.

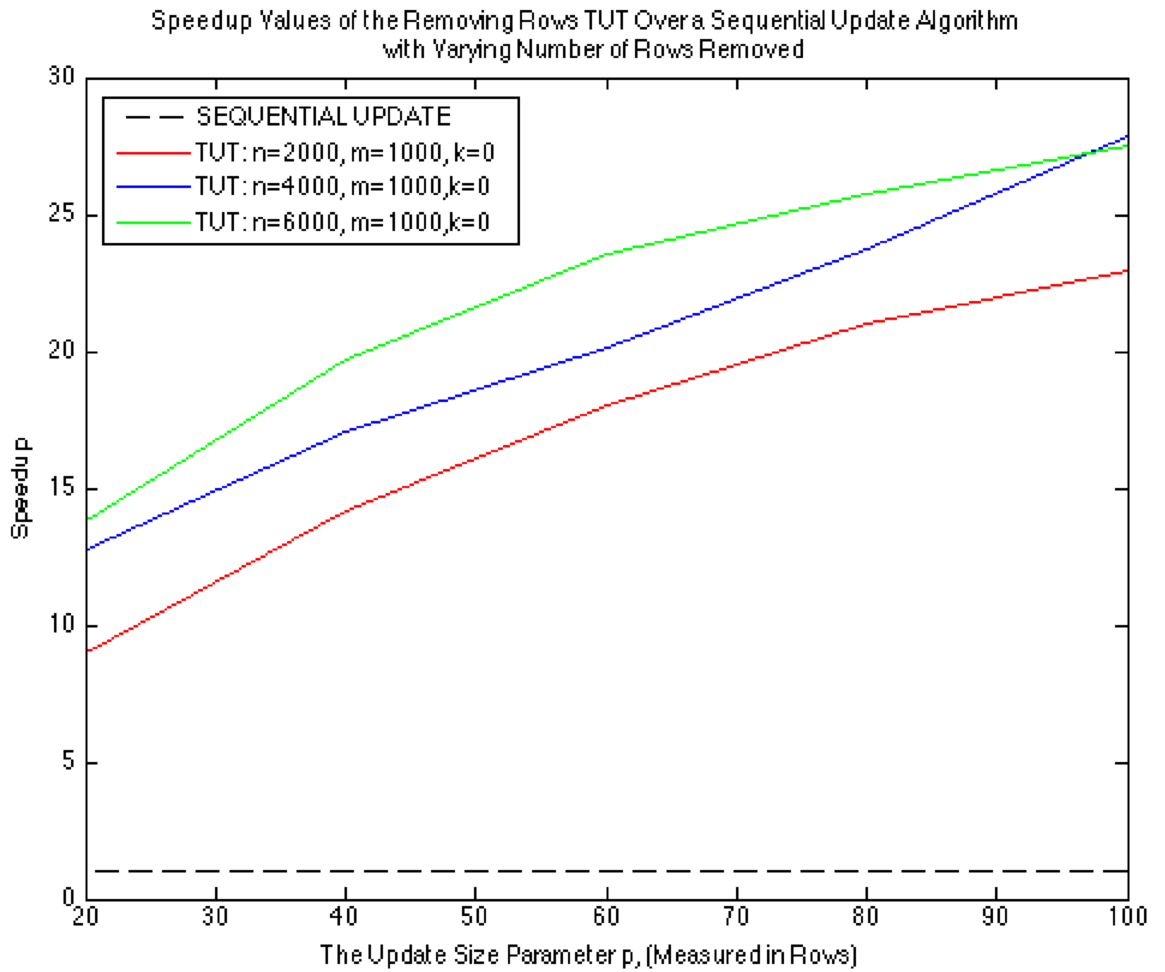

Figure 5.14: A plot of the speedup of GPU TUT for removing rows against the sequential update for $m = 1000, k = 0$. Each line exhibits a different n value and each line shows the trend of speedup of of GPU TUT over the sequential update with varying p. Values are given in Table 5.17.

	n = 2000		n = 4000		n = 6000	
p	Sequential Update	TUT	Sequential Update	TUT	Sequential Update	TUT
20	1.954	0.218	6.605	0.519	12.439	0.904
40	3.850	0.271	11.296	0.663	24.538	1.245
60	5.740	0.319	16.847	0.838	36.669	1.557
80	7.642	0.363	23.769	1.001	48.711	1.889
100	9.483	0.414	32.244	1.158	60.760	2.213

Table 5.17: Runtimes in seconds of GPU TUT for removing rows and the sequential update for parameters $m = 1000, k = 0$. A plot of speedups is shown in Figure 5.14.

