UPDATED QR DECOMPOSITIONS FOR EFFICIENT NNLS AND ITS GPU PARALLELIZATION

YUANCHENG LUO AND RAMANI DURAISWAMI
UNIVERSITY OF MARYLAND, COLLEGE PARK

Abstract. In many signal processing applications, non-negative least squares problem of "moderate" size (in a few hundred to a thousand variables) arise. Efficient solutions of these problems would enable online applications, in which the estimation can be performed as data is acquired. We parallelize a version of the *active-set* iterative algorithm derived from the original algorithm of Lawson and Hanson (1974) on a graphics processor. This algorithm requires the solution of an unconstrained least squares problem in every step of the iteration for a matrix composed of the "passive columns" of the original system matrix. To achieve improved performance we use parallelizable procedures to efficiently update and "downdate" the QR factorization of the matrix at the current iteration to account for inserted and removed columns, and efficient data structures that account for GPU memory access patterns. Results of the algorithm are presented.

Key words. Non-negative least squares, active-set, GPUs, updated QR factorization

1. Introduction. A central problem to data-modelling is the optimization of the underlying parameters specifying the model used to describe observed data. The underlying parameters denote a set m variables in a $m \times 1$ vector $x = \{x_1, x_2, \cdots, x_m\}^T$. The observed data is composed of n observations in a $(n \times 1)$ vector $b = \{b_1, b_2, \cdots, b_n\}^T$. Suppose that the observed data are linear functions of the underlying parameters in the model, then the linear functions may be expressed as a $n \times m$ matrix A where $Ax = b$ describes a linear mapping from the parameters in x to the observations in b.

In the general case where $n \geq m$, the dense overdetermined system of linear equations may be solved via a least squares approach. The usually recommended way to solve the least squares problem is with the QR decomposition of the matrix A where $A = QR$, Q is an orthogonal $n \times m$ matrix, and R is an upper-triangular $m \times m$ matrix. Modern implementations for general matrices use successive applications of the Householder transform to form QR, though variants based on the use of the Givens rotation and Gram-Schmidt orthogonalization are viable. Such algorithms carry an associated $O(nm^2)$ time-complexity. The resulting matrix equation may be rearranged to $Rx = Q^T b$ and solved via back-substitution in $O(nm)$ time.

Sometimes, the underlying parameters are constrained to be non-negative in order to reflect real-world prior information. When the data is corrupted by noise, the estimated parameters may not satisfy these constraints, producing answers which are not usable. In these cases, it is necessary to explicitly enforce the non-negativity constraints in algorithms.

The seminal work of Lawson and Hanson [17] provided the first widely used method for solving this non-negative least squares (NNLS) problem. This algorithm, later referred to as the active-set method, partitions the set of parameters or variables into the active and passive-sets. The active-set contains the variables with value zero and those that violate the constraints in the problem. The passive-set contains the variables that do not violate the constraint. By iteratively updating a feasibility vector with components from the passive-set, each iteration is reduced to an unconstrained linear least squares sub-problem that is solvable via QR.

In this paper, we present a new method for updating the QR decompositions between the iterations in the active-set algorithm in section 3. We show that our method is parallelizable and describe our implementation in sections 4.1 and 4.2. We summarize alternative solutions to the NNLS problem in section 1.2. Lastly, we demonstrate a sample application in section 5 and compare the run-time results in section 6.

1.1. Non-negative Least Squares. We formally state the NNLS problem: Given a $n \times m$ matrix $A \in \mathbb{R}^{n \times m}$, find a non-negative $m \times 1$ vector $x \in \mathbb{R}^m$ that minimizes the functional $f(x) = \frac{1}{2}\|Ax - b\|^2$ i.e.

$$min_x f(x) = \tfrac{1}{2}\|Ax - b\|^2, \qquad x_i \geq 0. \tag{1.1}$$

First consider the necessary Karush-Kuhn-Tucker (KKT) conditions for an optimal solution in non-linear programming in ref. [16]. Suppose $\hat{x} \in \mathbb{R}^m$ is a local minimum subject to inequality constraints $g_j(x) \leq 0$ and equality constraints $h_k(x) = 0$, then there exists vectors μ, λ such that

$$\bigtriangledown f(\hat{x}) + \lambda^T \bigtriangledown h(\hat{x}) + \mu^T \bigtriangledown g(\hat{x}) = 0, \quad \mu \geq 0, \quad \mu^T g(\hat{x}) = 0. \tag{1.2}$$

In order to apply the KKT conditions to the minimization function (1.1), we let $\bigtriangledown f(x) = A^T(Ax - b)$, $g_j(x) = -x_j$, and $h_k(x) = 0$. This leads to the necessary conditions,

$$\mu = \bigtriangledown f(\hat{x}), \quad \bigtriangledown f(\hat{x})^T \hat{x} = 0, \quad \bigtriangledown f(\hat{x}) \geq 0, \quad \hat{x} \geq 0 \tag{1.3}$$

that must be satisfied at the optimal solution.

1.2. Survey of NNLS Algorithms. A comprehensive review of the methods for solving the NNLS problem can be found in ref. [8]. The first widely used algorithm, proposed by Lawson and Hanson in ref. [17], is the active-set method that we implement on the GPU. Although many newer methods have since surpassed the active-set method in computation costs and run-time efficiency for large and sparse matrix systems, the active-set method remains competitive for small to moderate sized systems with unstructured and dense matrices.

