# *Implementing QR Factorization Updating Algorithms on GPUs*

Andrew, Robert and Dingle, Nicholas J.

2014

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, Manchester, M13 9PL*

**Abstract**

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

*Keywords:* QR factorization, QR updating, GPGPU computing

## 1. Introduction

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

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

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

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

$R_1$ is upper triangular and thus $||R_1 x - f||_2^2 = 0$ can be solved by back substitution, leaving $||g||_2^2$ as the minimum residual.

This approach fits a linear model to observed data. For example, we can use it to model the relationship between an individual's wage and variables such as their age, education and length of employment. Each row of $A$ will contain the

*Corresponding author

*Email addresses:* `robert.andrew@postgrad.manchester.ac.uk` (Robert Andrew), `nicholas.dingle@manchester.ac.uk` (Nicholas Dingle) values of these variables for one person, with the corresponding entry in $b$ being the observed value of their pay. In order to incorporate extra observations (in this example, people) we add rows to $A$, while if we wish to remove observations we must delete rows. Similarly, we can add variables to the problem by adding columns to $A$ and remove them by deleting columns.

QR factorizations are computationally expensive, but when elements are added to or removed from $A$ it is not always necessary to recompute $Q$ and $R$ from scratch. Instead, it can be cheaper to update the existing factorization to incorporate the changes to $A$. We aim to accelerate the updating algorithms originally presented in [1] by implementing them on a GPU using CUDA. These algorithms have been shown to outperform full QR factorization in a serial environment [1], and we have previously demonstrated that their implementation on a GPU can outperform a serial implementation by a wide 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 achieved speed-ups of 13x over the CULA library [6, 7] for tall-and-skinny matrices by applying communication-avoiding QR [8]. Updating and downdating algorithms [9] have been implemented in shared- and distributed-memory parallel environments, including parallel out-of-core updating for adding rows [10], block-based downdating [11] and MPI-based parallel downdating [12]. To the best of our knowledge there is no prior work on implementing all four updating algorithms on GPUs.

QR factorization decomposes the $m \times n$ matrix $A$ into the $m \times m$ orthogonal matrix $Q$ and the $m \times n$ 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 $\tilde{A}$ and has the corresponding updated factorization $\tilde{Q}\tilde{R}$.

Section 2 details the implementation of Givens and Householder transformations on the GPU. Section 3 summarizes the updating algorithms and describes how we implemented them, before Section 4 presents performance results. We show the speed-ups achieved by our implementation over the full QR factorization routine used by CULA. We also investigate the accuracy and stability of the updating algorithms. Section 5 concludes and discusses future work.

## 2. Householder and Givens Transformations on the GPU

Householder and Givens transformations are standard tools for computing the QR factorization [13, 14] and they are also key components of the factorization updating algorithms. Householder transformations can be readily implemented on GPUs using CUBLAS [15]. To better exploit the instruction bandwidth of GPUs we use a blocked Householder approach [3, 14] built on BLAS level 3 operations that combines $n_b$ Householder transformations into a single matrix:

$$P = P_1 P_2 \dots P_{n_b} = I + WY^T$$

where $W$ and $Y$ are matrices with the number of rows equal to the length of the longest Householder vector and number of columns equal to the number of transformations.

Efficient GPU parallelization of Givens rotations is more challenging. A Givens rotation alters entries in two rows of the matrix and therefore multiple rotations can only be conducted in parallel if they affect distinct rows. This leads to the following scheme, illustrated in Figure 1, which has been applied on both distributed and shared memory systems [16]:

1. Each processor $p_1, \dots, p_k$ is assigned a strip of rows (Stage 1).
2. Starting at the lower left corner of the matrix, $p_1$ zeroes the entries in the first column of its strip using Givens rotations (Stage 2).
3. When $p_1$ reaches the top of the first column of its strip, $p_2$ takes over zeroing entries in the first column. In parallel $p_1$ begins zeroing entries in the second column (Stage 3).
4. The algorithm continues in this way until the matrix is upper trapezoidal.

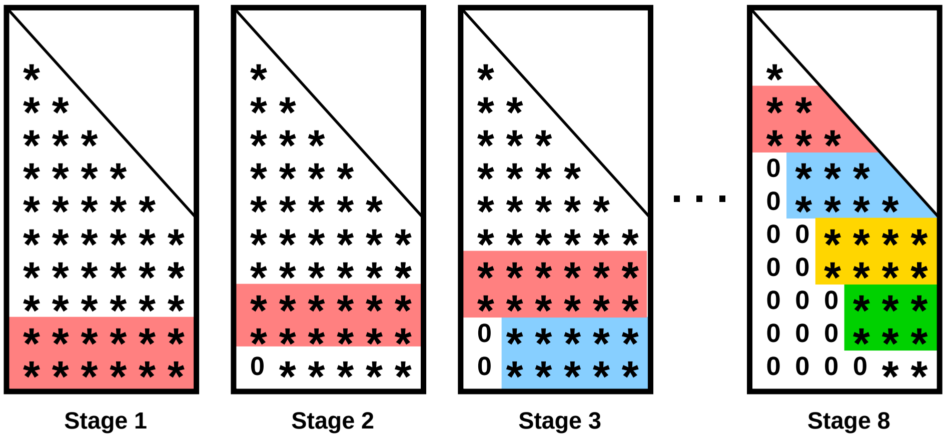

Figure 1: Applying Givens rotations in parallel.

