Hindawi Publishing Corporation
Scientific Programming
Volume 2015, Article ID 246019, 17 pages
http://dx.doi.org/10.1155/2015/246019

Research Article

Computing Low-Rank Approximation of a Dense Matrix on Multicore CPUs with a GPU and Its Application to Solving a Hierarchically Semiseparable Linear System of Equations

Ichitaro Yamazaki, Stanimire Tomov, and Jack Dongarra

Department of Electrical Engineering and Computer Science, University of Tennessee, Knoxville, USA

Correspondence should be addressed to Ichitaro Yamazaki; iyamazak@icl.utk.edu

Received 10 September 2014; Revised 14 January 2015; Accepted 20 January 2015

Academic Editor: Rafael Mayo

Low-rank matrices arise in many scientific and engineering computations. Both computational and storage costs of manipulating such matrices may be reduced by taking advantages of their low-rank properties. To compute a low-rank approximation of a dense matrix, in this paper, we study the performance of QR factorization with column pivoting or with restricted pivoting on multicore CPUs with a GPU. We first propose several techniques to reduce the postprocessing time, which is required for restricted pivoting, on a modern CPU. We then examine the potential of using a GPU to accelerate the factorization process with both column and restricted pivoting. Our performance results on two eight-core Intel Sandy Bridge CPUs with one NVIDIA Kepler GPU demonstrate that using the GPU, the factorization time can be reduced by a factor of more than two. In addition, to study the performance of our implementations in practice, we integrate them into a recently developed software StruMF which algebraically exploits such low-rank structures for solving a general sparse linear system of equations. Our performance results for solving Poisson's equations demonstrate that the proposed techniques can significantly reduce the preconditioner construction time of StruMF on the CPUs, and the construction time can be further reduced by 10%–50% using the GPU.

1. Introduction

In applied and numerical mathematics or in scientific and engineering simulations, we often encounter low-rank matrices, and more frequently, we encounter matrices whose submatrices are low-rank. We can reduce both computational and storage requirements of manipulating many of these matrices by taking advantages of their low-rank properties. In this paper, we study the performance of the following two algorithms for computing a low-rank approximation of a dense matrix on multicore CPUs and investigate the potential of using a GPU to accelerate the process: (1) the QR factorization with column pivoting (QP3) [1, 2] and (2) the QR factorization with restricted pivoting (QPR) [3, 4]. In addition, to study the performance of QP3 and QPR in practice, we integrate our implementations of QP3 and QPR into StruMF [5, 6] which is a recently developed software for solving a general sparse linear system of equations. StruMF algebraically exploits a low-rank structure, referred to as a hierarchically semiseparable (HSS) structure [7], of a coefficient matrix while computing and applying a preconditioner, and uses QP3 and QPR for computing the low-rank approximation of the dense submatrices. In many cases, the preconditioner construction time of StruMF is dominated by the low-rank approximation of the submatrices.

The rest of the paper is organized as follows. First, in Section 2, we review QP3 and QPR. Then, in Section 3, after analyzing the performance of StruMF on a CPU, we propose several techniques to improve the QPR performance on the CPU. Next, in Section 4, we describe our implementations of QP3 and QPR using a GPU. Finally, in Section 5, we show their performance on multicore CPUs with a GPU and its impact on the performance of StruMF. We provide our final remarks in Section 6. Throughout this paper, the jth column of a matrix A is denoted by $\mathbf{a}_j$, while $A_{i_1:i_2,j_1:j_2}$ is the submatrix consisting of the i_1th through the i_2th rows and the j_1th through the j_2th columns of A.

