Implementing QR Factorization Updating Algorithms on GPUs

Andrew, Robert and Dingle, Nicholas J.

2012

MIMS EPrint: **2012.114**

ISSN 1749-9097

Implementing QR Factorization Updating Algorithms on GPUs

Robert Andrew[a], Nicholas Dingle[a,*]

[a]*School of Mathematics, University of Manchester, Oxford Road, M13 9PL*

Abstract

Linear least squares problems are commonly solved by QR factorization. When multiple solutions have to be computed with only minor changes in the underlying data, knowledge of the difference between the old data set and the new one can be used to update an existing factorization at reduced computational cost. This paper investigates the viability of implementing QR updating algorithms on GPUs. We demonstrate that GPU-based updating for removing columns achieves speed-ups of up to 13.5x compared with full GPU QR factorization. Other updates achieve speed-ups under certain conditions, and we characterize what these conditions are.

Keywords: QR factorization, QR updating, GPGPU computing

1. Introduction

There are many methods for solving overdetermined linear systems in the least square sense, but the one with which this paper is concerned is QR factorization. QR factorizations are computationally expensive, but when many need to be calculated with small adjustments in the underlying data some of this cost can be amortized by updating the factorizations.

There are four types of updates to the data matrix: adding a block of columns, removing a block of columns, adding a block of rows, and removing a block of rows. Adding and removing columns from the data matrix corresponds respectively to adding and removing variables from the overdetermined system, while adding and removing rows corresponds to adding and removing equations.

In this paper we aim to accelerate the updating algorithms originally presented in [1] by implementing them on a GPU using CUDA. The original paper demonstrates that these algorithms outperform full QR factorization in a serial environment, and our own prior work shows that their implementation on a GPU correspondingly outperforms a serial implementation by a wide

*Corresponding author

Email addresses: `robert.andrew@postgrad.manchester.ac.uk` (Robert Andrew), `nicholas.dingle@manchester.ac.uk` (Nicholas Dingle) margin [2]. Other papers have investigated implementing full QR factorization on GPUs, for example by using blocked Householder transformations [3] or a tile-based approach across multicore CPUs and multiple GPUs [4, 5]. Another study [6] achieved speed-ups of 13x over the CULA library [7, 8] for tall-and-skinny matrices by applying communication-avoiding QR. There is less work on implementing updating algorithms in general, although [9] does investigate parallel out-of-core updating of QR factorizations for least squares problems when adding extra rows. To the best of our knowledge there has been no prior work that attempts to implement all four updating algorithms on GPUs.

QR factorization decomposes the $n \times m$ matrix A into the $n \times n$ orthogonal matrix Q and the $n \times m$ upper-trapezoidal matrix R. Matrix updates 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 A is given as an offset, k, in columns or rows from the top left corner. The updated matrix is denoted $\tilde{A}$, with the corresponding updated factorization $\tilde{Q}\tilde{R}$.

Section 2 presents background material on the least squares problem, QR factorization and updating procedures. Section 3 details the implementation of Givens and Householder transforms on the GPU, then Section 4 describes their use in the four updating algorithms introduced in Section 2. Section 5 presents the results of parallelizing the QR updating algorithms. We compare run-times of the implemented algorithms to an existing full QR factorization routine implemented by CULA. We also investigate the accuracy and stability of the updating algorithms. Section 6 concludes and discusses future work.

2. Background

We describe the least squares problem and Householder and Givens methods of QR factorization. We also introduce the four QR factorization updating algorithms from [1].

2.1. QR Factorization and the Least Squares Problem

In a least squares problem we wish to find a vector x such that:

$$\min_x ||Ax - b||_2$$

where A is an $n \times m$ matrix of input coefficients with $n \geq m$, and b is a vector of observations. To do this we can use the QR factorization of A:

$$\begin{aligned} ||Ax - b||_2 = ||Q^T Ax - Q^T b||_2^2 = ||Rx - d||_2^2 &= \left|\left| \begin{bmatrix} R_1 \\ 0 \end{bmatrix} x - \begin{bmatrix} f \\ g \end{bmatrix} \right|\right|_2^2 \\ &= ||R_1 x - f||_2^2 + ||g||_2^2 \end{aligned}$$

where R_1 is upper triangular and $d = Q^T b$. $||R_1 x - f||_2^2 = 0$ can be solved by back substitution, leaving $||g||_2^2$ as the minimum residual or error [10].

2.2. QR Factorizations

Givens and Householder transformations are two of the most widely used methods for computing QR factorizations.

2.2.1. Householder Transformations

A Householder transformation is an orthogonal matrix:

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

P can be chosen for a vector x so that $Px = ||x||e_1$, where e_1 is a vector of zeros with a 1 as the first element and $||\cdot||$ is the vector 2-norm. This means that we can use Householder vectors to transform A into R by progressively zeroing A's sub-diagonal elements starting from its first column. If A_i is A with the first $i-1$ columns so transformed, we can transform the i^{th} column by computing a Householder matrix P_i and embedding it in an $n \times n$ matrix:

$$H_i = \begin{bmatrix} I_{(i-1)} & 0 \\ 0 & P_i \end{bmatrix}$$

The factorization then proceeds by forming $A_{i+1} = H_i A_i$ [11].

2.2.2. Givens Rotations

A Givens rotation is a matrix of the form

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

When 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 Householder transformations, Givens rotations introduce just one zero each. To apply a Givens transformation to the i^{th} and $(i+1)^{th}$ elements within a column of A, we embed a Givens matrix within an identity matrix, overwriting the i^{th} to $(i+1)^{th}$ elements in the i^{th} 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.

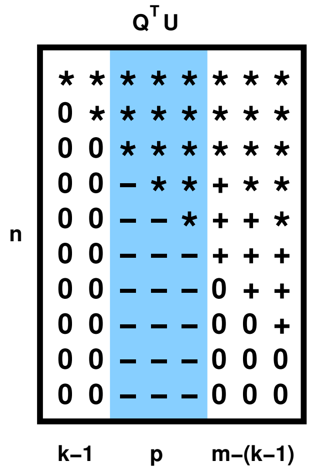

Figure 1: A 10×5 matrix R in which the $p = 3$ columns inserted before column $k = 3$ are shaded in blue. The symbol '-' denotes a nonzero which must be made zero, whereas '+' denotes a zero which must be made nonzero. '*' denotes a general nonzero element.

2.3. *QR Updating Algorithms*

Here we summarize the QR updating algorithms originally presented in [1].

2.3.1. *Adding Columns*

When a block of p columns, denoted U, is added to A before column k, the modified data matrix $\tilde{A}$ is:

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

Multiplying through by Q^T gives:

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

We note that $\tilde{R}(1:n, 1:k-1) = R(1:n, 1:k-1)$, as shown in Figure 1, so we need only modify Q^TU and $R(1:n, k:m)$. To achieve this we require an orthogonal matrix $X \in \mathbb{R}^{(n-k+1)\times(n-k+1)}$ such that:

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

We use Givens rotations to compute this, as using Householder transformations would result in $\tilde{R}$ being full. We can, however, use Householder transformations to reduce the submatrix $Q^TU(m+1:n, 1:p)$ to upper-trapezoidal form, before applying Givens rotations to finish the update. This use of Householder transformations is possible because the trailing submatrix $R(m+1:n, k:m)$ is guaranteed to be entirely zero.