In ref. [5], a number of improvements to original active-set method were made in their version of fast NNLS. Specifically, they reformulated the normal equations that appear in the pseudo-inverse for the least squares sub-problem to allow for pre-computed cross-product matrices A^TA and A^Tb. This contribution led to significant speed-ups in the presence of multiple right-hand-sides. In ref. [2], further improvements were made by grouping right-hand-side observations that led to similar pseudo-inverses.

A second class of algorithms that solved the NNLS problem belong to iterative optimization methods. Unlike the active-set methods, iterative methods are not limited to a single active constraint at each iteration. In ref. [14], a Projective Quasi-Newton NNLS approach used gradient projections to avoid pre-computing A^TA and non-diagonal gradient scaling to improve convergence and reduce zigzagging. Another approach in ref. [9] produced a sequence of vectors that converge to the solution $\hat{x}$. Each vector was optimized at a single coordinate with all other coordinates fixed via an efficiently computable analytical solution.

Other methods outside the scope of this review include the Principal Block Pivoting method for large sparse NNLS in ref. [6], and the Interior Point Newton-like method in ref. [1], [15] for moderate and large problems.

2. Active-set Method. Given a set of n equations and m unknowns in the system of linear equations, let the active-set Z be the subset of variables with negative or zero values. Let the passive-set P be the variables with positive values. Let the feasibility vector x be a solution vector that does not violate the set of constraints. Let gradient $w = \bigtriangledown f(x) = A^T(b - Ax)$ be the dual vector. Lawson and Hanson observe that only a small subset of variables remains in the candidate active-set Z at the solution. If the true active-set Z is known, then the NNLS problem is solved by the unconstrained least squares problem using only the variables from the passive-set.

Algorithm 1 Active-set method for non-negative least squares

Require: $A \in \mathbb{R}^{n \times m}$, $x = \vec{0} \in \mathbb{R}^m$, $b \in \mathbb{R}^n$, set $Z = \{1, 2, \ldots, m\}$, $P = \emptyset$
Ensure: Solution $\hat{x} \geq 0$ s.t. $\hat{x} = arg\ min \frac{1}{2}\|Ax - b\|^2$

```
1:  while true do
2:    Compute gradient w = A^T(b − Ax)
3:    if Z ≠ ∅ and max_{i∈Z}(w_i) > 0 then
4:      Let j = arg max_{i∈Z}(w_i)
5:      Move j from set Z to P
6:      while true do
7:        Let matrix A^P ∈ R^{n×*} s.t. A^P = {columns A_i s.t. i ∈ P}
8:        Compute least squares solution y for A^P y = b
9:        if min(y_i) ≤ 0 then
10:         Let α = −min_{i∈P}(x_i/(x_i − y_j)) s.t. (column j ∈ A^P) = (column i ∈ A)
11:         Update feasibility vector x = x + α(y − x)
12:         Move from P to Z, all i ∈ P s.t. x_i = 0
13:       else
14:         Update x = y
15:         break
16:       end if
17:     end while
18:   else
19:     return x
20:   end if
21: end while
```

The candidate active-set Z is updated by moving the largest positive gradient variable in w to the passive-set in

line 5. Variables in the passive-set form a candidate linear least squares system $A^P y = b$ where matrix A^P contain the column vectors in matrix A that correspond to indices in the passive-set in lines 7 to 8. A feasibility vector x moves towards the solution vector y while preserving non-negativity at the end of each iteration in line 11. Convergence to the optimal solution is proven by Lawson and Hanson.

At termination, the following relations satisfy the optimality conditions in eq. (1.3):

1. $w_i \leq 0 \quad i \in Z$ termination condition, line 3.
2. $w_i = 0 \quad i \in P$ solving least squares sub-problem, line 8.
3. $x_i = 0 \quad i \in Z$ updating sets, line 12.
4. $x_i > 0 \quad i \in P$ updating x, lines 10-11.

The variables in the passive-set form the corresponding columns of the matrix A^P in the unconstrained least squares sub-problem $A^P y = b$. As discussed previously, the cost of solving the unconstrained least squares sub-problem is $O(nm^2)$ via QR. If there are k iterations, then the cost of k independent decompositions is $O(knm^2)$. However, the decompositions are related to each other as noted in two observations regarding the structure of A^P.

1. Any changes between the active and passive-sets at each iteration are generally limited to the exchange of a single variable; usually one column vector is added or removed from A^P at each iteration.
2. Exchanged variables that have remained in the same set throughout more iterations have a lower propensity for future exchanges.

Hence, we develop a general method for QR column updating and downdating that take advantage of both the limited movement and pattern of movement between variables in the active and passive-sets. To achieve real-time and on-line processing, our method must be parallelizable on GPUs. We note that the improvements in refs. [5], [2] do not apply to our problem, and moreover do not account for possible efficiencies suggested by the observations above.