2. QR Algorithms with Column Pivoting

An m-by-n matrix A has a numerical rank r with respect to a threshold τ when

$$\frac{\sigma_1(A)}{\sigma_r(A)} \le \tau < \frac{\sigma_1(A)}{\sigma_{r+1}(A)}, \tag{1}$$

where $\sigma_1(A), \sigma_2(A), \ldots, \sigma_m(A)$ are the singular values of A in the descending order (i.e., $\sigma_1(A) \ge \sigma_2(A) \ge \cdots \ge \sigma_m(A)$) (we assume that $m \le n$ without the loss of generality). Then, an RRQR factorization of A has a form

$$AP = QR, \tag{2}$$

where Q is an m-by-m orthonormal matrix, R is an m-by-n upper-triangular matrix, and P is a permutation matrix chosen to reveal the numerical rank:

$$\sigma_{\min}(R_{1:r,1:r}) \approx \sigma_r(A), \tag{3}$$

$$\sigma_{\max}(R_{r+1:m,r+1:n}) \approx \sigma_{r+1}(A). \tag{4}$$

Since the interlacing properties of the singular values [8] states that

$$\sigma_{\min}(R_{1:r,1:r}) \le \sigma_r(A), \tag{5}$$

$$\sigma_{\max}(R_{r+1:m,r+1:n}) \ge \sigma_{r+1}(A), \tag{6}$$

satisfying (3) or (4) is equivalent to finding the permutation matrix P such that one of the following two tasks is satisfied, respectively:

$$\text{task-1: } \max_P \sigma_{\min}(R_{1:r,1:r}), \tag{7}$$

$$\text{task-2: } \min_P \sigma_{\max}(R_{r+1:m,r+1:n}). \tag{8}$$

More detailed discussion on the RRQR factorizations can be found in [9] and the references therein.

In the following subsections, we review the existing algorithms to compute such RRQR factorizations. Namely, we first review the blocked versions of Householder QR [8] (Section 2.1). We then outline QP3 [1, 2] which is a greedy algorithm for solving task-1 and is implemented in LAPACK [10] (Section 2.2). Next, we describe how an algorithm solving task-1 can be modified to solve task-2 and present three types of so-called hybrid algorithms [9] that are theoretically guaranteed to solve task-1 or task-2, or both (Section 2.3). Finally, we discuss QPR [3] that uses the hybrid algorithm as a postprocessing scheme to reduce the computational bottleneck of QP3, while ensuring the rank-revealing properties of the computed factorization (Section 2.4).

2.1. Blocked QR Algorithm. The jth step of Householder QR [8] generates the Householder transformations to zero out the off-diagonal elements in the jth column of A (for $j = 1, 2, \ldots, m$). To improve the data locality of the factorization, a blocked version of the algorithm (QR3) accumulates n_b Householder transformations and uses a BLAS-3 to apply the accumulated transformations at once to the trailing submatrix; that is, for $k = 1, 2, \ldots, m/n_b$, and $r = (k-1)n_b$ (we assume that both m and n are multiples of n_b, but the discussion can be easily extended for other cases),

$$A_{r:m,r+n_b:n} := H^{[k]}_{r:m,r:m} A_{r:m,r+n_b:n}, \tag{9}$$

where $H^{[k]}$ is the product of the n_b Householder matrices; that is, $H^{[k]} = H^{[k,n_b:-1:1]} = H^{[k,n_b]} H^{[k,n_b-1]} \cdots H^{[k,1]}$ and $H^{[k,j]} = I - \tau^{[k]}_j \mathbf{v}^{[k]}_j \mathbf{v}^{[k]T}_j$. (The matrices $H^{[k]}$ is not explicitly formed. Instead, we store $\mathbf{v}^{[k]}_{j:m,j}$ in the lower-triangular part of $\mathbf{a}_j$ and store $\tau^{[k]}_j$ in an n-length vector.) This matrix $H^{[k]}$ can be represented in a so-called VW form [11]:

$$H^{[k]} = I - V^{[k]} W^{[k]T}, \tag{10}$$

where $\mathbf{w}^{[k]}_j = \tau^{[k]}_j H^{[k,j-1:-1:1]T} \mathbf{v}^{[k]}_j$. Since $j = 1, 2, \ldots, n_b$, these $j-1$ Householder matrices are applied to the vector $\mathbf{v}^{[k]}_j$ in addition to the matrix A; this blocked algorithm requires about n_b/m times more floating point operations (flops) than the unblocked algorithm. However, on a modern computer, the blocked algorithm takes advantage of the memory hierarchy and often obtains a significant speedup over an unblocked algorithm.