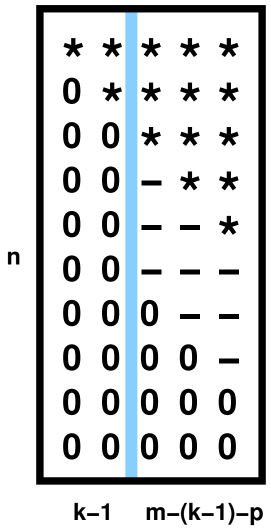

Figure 2: The 10×8 matrix R with the $p = 3$ columns removed from column $k = 3$ onwards represented by the blue line.

2.3.2. Removing Columns

When a block of p columns is deleted from A starting at column k, the modified data matrix becomes:

$$\tilde{A} = \left[\begin{array}{cc} A(1:n, 1:k-1) & A(1:n, k+p:m) \end{array} \right]$$

Multiplying through by Q^T:

$$Q^T \tilde{A} = \left[\begin{array}{cc} R(1:n, 1:k-1) & R(1:n, k+p:m) \end{array} \right]$$

We can see that $\tilde{R}(1:n, 1:k-1) = R(1:n, 1:k-1)$, as shown in Figure 2. This allows us to define $m - (k-1) - p$ Householder transformations $H_{k,k+1,\ldots,m-p}$ to reduce just the right-hand portion of R

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

2.3.3. Adding Rows

When a block of p rows, U, are added to A, the updated data matrix is:

$$\tilde{A} = \left[\begin{array}{c} A(1:(k-1), 1:m) \\ U \\ A((k):n, 1:m) \end{array} \right]$$

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

$$P\tilde{A} = \left[\begin{array}{c} A \\ U \end{array} \right]$$

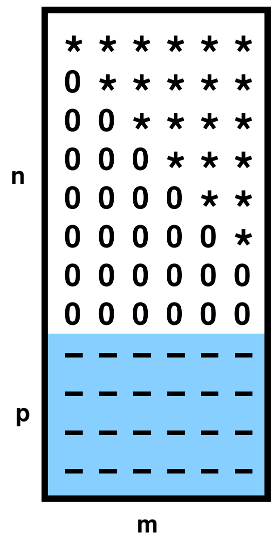

Figure 3: The 8×6 matrix R with the $p = 4$ rows added shaded in blue.

and thus

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

With m Householder transforms $H_{1,2,3,\ldots,m}$ we can form $\tilde{R}$ as shown in Figure 3:

$$H_m \ldots H_2 H_1 \begin{bmatrix} R \\ U \end{bmatrix} = \tilde{R}$$

and because $A = QR$:

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

2.3.4. Removing Rows

When a block of p rows are removed from A from row k onwards, the updated data matrix is:

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

In order to show that $\tilde{Q}$ and $\tilde{R}$ can be calculated using just Q and R from the original factorization, we must first permute the deleted rows to the top of A:

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

As shown in Figure 4, a series of Givens matrices, represented by the orthogonal matrix G, are then employed to introduce zeros directly into PQ to create

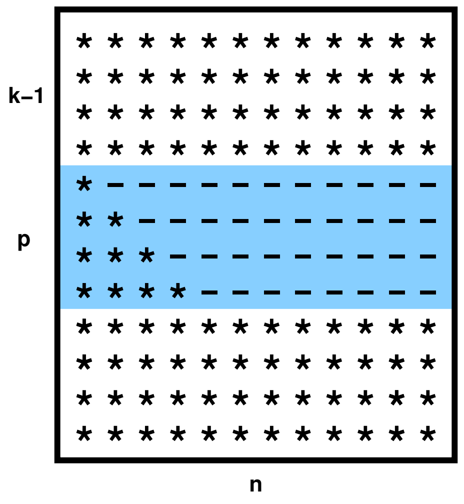

Figure 4: The 12×12 matrix Q with the $p = 4$ rows removed shaded in blue. Transformations used to introduce zeros into Q are also applied to R to produce $\tilde{R}$

$PQG = \begin{bmatrix} I & 0 \\ 0 & \tilde{Q} \end{bmatrix}$. These transformations are applied to the non-permuted R:

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

Note that again Householder transformations may not be used here because they would lead to $\tilde{R}$ being full.

3. Householder and Givens on the GPU

The updating algorithms rely on Givens and Householder transformations and therefore the efficient parallelization of these operations on the GPU is vital.

3.1. Parallelizing Householder Transformations

We implement blocked Householder QR factorization, which uses BLAS level 3 operations, to better exploit the instruction bandwidth of GPUs [3]. In a blocked QR factorization, multiple Householder transformations are combined into a single transformation matrix:

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

A block of r columns is reduced as if it were undergoing a full QR factorization, and then the composite of the block's Householder transforms is applied to the remainder of the matrix. This can be represented as [3]:

$$P = I + WY^T$$

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 transforms that comprise the block. We apply the Householder matrices to the trailing submatrix of the matrix undergoing reduction using elementary BLAS subroutines implemented in CUBLAS [12].

3.2. Parallelizing Givens Rotations

There are two main considerations in using Givens rotations: within the rows to the left of the subject elements there must be no other nonzero elements, and there must be no nonzero elements in the column below the element to be zeroed. Prior work exists on the efficient parallelization of Givens rotations on distributed and multicore systems. One such approach is described in [13]:

- Each processor is assigned a strip of rows.
- The first zero is introduced in the leftmost column of the lowest strip.
- 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.

We adapt this approach for use on GPUs, with the main difference being that we use a strip height of 2 rows to maximize the number of cores on the GPU which are doing work. This reflects the fact that GPUs prefer parallelism of a much finer grain (less work per thread, more threads) compared with CPUs.

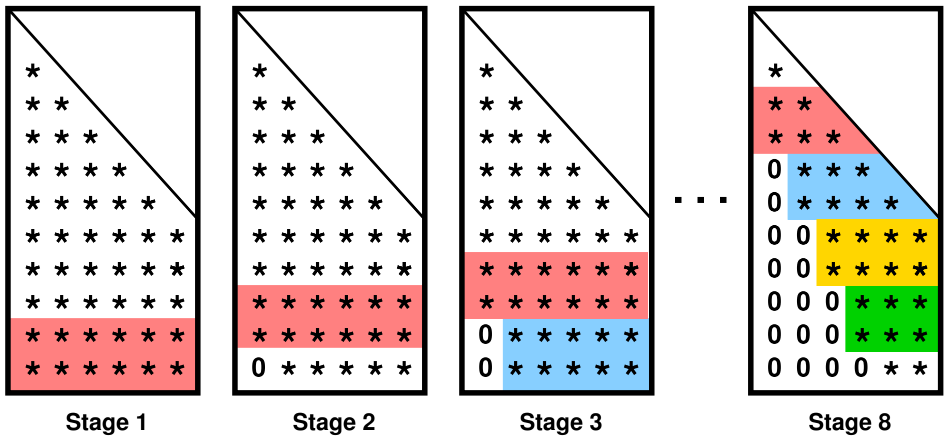

Figure 5: The application of Givens matrices on the GPU.

Figure 5 shows this modified approach in action. The stage numbers correspond to the number of kernel invocations, while the coloured areas denote different block assignments within a kernel; red denotes block id $y = 0$ within a

2-dimensional grid, blue denotes $y = 1$, yellow denotes $y = 2$, and finally green denotes $y = 3$. In addition, the matrix is also partitioned in the x direction into blocks of 128 elements. This requires the matrix to be accessed in row-major order, but as other algorithms in this paper require column-major access we implement an efficient transpose kernel based on that of [14].

Due to the fact that the matrix is partitioned along its width as well as its height, dependencies are created between blocks regarding the calculation and application of the Givens coefficients s and c. To ensure that application of the coefficients does not start before they have actually been computed, our implementation comprises two kernels: `makeGivens`, which calculates the coefficients, and `applyGivens`, which performs the actual rotations.