2.1. General QR Decompositions. Various implementations of the full QR decomposition are discussed in ref. [10]. The modified Gram-Schmidt (MGS) method computes QR via a series of projection operations on the columns of A. The classical and modified orthogonalization processes for column q_i in $Q = [q_1, q_2, \cdots, q_m]$ are

$$u_i = a_i - \sum_{j=1}^{i-1} \langle q_j, a_i \rangle q_j, \qquad q_i = \frac{u_i}{\|u_i\|}, \tag{2.1}$$

$$\begin{aligned} u_i^{(1)} &= a_i - \langle q_1, a_i \rangle q_1, \\ u_i^{(2)} &= u_i^{(1)} - \langle q_2, u_i^{(1)} \rangle q_2, \\ &\vdots \\ u_i &= u_i^{(i-2)} - \langle q_{i-1}, u_i^{(i-2)} \rangle q_{i-1}, \\ q_i &= \tfrac{u_i}{\|u_i\|}. \end{aligned} \tag{2.2}$$

Elements in the upper-triangular matrix

$$R = \begin{bmatrix} \langle q_1, a_1 \rangle & \langle q_1, a_2 \rangle & \langle q_1, a_3 \rangle & \cdots \\ 0 & \langle q_2, a_2 \rangle & \langle q_2, a_3 \rangle & \cdots \\ 0 & 0 & \langle q_3, a_3 \rangle & \cdots \\ \vdots & \vdots & \vdots & \ddots \end{bmatrix} \tag{2.3}$$

are denoted as pairwise inner products between the columns of Q and A.

The QR decomposition may also be computed with a series of Givens rotations. Each rotation is applied on a plane spanning two coordinate axes such

$$G(\theta) \begin{bmatrix} a \\ b \end{bmatrix} = \begin{bmatrix} c & s \\ -s & c \end{bmatrix} \begin{bmatrix} a \\ b \end{bmatrix} = \begin{bmatrix} r \\ 0 \end{bmatrix}, \qquad \begin{matrix} r = \sqrt{a^2 + b^2} \\ c = a/r, \quad s = b/r \end{matrix}$$

introduces a zero into a column vector. We embed g_{ii}, g_{jj}, g_{ij}, and g_{ji} in an identity matrix to get orthogonal matrix

$$G(\theta,i,j) = \begin{bmatrix} 1 & \cdots & 0 & \cdots & 0 & \cdots & 0 \\ \vdots & \ddots & \vdots & & \vdots & & \vdots \\ 0 & \cdots & c_{ii} & \cdots & s_{ij} & \cdots & 0 \\ \vdots & & \vdots & \ddots & \vdots & & \vdots \\ 0 & \cdots & -s_{ji} & \cdots & c_{jj} & \cdots & 0 \\ \vdots & & \vdots & & \vdots & \ddots & \vdots \\ 0 & \cdots & 0 & \cdots & 0 & \cdots & 1 \end{bmatrix} \tag{2.4}$$

for rotating coordinates on the i, j planes. The upper-triangular matrix R is formed by applying Givens rotations $G(\theta,1,n)$ starting with the left-most column vector of A and successive rotations upto the diagonal before moving to the next column. The product of the Given rotation matrices form the orthogonal matrix $Q^T = \cdots G_3 G_2 G_1$ where $R = Q^T A$. Now define matrix $R^{(i')} = (\prod_{j'=i'}^{1} G_{j'})A$ where $R^{(0)} = A$. In practice, the rotation matrices, $G(\theta,i,j)$, are not used as a single Givens transformation only modifies elements in rows i, j of matrix $R^{(i')}$. To compute $R^{(i')}$, we denote the column-independent relations as

$$R_{ik}^{(i')} = cR_{ik}^{(i'-1)} + sR_{jk}^{(i'-1)}, \qquad R_{jk}^{(i')} = -sR_{ik}^{(i'-1)} + cR_{jk}^{(i'-1)} \tag{2.5}$$

when applied to all $1 \le k \le m$.

3. Proposed Algorithm. Our observations lead to an efficient algorithm that does not recompute the entire $A^P = QR$ decomposition. Instead, we modify previous Q and R matrices with regards to two cases

1. A new variable added to set P expands or updates matrix A^P by a single column.
2. The removal of a variable from set P shrinks or downdates matrix A^P by a single column.

Furthermore, we maintain a separate ordering for the columns of A^P by the relative time of insertions and deletions over algorithm 1's iterations. This stack-like ordering ensures that variables more recently added to set P are placed near the top and computationally less expensive to update. Similarly, variables to be removed from set P are likely located near the top and are computationally less expensive to downdate.

3.1. QR Column Updating. Suppose that column a_i is added to matrix $A^P = [a_1, \cdots, a_i, \cdots, a_m]$. Define list $\hat{P}$ as an ordered list of column indices from set P such that the associated column $\hat{P}_{i-1}$ was inserted in a prior iteration to $\hat{P}_i$. Now consider the formulation for $\hat{A^P} = \hat{Q}\hat{R}$ as

$$\begin{aligned} \hat{A^P} &= [a_{\hat{p}_1}, a_{\hat{p}_2}, \cdots, a_{\hat{p}_{i-1}}, a_{\hat{p}_i}], \\ \hat{Q} &= [q_{\hat{p}_1}, q_{\hat{p}_2}, \cdots, q_{\hat{p}_{i-1}}, q_{\hat{p}_i}], \\ \hat{R} &= [r_{\hat{p}_1}, r_{\hat{p}_2}, \cdots, r_{\hat{p}_{i-1}}, r_{\hat{p}_i}] \end{aligned} \tag{3.1}$$

where $\hat{Q}$ is a $n \times i$ matrix and $\hat{R}$ is an $i \times i$ matrix. To compute $q_{\hat{p}_i}$, we need only to orthogonalize it with all other columns in $\hat{Q}$ via projections. To compute $r_{\hat{p}_i}$, we need to take the inner products between a_i and columns in $\hat{Q}$, or $\hat{Q}^T a_i$. Both quantities are found using the MGS procedure below.

