SIAM J. SCI. COMPUT.
Vol. 33, No. 5, pp. 2848–2863

EFFICIENT PARALLEL NONNEGATIVE LEAST SQUARES ON MULTICORE ARCHITECTURES*

YUANCHENG LUO† AND RAMANI DURAISWAMI†

Abstract. We parallelize a version of the *active-set* iterative algorithm derived from the original works of Lawson and Hanson [*Solving Least Squares Problems*, Prentice-Hall, 1974] on multicore architectures. 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 each iteration, to account for inserted and removed columns. We use a reordering strategy of the columns in the decomposition to reduce computation and memory access costs. We consider graphics processing units (GPUs) as a new mode for efficient parallel computations and compare our implementations to that of multicore CPUs. Both synthetic and nonsynthetic data are used in the experiments.

Key words. nonnegative least squares, active-set, QR updating, parallelism, multicore, graphics processing unit, deconvolution

AMS subject classifications. 15A06, 15A23, 65Y05, 65Y20

DOI. 10.1137/100799083

1. Introduction.

A central problem in data modeling is the optimization of underlying parameters specifying a linear model used to describe observed data. The underlying parameters of the model form a set n variables in an $n \times 1$ vector $x = \{x_1, \ldots, x_n\}^T$. The observed data is composed of m observations in an $m \times 1$ vector $b = \{b_1, \ldots, b_m\}^T$. Suppose that the observed data are linear functions of the underlying parameters in the model; then the functions' values at data points may be expressed as an $m \times n$ matrix A, where $Ax = b$ describes a linear mapping from the parameters in x to the observations in b.

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

Sometimes, the underlying parameters are constrained to be nonnegative 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 nonnegativity, leading to the nonnegative least squares (NNLS) problem considered in this paper.

The seminal work of Lawson and Hanson [19] provides the first widely used method for solving this NNLS problem. This algorithm, later referred to as the

*Received by the editors June 16, 2010; accepted for publication (in revised form) May 2, 2011; published electronically October 27, 2011. The authors received NVIDIA and NSF award 0403313 for facilities and ONR award N00014-08-10638 for support.

http://www.siam.org/journals/sisc/33-5/79908.html

†Department of Computer Science, University of Maryland, Room 3368 A.V. Williams, College Park, MD 20740 (yluo1@umd.edu, ramani@umiacs.umd.edu).

active-set method, partitions the set of parameters or variables into the active- and passive-sets. The active-set contains the variables with values forcibly set to zero and which 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 subproblem that is solvable via QR.

In many signal processing applications, NNLS problems in a few hundred to a thousand variables arise. In time-delay estimation, for example, multiple systems are continuously stored or streamed for processing. A parallel method for solving multiple NNLS problems would enable online applications, in which the estimation can be performed as data is acquired. Motivated by such an application, we develop an efficient algorithm and its implementations on both multicore CPUs and modern graphics processing units (GPUs).

Section 1.2 summarizes alternative solutions to the NNLS problem. Section 2 establishes notation and formally describes the active-set algorithm. Section 3 presents a new method for updating the QR decompositions for the active-set algorithm. Sections 4–5 describe parallelism on multicore CPUs and GPU like architectures. Section 6 provides a motivating application from remote estimation, and section 7 compares the GPU and CPU results from experiments.

1.1. Nonnegative least squares. We formally state the NNLS problem. Given an $m \times n$ matrix $A \in \mathbb{R}^{m \times n}$, find a nonnegative $n \times 1$ vector $x \in \mathbb{R}^n$ that minimizes the functional $f(x) = \frac{1}{2}\|Ax - b\|^2$; i.e.,

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

The Karush–Kuhn–Tucker (KKT) conditions necessary for an optimal constrained solution to an objective function $f(x)$ can be stated as follows [18]. Suppose $\hat{x} \in \mathbb{R}^n$ 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), 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 [9]. The first widely used algorithm, proposed by Lawson and Hanson in [19], is the active-set method that we implement on the GPU. Although many newer methods have since surpassed the active-set method for large and sparse matrix systems from our survey, the active-set method remains competitive for *small* to moderate sized systems with unstructured and dense matrices.

In [6], improvements to the original active-set method are developed for the fast NNLS (FNNLS) variant. By reformulating the normal equations that appear in the pseudoinverse for the least squares subproblem, the cross product matrices A^TA and A^Tb can be precomputed. This contribution leads to significant speedups in the presence of multiple right-hand sides. In [2], further redundant computations are avoided by grouping similar right-hand-side observations that would lead to similar pseudoinverses.

A second class of algorithms is iterative optimization methods. Unlike the active-set approach, these methods are not limited to a single active constraint at each iteration. In [17], a projective quasi-Newton NNLS approach uses gradient projections to avoid precomputing $A^T A$ and nondiagonal gradient scaling to improve convergence and reduce zigzagging. Another approach in [10] produces a sequence of vectors optimized at a single coordinate with all other coordinates fixed. These vectors have an efficiently computable analytical solution that converges to the solution.

Other methods outside the scope of this review include the principal block pivoting method for large sparse NNLS in [7], and the interior point Newton-like method in [1], [16] for moderate and large problems.

2. Active-set method. Given a set of m linear equations in n unknowns which are constrained to be nonnegative, let the active-set Z be the subset of variables which violate the nonnegativity constraint or are zero and the passive-set P be the variables with positive values. 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 an unconstrained least squares problem using the variables from the passive-set.

ALGORITHM 1. Active-set method for NNLS [19]

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

```
1: while true do
2:   Compute negative 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^{m×*} 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
```

In Algorithm 1, the candidate active-set Z is updated by first moving the largest positive component variable in the negative gradient w to the passive-set (line 5). This selects the component with the most negative gradient that reduces the residual 2-norm. The variables in the passive-set form a candidate linear least squares system $A^P y = b$, where matrix A^P contains the column vectors in matrix A that correspond to

indices in the passive-set (lines 7, 8). At each iteration, the feasibility vector x moves towards the solution vector y while preserving nonnegativity (line 11). Convergence to the optimal solution is proven in [19].