4. Implementation

We now describe how our Householder and Givens kernels are used to implement the four QR factorization updating algorithms summarized in Section 2.3.

4.1. Adding Columns

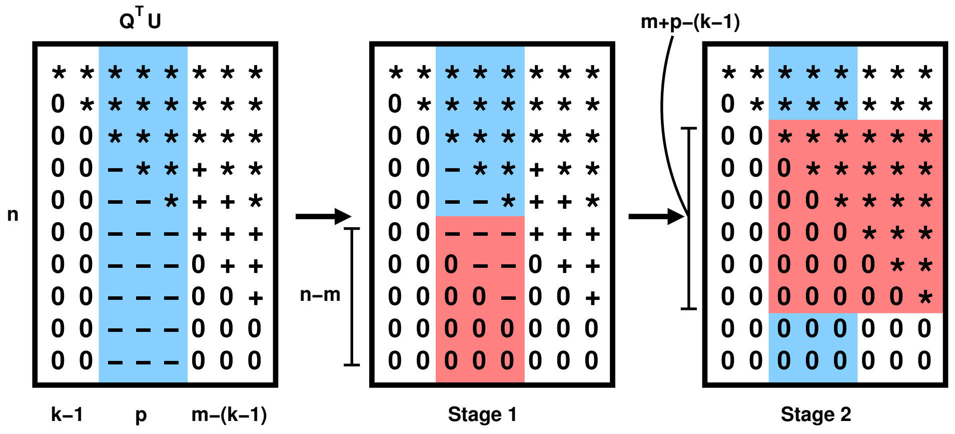

Figure 6: The adding columns update of R. The shaded blue area shows the added columns, while the shaded red area denotes the active section for each stage in the algorithm.

Figure 6 shows the stages in the reduction process for an example where $m = 5$, $n = 10$, $p = 3$ and $k = 3$. Taking Q, R, an $n \times p$ block of columns U and an index k, $0 \leq k \leq m + 1$ as input, the implementation proceeds as follows:

1. Apply Q to the added columns U to form $Q^T U$.
2. QR factorize the lower $(n - m) \times p$ block of $Q^T U$ using Householder transformations. This is the area shaded in red in Stage 1 of Figure 6.
3. If $k = m + 1$ then the update is complete.
4. If not, transposes the section shown in red in Stage 2 of Figure 6.

5. Apply Givens rotations to the transposed section. This can be done at the same time as Givens rotations are also applied Q and d, so separate CUDA streams can be used simultaneously for each matrix/vector.
6. Transpose back the (now reduced) red section in Stage 2 of Figure 6.

4.2. Removing Columns

Figure 7 shows the stages in the reduction process for an example where $m = 8$, $n = 10$, $p = 3$ and $k = 3$. Unlike adding columns, Q is not required for the update but R, an index k, $0 \leq k \leq m - p + 1$, and a block width n_b are. The implementation proceeds as follows:

1. If the right-most p columns were deleted, the update is complete.
2. Blocked Householder QR factorization is applied to the $(p + n_b) \times n_b$ submatrix to the right of the removed columns (Stage 1). Note that here the Householder vectors are only $p+1$ elements long and the matrices W and Y matrices need only be of dimension $(p + n_b) \times n_b$. This is repeated across submatrices to the right until R is upper triangular (Stage 2).

4.3. Adding Rows

We calculate this update using R, an index k, $0 \leq k \leq n + 1$, and a $p \times m$ block of rows U. A visualization of the reduction of a single block within an update where $m = 6$, $n = 8$, $p = 4$ is given in Figure 8. Note that the value of k does not affect the algorithm as the added rows can be permuted to the bottom of A. Adding rows also adds elements to b, and we denote these as e.

As can be seen in Stage 1 in Figure 8, a transformation is applied to a small section of the nonzero triangular part of R separately to U to avoid arithmetic involving zero. As blocked Householder transformations would apply the full W and Y matrices to both R and U, an alternative approach is to use an upper triangular square matrix T and a matrix $V = [v_1\, v_2 \ldots v_{n_b}]$ containing Householder vectors as its columns [1]. Given a set of Householder matrices, their product can be defined in terms of V and T as:

$$H_{n_b} \ldots H_2 H_1 = I - V T^T V^T$$

where n_b is the block size and:

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

T 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 is the Householder coefficient corresponding to the i^{th} Householder vector in the i^{th} column of V. As the assignment of individual τ_i elements

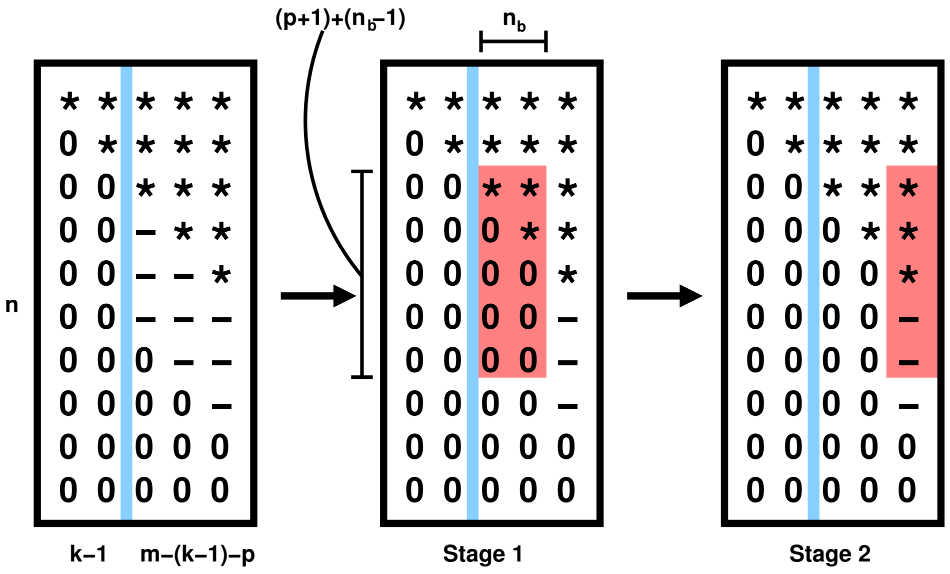

Figure 7: The reduction of one block within the removing columns update of R for block size $n_b = 2$. The shaded blue line shows where the removed columns used to be, while the shaded red area shows the active section for that stage in the algorithm.

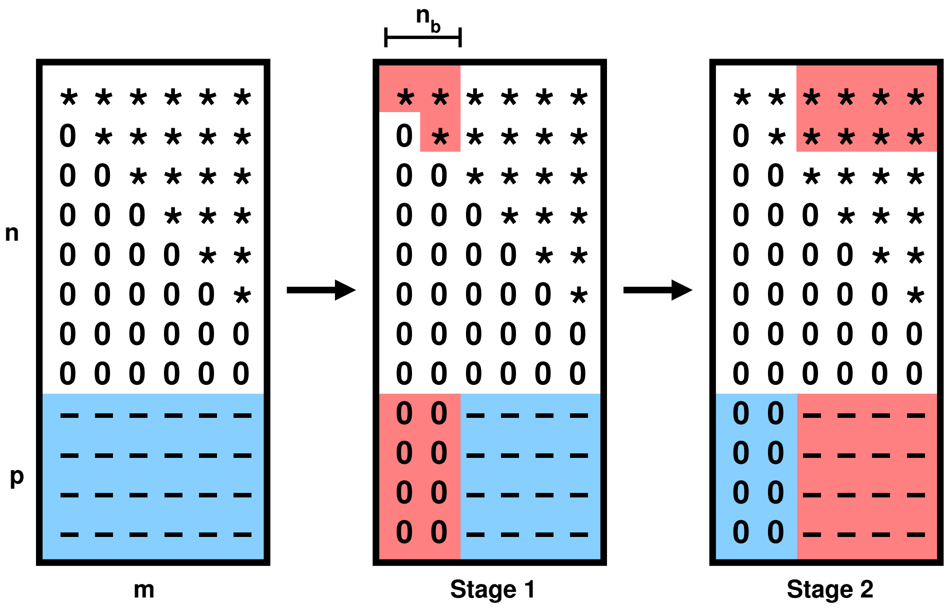

Figure 8: The reduction of one block within the adding rows update of R for block size $n_b = 2$. The shaded blue area shows the added rows, while the elements shaded in red in Stage 1 are the elements involved in reduction of a block via Householder transformations. The elements in the red area in Stage 2 are multiplied by the matrices produced in the previous stage.