Algorithm 2 Modified Gram-Schmidt (MGS) QR Column Updating

Require: list $\hat{P}$ is the elements in set P, index i the variable added to P, column a_i the new column in A^P, columns $q_j \in Q$

Ensure: $\hat{A^P} = \hat{Q}\hat{R}$, update vector $\hat{Q}^T b$, list $\hat{P}$

1: Let vector $u = a_i$
2: **for all** column index $k \in$ list $\hat{P}$ **do**
3: $\quad u = u - \langle q_k, u \rangle q_k$
4: $\quad \hat{R}_{ki} = \langle a_i, q_k \rangle$
5: **end for**
6: $q_i = \frac{u}{\|u\|}$
7: $\hat{R}_{ii} = \|u\|$
8: $\hat{Q}^T b_i = \langle q_k, b \rangle$
9: Add i to list $\hat{P}$

The time-complexity of the update procedure is $O(nm)$. For memory, the algorithm stores the $n \times m$ matrix $\hat{Q}$ and the matrix $\hat{R}$ in additional to $n \times 1$ vector $\hat{Q}^T b$.

Rotation based methods for updating QR may also be used. In ref. [11], $\hat{Q}$ and $\hat{R}$ are treated as $n \times n$ matrices where $\hat{Q}$ is initially an identity matrix. The method computes $r_{\hat{p}_i} = \hat{Q}^T a_i$ as an $n \times 1$ column vector. A series of rotation transformations introduces zeros to rows $\{i+1, i+2, \cdots, n\}$ in $r_{\hat{p}_i}$. The rotation coefficients then update the columns following i in $\hat{Q}$. The time-complexity of this procedure follows $O(n^2)$.

We compare the rotation and MGS methods with regards to the active-set algorithm. The rotation method updates $n-i$ columns of $\hat{Q}$ whereas the MGS projects i columns to generate one new orthogonal column in $\hat{Q}$. The MGS may take advantage of fewer column accesses if $n > m$ for an overdetermined NNLS problem. Furthermore, the sparsity in the solution guarantees that $i < n$ for constrained problems. The rotation method may take advantage of fast Givens rotations when applied to the columns of $\hat{Q}$ as opposed to projections via inner products in the MGS.

Householder reflection based approaches are computationally more expensive. $\hat{Q}$ and $\hat{R}$ are $n \times n$ matrices and $r_{\hat{p}_i} = \hat{Q}^T a_i$ is computed the same way as the rotation method. A Householder reflection matrix $\hat{Q}_i$ introduces zeros to rows $\{i+1, i+2, \cdots, n\}$ in $r_{\hat{p}_i}$ where $\hat{Q}_i = I - 2vv^T$ and vector v the Householder reflector. Updating matrix $\hat{Q}_{new}$ follows $\hat{Q}_{new} = \hat{Q}\hat{Q}_i^T = \hat{Q} - 2\hat{Q}vv^T$. This requires a matrix-vector product, an outer product, matrix-matrix subtraction, and costly row accesses to $\hat{Q}$.

3.2. QR Column Downdating. Suppose column a_i is deleted from matrix $A^P = [a_1, \cdots, a_{i-1}, a_{i+1}, \cdots, a_m]$. Let $\hat{p}_j$ be the corresponding column index in the ordered list. Now consider the reformulation of eq. 3.1 without column $\hat{p}_j$ as

$$\begin{array}{rcl} \tilde{A^P} & = & [a_{\hat{p}_1}, \cdots, a_{\hat{p}_{j-1}}, a_{\hat{p}_{j+1}}, \cdots, a_{\hat{p}_i}], \\ \tilde{Q} & = & [q_{\hat{p}_1}, \cdots, q_{\hat{p}_{j-1}}, q_{\hat{p}_j}, q_{\hat{p}_{j+1}}, \cdots, q_{\hat{p}_i}], \\ \tilde{R} & = & [r_{\hat{p}_1}, \cdots, a_{\hat{p}_{j-1}}, a_{\hat{p}_{j+1}}, \cdots, r_{\hat{p}_m}]. \end{array} \tag{3.2}$$

Eq. (3.2) shows that $\tilde{A^P} = \tilde{Q}\tilde{R}$ where $\tilde{A^P}$ and $\tilde{R}$ has column $\hat{p}_j$ removed and $\tilde{Q} = \hat{Q}$. The first issue is that column $\hat{p}_j$ still persists in $\tilde{Q}$. The second issue is that $\tilde{R}$ is no longer upper-triangular as the columns to right of $\hat{p}_j$ have shifted left.