GPUs prefer finer-grained parallelism compared with CPUs (less work per thread, more threads) so we adapt this approach by assigning two rows to each thread. Each stage of the algorithm requires the invocation of separate CUDA kernels to calculate and then apply the Givens rotations. Note that this approach requires the matrix to be accessed by row, but as other algorithms in this paper require access by column we have implemented an efficient CUDA transpose kernel [17].

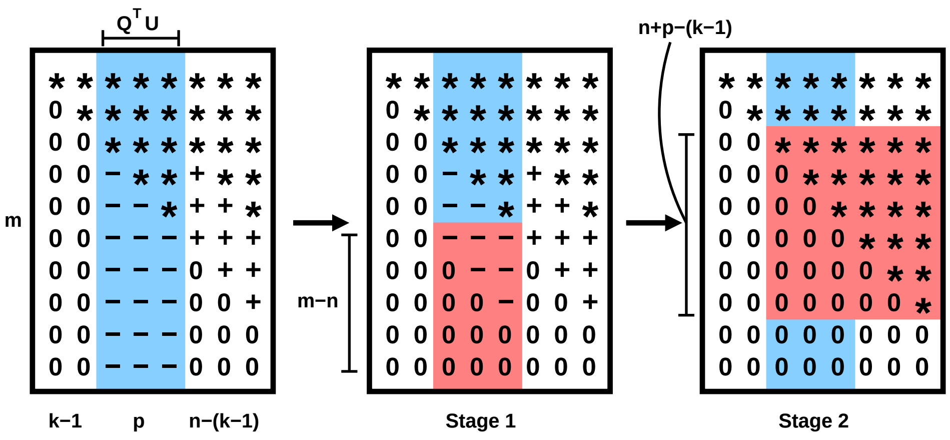

Figure 2: Adding columns update of $R$. The symbol '-' denotes a nonzero which must be made zero, '+' denotes a zero which must be made nonzero and '*' denotes a general nonzero element. The shaded blue area is the added columns, while the shaded red area is the area affected by each stage of the algorithm.

## 3. Implementing the QR Updating Algorithms

We now summarize the QR updating algorithms originally presented in [1] and describe how we implemented them for GPUs.

### *3.1. Adding Columns*

Adding a block of $p$ columns, denoted $U$, to $A$ before column $k$ gives:

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

Multiplying through by $Q^T$ yields:

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

Since $\tilde{R}(1:m, 1:k-1) = R(1:m, 1:k-1)$ we need only perform a QR factorization of $\left[Q^T U \; R(1:m, k:n)\right]$ to form the remainder of $\tilde{R}$. To do this efficiently we use Householder transformations to make $Q^T U(n+1:m, 1:p)$ upper trapezoidal, and then selectively apply Givens rotations to make $\tilde{R}$ upper trapezoidal. We cannot use Householder transformations for the second stage as they would make $\tilde{R}$ full.

Note that the use of $Q^T U$ to update $R$ means that $\tilde{Q}$ must also be formed if further updates are required. This can by accomplished by post-multiplying $Q$ with the same Householder and Givens transformations used to form $\tilde{R}$.

Figure 2 illustrates the algorithm for an example where $m = 10$, $n = 5$, $p = 3$ and $k = 3$. Taking $Q$, $R$, an $m \times p$ block of columns $U$ and an index $k$, $0 \leq k \leq n+1$ as input, the implementation proceeds as follows:

1. QR factorize the lower $(m-n) \times p$ block of $Q^T U$ using Householder transformations. This is the area shaded in red in Stage 1 of Figure 2.

2. If $k = n + 1$ then the update is complete.
3. If not, transposes the section shown in red in Stage 2 of Figure 2.
4. Apply Givens rotations to make the transposed section upper triangular and also to update $Q$ and $d$. These three updates are independent so separate CUDA streams can be used simultaneously for each.
5. Transpose back the (now reduced) red section in Stage 2 of Figure 2.

### *3.2. Removing Columns*

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

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

Multiplying through by $Q^T$:

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

Again $\tilde{R}(1:m, 1:k-1) = R(1:m, 1:k-1)$ and we can form $n-(k-1)-p$ Householder transformations to reduce just the right-hand portion of $R$:

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

This QR factorization of a submatrix of $R$ can be performed efficiently using blocked Householder transformations.