along the diagonal of T is independent of the remainder of the formula for T, the entire diagonal can be assigned in parallel within a simple kernel before repeated calls of CUBLAS `gemv` are used to form the rest of T.

V and T are applied to the trailing submatrix of $\begin{bmatrix} R \\ U \end{bmatrix}$ by:

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

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

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

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 [1].

We proceed as follows for each $p \times n_b$ block of columns in the added rows:

1. Stage 1 in Figure 8: use Householder transformations to reduce the block's entries to zeros and to modify R's corresponding nonzero entries.
2. Construct T as described above.
3. Stage 2 in Figure 8: update R and b by multiplying with T and V.

This updating algorithm is largely implemented using the CUBLAS routines `gemm`, `gemv`, `ger`, `axpy`, and `copy`.

4.4. Removing Rows

We calculate this update using Q, R, an index k, $0 \leq k \leq n - p + 1$, and a block height p. In contrast to the other updating algorithms, Givens rotations are applied to Q as opposed to R. Figure 9 shows an example of the reduction process where $n = 12$, $m = 5$, $p = 4$ and $k = 5$. A strip of rows Z is identified within Q corresponding to the removed rows from A, $Z := Q(k : k+p-1, 1 : n)$; these are the rows shaded blue in Figure 9.

We first calculate Givens rotations for Z. For efficiency, the number of kernels being spawned per stage can be greatly reduced by allocating a contiguous block of memory to house the matrices Q and R along with the vector d. An `applyGivens` kernel can then be applied once to the entire composite matrix as

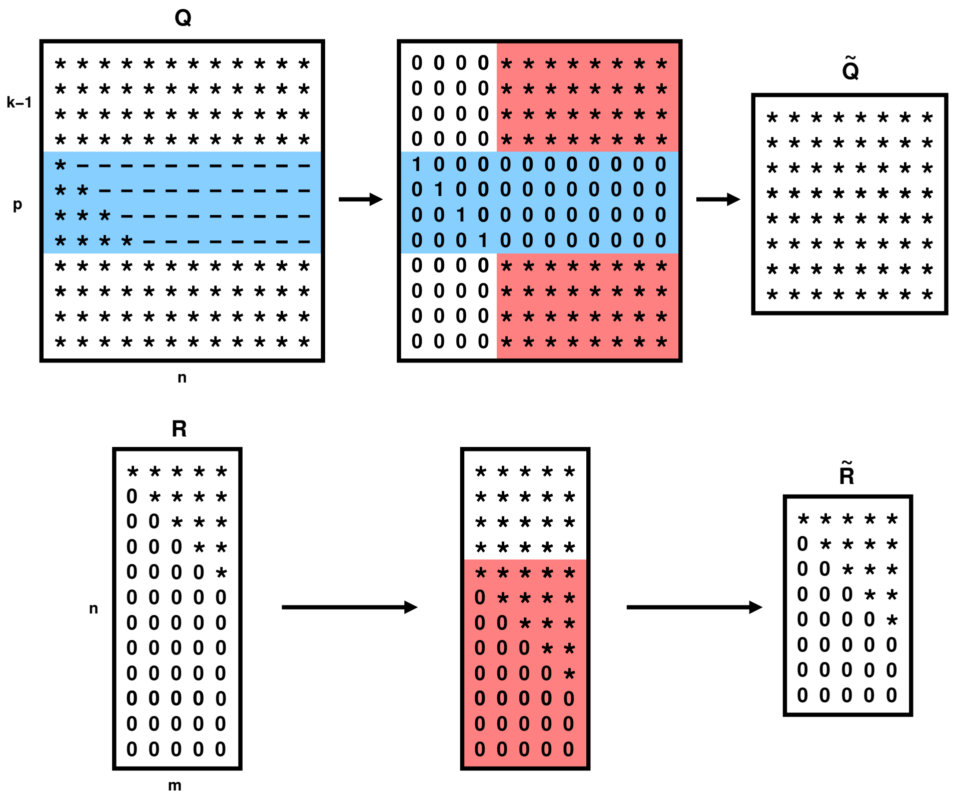

Figure 9: Removing rows in Q and R. The shaded blue area shows the rows in Q corresponding to the rows removed in A. The elements shaded in red are the elements that form $\tilde{Q}$ and $\tilde{R}$.

opposed to applying one kernel per constituent matrix. This also presents the opportunity to align memory for all accesses by allocating the composite matrix via `cudaMallocPitch`. The larger matrix also means that the thread blocks for the `applyGivens` kernel are increased in size from 128 elements to 256, which reduces the scheduling overhead and allows us to exploit shared memory for the storage of Givens coefficients.

The implementation proceeds as follows:

1. Allocate memory on the GPU for the composite matrix C, and copy Q and d from host memory into C.
2. Assign the variable Z to the strip to be zeroed by Givens transformations.
3. Transpose R to its place in C using the efficient GPU transpose procedure.
4. Apply the Givens transform algorithm described in Figure 5 to reduce Q to the form shown in the centre of the top of Figure 9, with zeros in the first p columns and in Z, and with the identity matrix embedded in the first p columns of Z.
5. Copy the reduced elements of Q and R from C into the smaller $\tilde{Q}$ and $\tilde{R}$.

5. Results

We evaluate the performance of our updating algorithms on an Nvidia Tesla M2050 (Fermi) GPU attached to a 12-core Intel Xeon E5649 2.53GHz host. All experiments are conducted in IEEE single precision arithmetic. We start measuring the run-time of our implementations, t_{update}, from when they initiate transfer of Q (for those updating algorithm that require it), R and d from host memory to the GPU. We then execute our updating algorithm and use the resulting $\tilde{R}$ and $\tilde{d}$ to solve the least squares problem via the CUBLAS back-substitution routine `cublasStrsm`. Timing stops when this CUBLAS routine returns. Q and R have been computed by a previous full QR factorization and we do not measure the time to do this.

We compare our implementation against the run-time, t_{full}, of QR-based least squares solve from the CULA library (version R14). We time the execution of the `culaDeviceSgels` routine, including the time taken to transfer $\tilde{A}$ and $\tilde{b}$ from host to GPU. Speed-up is then calculated as $\frac{t_{full}}{t_{update}}$.

The coefficients of our overdetermined systems of equations are uniformly-distributed random numbers in the interval $(-1, 1)$. All run-times are measured in seconds and all values presented are averages over 5 executions.

5.1. Choosing the Block Size Parameter

Table 1: Run-times in seconds for applying 1000 Householder transformations with different block sizes n_b.