Observe that the right sub-matrix shifted in $\tilde{R}$ is of Hessenberg form. In ref. [10], a series of Givens rotations introduces zeros along the sub-diagonal below the main diagonal. However, this does not directly address the first issue with the $\hat{p}_j$ column in $\tilde{Q}$. Instead, we apply a series of Given rotations to introduce zeros in the j^{th} row of $\tilde{R}$. Since the columns to the left of the deleted $\hat{p}_j$ in R are upper triangular, we only apply the rotations to column vectors to the right side in the transformation

$$\tilde{A^P} = (\tilde{Q}G_j^T G_{j+1}^T \cdots G_{i-1}^T G_i^T)(G_i G_{i-1} \cdots G_{j+2} G_{j+1} \tilde{R}).$$

$$(G_i G_{i-1} \cdots G_{j+2} G_{j+1} \tilde{R}) = \begin{bmatrix} \ddots & & & & & & \\ & * & * & * & * & & \\ & 0 & * & * & * & & \\ \cdots & 0 & 0 & 0 & 0 & \cdots \\ & 0 & 0 & * & * & & \\ & 0 & 0 & 0 & * & & \\ & & & & & \ddots \end{bmatrix}. \tag{3.3}$$

Transformation 3.3 preserves $\tilde{A^P} = \tilde{Q}\tilde{R}$ while introducing zeros along the j^{th} row of $\tilde{R}$ and modifying $\tilde{Q}$. Now, both row j in the updated $\tilde{R}$ and column j in updated $\tilde{Q}$ can be removed without violating $\tilde{A^P}$. Lastly, $\tilde{Q}^T b$ can be updated via a similar transformation.

Algorithm 3 Givens Rotation QR Column Downdating

Require: list $\hat{P}$ is the elements in set P, index i the variable removed from P, column a_i the removed column in A^P, columns $q_j \in Q$

Ensure: $\tilde{A^P} = \tilde{Q}\tilde{R}$, update vector $\tilde{Q}^T b$, list $\hat{P}$

1: **for all** column index k after $i \in$ list $\hat{P}$ **do**

2: Let $r = \sqrt{\tilde{R}_{kk}^2 + \tilde{R}_{ik}^2}$, Let $c = \frac{\tilde{R}_{ik}}{r}$, Let $s = \frac{\tilde{R}_{kk}}{r}$

3: **for all** column indices $j \in$ list $\hat{P}$ **do**

4: Let vector $r\_h_j = c * \tilde{R}_{kj} + s * \tilde{R}_{ij}$, Let vector $r\_l_j = -s * \tilde{R}_{kj} + c * \tilde{R}_{ij}$
Let vector $q\_h_j = c * q_{kj} + s * q_{ij}$, Let vector $q\_l_j = -s * q_{kj} + c * q_{ij}$

5: **end for**

6: Set row k of $\tilde{R} = r\_h^T$, Set row i of $\tilde{R} = r\_l^T$
Set row k of $\tilde{Q} = q\_h^T$, Set row i of $\tilde{Q} = q\_l^T$

7:

8: Let coefficient $b\_h = \tilde{Q}^T b_k$, Let coefficient $b\_l = \tilde{Q}^T b_i$
Set $\tilde{Q}^T b_k = c * b\_h + s * b\_l$, Set $\tilde{Q}^T b_i = -s * b\_h + c * b\_l$

9: **end for**

10: Remove i from list $\hat{P}$

We note that more robust methods for computing the rotation coefficients c, r are in ref. [10]. The time-complexity for our downdate procedure is $O(nm)$.

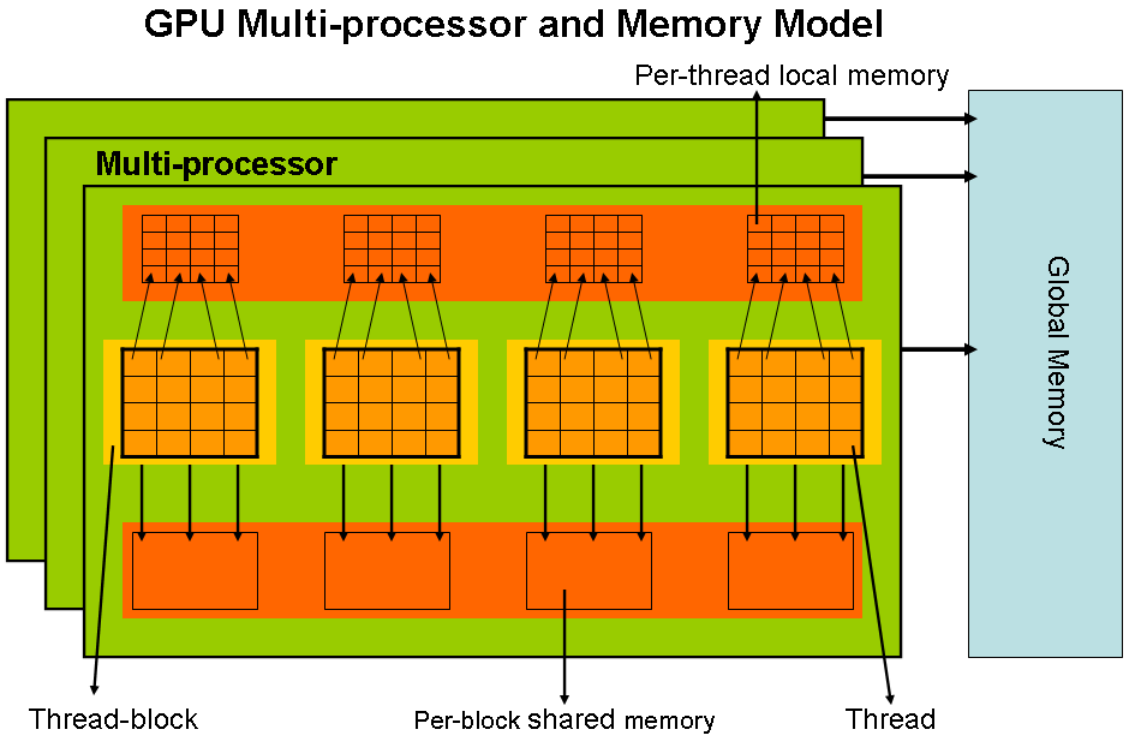
4. General Purpose GPUs. Recent advances in general purpose graphics processing units (GPUs) have given rise to highly programmable architectures designed with parallel applications in mind. Moreover, GPUs are considered to be typical of future generations of highly parallel, multi-threaded, multi-core processors with tremendous computational horsepower. They are well-suited for algorithms that map to a Single-Instruction-Multiple-Thread (SIMT) architecture. Hence, GPUs achieve a high arithmetic intensity (ratio of arithmetic operation to memory operations) when performing the same operations across multiple threads on a multi-processor.