5.6.3 Removing Rows GPU TUS vs CULA Solve

Even though the GPU removing rows updating algorithm performs well compared to the sequential version of the same algorithm, GPU TUS performs poorly compared to the full CULA QR factorisation and solve. The trends in speedup values of the runtime of GPU TUS against the CULA solve with varying p, n and m parameter values are illustrated in Figure 5.15 and Figure 5.16. The general trend with increasing n and p illustrated in both plots is a marked decrease in performance of the GPU TUS algorithm. This observation is further evidence towards the conclusion that the performance issues exhibited by this implementation of Givens rotations on the GPU is due to kernel invocation overhead as, in big-O notation, the number of kernel invocations within GPU TUS is:

$$O(n + p) \tag{5.10}$$

kernels. The trend of decreased efficiency in GPU TUS with n however is not quite as severe as is suggested by the pn^2 complexity from (5.8) and the conclusion drawn previously from (5.10), with only a 10% decrease in relative speedup in response to a 100% increase in the n parameter from 8000 to 16000 in Figure 5.16. Referring to the Table 5.19 however it is shown that the runtime of full CULA QR factorisation more than doubles between matrix height $n = 8000$ and $n = 16000$ as by (5.9), complexity of a full QR factorisation increases linearly with n. This, paired with the fact that complexity of full QR decreases with increased p, also by (5.9), means that the slowdown rate of GPU TUS over full CULA QR factorisation is more sensitive to increased p than to increased n.

The effect of m is illustrated in Figure 5.15 and, with m close to n, runtimes of GPU TUS that are faster than the full QR factorisation are observed. Complexity of the full QR factorisation increases with m quadratically by (5.9) and illustrated as runtimes in Table 5.18. The complexity added by increased m in TUS on the other hand is linear and the corresponding m operations are executed within a larger kernel call, this results in an overall increase in speedup of TUS relative to full CULA QR with increased m.