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

Each updating algorithm that involves blocked Householder transformations (i.e. adding and removing columns and adding rows) depends on a block size n_b. The optimum value for n_b is likely to be problem-dependent, and it was not possible for us to determine this optimum for each of the test cases considered. Instead, we choose n_b for each algorithm based on the performance of a test case. Table 1 shows the times taken for each of the three algorithms to apply 1000 Householder vectors to 1000 columns, and we observe that the optimum block size lies between 50 and 100 columns per block. In all further tests we therefore pick a block size that results in ten blocks per factorization.

5.2. Adding Columns

Updating by adding a block of columns requires Q to calculate $\tilde{R}$, which means we must output $\tilde{Q}$ as well as $\tilde{R}$ to enable subsequent updates. The complexity of updating a QR factorization by adding a block of p columns is:

$$O(nmp + (n-m)p^2 + (n-m)np + (m+p-k)^2p + (m+p-k)pn)$$

The first term comes from the matrix multiplication Q^TU, while the second and third terms come from the QR factorization of the lower part of U and the subsequent application of the transforms to Q. The final two terms come from the application of Givens matrices to R and Q. The complexity of a QR full factorization with the addition of p columns is $O(n(m+p)^2)$.

Table 2: Run-times in seconds for adding $p = 200$ columns and for CULA full QR factorization, for $m = 3000$. CULA run-time is constant because k does not change the problem size.

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

Table 3: Run-times in seconds for adding $p = 200$ columns and for CULA full QR factorization, for $m = 6000$. CULA run-time is constant because k does not change the problem size.

k	$n = 8000$		$n = 12000$		$n = 16000$	
	CULA	Updating	CULA	Updating	CULA	Updating
0		8.585		11.385		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	1.797	3.720	3.003	5.423	4.241	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

As k gets smaller the complexity of updating approaches that of computing a whole new factorization. We investigate this for a range of problem sizes (n and m values). The run-times are given in Tables 2 and 3 and the corresponding speed-ups of updating relative to CULA full factorization are shown in Figures 10 and 11. We observe the expected inverse relationship between k

and run-time of the update algorithm, with only the highest values of k showing speed-up over CULA. For the updating algorithm the lower values of n exhibit larger speed-ups over the full factorization, which is possibly due to the involvement of Q in updating.

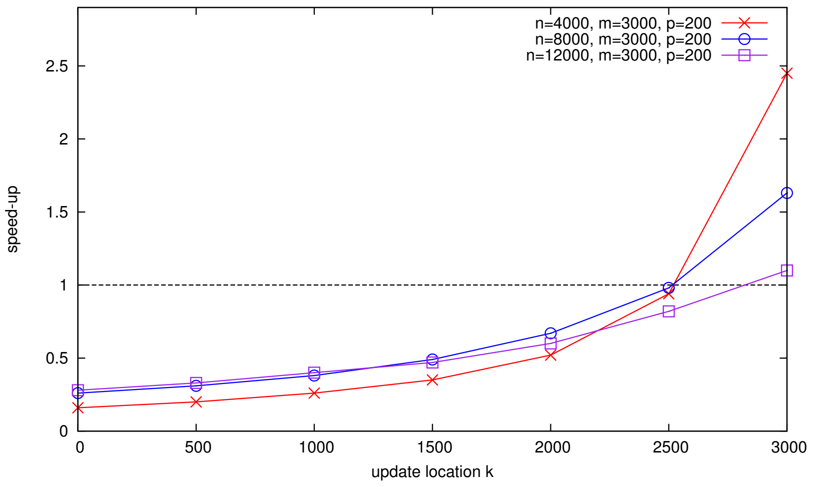

Figure 10: Speed-up of adding columns update relative to full QR factorization for $m = 3000$.

Table 4: Run-times of the individual stages of adding $p = 200$ columns update for $n = 16000$, $m = 6000$.

Algorithm Stage	$k = 500$	$k = 5500$
Memory Transfer (HOST$\rightarrow$GPU)	0.4913	0.5102
Q^TU	0.1922	0.1922
Householder Transformations	1.3950	1.3936
Transpose Procedures	0.0291	0.0006
Givens Rotations	11.0144	0.7949

Figure 11 shows the results with larger m than in Figure 10. The complexity of a full QR factorization increases with m whereas the complexity of updating increases with the difference between m and k. This is because the size of the sub-matrix to which Givens rotations must be applied (the red area in Stage 2 of Figure 6) increases with $m - k$, which means that the GPU updating algorithm performs better relative to CULA when adding columns to the end of matrix that has a large number of columns. As shown in Table 4, when k is much smaller than m (corresponding to adding columns near the beginning of the matrix), the run-time of the updating algorithm is dominated by the time required to execute the $O(m - k)$ Givens rotations kernels.

The complexity of updating also increases quadratically with p in the second term due to the Householder transformations in Stage 1 of the algorithm. As

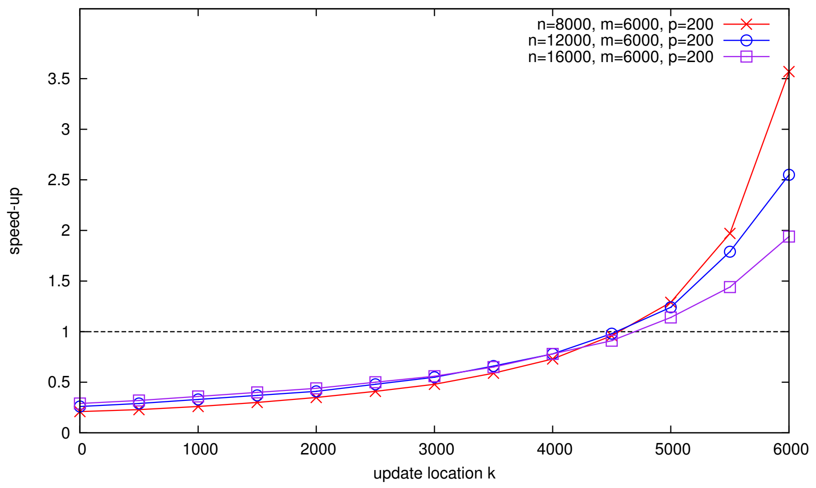

Figure 11: Speed-up of adding columns update relative to full QR factorization for $m = 6000$.

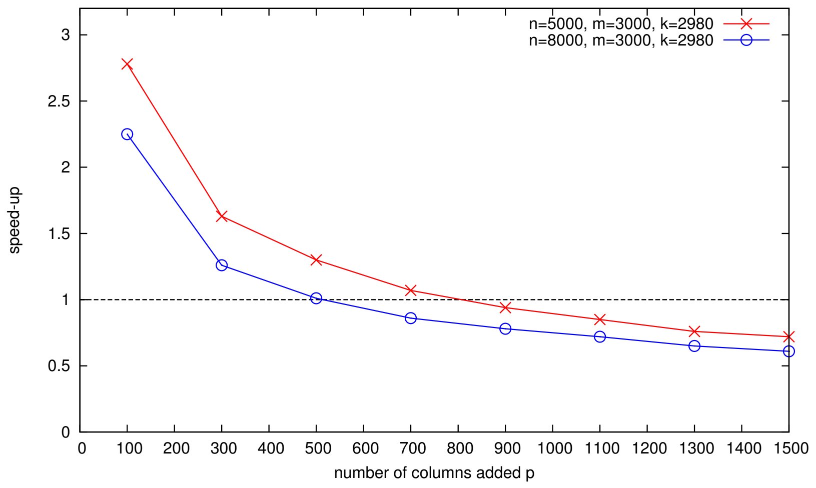

Figure 12: Speed-up of adding columns update relative to full QR factorization for $k = 2980, m = 3000$ for numbers of additional columns p.