The termination condition (line 3) checks if the gradient is strictly positive or if the residual can no longer be minimized. At termination, the following relations satisfy the optimality conditions in (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 subproblem (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 subproblem $A^P y = b$. As discussed previously, the cost of solving the unconstrained least squares subproblem is $O(mn^2)$ via QR. If there are k iterations, then the cost of k independent decompositions is $O(kmn^2)$. However, the decompositions at each iteration share a similar structure in matrix A^P, and this can be taken advantage of. We observe the following properties of matrix A^P as the iterations proceed:

1. The active- and passive-sets generally exchange a single variable per iteration; one column is added or removed from matrix A^P.
2. Most exchanges move variables from the active-set into the passive-set; early iterations add variables to an empty passive-set to build the feasible solution, while later iterations add and remove variables to refine the solution.

Hence, we develop a general method for QR column updating and downdating that takes advantage of the pattern of movement between variables in the active- and passive-sets. To achieve real-time and online processing, the method must be parallelizable on GPUs or other multicore architectures. We note that the improvements made to the active-set NNLS proposed in [6], [2] do not apply to our problem and, moreover, do not account for possible efficiencies suggested by the observations above.

3. Proposed algorithm. The first property of matrix A^P suggests that a full $A^P = QR$ decomposition is unnecessary. Instead, we consider an efficient QR column updating and downdating method.

1. QR Updating: A new variable added to set P expands matrix A^P by a single column. Update previous matrices Q, R with this column insertion.
2. QR Downdating: The removal of a variable from set P shrinks matrix A^P by a single column. Downdate previous matrices Q, R with this column deletion.

The second property of matrix A^P suggests that we can optimize the cost for QR updating in terms of floating point operations (flops) and column or row memory accesses. We observe that many QR updating methods minimize computations when inserting columns at the right-most index. Our method takes advantage of this by maintaining a separate ordering for the columns of matrix A^P by the relative times of insertions and deletions across iterations. That is, a column insertion always appends to the end of a reordered matrix $\hat{A^P}$. We describe the effects of the reordering strategy for various updating methods in sections 3.1–3.3. We also show that the modified Gram–Schmidt (MGS) and Givens rotation methods are the most cost efficient with respect to the reordering strategy for overdetermined and square systems.

3.1. *QR* updating by MGS. The reordering strategy allows a new column a_i from the matrix $A^P = [a_1, \ldots, a_i, \ldots, a_n]$ to be treated as the right-most column in the decomposition. We define list $\hat{P}$ as an ordered list of column indices from set P

such that the associated column $\hat{p}_{i-1}$ is added in a prior iteration to column $\hat{p}_i$. The reordered decomposition $\hat{A^P} = \hat{Q}\hat{R}$ is

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

where $\hat{Q}$ is an $m \times i$ matrix and $\hat{R}$ is an $i \times i$ matrix. To compute column $q_{\hat{p}_i}$, we orthogonalize the inserted column a_i with all the previous columns in matrix $\hat{Q}$ via vector projections. To compute column $r_{\hat{p}_i}$, we take the inner products between column a_i and columns in $\hat{Q}$, or the equivalent matrix-vector product $\hat{Q}^T a_i$. Both quantities are found using the MGS procedure in Algorithm 2.

ALGORITHM 2. Reordered MGS QR Column Updating

Require: Reordered list $\hat{P}$ contains the elements in set P, index i the variable added to set 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: $\hat{Q}^T b_i = \langle q_i, b \rangle$
9: Add i to list $\hat{P}$

With the reordering strategy in Algorithm 2, a new column a_i is always inserted in the right-most position of matrix $\hat{A^P}$. The number of columns read from memory in matrix $\hat{Q}$ is the size of set P, denoted as $\ell \leq n$, and is used to form column q_i. The number of column memory writes per step is two, as column q_i appends to matrix $\hat{Q}$ and the projection step writes a single column to matrix $\hat{R}$. Updating matrices $\hat{Q}$ and $\hat{R}$ requires $6m\ell + 3m + 1$ flops. The asymptotic complexity is $O(mn)$.

Without the reordering strategy, column a_i can be inserted into the middle of matrix $\hat{A^P}$. This requires computing column q_i and reorthogonalizing the $\ell - i$ columns to its right. The memory access costs of computing q_i is i number of column reads from matrix $\hat{Q}$ and two column writes to matrices $\hat{Q}$ and $\hat{R}$. The reorthogonalization costs of column q_j, where $j > i$, is equivalent to a new column insertion into matrix A^P. This is because the MGS method does not compute the null-space of the basis vectors in matrix $\hat{Q}$. Orthogonalizing columns q_j and q_{j+1} with respect to column q_i does not preserve the orthogonality between q_j and q_{j+1}. Thus, each of the $\ell - i + 1$ columns must be reinserted with an additional $\ell(\ell - i + 1)$ column reads and $2(\ell - i + 1)$ column writes. Updating matrices $\hat{R}$ and $\hat{Q}$ requires a total of $(3m\ell + 3mi + 1)(\ell - i + 2)$ flops. The asymptotic complexity is $O(mn^2)$.

3.2. Alternative QR updating by rotations. Rotation-based methods for updating QR are possible. In [12], $\hat{Q}$ and $\hat{R}$ are treated as $m \times m$ matrices where matrix $\hat{Q}$ is initially the identity. When inserting column a_i, the method appends $m \times 1$ column vector $r_{\hat{p}_i} = \hat{Q}^T a_i$ to matrix $\hat{R}$. A series of rotation transformations

introduces zeros to rows $\{i+1, i+2, \ldots, m\}$ of column $r_{\hat{p}_i}$ to preserve the upper-triangular property. The rotation transformations then update the columns to the right of index i in matrices $\hat{R}$ and $\hat{Q}$. A similar step follows updating the right-hand side $\hat{Q}^T b_i$.