The NVIDIA GPU is a set of multiprocessors, each with set of 8 scalar-processors (SP) with a Single-Instruction-Multiple-Data (SIMD) architecture. Hardware multi-threading under the SIMT architecture allows each thread to be map to a single SP where the thread executes independently with its own instruction address and register state. The multiprocessor SIMT unit schedules groups of 32 threads called a *warp* where a common instruction is executed at the same time. If threads of the same warp diverge via a data-dependent conditional branch, then threads along the separate branches are serially executed until they converge back to the same execution path.

All multiprocessors may access a global memory pool, which can be as high as 4GB in current NVIDIA chips. The 8 SP in each multiprocessor have access to a local shared memory space and a local set of registers. Note that the access times to each type of memory in the device are significantly different. A single warp access to global memory has a 400 to 600 clock cycles latency if coalesced. See ref. [18] for coalesced global memory accesses on the GPU. Accesses from the cached constant and texture memory, which can be written to from the host, are cheaper. Accesses to shared memory and per-processor register memory normally adds zero extra clock cycles to the instruction.

To a programmer using NVIDIA's Compute Unified Device Architecture (CUDA) in ref. [18], the threads are organized into thread-blocks, which in turn are arranged into grids. A thread-block refers to a 1-D or 2-D patch of threads that are executed on a single multiprocessor. The thread-blocks independently schedule warps and do not synchronize until each thread completes its task. A maximum of 512 threads can be housed in a single thread-block on current GPUs. Conceptually, each thread performs its operations independently and halts when a fast synchronization barrier is reached. Thus, threads in the same thread-block may coordinate on similar tasks while the thread-blocks remain asynchronous.

FIG. 4.1. *GPU Multiprocessor and Memory Model*



While an efficient algorithm on sequential processors must reduce the number of computations, parallel algorithms on GPUs must limit data-dependent operations and intermediate memory storage. Data dependencies increase the number of barrier synchronizations required between threads. Memory access to temporary data stored in global memory is costly.

4.1. GPU QR. Full QR decompositions have been implemented on the GPU in ref. [19], [13]. Ref. [13] evaluates that the blocked MGS and Givens rotation methods compared to the Householder reflections are ill-suited for large systems on GPUs because of instability and synchronization overhead. However, for moderate sized systems that can fit on a single multi-processor or thread-block on the GPU, synchronization is less costly. In this section, we review the parallelism in the MGS and Givens rotations methods in our update and downdate procedures.

For MGS, most of the work in eq. (2.2) (2.3) lies in computing vector inner products, scalar-vector products, and vector-vector differences. These vector based operations lead to a one-to-one mapping between vector coordinates and threads in a block. Such operations are efficiently computable via parallel reduction techniques from ref. [3] in section 4.2. For algorithm 2, we may parallelize all four inner products on lines 3, 4, 6, and 8 in $log(n)$ time each. The inner loop runs for $\hat{P}$ or at most m times. Thus, the parallel time-complexity for our update procedure is $O(m * log(n))$.

For Givens rotation, eq. (2.5) gives a one-to-one mapping between the coordinates of a row vector and threads in a block for computing R. Computing $Q^T b$ follows a similar relation. Lastly, a single thread may compute the coefficients c, s to be broadcast across all threads in the same block. For algorithm 3, the inner loop from lines 3 to 5 updates relation (2.5) for both $\tilde{R}$ and $\tilde{Q}$ in parallel $O(1)$ time. Writing row data in lines 6 and updating $\tilde{Q}^T b$ are thread-independent and performed in $O(1)$ time. Thus, the parallel time-complexity of the downdate procedure is $O(m)$.