The additional m-by-n_b workspace to store $W^{[k]}$ can be reduced by factorizing $W^{[k]}$ into the following form [12]:

$$W^{[k]} = V^{[k]} T^{[k]}, \tag{11}$$

where $T^{[k]}$ is an n_b-by-n_b upper-triangular matrix such that $T^{[k]}_{1:i,i} = \tau^{[k]}_i \begin{bmatrix} \mathbf{z} \\ 1.0 \end{bmatrix}$ and $\mathbf{z}$ is an $(i-1)$-length vector given by $\mathbf{z} = -T^{[k]}_{1:i-1,1:i-1} V^{[k]T}_{:,1:i-1} \mathbf{v}^{[k]}_i$. Algorithm 1 shows the pseudocode of the resulting blocked QR algorithm with BLAS-3 (QR3).

2.2. QP3 Algorithm. At each step of an RRQR factorization, selecting an optimal pivot to satisfy task-1 (7) or task-2 (8), or both, is likely to be a combinatorial optimization problem. To reduce the computational cost, a greedy algorithm is generally used. In this section, we discuss such a greedy algorithm for solving task-1. Specifically, at the $(r+1)$th step (for $r = 0, 1, \ldots, m-1$), assuming that the r well-conditioned columns of A have been already selected and factorized, the next pivot column is selected from the remaining $n-r$ columns such that the smallest singular value of the $r+1$ selected columns are maximized:

$$\begin{pmatrix} R_{1:r,1:r} & \mathbf{r}_{1:r,r+1} \\ & \mathbf{r}_{r+1:m,r+1} \end{pmatrix} = \arg\max_{j=r+1,\ldots,n} \sigma_{\min} \begin{pmatrix} R_{1:r,1:r} & \mathbf{a}_{1:r,j} \\ & \mathbf{a}_{r+1:m,j} \end{pmatrix}. \tag{12}$$

Since selecting such a column is still computationally expensive, heuristics are used. First, since a Householder transformation can zero out the elements of $\mathbf{a}_{r+1:m,j}$ below the diagonal, we have

$$\sigma_{\min} \begin{pmatrix} R_{1:r,1:r} & \mathbf{a}_{1:r,j} \\ & \mathbf{a}_{r+1:m,j} \end{pmatrix} = \sigma_{\min} \begin{pmatrix} R_{1:r,1:r} & \mathbf{a}_{1:r,j} \\ & \gamma_j \end{pmatrix}, \tag{13}$$

setup: $r := 0$ and $k := 0$
`while` $r < m$ do
(1) **panel factorization:**
`for` $j = 1, \ldots, n_b$ do
generation of Householder transformation:
$(\tau_j^{[k]}, \mathbf{v}_j^{[k]})$ such that $H^{[k,j]} := I - \tau_j^{[k]} \mathbf{v}_j^{[k]} \mathbf{v}_j^{[k]T}$.
right-looking update of panel:
$A_{r+j:m,r+j:r+n_b} := H^{[k,j]}_{r+j:m,r+j:m} A_{r+j:m,r+j:r+n_b}$.
`end for`
(2) **computation of matrix** $T^{[k]}$**:**
`for` $j = 1, 2, \ldots, n_b$ do
$T^{[k]}_{1:j-1,j} := T^{[k]}_{1:j-1,j} - \tau_j^{[k]} T^{[k]}_{1:j-1,1:j-1} V^{[k]T}_{j:n,1:j-1} \mathbf{v}^{[k]}_{j:n,j}$
$T^{[k]}_{j,j} := \tau_j^{[k]}$
`end for`
(3) **right-looking update of trailing submatrix:**
$A_{r:m,r+n_b:n} := H^{[k]}_{r:m,r:m} A_{r:m,r+n_b:n}$.
$k := k + 1$ and $r := r + n_b$.
`end while`

Algorithm 1: Blocked QR factorization algorithm.