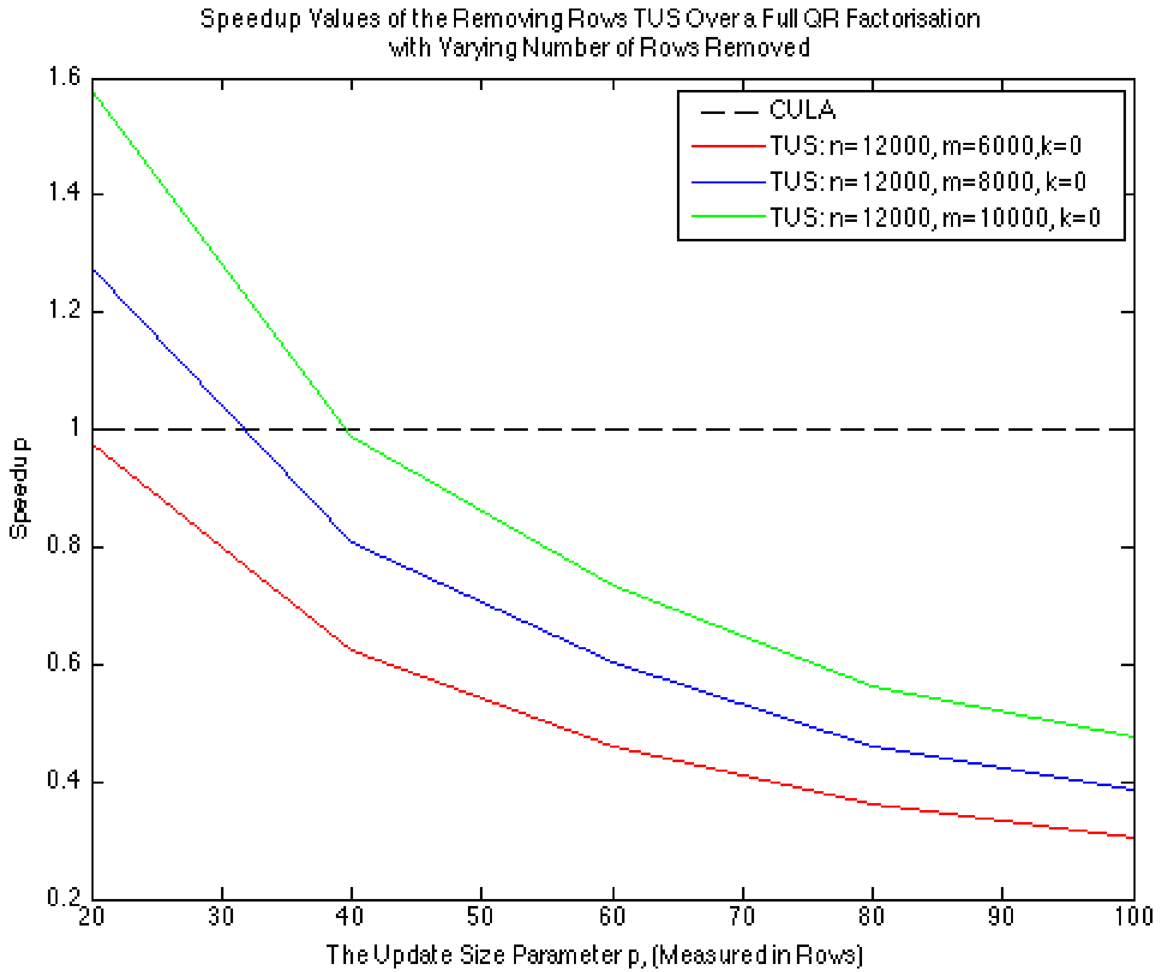

Figure 5.15: A plot of the speedup of GPU TUS for removing rows against a full GPU CULA QR factorisation. Each line exhibits a different m value and each line shows the trend of speedup with varying p. Values are given in Table 5.18.

	m = 6000		m = 8000		m = 10000	
p	CULA Solve	TUS	CULA Solve	TUS	CULA Solve	TUS
20	2.999	3.068	4.240	3.325	5.680	3.592
40	2.895	4.646	4.185	5.189	5.563	5.631
60	2.963	6.417	4.202	6.989	5.631	7.674
80	2.921	8.111	4.088	8.851	5.435	9.679
100	2.954	9.752	4.144	10.706	5.575	11.742

Table 5.18: Runtimes in seconds of GPU TUS for removing rows and a full CULA QR factorisation for parameters $n = 12000, k = 0$. A plot of speedups is shown in Figure 5.15.