Table 5: Run-times in seconds for adding p columns and for CULA full QR factorization, for $k = 2980$, $m = 3000$.

p	$n = 5000$		$n = 8000$	
	CULA	Updating	CULA	Updating
100	0.500	0.180	0.870	0.386
300	0.528	0.323	0.919	0.732
500	0.555	0.428	0.973	0.967
700	0.564	0.526	1.028	1.198
900	0.603	0.643	1.085	1.392
1100	0.620	0.731	1.149	1.606
1300	0.648	0.848	1.221	1.885
1500	0.683	0.954	1.230	2.027

shown in Figure 12 and Table 5, the updating algorithm only runs faster than full factorization for smaller values of p. This is expected because the Householder transformations that reduce the lower part of U (shaded red in Stage 1 of Figure 6) also have to be applied to Q. As p approaches m, therefore, the complexity of the updating algorithm approaches that of full QR factorization.

5.3. Removing Columns

Unlike the update by adding columns, removing columns can be implemented without the involvement of Q. We therefore expect that these updates will perform better than those that do involve Q. The complexity of updating a QR factorization by removing a block of p columns is $O((m - k - p)^2 p)$, compared with the complexity of $O(n(m-p)^2)$ for a full QR factorization of an $n \times m$ matrix with p columns removed. Note that the complexity of an update via removing columns is not dependent on the matrix height n, whereas the complexity of a full QR factorization is. We therefore expect that, for a sufficiently large n, our updating algorithm will outperform a full GPU QR factorization. Our experiments confirm this: as shown in Table 6, the run-time of the CULA solve increases with n, while that of our updating algorithm remains essentially constant. The corresponding speed-up is plotted in Figure 13.

Table 6: Run-times in seconds for removing $p = 500$ columns and for CULA full QR factorization, for $m = 3000$, $k = 0$.

n	CULA	Updating
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

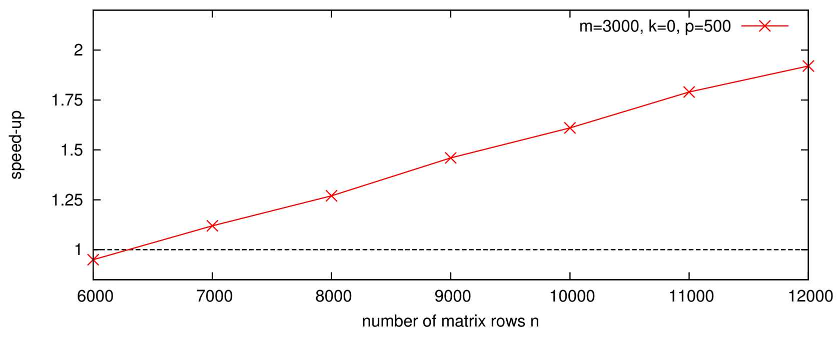

Figure 13: Speed-up of removing columns update relative to full QR factorization.

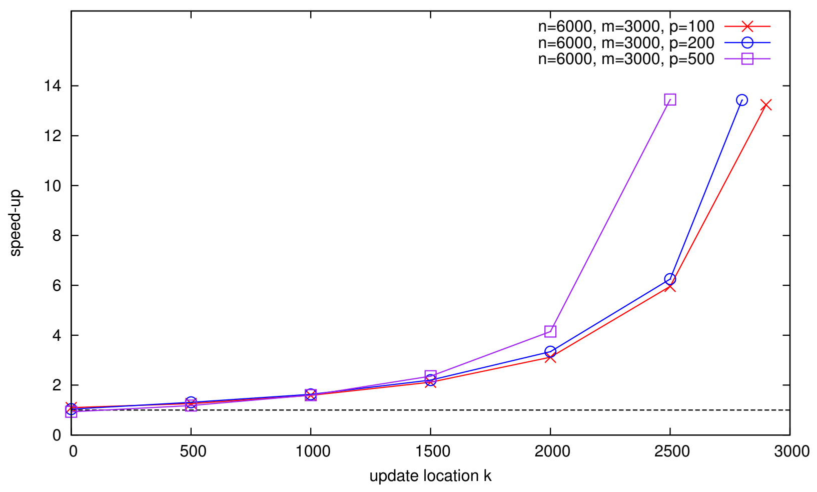

Figure 14: Speed-up of removing columns update relative to full QR factorization in CULA with varying k and p.

Table 7: Run-times in seconds for removing p columns and for CULA full QR factorization, for $m = 3000$, $n = 6000$.

k	$p = 100$		$p = 200$		$p = 500$	
	CULA	Updating	CULA	Updating	CULA	Updating
0	0.662	0.601	0.631	0.605	0.565	0.609
500		0.526		0.482		0.479
1000		0.417		0.387		0.353
1500		0.312		0.287		0.239
2000		0.212		0.189		0.136
2500		0.111		0.101		0.042
2800		–		0.047		–
2900		0.050		–		–

Figure 7 shows that $m - (k-1) - p$ Householder transforms are required per update. Removing more columns (which corresponds to larger values of p) therefore decreases the number of the Householder transforms required to carry out an update, but also decreases the amount of work required for a full QR factorization. As reducing the number of Householder transformations in our implementation reduces the number of kernel invocations on the GPU, our implementation performs better as p increases. This can be seen from the run-times in Table 7, and the corresponding speed-ups over CULA (shown in Figure 14) reach over 13x for large p and k.

5.4. Adding Rows

Unlike the other updating algorithms, the block of rows U added to A can be permuted to the bottom of the matrix without altering the algorithm. We therefore set $k = 0$ for all tests in this section. As with removing columns, updating by adding rows does not require Q.

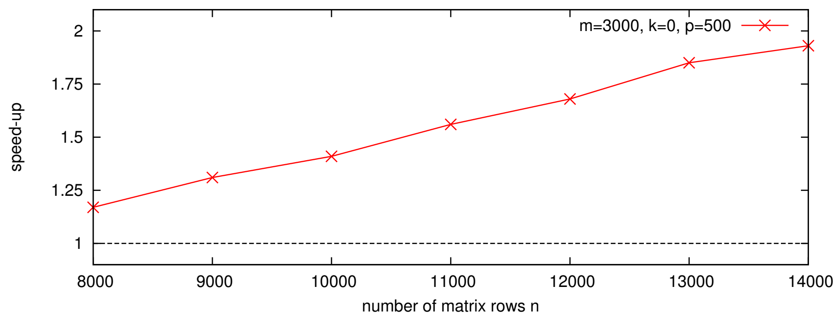

Figure 15: Speed-up of adding rows update relative to full QR factorization with varying n.

Table 8: Run-times in seconds for adding $p = 500$ rows and for CULA full QR factorization, for $m = 3000$, $k = 0$.

n	CULA	Updating
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

The complexity of updating a QR factorization by adding a block of p rows is $O(m^2p + m^2)$, compared with the $O((n+p)m^2)$ complexity of a full QR factorization of a matrix with p rows added. Once again, the complexity of the updating algorithm does not depend on the matrix height n. This is because, as shown in Figure 8, each Householder transformation is only applied to the $p \times m$ matrix U and a single row of R. As shown in Figure 15 and Table 8, the run-time of the updating algorithm therefore remains essentially constant with increasing n while the run-time of the full QR factorization increases linearly.