where $\gamma_j = \|\mathbf{a}_{r+1:m,j}\|_2$. Furthermore, we can approximate the smallest singular value of the matrix by the reciprocal of the largest row norm of its inverse [9, 13]:

$$\sigma_{\min}\begin{pmatrix} R_{1:r,1:r} & \mathbf{a}_{1:r,j} \\ & \gamma_j \end{pmatrix} \leq \max_{i=1,2,\ldots,r+1} \left\| \mathbf{e}_i^T \begin{pmatrix} R_{1:r,1:r} & \mathbf{a}_{1:r,j} \\ & \gamma_j \end{pmatrix}^{-1} \right\|_2^{-1} \leq \sqrt{n}\sigma_{\min}\begin{pmatrix} R_{1:r,1:r} & \mathbf{a}_{1:r,j} \\ & \gamma_j \end{pmatrix}. \tag{14}$$

In addition, since $R_{1:r,1:r}$ is assumed to be well-conditioned, all the row norms of $R^{-1}_{1:r,1:r}$ are expected to be small. Hence, it leads to the following approximation:

$$\min_{i=1,2,\ldots,r+1} \left\| \mathbf{e}_i^T \begin{pmatrix} R^{-1}_{1:r,1:r} & -R^{-1}_{1:r,1:r}\mathbf{a}_{1:r,j}\gamma_j^{-1} \\ & \gamma_j^{-1} \end{pmatrix} \right\|_2 \approx \min_{i=1,2,\ldots,r+1} \left\| \mathbf{e}_i^T \begin{pmatrix} -R^{-1}_{1:r,1:r}\mathbf{a}_{1:r,j}\gamma_j^{-1} \\ \gamma_j^{-1} \end{pmatrix} \right\|_2 \approx \gamma_j^{-1}, \tag{15}$$

where

$$\begin{pmatrix} R_{1:r,1:r} & \mathbf{a}_{1:r,j} \\ & \gamma_j \end{pmatrix}^{-1} = \begin{pmatrix} R^{-1}_{1:r,1:r} & -R^{-1}_{1:r,1:r}\mathbf{a}_{1:r,j}\gamma_j^{-1} \\ & \gamma_j^{-1} \end{pmatrix}. \tag{16}$$

Based on these heuristics, QR with column pivoting (QRP) [1] selects the next pivot column that is farthest away in the Euclidean norm from the subspace spanned by the already selected columns (i.e., the column with the largest $\gamma_j = \|\mathbf{a}_{r+1:m,j}\|_2$). Since an orthogonal transformation does not change the column norms, once these norms are initialized ($\gamma_j := \|\mathbf{a}_j\|_2$ for $j = 1, 2, \ldots n$), it can be cheaply downdated at the end of the $(r+1)$th step ($\gamma_j := \gamma_j - r^2_{r+1,j}$ for $j = r+2, r+3, \ldots, n$).

Algorithm 2 shows a blocked version (QP3) of QRP [2]. One difference between QP3 and QR3 of Algorithm 1 is at the trailing submatrix update. Specifically, QR3 stores the product of the n_b Householder matrices in the VW form (10) (or using the matrix $T^{[k]}$ to reduce the memory requirement) and updates the trailing submatrix by two matrix-matrix multiplies (our implementation takes advantage of the triangular structures of both $V^{[k]}$ and $T^{[k]}$); that is, after the kth panel factorization, the trailing submatrix is updated by $A := A - V^{[k]}F^{[k]T}$ where $F^{[k]} = A^T W^{[k]}$. On the other hand, the jth step of the kth QP3 panel factorization computes the jth column of the matrix-matrix product $A^T W^{[k]}$ and uses the resulting vector $\mathbf{f}_j^{[k]}$ to update the jth row of A, which, in return, is needed to downdate the column norms of the trailing submatrix (Steps 1.5 and 1.6 of Algorithm 2). Finally, QP3 updates the trailing submatrix by a single matrix-matrix multiply, $A := A - V^{[k]}F^{[k]T}$. Since QP3 saves the result of the matrix-matrix product $A^T W^{[k]}$ in the auxiliary matrix $F^{[k]}$, QP3 and QR3 require about the same numbers of flops. However, QP3 performs about a half of the total flops using BLAS-2, while QR3 performs most of its flops using BLAS-3.