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

1. If the right-most $p$ columns of $A$ were deleted, the update is complete.
2. Blocked Householder QR factorization is applied to the $(p + n_b + 1) \times n_b$ submatrix to the right of the removed columns (Stage 1). This is repeated across submatrices to the right until $R$ is upper trapezoidal (Stage 2).

### *3.3. Adding Rows*

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

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

We can permute $U$ to the bottom of $\tilde{A}$:

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

and thus:

$$\left[ \begin{array}{cc} Q^T & 0 \\ 0 & I_p \end{array} \right] P\tilde{A} = \left[ \begin{array}{c} R \\ U \end{array} \right].$$

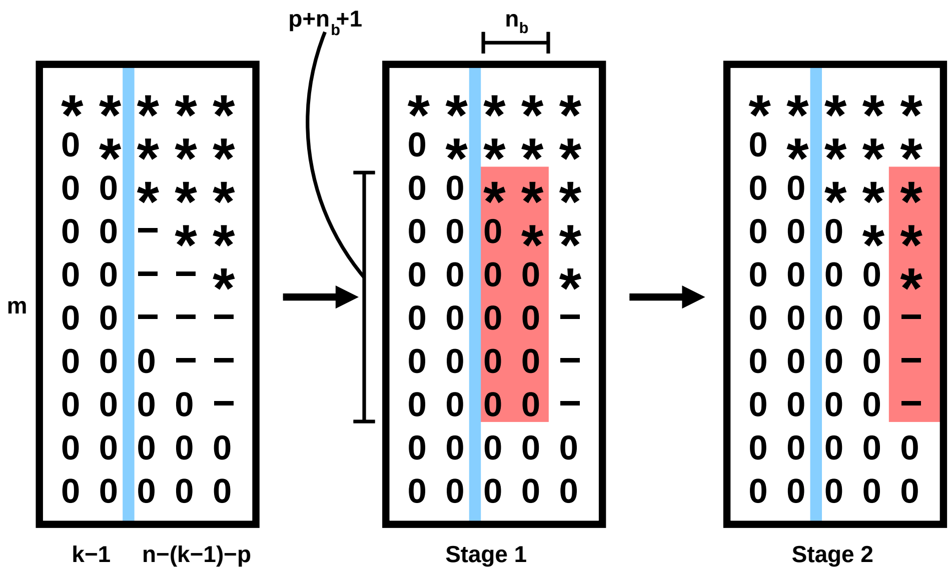

Figure 3: Reducing one block within the removing columns update of $R$ for block size $n_b = 2$. The shaded blue line shows where the removed columns used to be, while the shaded red area shows the active section for that stage in the algorithm.

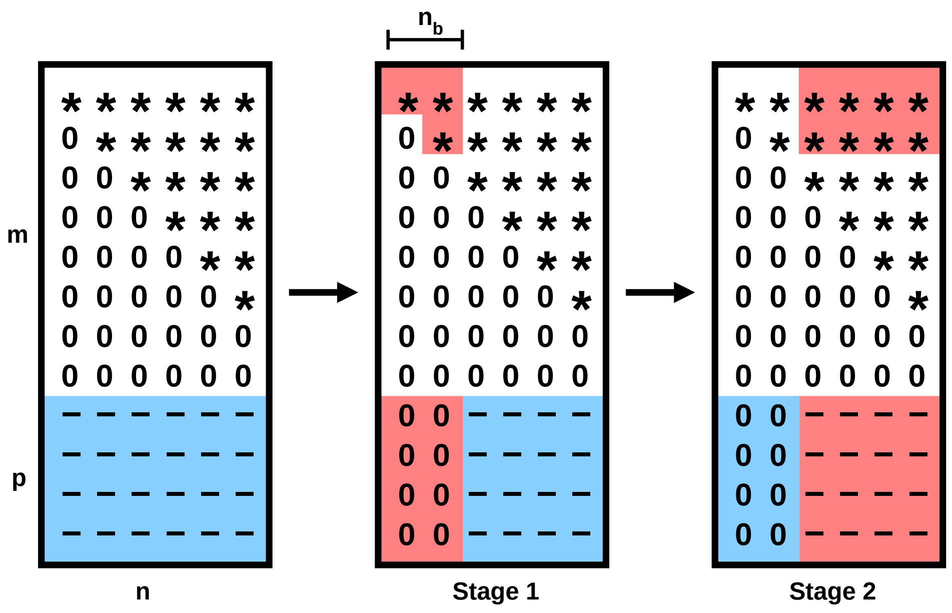

Figure 4: Reducing one block within the adding rows update of $R$ for block size $n_b = 2$. The shaded blue area shows the added rows, while the elements shaded in red in Stage 1 are the elements involved in reduction of a block via Householder transformations. The elements in the red area in Stage 2 are multiplied by the matrices produced in the previous stage.