Table 9: Run-times in seconds for adding p rows and for CULA full QR factorization, for $n = 8000$, $k = 0$.

m	$p = 200$		$p = 1500$	
	CULA	Updating	CULA	Updating
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

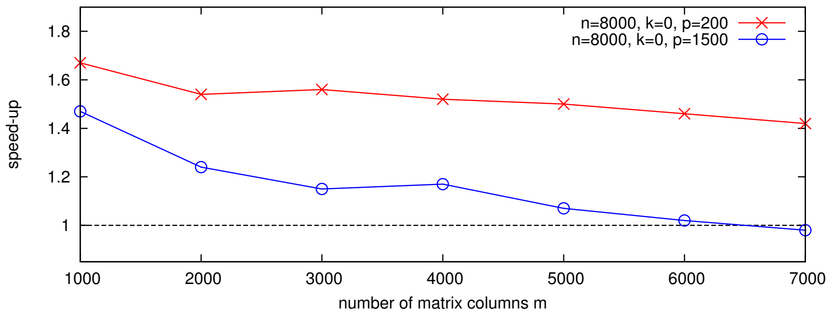

Figure 16: Speed-up of adding rows update relative to full QR factorization for varying m and p.

We investigate the effect of varying m and p. The run-times are reported in Table 9 and the speed-up of the updating method versus full factorization is shown in Figure 16. We note that the speed-up decreases as m and p increase. We might expect that the updating algorithm would never run slower than full QR factorization because we are essentially performing a full QR reduction and skipping over those entries of R that are already zero. However, to do this we require twice as many CUBLAS calls as an efficient full QR factorization. This is worthwhile for low values of p, but for large values the overhead dominates. Given the processing power of GPUs, it might be beneficial to do the unnecessary work and avoid the expense of additional function calls.

We also note that the efficiency of updating for adding rows decreases with increasing m because each of the m Householder reflections is performed using multiple CUBLAS calls. Implementing a Householder transformation as a single dedicated kernel is the subject of future work.

5.5. Removing Rows

Updating by removing rows requires Q, as was also the case for adding rows. Alone out of all the updating algorithms described here, however, it is entirely implemented using Givens rotations. These two facts combine to make it the most computationally demanding of all four updates. Its theoretical complexity is $O(pn(n+m+1))$, compared with the $O((n-p)m^2)$ complexity of a full QR factorization of a matrix with p rows removed. The complexity of the update is not dependent on the update location parameter k because all Givens transformations must be applied to the entire length n of Q and the entire width m of R, regardless of the value of k.

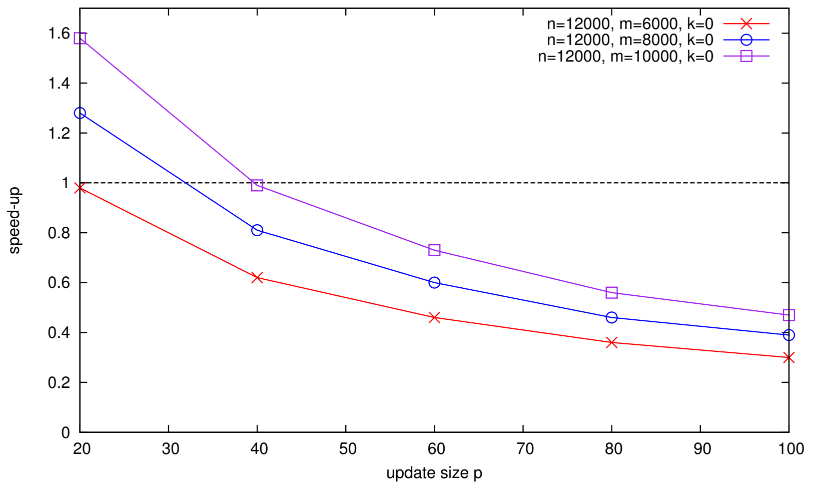

Figure 17: Speed-up of removing rows update relative to full QR factorization for varying m and p.

Table 10: Run-times in seconds for removing p rows and for CULA full QR factorization, for $n = 12000$, $k = 0$.

p	$m = 6000$		$m = 8000$		$m = 10000$	
	CULA	Updating	CULA	Updating	CULA	Updating
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

Figures 17 and 18 illustrate the speed-up of removing rows updating over full QR factorization with varying m, n and p values, and accompanying run-times are shown in Tables 10 and 11. The general trend with increasing m, n and p is that updating becomes progressively more slow than full QR factorization. This

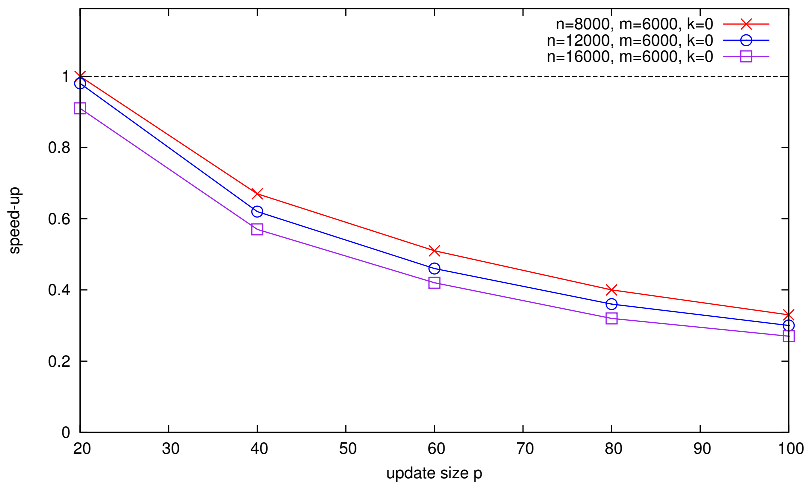

Figure 18: Speed-up of removing rows update relative to full QR factorization for varying n and p.

Table 11: Run-times in seconds for removing p rows and for CULA full QR factorization, for $m = 6000$, $k = 0$.

p	$n = 8000$		$n = 12000$		$n = 16000$	
	CULA	Updating	CULA	Updating	CULA	Updating
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

is probably attributable to the $O(n+p)$ kernel invocations required to implement the Givens rotations. We do observe speed-ups for updating compared with full factorization when p is small and m approaches n (see Figure 17). This is because the complexity of full QR factorization increases quadratically with m while for updating it only increases linearly.

5.6. Accuracy and Stability

Table 12: Table of normwise relative forward errors for $n = 4000$, $m = 2000$, $k = 0$.

p	Adding Rows		Removing Columns	
	GPU	Serial	GPU	Serial
100	2×10^{-6}	3×10^{-6}	3×10^{-6}	3×10^{-6}
300	2×10^{-6}	2×10^{-6}	3×10^{-6}	3×10^{-6}
500	2×10^{-6}	3×10^{-6}	2×10^{-6}	2×10^{-6}
700	1×10^{-6}	3×10^{-6}	2×10^{-6}	2×10^{-6}
900	3×10^{-6}	3×10^{-6}	2×10^{-6}	2×10^{-6}

Table 13: Table of normwise relative forward errors for $n = 4000$, $m = 2000$, $k = 0$.

p	Removing Rows		Adding Columns	
	GPU	Serial	GPU	Serial
100	5×10^{-6}	4×10^{-6}	3×10^{-6}	3×10^{-6}
300	4×10^{-6}	4×10^{-6}	5×10^{-6}	5×10^{-6}
500	5×10^{-6}	4×10^{-6}	6×10^{-6}	6×10^{-6}
700	6×10^{-6}	5×10^{-6}	6×10^{-6}	6×10^{-6}
900	6×10^{-6}	5×10^{-6}	7×10^{-6}	8×10^{-6}

We compare the normwise relative error $\frac{||x-\hat{x}||_2}{||x||_2}$ of the least squares solution calculated from the output of an updated factorization, $\hat{x}$, versus that of a solution from a full factorization calculated by CULA, x. We perform this comparison for updated factorizations calculated by our GPU implementation, and also for a serial implementation of the updating algorithms. Tables 12 and 13 show that the errors relative to the CULA solution are comparable between the GPU and CPU updated factorizations.