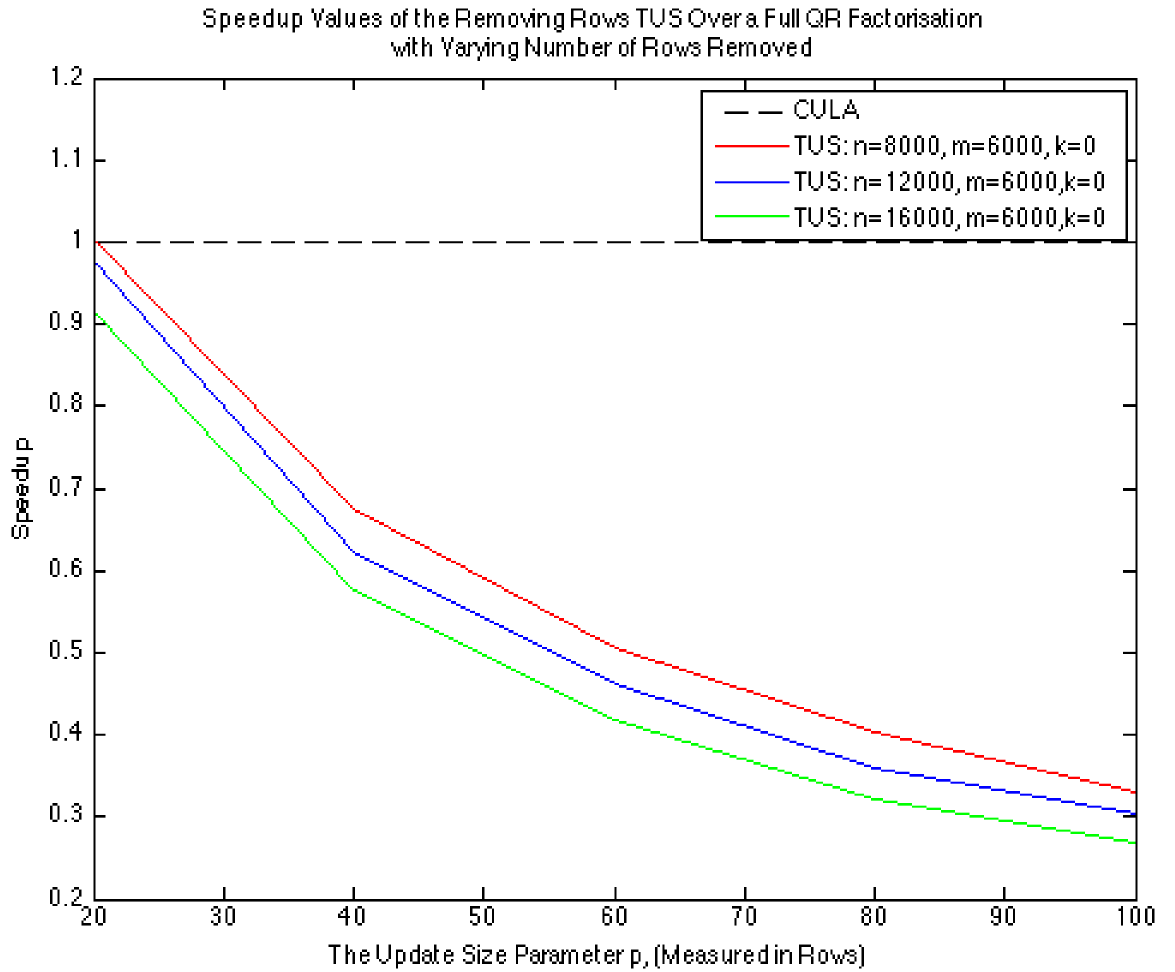

Figure 5.16: A plot of the speedup of GPU TUS for removing rows against a full GPU CULA QR factorisation. Each line exhibits a different n value and each line shows the trend of speedup with varying p. Values are given in Table 5.19.

	n = 8000		n = 12000		n = 16000	
p	CULA Solve	TUS	CULA Solve	TUS	CULA Solve	TUS
20	1.745	1.736	2.999	3.068	4.197	4.587
40	1.775	2.635	2.895	4.646	4.176	7.266
60	1.773	3.501	2.963	6.417	4.205	10.064
80	1.757	4.371	2.921	8.111	4.116	12.773
100	1.713	5.181	2.954	9.752	4.116	15.335

Table 5.19: Runtimes in seconds of GPU TUS for removing rows and a full CULA QR factorisation for parameters $m = 6000, k = 0$. A plot of speedups is shown in Figure 5.16.