4.2. GPU Implementation. Coalesced global memory access on the GPU is necessary when we update $\hat{Q}$ and $\hat{R}$. First, we store $\hat{Q}$ as its transpose so that access to column vectors are coalesced in row-oriented programming models. Second, we do not explicitly store $\hat{Q}$ and $\hat{R}$ in the dense format presented in the paper. Instead, we store

the two matrices in the same block-allocated format as matrix A and impose the column ordering from list $\hat{P}$ for the update, downdate, and back-substitution steps. This is to avoid any physical shifts of column vectors in global memory. Rather, we perform a parallel shift of the elements in list $\hat{P}$ when an element is removed from the passive-set. The following NVIDIA Compute Unified Device Architecture device function performs a parallel left shift of all elements to the right of index mark in 2 operations and 1 thread-synchronization.

Algorithm 4 CUDA parallel left shift routine

```
__device__ void shiftLeft512( int P[], short kCols, short mark, short tID){
        int i;
        bool toShift = (tID >= mark && tID < kCols - 1);
        if(toShift) i = P[tID + 1];
        __syncthreads();
        if(toShift) P[tID] = i;
}
```

Parallel reductions are often performed on the GPU in place of common vector-vector operations in ref. [12]. The following CUDA device function sums 512 elements in 9 operations and 5 thread-synchronizations:

Algorithm 5 CUDA parallel floating-point summation routine

```
__device__ float reduce512( float smem512[], unsigned short tID){
        __syncthreads();
        if(tID < 256)   smem512[tID] += smem512[tID + 256];
        __syncthreads();
        if(tID < 128)   smem512[tID] += smem512[tID + 128];
        __syncthreads();
        if(tID < 64)    smem512[tID] += smem512[tID + 64];
        __syncthreads();
        if(tID < 32){
                smem512[tID] += smem512[tID + 32];
                smem512[tID] += smem512[tID + 16];
                smem512[tID] += smem512[tID + 8];
                smem512[tID] += smem512[tID + 4];
                smem512[tID] += smem512[tID + 2];
                smem512[tID] += smem512[tID + 1];
        }
        __syncthreads();
        return smem512[0];
}
```

In the parallel summation, each of the 512 threads has reserved a memory slot in shared memory. The unique thread ID or tID also denotes the corresponding data index in the shared memory array. At each step, half the threads from the previous step sum up the data entries stored in the other half of shared memory. The process continues until index 0 in the shared memory array contains the total summation.

We note that modern GPUs contain a grid of multi-processors. Each multi-processor may map to a different linear system of equations and independently run in parallel without memory and thread conflicts. If there are more systems then the number of multi-processors, then a queue is formed awaiting the next available multi-processor.

5. Application. In the discrete-time deconvolution problem, a known signal s is convolved with an unknown filter x that satisfies certain characteristics.

$$\begin{aligned} y(t) = s(t) * x(t) &= \int_{-\infty}^{\infty} s(\tau)x(t-\tau)\,d\tau = \int_{-\infty}^{\infty} x(\tau)s(t-\tau)\,d\tau \\ &= \sum_{\tau=-\infty}^{\infty} x(\tau)s(t-\tau)\,d\tau = \sum_{\tau=1}^{n} x(\tau)s(t-\tau)\,d\tau \end{aligned} \tag{5.1}$$

where t is the sample's time, $y(t)$ the observed signal, and n the number of samples over time. To solve for the unknown filter x, we rewrite eq. (5.1) as a linear system of equations $Ax = b$, where A is Toeplitz.

$$
\begin{aligned}
A &= \begin{bmatrix} s(0) & s(-1) & \cdots & s(-(n-1)) \\ s(1) & s(0) & \cdots & s(-(n-2)) \\ \vdots & \vdots & \ddots & \vdots \\ s(n-1) & s(n-2) & \cdots & s(0) \end{bmatrix} \\
x &= \begin{bmatrix} x(1) & \cdots & x(n) \end{bmatrix}^T \\
b &= \begin{bmatrix} y(1) & \cdots & y(n) \end{bmatrix}^T \\
Ax &= b
\end{aligned}
\tag{5.2}
$$

Efficient algorithms for the deconvolution problem which either exploit the simple structure of the convolution in Fourier space, or which exploit the Toeplitz structure of the matrix are available in ref. [4], [7]. However, if x is known to be non-negative and the data $y(t)$ is corrupted by noise, then we may treat the deconvolution as a NNLS problem to guarantee a non-negative solution.

6. Experiments. We solve for a non-negative deconvolution problem using terrain laser imaging data sets obtained from[2] for $n = 432$. For a comparison, Matlab's lsqnonneg function implements the same active-set algorithm but with either the Intel MKL pseudo-inverse or the backslash QR to solve the least squares sub-problem.