With $n$ Householder transformations we can eliminate $U$ and form $\tilde{R}$:

$$H_n \dots H_2 H_1 \begin{bmatrix} R \\ U \end{bmatrix} = \tilde{R}$$

and because $A = QR$:

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

As with the removing columns update we are performing a QR factorization on $R$ to give an upper trapezoidal $\tilde{R}$. This can be done efficiently with blocked Householder transformations, and we can further reduce the operation count by exploiting the fact that $R$ already has many zero entries. Instead of applying the full $W$ and $Y$ matrices to all of $U$ and $R$, we instead apply the transformations separately to $U$ and the non-zero part of $R$ using a matrix $V = [v_1\ v_2 \dots v_{n_b}]$ with Householder vectors as its columns [1] and an upper triangular matrix $T$. The product of $n_b$ Householder matrices can be defined as:

$$H_1 H_2 \dots H_{n_b} = I - VTV^T,$$

with:

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

and:

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

where $\tau_i$ is the Householder coefficient corresponding to the $i^{th}$ Householder vector contained in the $i^{th}$ column of $V$. As the assignment of individual $\tau_i$ elements along the diagonal of $T$ is independent of the calculation of the other elements of $T$, the entire diagonal can be assigned in parallel within a single 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:

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

and also used to update the right-hand side of the original problem:

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

where $k_b$ is the column index in the blocked update where the recently reduced block began, and $e$ contains values added to $b$ corresponding to the rows added to $A$ [1].

We calculate this update using $R$, an index $k$, $0 \leq k \leq m+1$, and a $p \times n$ block of rows $U$. Note that the value of $k$ does not affect the algorithm as the added rows can be permuted to the bottom of $A$. Figure 4 shows an example where $m = 8$, $n = 6$, $p = 4$. We proceed as follows for each $p \times n_b$ block in $U$:

1. Stage 1 in Figure 4: 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 4: update $R$ and $b$ by multiplying with $T$ and $V$.

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

### *3.4. Removing Rows*

Removing a block of $p$ rows from $A$ from row $k$ onwards gives:

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

We first permute the deleted rows to the top of $A$:

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

and then construct a series of Givens matrices, $G$, to introduce zeros to the right of the diagonal in the first $p$ rows of $PQ$, thus removing the rows of $Q$ that correspond to the rows deleted from $A$. These transformations are also applied to $R$, which yields:

$$PA = \left[\begin{array}{c} A(k:k+p-1,1:n) \\ \tilde{A} \end{array}\right] = P(QG)(G^TR) = \left[\begin{array}{cc} I & 0 \\ 0 & \tilde{Q} \end{array}\right]\left[\begin{array}{c} S \\ \tilde{R} \end{array}\right]$$

giving $\tilde{A} = \tilde{Q}\tilde{R}$. Note that Householder transformations cannot be used because $H$ would be full and constructed so as to eliminate elements of $PQ$, which would cause $\tilde{R}$ to be full.

We calculate this update using $Q$, $R$, an index $k$, $0 \leq k \leq m-p+1$, and a block height $p$. Figure 5 shows an example of the reduction process where $m = 12$, $n = 5$, $p = 4$ and $k = 5$. A strip of rows $Z$ is identified within $Q$ corresponding to the removed rows from $A$, $Z := Q(k:k+p-1,1:n)$.

The implementation proceeds as follows:

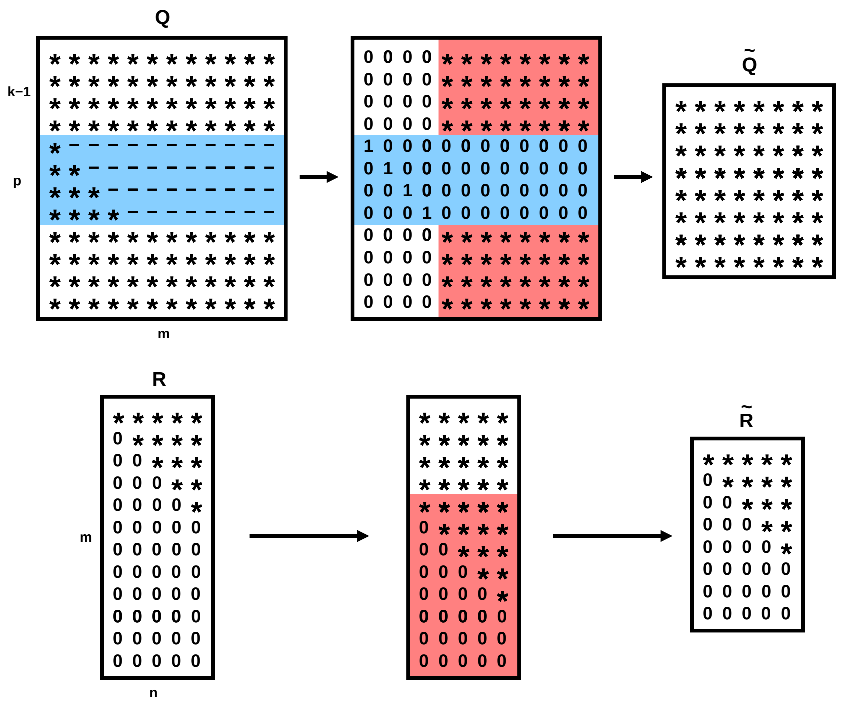

Figure 5: Removing rows in $Q$ and $R$. The shaded blue area shows the rows in $Q$ corresponding to the rows removed in $A$. The elements shaded in red are the elements that form $\tilde{Q}$ and $\tilde{R}$.