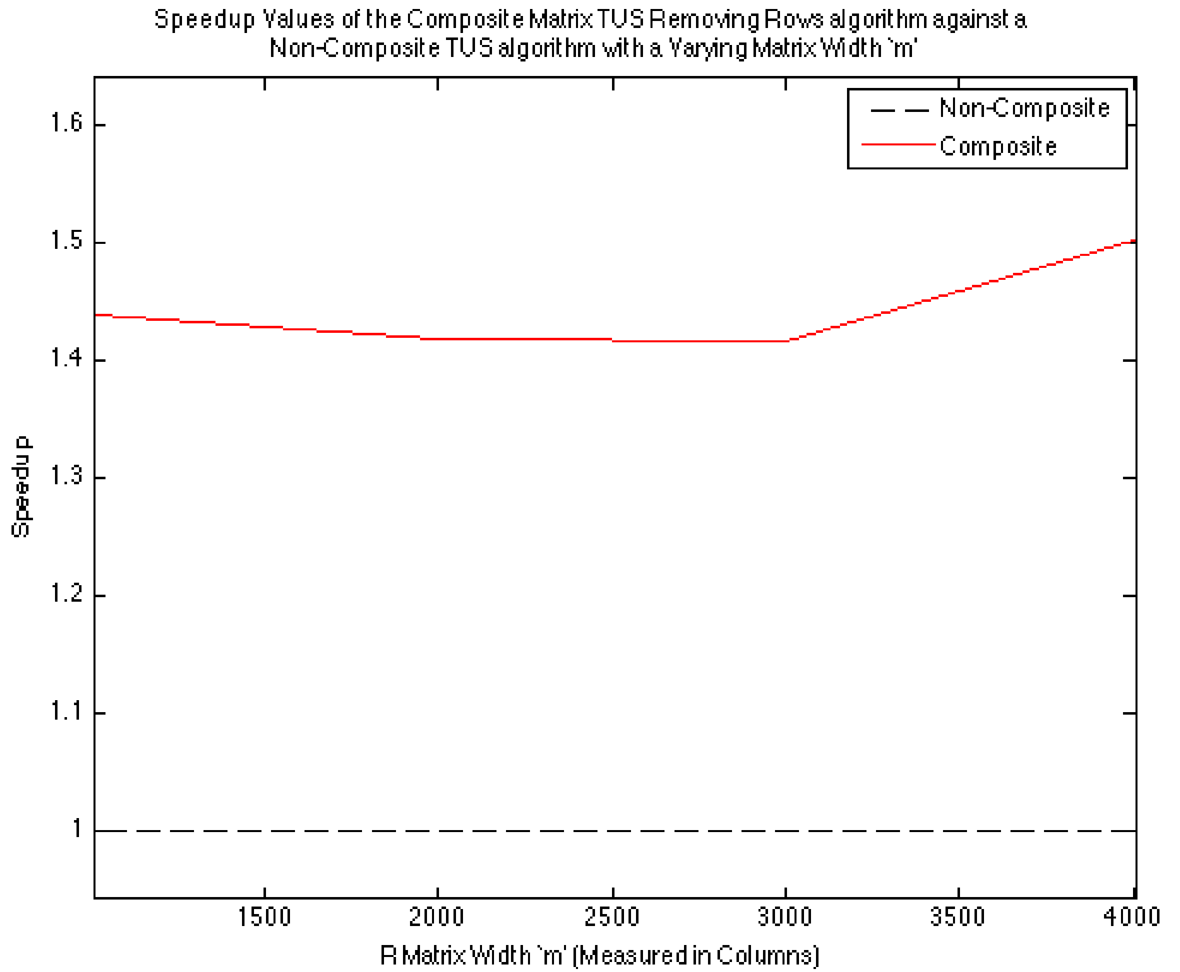

Figure 5.17: Speedup of a GPU TUS removing rows update algorithm using a composite matrix over a GPU TUS removing rows update algorithm which makes use of CUDA streams as opposed to a composite matrix, see Figure 4.5. Values are given in Table 5.20.

m	composite	non-composite
1000	1.001	1.441
2000	1.116	1.583
3000	1.244	1.762
4000	1.363	2.048

Table 5.20: Runtimes in seconds for TUS where $n = 5000, k = 0, p = 50$. The plot is in Figure 5.17.

5.7 Stability

Throughout all of the experiments carried out for this project, various checks for correctness and stability were carried out after timing had ceased, against the output of a model QR factorisation calculated via CULA. The first of the correctness tests is the comparison of the least squares solution calculated from the output of the given algorithm x, to a model least squares solution $\hat{x}$ by:

$$\frac{||x - \hat{x}||_2}{||x||_2} = e_x \qquad (5.11)$$

which is the relative error of the least squares solution after an update. The second test outputs the least squares residual of both the reference solution $d = Q^T b$ and the solution following the update $\tilde{d}$:

$$||d(m+1:n)||_2 = r \text{ and } ||\tilde{d}(m+1:n)||_2 = \tilde{r} \qquad (5.12)$$

which are compared visually to ensure correctness of the update. Some of the e_x values calculated as in (5.11) for GPU and sequential implementations of all four update algorithms are shown in Table 5.21 and Table 5.22. All relative errors presented in this table are approximately of the order of the unit round off for single-precision floating point accuracy, this suggests the algorithms implemented in this project are accurate for the application they were intended.

	Adding Rows		Removing Columns	
p	TUT	Sequential Update	TUT	Sequential Update
100	0.000002	0.000003	0.000003	0.000003
300	0.000002	0.000002	0.000003	0.000003
500	0.000002	0.000003	0.000002	0.000002
700	0.000001	0.000003	0.000002	0.000002
900	0.000003	0.000003	0.000002	0.000002

Table 5.21: Table of (5.11) values $n = 4000, m = 2000, k = 0$.

	Removing Rows		Adding Columns	
p	TUT	Sequential Update	TUT	Sequential Update
100	0.000005	0.000004	0.000003	0.000003
300	0.000004	0.000004	0.000005	0.000005
500	0.000005	0.000004	0.000006	0.000006
700	0.000006	0.000005	0.000006	0.000006
900	0.000006	0.000005	0.000007	0.000008

Table 5.22: Table of (5.11) values $n = 4000, m = 2000, k = 0$.