Without the reordering strategy, the costs of this rotation method depend on index i. Column $r_{\hat{p}_i}$ requires $m-i$ rotation transformations. Each transformation requires two row memory reads and writes to matrix $\hat{R}$ and two column memory reads and writes to matrix $\hat{Q}$ for a total of $2(m-i)$. This is disadvantageous as the number of column and row accesses is bound by m and multiple columns and rows of matrices $\hat{R}$ and $\hat{Q}$ are modified. Updating matrix $\hat{Q}$ and $\hat{R}$ requires $6m(m-i)$ and $2m^2 + 8(m-i) + 6(\ell - i + 1)(\ell/2 - i/2 - 1)$ flops, respectively. The asymptotic complexity is $O(m^2 + n^2)$.

With the reordering strategy, index $i = \ell + 1$ and so many of the costs are reduced. There are no columns to the right of index i in matrix $\hat{R}$ so updating is limited to single column memory write of column $r_{\hat{p}_i}$. Updating matrix $\hat{Q}$ now requires $m - \ell - 1$ column reads and writes each while applying the transformations. Updating matrices $\hat{Q}$ and $\hat{R}$ requires $6m(m-\ell-1)$ and $2m^2 + 8(m-\ell-1)$ flops, respectively. The asymptotic complexity is $O(m^2)$.

3.3. Alternative *QR* updating by seminormal equations. The corrected seminormal equations (CSNE) can be used to update an $\ell \times \ell$ matrix $\hat{R}$ without the construction of matrix $\hat{Q}$. The stability analysis of this method is provided by [3]. With the reordering strategy, the problem treats $\hat{R}^T \hat{R} x = \hat{A^P} b$, where column $r_{\hat{p}_i}$ is computed by

$$
\begin{aligned}
\hat{R}^T \hat{R} z &= \hat{A^P} a_i, \\
s &= a_i - \hat{A^P} z, \\
\hat{R}^T \hat{R} \delta z &= s, \\
z &= z + \delta z, \\
r_{\hat{p}_i} &= \begin{bmatrix} \hat{R} z \\ \| \hat{A^P} z - a_i \| \end{bmatrix}.
\end{aligned} \tag{3.2}
$$

Although the method does not compute and store matrix $\hat{Q}$, it requires both row and column access to matrix $\hat{R}$ and more operations to produce column $r_{\hat{p}_i}$. Computing and correcting for vector z entails four back-substitutions using matrices $\hat{R}^T$, $\hat{R}$, and $\hat{A^P}$. All four back-substitutions require ℓ row or column memory reads from matrix $\hat{R}$ each. Two of the back-substitutions require m row memory reads from matrix $\hat{A^P}$. The total number of column and row memory reads in the method is $3m + 2\ell$ and one column memory write to update matrix $\hat{R}$. The entire procedure requires $6m^2 + 4m + 1 + 3\ell^2 + 3\ell$ flops. The asymptotic complexity is $O(m^2 + n^2)$.

Without the reordering strategy, the same CSNE method computes column $r_{\hat{p}_i}$ and a series of rotations introduces zeros below index i. The rotation transformations are then applied to the columns to the right of index i in matrix $\hat{R}$. This requires an additional $\ell - i$ row memory reads and writes to matrix $\hat{R}$ each and $6(\ell - i + 1)(\ell/2 - i/2 - 1) + 3(\ell - i)$ flops. The asymptotic complexity is $O(n^2)$. The costs for the updates are summarized in Table 3.1.

3.4. *QR* downdating by rotations. The reordering strategy is less applicable to the downdating scheme, as deleted columns may not be in the right-most index. Suppose that column a_i is removed from matrix $A^P = [a_1, \ldots, a_{i-1}, a_{i+1}, \ldots, a_n]$. Let

TABLE 3.1

Costs for QR updating/downdating methods with respect to the reordering strategy. The rotation and CSNE methods have flops of order m^2. For overdetermined and square systems, where $\ell \leq n \leq m$, this quantity is minimized for the MGS method.

Algorithm	Col/row accesses	Up/down Q flops	Up/down R flops
MGS/up/reorder	$\ell+2$	$6m\ell+3m+1$	included in Q
MGS/up/unorder	$\ell(\ell-i+2)+2(\ell-i+2)$	$(3m\ell+3mi+1)(\ell-i+2)$	included in Q
Rot/up/reorder	$2(m-\ell-1)$	$6m(m-\ell-1)$	$2m^2+8(m-\ell-1)$
Rot/up/unorder	$4(m-\ell-1)$	$6m(m-i)$	$2m^2+8(m-i)+6(\ell-i+1)(\ell/2-i/2-1)$
CSNE/up/reorder	$3m+2\ell+1$	NA	$6m^2+4m+1+3\ell^2+3\ell$
CSNE/up/unorder	$3m+4\ell-2i+1$	NA	$6m^2+4m+1+3\ell^2+6(\ell-i+1)(\ell/2-i/2-1)+6\ell-3i$
Rot/down/NA	$4(\ell-i+1)$	$6m(m-i)$	$6(\ell-i)+6(\ell-i+1)(\ell/2-i/2-1)$

$\hat{p}_j$ be the corresponding column index in the ordered list. We consider the reformulation of (3.1) without column $\hat{p}_j$ as

$$\begin{aligned} \tilde{A^P} &= [a_{\hat{p}_1}, \ldots, a_{\hat{p}_{j-1}}, a_{\hat{p}_{j+1}}, \ldots, a_{\hat{p}_i}], \\ \tilde{Q} &= [q_{\hat{p}_1}, \ldots, q_{\hat{p}_{j-1}}, q_{\hat{p}_j}, q_{\hat{p}_{j+1}}, \ldots, q_{\hat{p}_i}], \\ \tilde{R} &= [r_{\hat{p}_1}, \ldots, r_{\hat{p}_{j-1}}, r_{\hat{p}_{j+1}}, \ldots, r_{\hat{p}_i}], \end{aligned} \tag{3.3}$$

where $\tilde{A^P} = \tilde{Q}\tilde{R}$, matrices $\tilde{A^P}$ and $\tilde{R}$ are missing column $\hat{p}_j$, and matrix $\tilde{Q} = \hat{Q}$ is unchanged. Column $q_{\hat{p}_j}$ still exists in matrix $\tilde{Q}$, and matrix $\tilde{R}$ is no longer upper-triangular as the columns to the right of index $\hat{p}_j$ have shifted left.