1. Assign the variable $Z$ to the strip to be reduced by Givens transformations.
2. Apply Givens transformations to reduce $Q$ to the form shown in the centre of the top of Figure 5, with zeros in the first $p$ columns and in $Z$, and with the identity matrix embedded in the first $p$ columns of $Z$.
3. Apply the same Givens transformations to $R$ as well.
4. Form $\tilde{Q}$ and $\tilde{R}$ from the elements shaded red in Figure 5.

## 4. 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 with ECC enabled. We start measuring the run-time of our implementations, $t_{update}$, from when they initiate transfer of $Q$ (for those updating algorithms that require it), $R$ and $d$ from host memory to the GPU. We then execute our updating algorithm and use $\tilde{R}$ 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 include the time to do this.

Table 1: Run-times in seconds for applying 1000 Householder transformations.

| $n_b$ | Adding Rows<br>$n = 4000, m = 1000$<br>$k = 250, p = 200$ | Adding Columns<br>$n = 4000, m = 1000$<br>$k = 250, p = 1000$ | Removing Columns<br>$n = 4000, m = 1200$<br>$k = 0, p = 200$ |
|---|---|---|---|
| 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 |

We compare our implementation against the run-time, $t_{full}$, of QR-based least squares solve from the CULA library (version R14). Note that CULA is closed-source and we do not know the details of its implementation. We time the execution of the `culaDeviceSgels` routine, including the time taken to transfer $\tilde{A}$ and $\tilde{b}$ from host to GPU. The speed-up is $\frac{t_{full}}{t_{update}}$.

The entries of $\tilde{A}$ are uniformly-distributed random numbers in the interval $(-1, 1)$ and all matrices fit in GPU memory. All run-times are measured in seconds and all values presented are averages over 5 executions.

## *4.1. Choosing the Block Size Parameter*

The optimum value of $n_b$ in the blocked Householder algorithm is problem-dependent. It was not possible to determine this value for each problem size, and we instead choose $n_b$ based on the performance of a test case. Table 1 shows run-times for each of the three algorithms that feature blocked Householder transformations when applying 1000 Householder vectors to 1000 columns. We observe that the optimum block size lies between 50 and 100 columns per block and so in all further tests we pick an $n_b$ that gives ten blocks per factorization.

## *4.2. Adding Columns*

The speed-ups of the adding columns update relative to CULA full factorization for varying $k$ and two different $n$ values are shown in Figures 6 and 7. As $k$ gets smaller the cost of updating approaches that of computing a whole new factorization, and consequently only the highest values of $k$ show any speed-up.

The updating algorithm performs better relative to CULA for large $k$ and $n$ (adding columns to the end of a matrix that has a large number of columns) because this reduces the size of the submatrix to which Givens rotations must be applied (the red area in Stage 2 of Figure 2). As shown in Table 2, when $k$ is much smaller than $n$ (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 $O(n - k)$ Givens rotations kernels.

Figure 8 shows that the updating algorithm only runs faster than full factorization for smaller values of $p$ (adding fewer columns). This is because the number of Givens rotations in Stage 2 increases with $p$ and must also be applied to $Q$ as well as $R$ in order to enable subsequent updates.

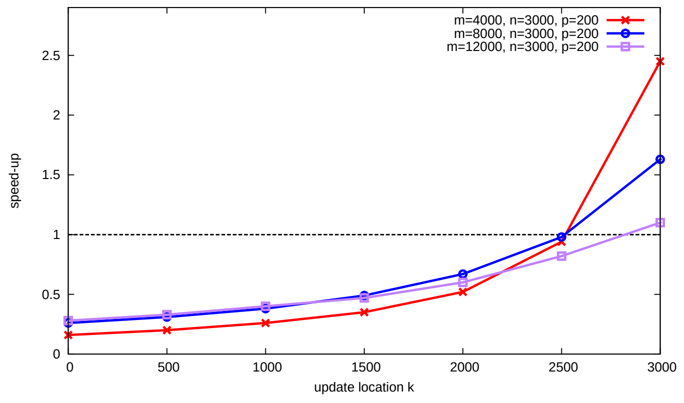

Figure 6: Speed-up of adding columns update over full QR factorization for $n = 3000$.