For all updating algorithms that form Q during their operation, namely removing rows and adding columns, two further accuracy tests were implemented. The first of which is an orthogonality test of the matrix $\tilde{Q}$ via:

$$||\tilde{Q}^T \tilde{Q} - I||_2 = e_Q \qquad (5.13)$$

and values for some tests are given in Table 5.23. Unfortunately, the error values in this case are not of the order of the unit roundoff. As the GPU based errors are comparable to the sequential ones, the

errors are accounted for as they are inherent to compounding multiple transformations into a single matrix. The second error check involving Q is the normwise relative backwards error:

$$\frac{||\tilde{Q}\tilde{R} - \tilde{A}||_2}{||A||_2} = e_A \tag{5.14}$$

and the values are given in Table 5.24. The errors again are larger than the unit roundoff but again are comparable between the sequential and GPU implementations so are acceptable. Note that there is a general increase of error with the value of p, this is due to the fact that the number of transformations applied during an update increase with p.

	Removing Rows		Adding Columns	
p	TUT	Sequential Update	TUT	Sequential Update
100	0.000234	0.000162	0.000242	0.000168
300	0.000339	0.000176	0.000367	0.000467
500	0.000364	0.000185	0.000500	0.000600
700	0.000389	0.000190	0.000479	0.000229
900	0.000470	0.000193	0.000564	0.000246

Table 5.23: Table of (5.13) values $n = 4000, m = 2000, k = 0$.

	Removing Rows		Adding Columns	
p	TUT	Sequential Update	TUT	Sequential Update
100	0.000139	0.000074	0.000095	0.000050
300	0.000374	0.000187	0.000156	0.000061
500	0.000581	0.000304	0.000202	0.000069
700	0.000733	0.000412	0.000234	0.000077
900	0.000902	0.000500	0.000251	0.000083

Table 5.24: Table of (5.14) values $n = 4000, m = 2000, k = 0$.

Chapter 6

Conclusion

The aim of this project was to accelerate updating algorithms presented in [4] on the GPU using a popular and widely used massively parallel GPU programming interface, CUDA. The challenge in this case was to develop QR updating algorithms on the GPU that achieve speedups over both QR updating algorithms executed sequentially on the CPU, and QR factorisations computed from scratch on the GPU. To achieve this end, a practical knowledge of the underlying architecture of the GPU as well as an awareness of the subtleties of the software interface was required.

The main components of the QR updating algorithms that were implemented in this project are Householder reflections and Givens rotations. In order to efficiently implement the updating algorithms on the GPU, we were first required to efficiently implement GPU Givens and Householder methods.

A full description of the method of application of Householder reflections used in this project is described in Section 3.1. The application of Householder reflections in their simplest form requires many BLAS level 2 operations. As BLAS level 2 operations do not achieve the highest throughput on the GPU, an implementation of a BLAS level 3 variant of the standard algorithm was found, as implemented in [5], namely a blocked Householder factorisation. All BLAS subroutines were implemented in a CUDA BLAS library, CUBLAS. To avoid expensive data transfer between the GPU and host, Householder vectors were calculated within a kernel.

A full description of the Givens rotation method used in this project on the other hand is given in Section 3.2. A possible method of parallelising the application of Givens rotations was given in [2]. Though this paper was is aimed at parallelising in an explicit message passing environment, the analysis of dependencies between Givens rotations was relevant. The main adjustment in the algorithm presented in [2] was to reduce the granularity of the parallelism to maximise the number of independent Givens transformations executed per kernel. This method is explained graphically in Figure 3.1, Figure 3.2, and Figure 3.3.

These developed methods to apply Givens and Householder transformations in parallel were then applied to attempt to efficiently parallelise each of the four updating algorithms in turn, adding columns, removing columns, adding rows, and removing rows. Runtime results were reported against both a sequential implementation of the respective algorithm and a full GPU QR factorisation calculated from scratch. The first experiment for each update involved a timed memory transfer to the GPU,

update, and subsequent memory transfer back, (or TUT) specifically for comparison with a sequential update. The second experiment involved again a timed transfer and update, but this time, the update is followed by a back-substitution solve on the GPU, (or TUS) specifically for comparison with a full QR factorisation and subsequent least squares solve on the GPU. Trends in relative runtimes were recorded and commented upon while varying input parameters such as the matrix A dimensions n in rows and m in columns, the location of the update k within the matrix A, and finally the update size p.

QR update by adding columns in GPU TUT greatly outperformed the sequential update achieving approximately 45 times speedup observed for parameter values such as $n = 4000, m = 8000$, and $p = 200$. QR update by adding columns in GPU TUS on the other hand theoretically does not perform well in comparison to the existing full QR factorisation. Performance is worsened mainly by increasing $(m - k)$, and p shown in Figure 5.3 and Figure 5.6. This was found to be due to the Givens rotations procedure in of the adding columns algorithm, with its large complexity and high rate of kernel invocations. GPU TUS runtime was also found to be increased with n in relation to the CULA solve.