Number of Systems	1	24	48	96	192	256
Active-set iterations	108	1477	2806	5695	12067	16176
Least squares updates + downdates	122	1697	3212	6529	13906	18681
Matlab 2006b lsqnonneg: p-inv	29.54	314.02	578.84	1187.99	2565.02	3447.56
Matlab 2008b lsqnonneg: QR	.24	2.15	3.86	8.10	17.82	23.46
GPU lsqnonneg: QR (up/downdate)	.29	.29	.55	1.06	2.09	2.76
Matlab FNNLS [5]	9.30	145.54	282.95	569.47	1168.40	1557.40
Matlab Interior-points method [15]	8.32	141.14	266.49	547.06	1124.60	1515.70
Matlab PQN-NNLS [14]	2.30	90.55	166.92	267.64	583.37	759.48

FIG. 6.1. *Runtime (seconds) comparisons of active-set algorithms for signal deconvolution on* [1] *for* $n = 432$*. Signal data taken from LVIS Sierra Nevada, USA (California, New Mexico), 2008.*

We note that both the Matlab lsqnonneg functions and our algorithm obtained the same solutions in the same number of iterations for all the sample data. For a single system of linear equations, our GPU version achieved over a 100x speed-up over the Matlab lsqnonneg implementation using pseudo-inverse. A more recent version of Matlab's lsqnonneg uses the backslash operator with the Intel MKL QR to solve the least squares sub-problems. While the version does not perform column-wise updates and downdates to the QR matrices, the optimized Intel MKL code is faster than all other algorithms tested for a single NNLS problem. For multiple problems, our algorithm scales with the number of multi-processors on the GPU and so outperforms the fastest Matlab implementation by 7-8x. This factor should increase in future GPU generations.

7. Concluding Remarks. In this paper, we have presented an efficient procedure for solving least squares sub-problems in the active-set algorithm. We have shown that prior QR decompositions may be used to update and solve similar least squares sub-problems. Furthermore, a stack-based ordering of variables in the passive-set yielded fewer computations in both the update and downdate steps. This has lead to substantial speed-ups over existing methods in our GPU implementation. Applications to satellite based terrain mapping and in microphone array signal processing are being worked on.

[1] Intel Core(TM)2 CPU 6600 @ 2.40GHz, 2.00 GB of Ram, NVIDIA GeForce GTX 275 (24 multi-processors).

[2] https://lvis.gsfc.nasa.gov/index.php

REFERENCES

[1] S. Bellavia, M. Macconi, and B. Morini, An interior point newton-like method for nonnegative least squares problems with degenerate solution, Numerical Linear Algebra with Applications, 13, pp. 825-846, 2006.

[2] M. H. van Benthem , M. R. Keenan, Fast algorithm for the solution of large-scale non-negativity-constrained least squares problems. Journal of Chemometrics, Vol. 18, p.441-450, 2004.

[3] G. E. Blelloch, Prefix Sums and Their Applications. Technical Report CMU-CS-90-190, Carnegie Mellon University, Pittsburgh, PA, Nov. 1990.

[4] A. W. Bojanczyk, R. P. Brent and F. R. de Hoog, QR factorization of Toeplitz matrices, Numerische Mathematik 49 (1986), 81-94.

[5] R. Bro, S. D. Jong, A fast non-negativity-constrained least squares algorithm, Journal of Chemometrics, Vol. 11, No. 5, pp. 393-401, 1997.

[6] M. Catral, L. Han, M. Neumann, and R. Plemmons, on reduced rank nonnegative matrix factorization for symmetric nonnegative matrices, Lin. Alg. Appl., 393:107-126, 2004.

[7] R. H. Chan, J. G. Nagy, and R. J. Plemmons, FFT-based preconditioners for Toeplitz-block least squares problems. SIAM J. Numer. Anal. 30 (1993), pp. 17401768.

[8] D. Chen, R.J. Plemmons, Nonnegativity Constraints in Numerical Analysis, Symp on the Birth of Numerical Analysis, Leuven, Belgium, October 2007.

[9] V. Franc, V. Hlavc, and M. Navara, Sequential coordinate-wise algorithm for non-negative least squares problem, Research report CTU-CMP-2005-06, Center for Machine Perception, Czech Technical University, Prague, Czech Republic, 2005.

[10] G. H. Golub and C. F. Van Loan, Matrix Computations (3rd ed.), Johns Hopkins Press, 1996. (pp. 223-236).

[11] S. Hammarling, C. Lucas, Updating the QR factorization and the least squares problem, MIMS EPrint, Manchester Institute for Mathematical Sciences, University of Manchester, Manchester, 2008.

[12] M. Harris, Optimizing parallel reduction in cuda (2007).

[13] A. Kerr, D. Campbell, and M. Richards, QR decomposition on GPUs, in GPGPU-2: Proceedings of 2nd Workshop on General Purpose Processing on Graphics Processing Units. New York, NY, USA: ACM, 2009, pp. 7178.

[14] D. Kim, S. Sra, and I. S. Dhillon, A New Projected Quasi-Newton Approach for the Non-negative Least Squares Problem. Technical Report TR-06-54, Computer Sciences, The Univ. of Texas at Austin, 2006.

[15] S. J. Kim, K. Koh, M. Lustig, S. Boyd, and D. Gorinevsky, An Interior-Point Method for Large-Scale l1-Regularized Least Squares, IEEE Journal of Selected Topics in Signal Processing 1 (2007), no. 4, 606617.

[16] H. W. Kuhn and A. W. Tucker, Nonlinear programming. In J. Neyman, editor, Proceedings of the Second Berkeley Symposium on Mathematical Statistics and Probability, pages 481-492, 1951. University of California Press, Berkeley, California.

[17] C. L. Lawson and R. J. Hanson, Solving Least Squares Problems, PrenticeHall, 1987.

[18] NVIDIA, NVIDIA CUDA Programming Guide 2.3.

[19] V. Volkov and J. W. Demmel, Benchmarking GPUs to tune dense linear algebra, SC08, 2008.