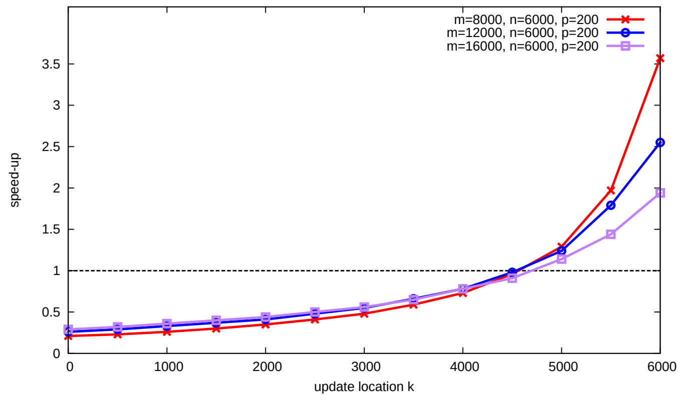

Figure 7: Speed-up of adding columns update over full QR factorization for $n = 6000$.

Table 2: Run-times of the individual stages of adding $p = 200$ columns update for $m = 16000$, $n = 6000$.

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

## *4.3. Removing Columns*

The cost of the removing columns updating algorithm is independent of $m$ because we only apply Householder transformations to a $(p + n_b + 1) \times (n - (k - 1) - p)$ submatrix of $R$. This is not the case for full factorization, and as shown in Figure 9 the speed-up of updating over CULA accordingly increases as $m$ increases.

Removing more columns (larger values of $p$) decreases the number of Householder transformations required to update the factorization as well as decreasing the amount of work required for a full QR factorization. As reducing the number of Householder transformations reduces the number of kernel invocations on the GPU, our updating implementation performs better as $p$ increases. We consequently observe speed-ups over CULA shown in Figure 10 that reach over 13x for large $p$ and $k$.

## *4.4. Adding Rows*

We set $k = 0$ for all tests in this section because the block of added rows can be permuted to the bottom of the matrix without altering the algorithm. The cost of the updating algorithm does not depend on $m$ because the Householder transformations are not applied to the zero elements of $R$. The run-time of the updating algorithm therefore remains essentially constant with increasing $m$ while the run-time of the full QR factorization increases, giving rise to the speed-ups shown in Figure 11.

As shown in Figure 12, however, the speed-up decreases as $n$ and $p$ increase. We might expect that the updating algorithm would never run slower than full QR factorization because it essentially performs a full QR reduction that skips over the zeros in $R$. To do this, however, requires twice as many CUBLAS calls as an efficient full QR factorization and so is only worthwhile for low values of $n$ and $p$. Given the processing power of GPUs, it might be beneficial to do the unnecessary arithmetic and avoid the expense of additional function calls.

## *4.5. Removing Rows*

This is the most computationally demanding update because it applies multiple Givens rotation kernels to both $Q$ and $R$. Figures 13 and 14 illustrate the speed-up of removing rows updating over full QR factorization with varying $m$, $n$ and $p$. Updating becomes progressively slower than full QR factorization as $m$, $n$ and $p$ increase because of the $O(m + p)$ kernel invocations required to implement the Givens rotations. Updating is only faster than full factorization when $p$ is small and $m$ approaches $n$ (see Figure 13).

## *4.6. Accuracy and Stability*

We compute the normwise relative error $\frac{||x-\hat{x}||_2}{||x||_2}$ of the least squares solution from an updated factorization, $\hat{x}$, against that of a solution from a full factorization calculated by CULA, $x$. We compare both GPU-based updating and a serial implementation of the updating algorithms. Table 3 shows that the errors are comparable between the GPU and CPU updated factorizations.

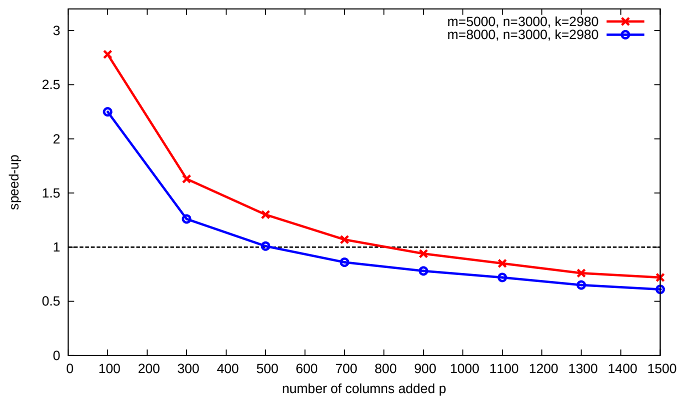

Figure 8: Speed-up of adding columns update over full QR factorization for $k = 2980, n = 3000$ for increasing numbers of additional columns $p$.

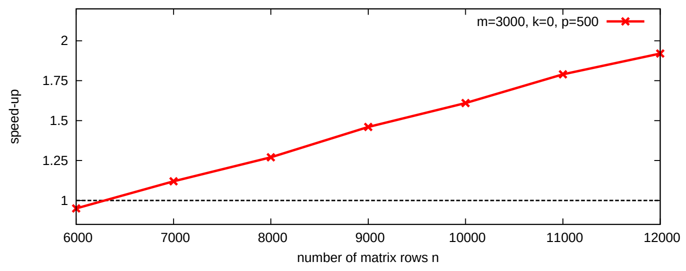

Figure 9: Speed-up of removing columns update over full QR factorization.