Observe that the right submatrix shifted in matrix $\tilde{R}$ has a Hessenberg form. In [11], a series of Givens rotations introduces zeros along the subdiagonal. However, this does not directly address the removal of column $\hat{p}_j$ in matrix $\tilde{Q}$. Instead, we apply a series of Givens rotations to introduce zeros along the jth row of matrix $\tilde{R}$. The rotations are applied to the right of column $\hat{p}_j$ in the transformation

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

$$(G_i G_{i-1} \ldots G_{j+2} G_{j+1} \tilde{R}) = \begin{bmatrix} \ddots & & & & & & \\ & * & * & * & * & & \\ & 0 & * & * & * & & \\ \cdots & 0 & 0 & 0 & 0 & \cdots & \\ & 0 & 0 & * & * & & \\ & 0 & 0 & 0 & * & & \\ & & & & & \ddots & \end{bmatrix}, \tag{3.4}$$

which preserves $\tilde{A^P} = \tilde{Q}\tilde{R}$ while introducing zeros along the jth row of matrix $\tilde{R}$ and modifying matrix $\tilde{Q}$. This enables both row j in the updated matrix $\tilde{R}$ and column j in the updated matrix $\tilde{Q}$ to be removed without violating matrix $\tilde{A^P}$. Vector $\tilde{Q}^T b$ is updated via a similar transformation.

We refer to [11] for precautions when computing the rotation coefficients c, r in Algorithm 3. When updating matrices $\tilde{R}$ and $\tilde{Q}$, a row or column is fixed so the transformation requires $2(\ell-i+1)$ row and column memory reads and writes each. Updating matrices $\tilde{Q}$ and $\tilde{R}$ requires $6m(m-i)$ and $6(\ell-i)+6(\ell-i+1)(\ell/2-i/2-1)$

ALGORITHM 3. Reordered QR Column Downdating with Givens Rotations

Require: Reordered list $\hat{P}$ contains the elements in set P, index i is the variable removed from set P

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

1: **for all** column indices k following ($\succ$) index $i \in$ list $\hat{P}$ **do**

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

3: $$\begin{bmatrix} \tilde{R}_{k,j\succeq k\in\hat{P}} \\ \tilde{R}_{i,j\succeq k\in\hat{P}} \end{bmatrix} = \begin{bmatrix} c & s \\ -s & c \end{bmatrix} \begin{bmatrix} \tilde{R}_{k,j\succeq k\in\hat{P}} \\ \tilde{R}_{i,j\succeq k\in\hat{P}} \end{bmatrix}$$

4: $$\begin{bmatrix} \tilde{Q}_{:,k} & \tilde{Q}_{:,i} \end{bmatrix} = \begin{bmatrix} \tilde{Q}_{:,k} & \tilde{Q}_{:,i} \end{bmatrix} \begin{bmatrix} c & -s \\ s & c \end{bmatrix}$$

5: Let coefficient $b\_h = \tilde{Q}^T b_k, \quad b\_l = \tilde{Q}^T b_i$

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

7: **end for**

8: Remove index i from list $\hat{P}$

flops, respectively. The asymptotic complexity is $O(m^2 + n^2)$. The costs for the downdates are summarized in Table 3.1.

4. Multicore CPU architectures. The multicore trend began as a response to the slowdown of Moore's Law while manufactures approached the limitations in single-core clock speeds. With additional cores added on chip, individual CPU threads can be assigned and processed by their own units in hardware. Thus, a single problem is decomposed and solved by several threads without overutilizing a single core. This gave multithreading an edge over traditional single-core processors as data and instruction level caches could be dedicated to a smaller subset of operations.

Such multiple-instruction-multiple-data (MIMD) architectures support task-level parallelism where each core can asynchronously execute separate threads on separate data regions. The individual cores are often superscalar and thus capable of processing out-of-order instructions in their pipeline. This allows multicore architectures to simulate data-level parallelism from single-instruction-multiple-data (SIMD) like architectures such as the GPU with added proficiency. Furthermore, multicore architectures have access to a common pool of main memory off-die and are capable of multilevel caching per core and per processor on-die. For both data- and task-level parallelism, this allows memory to be decomposed and cached on a per-core basis for efficient reuse.

Several application programming interfaces (APIs) and libraries take advantage of these shared memory multiprocessing environments for high performance computing. Open Message Passing (OpenMP) is an API based on fork-join operations where the program enters into a designated parallel region [22]. Each thread exhibits both task- and data-level parallelism as it independently executes code within a same parallel region. The Intel Math Kernel Library (MKL) is a set of optimized math routines with calls to Basic Linear Algebra Subprograms (BLAS) and Linear Algebra PACKage (LAPACK) libraries [14]. Many of its fundamental matrix and vector routines are blocked and solved across multiple threads.

4.1. CPU implementation. To exploit the advantages of multithreading, we adopt both the OpenMP API and the Intel MKL in the CPU implementation. One

way to map each linear system to a thread is to declare the entire NNLS algorithm within a parallel OpenMP region. That is, a specified fraction of threads execute NNLS on a mutually exclusive set of linear system of equations. The remaining threads are dedicated to the MKL in order to accelerate common matrix-vector and vector-vector operations used to solve the unconstrained least squares subproblem.

5. GPU architectures. Recent advances in general purpose 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, multithreaded, multicore 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 multiprocessor.