All performance problems with QR adding columns update can be in some way attributed to the involvement of the matrix Q, required by the updating algorithm, and the parallel application of Givens rotations. In practical applications however, due to the nature of the commutativity of linear equations, there is little reason for $k < m$. This is due to the fact that adding a linear combination of variables to a linear system practically has the same effect wherever in the system you add them. Also, as p represents a number of variables added to the linear system, p is in practice is often small. So by the findings in this section therefore, this GPU implementation of QR update by adding columns is an often efficient, viable alternative to a full QR factorisation for practical applications. With low p values and high k values, $n = 8000, m = 6000, p = 200$ and $k = m$, adding columns on the GPU gained observed speedups of up to 3.5 times over a full QR factorisation. It was concluded by the runtime chart of the various components of the adding columns algorithm in Figure 5.5 that the application of the Givens rotations in the algorithm was a bottleneck to performance in GPU TUT as k decreased. This was concluded to be due to the fact that the number of kernel invocations per update increases with decreased k, causing large amounts of kernel invocation overhead.

QR update by removing columns on the GPU performed very well compared to the sequential version with increased p values and decreased k. This trend is shown in Figure 5.7 and is due to the increased complexity of the algorithm at increased p and $(m-k)$ taking advantage of the instruction throughput of the GPU. At higher values of k for all values of p however we observe that GPU TUT has up to double the runtime of the sequential version, this is due to memory transfer overhead dominating algorithm runtime due to extremely low computational complexity.

QR update TUS on the other hand was observed to perform arbitrarily better than the full QR factorisation with arbitrarily high n values, shown in Figure 5.9. This is due to the amount of computations per update staying constant for increasing n. Increasing k and p produced increased speedups of TUS over the full factorisation due to a decreasing number of kernel invocations per update in both cases, shown in Figure 5.8. Speedups for the GPU update were observed for all but the smallest k values achieving as much as 13 times speedup for $k = m - p$, for $n = 6000, m = 3000$.

Also due to increasing instruction throughput on the GPU via increased complexity, the GPU QR update TUT for adding rows increases speedup over the sequential version with increased m and p, shown in Figure 5.10. GPU TUT gains observed speedups of over 40 times over the sequential update with $m = 3000, n = 8000, k = 0$, and $p = 1500$.

Complexity of the adding rows update does not vary with both k, and n, this provides a performance advantage for GPU TUS over the full QR factorisation with increasing n, as shown in Figure 5.13. There is also observed, if fairly minor, speedups for all but the larger values of m and p for GPU TUS over the full factorisation. Approximately 1.5 times speedup was observed in Figure 5.11 for values $p = 200$ and $m = 3000$, however runtimes were approximately the same for $m = 6000$ and $p = 1500$.

The GPU TUT algorithm for removing rows performs extremely well compared to the sequential version of the algorithm, with speedups of over 25 times observed for $n = 6000, m = 1000, k = 0$, and $p = 100$, by Figure 5.14. However for larger values of n and p increase in speedups seem to level out, this is possibly due to the increased number of kernel invocations per update with an increase in both n and p.

Again, as in the adding rows case, complexity and therefore runtime of the removing rows updating algorithm is invariant upon k. Following the poor performance of applying Givens factorisations in the adding columns update on the GPU, an alternative data structure was sought in an attempt to reduce the number of kernel invocations per update. A 'composite' representation of the two input matrices Q, R and the input vector d was tested, shown graphically in Figure 4.5. The results of the runtime test of the composite representation over the alternative are shown in Figure 5.17, a speedup of approximately 1.45 is observed after the change. However, even with this improvement in runtime, the GPU TUS algorithm performs poorly in general compared to the full GPU QR factorisation. Speedups decrease harshly with increases in both p and n, leading to longer runtimes of the GPU update compared to a full QR factorisation with p values as small as 40, with $n = 12000$ and $m = 6000$, from Figure 5.15. Complexity of the full factorisation however increases quadratically with the parameter m whereas the complexity of the updating algorithm increases only linearly. This produces speedup of the GPU TUS algorithm over the full factorisation for higher values of m, as observed in Figure 5.15, with a speedup value of approximately 1.6 for $p = 20$, $n = 1200$ and $m = 1000$.