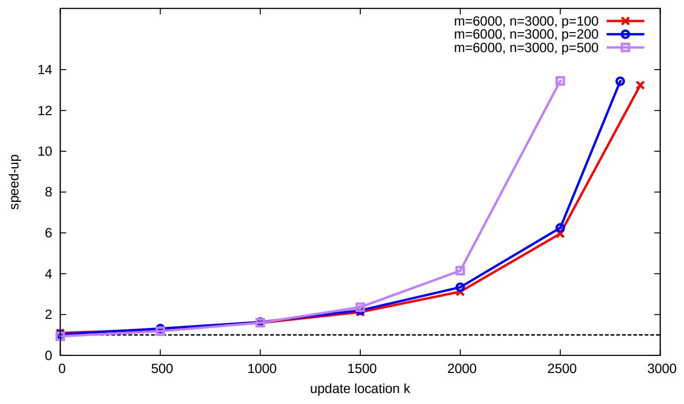

Figure 10: Speed-up of removing columns update over full QR factorization in CULA with varying $k$ and $p$.

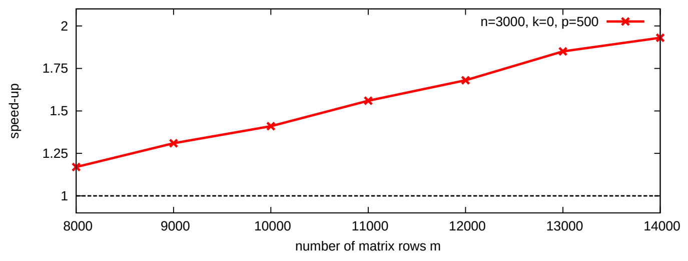

Figure 11: Speed-up of adding rows update over full QR factorization with varying $m$.

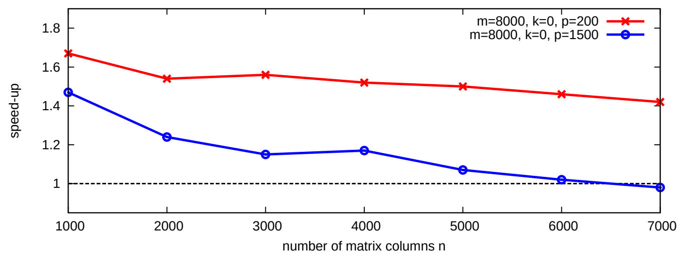

Figure 12: Speed-up of adding rows update over full QR factorization for varying $n$ and $p$.

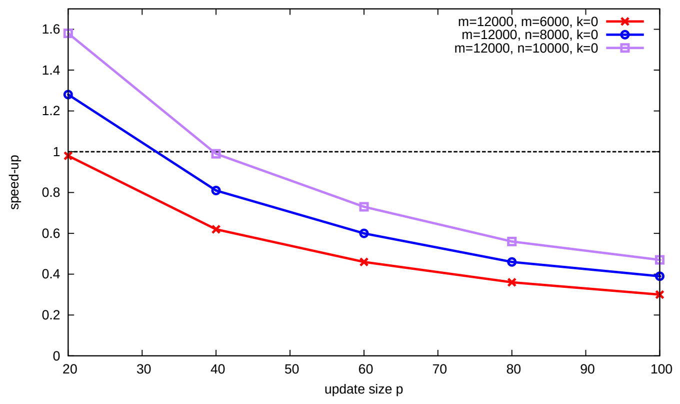

Figure 13: Speed-up of removing rows update over full QR factorization for varying $n$ and $p$.

Table 3: Table of normwise relative forward errors for $m = 4000$, $n = 2000$, $k = 0$.

| $p$ | Adding Rows | | Removing Columns | | Removing Rows | | Adding Columns | |
|---|---|---|---|---|---|---|---|---|
| | GPU | Serial | GPU | Serial | GPU | Serial | GPU | Serial |
| 100 | $2\times10^{-6}$ | $3\times10^{-6}$ | $3\times10^{-6}$ | $3\times10^{-6}$ | $5\times10^{-6}$ | $4\times10^{-6}$ | $3\times10^{-6}$ | $3\times10^{-6}$ |
| 300 | $2\times10^{-6}$ | $2\times10^{-6}$ | $3\times10^{-6}$ | $3\times10^{-6}$ | $4\times10^{-6}$ | $4\times10^{-6}$ | $5\times10^{-6}$ | $5\times10^{-6}$ |
| 500 | $2\times10^{-6}$ | $3\times10^{-6}$ | $2\times10^{-6}$ | $2\times10^{-6}$ | $5\times10^{-6}$ | $4\times10^{-6}$ | $6\times10^{-6}$ | $6\times10^{-6}$ |
| 700 | $1\times10^{-6}$ | $3\times10^{-6}$ | $2\times10^{-6}$ | $2\times10^{-6}$ | $6\times10^{-6}$ | $5\times10^{-6}$ | $6\times10^{-6}$ | $6\times10^{-6}$ |
| 900 | $3\times10^{-6}$ | $3\times10^{-6}$ | $2\times10^{-6}$ | $2\times10^{-6}$ | $6\times10^{-6}$ | $5\times10^{-6}$ | $7\times10^{-6}$ | $8\times10^{-6}$ |