In some cases, due to the round-off errors, the downdated norms γ_j diverge significantly from the true norms [14]. When this occurs, the trailing submatrix is immediately updated with the outstanding Householder transformations, and the column norms are recomputed. If the column norms must be frequently recomputed, then in comparison to QR3, QP3 not only requires significantly more flops for recomputing the norms but also exhibits a poorer data locality since the trailing submatrices are updated using smaller blocks.

2.3. Hybrid Algorithms. In this section, we first show how an algorithm solving task-1 can be used to solve task-2, and then describe so-called hybrid algorithms for solving task-1 or task-2, or both. The algorithms presented in this subsection

setup: Compute the column norms
$\gamma_j = \|\mathbf{a}_j\|_2^2$ for $j = 1, 2, \ldots, n$ $r = 0$, and $k = 0$.
`While` $r < m$ do
(1) **panel factorization:**
`for` $j = 1, 2, \ldots, n_b$ do
(1.1) **pivoting:**
select the pivot column with the largest γ_h.
$A_{:,[r+1,h]} = A_{:,[h,r+1]}$ and $\gamma_{[r+1,h]} = \gamma_{[h,r+1]}$.
$F^{[k]}_{[h,j],1:j-1} = F^{[k]}_{[j,h],1:j-1}$.
$r := r + 1$ (update numerical rank).
(1.2) **left-looking update of pivot column:**
$A_{r:m,r} := A_{r:m,r} - V^{[k]}_{r:m,1:j-1}\mathbf{f}^{[k]T}_{r,1:j-1}$.
(1.3) **Householder matrix computation:**
$(\tau_j^{[k]}, \mathbf{v}_j^{[k]})$ such that $H_j^{[k]} = I - \tau_j^{[k]}\mathbf{v}_j^{[k]}\mathbf{v}_j^{[k]T}$.
(1.4) **Computation of the j-auxiliary vector $\mathbf{f}_j$:**
$\mathbf{f}^{[k]}_{r:n,j} = \tau_j^{[k]}(A_{r:m,r:n} - V^{[k]}_{r:m,1:j-1}F^T_{r:n,1:j-1})^T\mathbf{v}_j^{[k]}$
(1.5) **right-look update of pivot row:**
$\mathbf{r}_{r,r:n} = \mathbf{a}_{r,r:n} - \mathbf{v}^{[k]}_{r,1:j-1}F^{[k]T}_{r:n,1:j-1}$
(1.6) **norm downdate:**
$\gamma_\ell = \gamma_\ell - r^2_{r-1,\ell}$ for $\ell = r, r+1, \ldots, n$.
(**if** norms must be recomputed **then** break)
`end for`
(2) **trailing submatrix update:**
$A_{r+1:m,r+1:n} := A_{r+1:m,r+1:n} - V^{[k]}_{r+1:m,1:n_b}F^{[k]T}_{r+1:n,1:n_b}$.
$k := k + 1$ (recompute norms if necessary).
`end while`

Algorithm 2: QP3 factorization algorithm, where $A_{:,[r+1,h]}$ is the submatrix consisting of the $(r+1)$th and hth columns of A.

are used as a postprocessing to improve the numerical properties of an RRQR factorization, and the matrix A is of upper-triangular form. For instance, Algorithm 3(a) shows the QP3 algorithm applied to an upper triangular matrix R. To simplify our notation, we write the RRQR factorization as

$$AP = Q\begin{bmatrix} R^{[1,1]} & R^{[1,2]} \\ & R^{[2,2]} \end{bmatrix}, \tag{17}$$

where $R^{[1,1]}$ is the r-by-r leading submatrix. Then, task-2 is equivalent to

$$\text{task-3: } \max_P \sigma_{\min}\left(\left(R^{[2,2]}\right)^{-T}\right). \tag{18}$$

Hence, to solve task-2, we apply an algorithm that solves task-1 to the transpose-inverse of the matrix A. Namely, if we have such an RRQR factorization,

$$A^{-T}P = Q\begin{bmatrix} R^{[1,1]} & R^{[1,2]} \\ & R^{[2,2]} \end{bmatrix}, \tag{19}$$

then we also have

$$AP = Q\begin{bmatrix} \left(R^{[1,1]}\right)^{-T} & \\ -\left(R^{[2,2]}\right)^{-T}R^{[1,2]T}R^{[1,1]^T} & \left(R^{[2,2]}\right)^{-T} \end{bmatrix}. \tag{20}$$