GPUs are often designed as a set of multiprocessors, each containing a smaller set of scalar processors (SP) with an SIMD architecture. Hardware multithreading under an SIMT architecture maps multiple threads to a single SP. A single SP handles the instruction address and register states of multiple threads so that they may execute independently. The multiprocessor's SIMT unit schedules batches of threads to execute a common instruction. If threads of the same batch diverge via a data-dependent conditional branch, then all the threads along the separate branches are serialized until they converge back to the same execution path.

GPUs have a hierarchical memory model with significantly different access times to each level. At the top, all multiprocessors may access a global memory pool on the device. This is the common space where input data is generally copied and stored from main memory by the host. It is also the slowest memory to access as a single query from a multiprocessor has a 400 to 600 clock cycle latency on a cache-miss. See [20] for a discussion on coalesced global memory accesses which read or write to a continuous chunk of memory at a cost of one query and implicit caching on the Fermi architecture. On the same level, texture memory is also located on the device but can only be written to from hosts. However, it is faster than global memory when access patterns are spatially local. On the next level, SPs on the same multiprocessor have access to a fast shared memory space. This enables explicit interthread communication and temporary storage for frequently accessed data. Constant memory, located on each multiprocessor, is cached and optimized for broadcasting to multiple threads. On the lowest level, an SP has its own private set of registers distributed amongst its assigned threads. The latency for accessing both shared and per-processor registers normally adds zero extra clock cycles to the instruction time. See Figure 5.1 for a visualization.

Programming models such as NVIDIA's Compute Unified Device Architecture (CUDA) [20] and OpenCL [21] organize threads into thread-blocks, which in turn are arranged in a two-dimensional (2D) grid. A thread-block refers to a 1D or 2D patch of threads that are executed on a single multiprocessor. These threads efficiently synchronize their instructions and pass data via shared memory. Instructions are generally executed in parallel until a conditional branch or an explicit synchronization barrier is declared. The synchronization barrier ensures that the thread-block waits for all its threads to complete its last instruction. Thus, two levels of data parallelism are achieved. The threads belonging to the same thread-block execute in lock-step as they process a set of data. Individual thread-blocks execute asynchronously but generally with the same set of instructions on a different set of data.

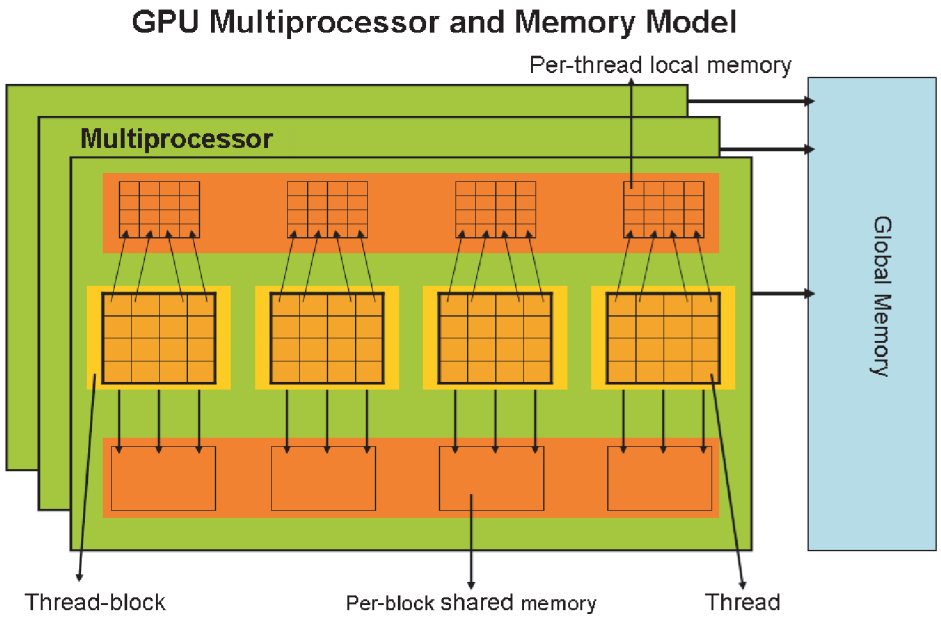

FIG. 5.1. *GPU multiprocessor and memory model.*

While efficient algorithms on sequential processors must reduce the number of computations and cache-misses, parallel algorithms on GPUs are more concerned with minimizing data dependencies and optimizing accesses to the memory hierarchy. Data dependency increases the number of barrier synchronizations amongst threads and is often subject to the choice of the algorithm. Memory access patterns present a difficult bottleneck on multiple levels. While latency is the first concern for smaller problems, we run into a larger issue with memory availability as the problem size grows. That is, the shared memory and register availability are hard limits that bound the size and efficiency of thread-blocks. A register memory bound per SP limits the number of threads assigned to each SP and so decreases the maximum number of threads and thread-blocks running per multiprocessor. A shared memory bound per multiprocessor limits the number of thread-blocks assigned to a multiprocessor and so decreases the total number of threads processed per multiprocessor.

5.1. GPU implementation. One way to map each linear system onto a GPU is to consider every thread-block as an independent vector processor. Each thread-block of size $m \times 1$ maps to the elements in a column vector and asynchronously solves for a mutually exclusive set of linear systems. The number of thread-blocks that fit onto a single multiprocessor depends on the column size m or the number of equations in the linear system. This poses a restriction on the size of linear systems that our GPU implementation can solve, as the maximum size m is constrained to a fraction of the amount of shared memory available per multiprocessor. Fortunately, this is not an issue for applications where m is small (500–1000) and the number of linear systems to be solved is large. However, for arbitrarily sized linear systems of equations, our GPU implementation is not generalizable. We note that this is not an algorithmic constraint but rather a design choice for our application. Our multicore CPU implementation of the same algorithm can solve for arbitrarily sized linear systems. We discuss the details of the GPU implementation in sections 5.2–5.3.

5.2. Parallelizing QR methods. Full QR decompositions on the GPU via blocked MGS, Givens rotations, and Householder reflections are implemented in [23] and [15]. While [15] cites that the blocked MGS and Givens rotation methods are ill suited for large systems on GPUs as they suffer from instability and synchronization overhead, we are interested in only the QR updating and downdating schemes for a large number of *small* systems. We show that it is possible for m threaded multiprocessors to efficiently perform the MGS updating and Givens rotation downdating steps.