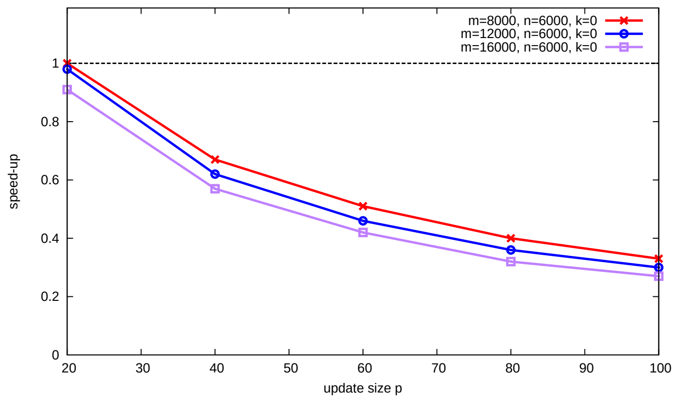

Figure 14: Speed-up of removing rows update over full QR factorization for varying $m$ and $p$.

Table 4: Table of $\tilde{Q}$ orthogonality values for $m = 4000$, $n = 2000$, $k = 0$.

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

Table 5: Table of normwise relative backward error values for $m = 4000$, $n = 2000$, $k = 0$.

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

For the two updating algorithms that form $\tilde{Q}$ (removing rows and adding columns) we assess the orthogonality of $\tilde{Q}$ by computing $||\tilde{Q}^T\tilde{Q} - I||_2$. Table 4 shows that the errors from the GPU computations are again 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 5. The errors are again comparable between serial and GPU implementations.

## 5. Conclusion

We have implemented four GPU updating algorithms using CUDA and have shown that they outperform full GPU QR factorization for certain values of $m$, $n$, $p$ and $k$. We observed the best performance 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. Adding smaller numbers of columns for large values of $k$, with speed-ups of up to 3.5x, and adding rows to a tall-and-skinny $A$, which gave speed-ups approaching 2x, also performed well. Removing rows, which required the application of Givens rotations to both $Q$ and $R$, performed worse than CULA for all cases except for removing a small number of rows from an almost-square $A$.

Many of the encountered performance issues were linked to high frequencies of kernel invocations. We could reduce the number of kernels needed to apply Givens rotations by increasing the number of rows in a strip and applying multiple dependent rotations per thread block by looping and synchronization statements within the kernel. Alternatively, we could reduce kernel invocation overheads by using the Nvidia Kepler architecture's dynamic parallelism [18] to spawn kernels directly on the GPU.

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

We have identified some situations in which updating is faster than full factorization, and others where it is not. Ideally, we would like to be able to identify in advance whether or not updating will save time by considering the size of the problem, the type of update and the number of rows/columns to be added/removed. The original presentation of the algorithms includes operation counts in terms of $m$, $n$, $p$ and $k$ that we could use as the basis for closed-form expressions to approximate the run-times of our GPU implementations.

Finally, it would be instructive to compare our implementations with the QR routines from other GPU linear algebra libraries such as MAGMA [19].

**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 authors would also like to thank the anonymous referees for their extremely helpful comments. The second author was 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.

[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.

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

[7] 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.

[8] 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.

[9] E. Kontoghiorghes, Parallel Algorithms for Linear Models – Numerical Methods and Estimation Problems, Springer, 2000.

[10] 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.

[11] P. Yanev, E. Kontoghiorghes, Efficient algorithms for block downdating of least squares solutions, Applied Numerical Mathematics 49 (2004) 3–15.

[12] P. Yanev, E. Kontoghiorghes, Parallel algorithms for downdating the least squares estimator of the regression model, Parallel Computing 34 (2008) 451–468.

[13] Å. Björck, Numerical Methods for Least Squares Problems, SIAM, Philadelphia, PA, USA, 1996.

[14] G. Golub, C. Van Loan, Matrix Computations, Johns Hopkins University Press, 4th edition, 2013.

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

[16] 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.

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

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

[19] J. Dongarra, T. Dong, M. Gates, A. Haidar, S. Tomov, I. Yamazaki, MAGMA: a new generation of linear algebra library for GPU and multicore architectures, Supercomputing 2012 (SC12). Salt Lake City, UT, USA.