In conclusion, the GPU updating algorithms implemented in this project outperform their respective sequential implementations for almost all update sizes, and dimensions of the overdetermined system. However, the GPU updating algorithms only outperform a full GPU QR factorisation for certain values of the input parameters k, p, m and n. This is especially relevant when a large number of kernel invocations are required, such the application of Givens transformations.

6.1 Future Work

Most of the performance issues presented in the investigation were linked to the high frequency of kernel invocations and the overhead that this incurs. A future area of work could therefore be to devise methods for decreasing the number of kernel invocations within an update procedure. A possible method for reducing the number of kernels invoked in applying Givens rotations for example, would be to increase the strip size to a number of rows greater than 2. It might then be possible to apply multiple dependent rotations per thread block by looping and synchronisation statements within the kernel code. Alternatively, a development currently specific only to the Nvidia Kepler architecture called 'dynamic parallelism' is introduced in [9]. It would be worth investigating whether this could be the answer to the problem of large kernel invocation overhead as kernels can be spawned dynamically on the GPU in volume, without involving the CPU.

Due to the exhibited difficulties in parallelising the application of Givens rotations, and the effectiveness of GPU Householder reflections, it may be worth investigating the use of Householder reflections throughout an adding column update despite the prospect of the increased work created by non-zero fill-in.

Bibliography

[1] E. Anderson, Z. Bai, C. Bischof, S. Blackford, J. Demmel, J. Dongarra, J. Du Croz, A. Greenbaum, S. Hammarling, A. McKenney, and D. Sorensen. *LAPACK Users' Guide*. Society for Industrial and Applied Mathematics, Philadelphia, PA, third edition, 1999.

[2] Omer Egecioglu and Ashok Srinivasan. Givens and Householder reductions for linear least squares on a cluster of workstations. Technical report, University of California at Santa Barbara, Santa Barbara, CA, USA, 1995.

[3] Gene H. Golub and Charles F. Van Loan. *Matrix Computations*. The John Hopkins University Press, 1996.

[4] Sven Hammarling and Craig Lucas. Updating the QR factorization and the least squares problem. Technical report, University of Manchester, Manchester, UK, 2008.

[5] Andrew Kerr, Dan Campbell, and Mark Richards. QR decomposition on GPUs. In *Proceedings of 2nd Workshop on General Purpose Processing on Graphics Processing Units*, GPGPU-2, pages 71–78, New York, NY, USA, 2009. ACM.

[6] Nvidia. *Nvidia's Next Generation CUDA Compute Architecture: Fermi*, 2009. http://www.nvidia.com/content/PDF/fermi_white_papers/NVIDIA_Fermi_Compute_Architecture_Whitepaper.pdf.

[7] Nvidia. *TESLA M2050/M2070 GPU Computing Module*, 2010. http://www.nvidia.com/docs/IO/43395/NV_DS_Tesla_M2050_M2070_Apr10_LowRes.pdf.

[8] Nvidia. *CUDA Toolkit 4.2 CUBLAS Library*, 2012. http://developer.download.nvidia.com/compute/DevZone/docs/html/CUDALibraries/doc/CUBLAS_Library.pdf.

[9] Nvidia. *Dynamic Parallelism in CUDA*, 2012. http://developer.download.nvidia.com/assets/cuda/docs/TechBrief_Dynamic_Parallelism_in_CUDA_v2.pdf.

[10] Nvidia. *Nvidia CUDA C Programming Guide*, 2012. http://developer.download.nvidia.com/compute/DevZone/docs/html/C/doc/CUDA_C_Programming_Guide.pdf.

[11] David A. Patterson and John L. Hennessy. *Computer Organization and Design: The Hardware/Software Interface*. Elsevier, Inc., 2012.

[12] David Poole. *Linear Algebra: A Modern Introduction.* Thomson Brooks/Cole, 2006.

[13] Greg Ruetsch and Paulius Micikevicius. Optimizing matrix transpose in CUDA. Technical report, Nvidia, 2009.
http://www.cs.colostate.edu/~cs675/MatrixTranspose.pdf.

[14] R. Clint Whaley, Antoine Petitet, and Jack J. Dongarra. Automated empirical optimization of software and the ATLAS project. *Parallel Computing*, 27(1–2):3–35, 2001. Also available as University of Tennessee LAPACK Working Note #147, UT-CS-00-448, 2000 (`www.netlib.org/lapack/lawns/lawn147.ps`).