For the MGS update step, most of the operations are formulated as vector inner products, scalar-vector products, and vector-vector summations. These operations lead to a one-to-one mapping between the $m \times 1$ column vector coordinates and the m threaded thread-block. Such operations are computable via parallel reduction techniques from [4]. In Algorithm 2, we parallelize all four inner products (lines 3, 4, 6, 8) in $\log m$ parallel time each. The inner loop iterates for ℓ or at most n times. Thus, we obtain an order reduction in parallel time-complexity to $O(n \log m)$.

For the Givens rotation downdate step, we obtain a one-to-many mapping between the $n \times 1$ row vector elements and the m threads in a thread-block for matrix R. We obtain a one-to-one mapping for the $m \times 1$ column vector elements in matrix Q. Computing vector $Q^T b$ follows a similar relation. For obtaining the rotation coefficients c, s, a single thread computes and broadcasts to the rest of the thread-block. In Algorithm 3, the inner loop (lines 3–4) updates both matrices $\tilde{R}$ and $\tilde{Q}$ in parallel $O(1)$ time. Writing row and column data and updating vector $\tilde{Q}^T b$ (line 6) are thread-independent and computable in $O(1)$ parallel time. Thus, we obtain an order reduction in parallel time-complexity to $O(n)$.

Parallel reductions are often performed on the GPU in place of common vector-vector operations using the prefix sum discussed in [13]. Algorithm 4 sums 512 elements in 9 parallel flops, 5 thread synchronizations, and 18 parallel shared memory accesses. Each of the 512 threads reserves a memory slot in shared memory. The unique thread ID or tID 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.

5.3. Memory usage. To take advantage of different access times on the GPU memory hierarchy, the input and intermediate data can be stored and accessed on different levels for efficient reuse. Local intermediate vectors can either be stored in shared memory or alternatively in dedicated registers spanning all threads in a thread-block. List $\hat{P}$ is stored in shared memory, as multiple threads require synchronization to update and downdate the same column. The right-hand-side vector $\hat{Q}^T b$ is stored in registers since no thread accesses elements outside its one-to-one mapping in the update and downdate steps.

Global memory accesses on the GPU are unavoidable for updating large matrices $\hat{Q}$ and $\hat{R}$. We store matrix $\hat{Q}^T$ so that column vector accesses are coalesced in row-oriented programming models and matrix $\hat{R}$ as the Givens rotations update the rows. Matrices $\hat{Q}$ and $\hat{R}$ are stored in place unlike the compact format in (3.1), (3.3). We allocate $m \times n$ blocks of global memory and use the reordered list $\hat{P}$ to associate column and row indices for the update and downdate steps. This is to avoid any physical shifts of column vectors in global memory. Rather, we parallel shift the list $\hat{P}$ when a variable is removed from the passive-set.

ALGORITHM 4. CUDA parallel floating-point summation routine [13]

```
__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];
}
```

The MGS update step reads ℓ number of columns in matrix $\hat{Q}$ from global memory into registers. Computing inner products and vector norms during the projections requires an intermediate shared memory vector for the parallel reduction function. The new column for matrix $\hat{R}$ is locally stored in registers before updated to global memory. A single element for vector $\hat{Q}^T b$ is updated and written to shared memory. The total number of parallel shared memory accesses is $39\ell + 2$. The total number of parallel global memory accesses is $\ell + 2$.

The Givens rotation downdate step accesses two columns of matrix $\tilde{Q}$ and two rows of matrix $\tilde{R}$ for each of the $\ell - i$ transformations. Since row i of matrix $\tilde{R}$ and column i of matrix $\tilde{Q}$ are fixed across transformations, they are stored and updated in shared memory. The other row and column are directly updated in global memory. Updating vector $\hat{Q}^T b$ requires two shared memory reads and writes. The total number of parallel shared memory accesses is $2(\ell - i) + 2$. The total number of parallel global memory accesses is $2(\ell - i + 1)$.

6. Application. In remote sensing, a discrete-time deconvolution recovers a signal x that has been convolved with a transfer function s. The known signal s is often convolved with an unknown signal x that satisfies certain characteristics.

$$
\begin{aligned}
y(t) = s(t) * x(t) &= \int_{-\infty}^{\infty} s(\tau)x(t-\tau)\,d\tau = \int_{-\infty}^{\infty} x(\tau)s(t-\tau)\,d\tau \\
&= \sum_{\tau=-\infty}^{\infty} x(\tau)s(t-\tau)\,d\tau = \sum_{\tau=1}^{n} x(\tau)s(t-\tau)\,d\tau,
\end{aligned} \tag{6.1}
$$

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 signal x, we rewrite (6.1) as the following linear

system of equations $Ax = b$, where A is a Toeplitz matrix:

$$
\begin{aligned}
A &= \begin{bmatrix} s(0) & s(-1) & \cdots & s(-(n-1)) \\ s(1) & s(0) & \cdots & s(-(n-2)) \\ \vdots & \vdots & \ddots & \vdots \\ s(n-1) & s(n-2) & \cdots & s(0) \end{bmatrix}, \\
x &= \begin{bmatrix} x(1) & \cdots & x(n) \end{bmatrix}^T, \\
b &= \begin{bmatrix} y(1) & \cdots & y(n) \end{bmatrix}^T, \\
Ax &= b.
\end{aligned} \tag{6.2}
$$

Efficient algorithms for the deconvolution problem, which either exploit the simple structure of the convolution in Fourier space or exploit the Toeplitz structure of the matrix, are available in [5], [8]. However, if signal x is known to be nonnegative and the data $y(t)$ is corrupted by noise, then we may treat the deconvolution as a NNLS problem to ensure nonnegativity.