Table 14: Table of $\tilde{Q}$ orthogonality values for $n = 4000$, $m = 2000$, $k = 0$.

p	Removing Rows		Adding Columns	
	GPU	Serial	GPU	Serial
100	2.34×10^{-4}	1.62×10^{-4}	2.42×10^{-4}	1.68×10^{-4}
300	3.39×10^{-4}	1.76×10^{-4}	3.67×10^{-4}	4.67×10^{-4}
500	3.64×10^{-4}	1.85×10^{-4}	5.00×10^{-4}	6.00×10^{-4}
700	3.89×10^{-4}	1.90×10^{-4}	4.79×10^{-4}	2.29×10^{-4}
900	4.70×10^{-4}	1.93×10^{-4}	5.64×10^{-4}	2.46×10^{-4}

Table 15: Table of normwise relative backward error values for $n = 4000$, $m = 2000$, $k = 0$.

p	Removing Rows		Adding Columns	
	GPU	Serial	GPU	Serial
100	1.39×10^{-4}	7.40×10^{-5}	9.50×10^{-5}	5.00×10^{-5}
300	3.74×10^{-4}	1.87×10^{-4}	1.56×10^{-4}	6.10×10^{-5}
500	5.81×10^{-4}	3.04×10^{-4}	2.02×10^{-4}	6.90×10^{-5}
700	7.33×10^{-4}	4.12×10^{-4}	2.34×10^{-4}	7.70×10^{-5}
900	9.02×10^{-4}	5.00×10^{-4}	2.51×10^{-4}	8.30×10^{-5}

For the two updating algorithms that form $\tilde{Q}$, namely removing rows and adding columns, we compute $||\tilde{Q}^T\tilde{Q} - I||_2$ to assess the orthogonality of $\tilde{Q}$. Table 14 shows that, once again, the errors from the GPU computations are of the same order as those from the serial implementation. We also compute the normwise relative backward error $\frac{||\tilde{Q}\tilde{R}-\tilde{A}||_2}{||A||_2}$ for these two algorithms, and the results are given in Table 15. The errors are again comparable between the serial and GPU implementations. Note that there the error increases with p, which is probably because the number of transformations increases with p.

6. Conclusion

The aim of this paper was to implement the updating algorithms presented in [1] on the GPU using CUDA in order to achieve speed-ups over recomputing full QR factorizations on the GPU. Evaluation of our implementation shows that the GPU updating algorithms outperform full GPU QR factorization for certain values of the input parameters m, n, p and k. In a situation where repeated QR factorizations needed to be computed we could therefore decide whether to update or to recompute from scratch by considering the size of the problem, the type of update and the number of rows/columns to be added/removed.

We observed that the best performance was achieved when removing large numbers of columns for large values of k (corresponding to removing columns from the right-hand portion of A), with speed-ups of up to 13.5x over full factorization (see Figure 14). Other good performers were adding smaller numbers of columns for large values of k, with speed-ups of up to 3.5x (Figure 11), and adding rows to a tall-and-skinny A, which gave speed-ups approaching 2x (Figure 15). We found that removing rows, which required the application of Givens rotations to both Q and R, performed worse than CULA for all cases except removing a small number of rows from an almost-square A.

Many of the performance issues we encountered were linked to high frequencies of kernel invocations. A possible method for reducing the number of kernels used to apply Givens rotations would be to increase the number of rows in a strip. It might then be possible to apply multiple dependent rotations per thread block by looping and synchronization statements within the kernel code. Alternatively, Nvidia have recently introduced "dynamic parallelism" to their Kepler architecture [15]. This permits the dynamic spawning of kernels on the

GPU without involving the CPU, and so could overcome our problem with large kernel invocation overheads.

Another approach would be to use Householder transformations in place of Givens rotations where possible. For example, in the adding columns update Householder transformations could be used even though they generate fill-in because the resulting extra floating point operations are likely to be cheaper on the GPU than invoking multiple kernels.

Acknowledgements

The authors would like to thank IT Services for Research (formerly Research Computing Services) at the University of Manchester for providing access to the GPU nodes on the Computational Shared Facility (CSF). The second author is supported by EPSRC grant EP/I006702/1 “Novel Asynchronous Algorithms and Software for Large Sparse Systems”.

References

[1] S. Hammarling, C. Lucas, Updating the QR factorization and the least squares problem, MIMS EPrint 2008.111, University of Manchester, Manchester, UK, 2008.

[2] R. Andrew, Implementation of QR updating algorithms on the GPU, Master’s thesis, University of Manchester, Manchester, UK, 2012. Available as MIMS EPrint 2012.80.

[3] A. Kerr, D. Campbell, M. Richards, QR decomposition on GPUs, in: Proc. 2nd Workshop on General Purpose Processing on Graphics Processing Units (GPGPU-2), Washington, D.C., USA, pp. 71–78. 2009.

[4] E. Agullo, C. Augonnet, J. Dongarra, M. Faverge, H. Ltaief, S. Thibault, S. Tomov, QR factorization on a multicore node enhanced with multiple GPU accelerators, in: Proc. 2011 IEEE International Parallel Distributed Processing Symposium (IPDPS’11), Anchorage, AK, USA, pp. 932–943. 2011.

[5] J. Kurzak, R. Nath, P. Du, J. Dongarra, An implementation of the tile QR factorization for a GPU and multiple CPUs, in: K. Jónasson (Ed.), Applied Parallel and Scientific Computing, volume 7134 of *Lecture Notes in Computer Science*, Springer, Berlin, Heidelberg, 2012, pp. 248–257.

[6] M. Anderson, G. Ballard, J. Demmel, K. Keutzer, Communication-avoiding QR decomposition for GPUs, Technical Report UCB/EECS-2010-131, University of California at Berkley, 2010.

[7] CULA: GPU accelerated linear algebra, EM Photonics, 2012. `http://www.culatools.com/`.

[8] J. Humphrey, D. Price, K. Spagnoli, A. Paolini, E. Kelmelis, CULA: Hybrid GPU accelerated linear algebra routines, in: Proc. SPIE Defense, Security and Sensing (DSS'10), Orlando, FL, USA. 2010.

[9] B. Gunter, R. Van De Geijn, Parallel out-of-core computation and updating of the QR factorization, ACM Trans. Math. Softw. 31 (2005) 60–78.

[10] G. Golub, C. V. Loan, Matrix Computations, Johns Hopkins University Press, 3rd edition, 1996.

[11] N. J. Higham, Accuracy and Stability of Numerical Algorithms, Society for Industrial and Applied Mathematics, Philadelphia, PA, USA, 2nd edition, 2002.

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

[13] O. Egecioglu, A. Srinivasan, Givens and Householder reductions for linear least squares on a cluster of workstations, in: Proc. International Conference on High Performance Computing (HiPC'95), New Delhi, India, pp. 734–739.

[14] G. Ruetsch, P. Micikevicius, Optimizing matrix transpose in CUDA, Technical Report, Nvidia, 2009. `http://www.cs.colostate.edu/~cs675/MatrixTranspose.pdf`.

[15] Dynamic parallelism in CUDA, Nvidia, 2012. `http://developer.download.nvidia.com/assets/cuda/docs/TechBrief_Dynamic_Parallelism_in_CUDA_v2.pdf`.