7. Experiments. As a baseline, we note that Matlab's lsqnonneg function implements the same active-set algorithm but with a full QR decomposition for the least squares subproblem. Matlab 2009b and later versions use Intel's MKL with multithreading to resolve the least squares subproblems. For a better comparison, we first port the lsqnonneg function into native C-code with calls to multithreaded MKL BLAS and LAPACK functions. The results from this implementation (CPU lsqnonneg) show a 1.5–3x speedup over the Matlab lsqnonneg function in our experiments. Next, we apply our updating and downdating strategies with column reordering using MKL BLAS functions to the CPU version. The results from this second implementation (CPU NNLS) show a 1–3x speedup that depends on the number of column updates and downdates. Last, we compare the lsqnonneg variants to alternative NNLS algorithms from literature.

To compare GPU implementation with the multithreaded CPU variants, we begin timing the point of entry and exit out of the GPU kernel function. Memory transfer and preprocessing times in the case of nonsynthetic data are omitted. Both the GPU and CPU variants also obtain identical solutions subject to rounding error within the same number of iterations for all data sets. We find that for a fewer number of linear systems, the CPU implementations outperform the GPU as only a fraction of the GPU cores are utilized. When the number of linear systems surpasses the number of multiprocessors, the GPU scales better on an order of 1–3x than our fastest CPU implementation.

For reference, we use a Dual Quad-Core Intel(R) Xeon(R) X5560 CPU @ 2.80GHz (8 cores) for testing our CPU implementations. The CPU codes compiled under both Intel icc 11.1, gcc 4.5.1, and linked to MKL 10.1.2.024 yield similar results for 8 run-time threads. The codes tested between Matlab 2010b and 2009b also yield comparable results. Mixing the number of threads assigned between OpenMP and MKL did not have a large impact on our system. We use a NVIDIA Tesla C2050 (448 cores across 14 multiprocessors) and codes compiled under CUDA 3.2 for the GPU implementation and testing.

TABLE 7.1
Run-time (seconds) comparisons of NNLS and lsqnonneg variants for signal deconvolution. Signal and observation data taken from LVIS Sierra Nevada, USA (California, New Mexico), 2008.

Number of systems	1	24	48	96	192
Number of updates	108	1477	2806	5695	12067
Number of downdates	14	220	406	834	1839
GPU NNLS	0.2172	0.2238	0.2356	0.4508	0.7203
CPU NNLS	0.0186	0.1636	0.2990	0.5989	1.2489
CPU lsqnonneg	0.0770	0.7163	1.2908	2.6225	5.5460
Matlab lsqnonneg	0.1342	1.1236	2.0152	4.1268	8.7176
Matlab FNNLS [6]	0.0493	0.5936	1.1135	2.2639	4.6148
Matlab interior-points [16]	8.3642	135.7896	261.3665	528.4468	1082.8714
Matlab PQN-NNLS [17]	1.4161	138.2953	217.6929	479.9867	862.9674

7.1. Nonsynthetic test data. For real-world data, we use terrain laser imaging sets obtained from the NASA Laser Vegetation Imaging Sensor (LVIS).[1] Each data set contains multiple 1D Gaussian-like signals s and observations of total return energy b of size $m = n = 432$. In this deconvolution problem, the transfer functions s represent the single impulse energy fired over time on ground terrain and the observed signals b produce a waveform that indicates the reflected energy over time. The signals s are generally 15–25 samples wide so the computed matrices A are Toeplitz banded and sparse. NNLS solves for corresponding pairs of matrix A and vector b to obtain the sparse nonnegative solutions x that represent the times of arrival for a series of fired impulses. This estimates the ranges or distances to a surface target.

For comparing the NNLS methods, we record the run-times in relation to the number of column updates and downdates for the least squares subproblem. The results from CPU NNLS in Table 7.1 show an 11x speedup over the GPU implementation when solving for a single system. This is due to the underutilization of cores in all but the multiprocessor currently assigned to the linear system of interest. For a larger number of systems, the GPU results show a 1–2x speedup over CPU NNLS due in part to the larger number of processing units suited for vector operations in the algorithm. The results between CPU NNLS and CPU lsqnonneg show the performance gains from fewer flops and memory accesses attained by the column reordering, updating, and downdating strategies.

7.2. Synthetic test data. For the first set of synthetic data, we generate mean shifted 1D Gaussians with $\sigma = 4.32$ to store as columns in matrix A of size $m = n = 512$. In this Gaussian fitting problem, each system uses the same matrix A but with nonnegative random vectors b. The choice of the σ parameter ensures that the mean shifted Gaussians are not too wide as to allow early convergence and not too narrow as to locally affect only a few variables. Furthermore, matrix A is now considered dense and vectors b no longer reflect real-world values. We expect the average number of iterations or column updates and downdates to exceed that of the real-world data cases.

The total speedup of GPU NNLS over the CPU variants are more pronounced (3x compared to CPU NNLS, 23x compared to CPU lsqnonneg) in Table 7.2. The larger ratio of column downdate to update steps suggests that our reordering strategy and fast Givens rotation method in the downdating step outperforms the lsqnonneg variants.

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

TABLE 7.2
Run-time (seconds) comparisons of NNLS and lsqnonneg variants on multiple systems of mean shifted Gaussian matrix A and random vectors b.

Number of systems	1	24	48	96	192
Number of updates	165	3907	7792	15541	30966
Number of downdates	92	2109	4196	8362	16599
GPU NNLS	0.4257	0.5094	0.9546	1.7862	3.2672
CPU NNLS	0.0654	1.2238	2.4141	4.8067	9.6250
CPU lsqnonneg	0.4791	8.7611	17.2483	34.5396	69.4889
Matlab lsqnonneg	0.9437	19.5904	38.6469	77.1072	155.7700
Matlab FNNLS [6]	0.4937	11.7176	23.9502	47.4635	91.4500
Matlab interior-points [16]	1.6317	40.7017	83.9788	164.4839	328.9569
Matlab PQN-NNLS [17]	3.1106	128.6616	253.3796	504.3153	989.2051

TABLE 7.3
Run-time (seconds) comparisons of NNLS and lsqnonneg variants on multiple systems of random matrices A and random vectors b.

Number of systems	1	24	48	96	192
Number of updates	52	1169	2361	4766	9525
Number of downdates	0	13	28	58	124
GPU NNLS	0.1194	0.1397	0.2526	0.4875	0.8667
CPU NNLS	0.0068	0.1157	0.2245	0.4352	0.8794
CPU lsqnonneg	0.0223	0.4665	0.8959	1.7269	3.5243
Matlab lsqnonneg	0.0330	0.8096	1.5583	3.0097	6.1627
Matlab FNNLS [6]	0.0248	0.5902	1.1602	2.2920	4.6022
Matlab interior-points [16]	0.4480	10.7729	21.1767	41.9441	83.6695
Matlab PQN-NNLS [17]	1.5935	55.8196	110.0892	221.7277	441.7190

For the second set of synthetic data, we generate both random matrices A of size $m = n = 512$ and nonnegative random vectors b. The number of column updates and downdates is less than that of the two previous experiments. Furthermore, the total number of column updates dominates the number of column downdates. The comparisons between GPU and CPU NNLS in Table 7.3 show that both implementations have similar run-time scaling as the number of systems increases. This suggests that most of the performance gains in prior experiments are from the GPU downdating steps. The comparisons between CPU NNLS and CPU lsqnonneg lead to a similar conclusion, as the performance gain (1.7x) is minimum compared to the prior two experiments.

8. Concluding remarks. In this paper, we have presented an efficient procedure for solving least squares subproblems in the active-set algorithm. We have shown that prior QR decompositions may be used to update and solve similar least squares subproblems. Furthermore, a reordering of variables in the passive-set yielded fewer computations in the update step. This has led to substantial speedups over existing methods in both the GPU and CPU implementations. Applications to satellite-based terrain mapping and to microphone array signal processing are being worked on. Both GPU and CPU source codes are available online at http://www.cs.umd.edu/~yluo1/Projects/NNLS.html.

Acknowledgments. We would like to acknowledge James Blair and Michelle Hofton at NASA for providing us with data for the deconvolution problem.

REFERENCES

[1] S. Bellavia, M. Macconi, and B. Morini, *An interior point Newton-like method for nonnegative least squares problems with degenerate solution*, Numer. Linear Algebra Appl., 13 (2006), pp. 825–846.
[2] M. H. van Benthem and M. R. Keenan, *Fast algorithm for the solution of large-scale nonnegativity-constrained least squares problems*, J. Chemometrics, 18 (2004), pp. 441–450.
[3] A. Björck, *Stability analysis of the method of semi-normal equations for least squares problems*, Linear Algebra Appl., 88/89 (1987), pp. 31–48.
[4] G. E. Blelloch, *Prefix Sums and Their Applications*, Technical report CMU-CS-90-190, Carnegie Mellon University, Pittsburgh, PA, 1990.
[5] A. W. Bojanczyk, R. P. Brent, and F. R. de Hoog, *QR factorization of Toeplitz matrices*, Numer. Math., 49 (1986), pp. 81–94.
[6] R. Bro and S. D. Jong, *A fast non-negativity-constrained least squares algorithm*, J. Chemometrics, 11 (1997), pp. 393–401.
[7] M. Catral, L. Han, M. Neumann, and R. Plemmons, *On reduced rank nonnegative matrix factorization for symmetric nonnegative matrices*, Linear Algebra Appl., 393 (2004), pp. 107–126.
[8] 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. 1740–1768.
[9] D. Chen and R. J. Plemmons, *Nonnegativity constraints in numerical analysis*, in Proceedings of the Symposium on the Birth of Numerical Analysis, A. Bultheel and R. Cools, eds., Leuven, Belgium, 2007, World Scientific Press, 2009.
[10] 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.
[11] G. H. Golub and C. F. Van Loan, *Matrix Computations*, 3rd ed., Johns Hopkins University Press, Baltimore, MD, 1996, pp. 223–236.
[12] S. Hammarling and C. Lucas, *Updating the QR factorization and the least squares problem*, MIMS EPrint, Manchester Institute for Mathematical Sciences, University of Manchester, Manchester, UK, 2008.
[13] M. Harris, *Optimizing Parallel Reduction in CUDA*, 2007.
[14] Intel, Math Kernel Library Reference Manual, 2010.
[15] A. Kerr, D. Campbell, and M. Richards, *QR decomposition on GPUs*, in GPGPU-2: Proceedings of the 2nd Workshop on General Purpose Processing on Graphics Processing Units, New York, ACM, 2009, pp. 71–78.
[16] S. J. Kim, K. Koh, M. Lustig, S. Boyd, and D. Gorinevsky, *An interior-point method for large-Scale l1-regularized least squares*, IEEE J. Selected Topics Signal Process., 1 (2007), 606617.
[17] D. Kim, S. Sra, and I. S. Dhillon, *A New Projected Quasi-Newton Approach for the Nonnegative Least Squares Problem*, Technical report TR-06-54, Computer Sciences, The University of Texas at Austin, TX, 2006.
[18] H. W. Kuhn and A. W. Tucker, *Nonlinear programming*, in Proceedings of the Second Berkeley Symposium on Mathematical Statistics and Probability, University of California Press, Berkeley, CA, 1951, pp. 481–492.
[19] C. L. Lawson and R. J. Hanson, *Solving Least Squares Problems*, Prentice-Hall, Englewood Cliffs, NJ, 1974.
[20] NVIDIA, *CUDA Programming Guide* 3.2, 2011.
[21] NVIDIA, OpenCL *Programming Guide for* CUDA *Architectures* 3.1, 2010.
[22] Sun Microsystems, Inc, OpenMP API *User Guide*, 2003.
[23] V. Volkov and J. W. Demmel, *Benchmarking* GPUs *to tune dense linear algebra*, in Proceedings of the ACM/IEEE Conference on Spercomputing, 2008.