Moreover, if $\widehat{P}$ is the permutation matrix with ones on the antidiagonal, then

$$A\left(P\widehat{P}\right) = \left(Q\widehat{P}\right)\begin{bmatrix} R^{[2,2]^{-1}} & -R^{[1,1]^{-1}}R^{[1,2]}R^{[2,2]^{-1}} \\ & R^{[1,1]^{-1}} \end{bmatrix}, \tag{21}$$

where $P\widehat{P}$ is a permutation matrix, $Q\widehat{P}$ is an orthogonal matrix, and $\sigma_{\max}(R^{[1,1]^{-1}})$ is minimized. Hence, the factorization (21) is an RRQR factorization solving task-2. This is called the unification principle of the algorithms solving task-1 and task-2 [9].

We now introduce the Chan-I algorithm for solving task-1, whose task-2 version is used as the postprocess scheme in Section 2.4. Just like the QRP algorithm, the Chan-I algorithm pivots the column with the largest norm, but it uses an additional heuristic. Namely, at the $(r+1)$th step, the column norm γ_{r+k} is approximated using the right-singular vector corresponding to the largest singular value of the submatrix $A^{[2,2]}$. Specifically, if $A^{[2,2]} = U\Sigma V^T$ is the singular value decomposition of $A^{[2,2]}$, then the kth column norm of $A^{[2,2]}$ is approximated by

$$\left\|\mathbf{a}_k^{[2,2]}\right\|_2 = \left\|\sum_{\ell=1,2,\ldots,n-r}\sigma_\ell\left(A^{[2,2]}\right)v_{k,\ell}\mathbf{u}_\ell\right\|_2$$

(a)
Golub algorithm for task-1.
`for` $j = 1, 2, \ldots, m-1$ do
  Golub-I (A, j, m, n)
`end for`

(b)
Chan algorithm for task-2.
`for` $j = m, m-1, \ldots, 2$ do
  Chan-II $(A, 1, j, n)$
`end for`

(c)
Golub-I (R, r, m, n) algorithm.
(1) compute the column norms:
  for $j = r+1, r+2, \ldots, n$
    $\gamma_j = \|\mathbf{r}_{r:m,j}\|_2$
(2) find a pivot column j such that
  $\gamma_j = \max_{k=r+1,r+2,\ldots,n} \gamma_k$
(3) shift columns for column pivot:
  $\mathbf{y} = \mathbf{r}_{1:j,j}$
  for $k = j, j-1, \ldots, r+1$
    $\mathbf{r}_{1:k,k} = \mathbf{r}_{1:k,k-1}$
  $\mathbf{r}_{1:j,r} = \mathbf{y}$
(4) re-triangularize R:
  `for` $k = j, j-1, \ldots, r+1$ do
  (4.1) generate a Given's rotation
    to zero out $r_{k,r}$ with $r_{k-1,r}$
  (4.2) apply the Given's rotation
    to the trailing submatrix
  `end for`

(d)
Chan-II (R, r, m, n) algorithm.
(1) compute right singular vector $\mathbf{v}$
  corresponding to $\sigma_{\min}(R_{1:r,1:r})$
(2) find a pivot column j such that
  $|v_j| = \min_{k=1,2,\ldots,r} |v_k|$
(3) shift columns for column pivot:
  $\mathbf{y} = \mathbf{r}_{1:r,j}$
  for $k = j, j+1, \ldots, r-1$
    $\mathbf{r}_{1:k+1,k} = \mathbf{r}_{1:k+1,k+1}$
  $\mathbf{r}_{1:r,r} = \mathbf{r}_{1:r,j}$
(4) re-triangularize R:
  `for` $k = j, j+1, \ldots, r-1$ do
  (4.1) generate a Given's rotation
    to zero out $r_{k+1,k}$ with $r_{k,k}$
  (4.2) apply the Given's rotation
    to the trailing submatrix
`end for`

Algorithm 3: Unification principle of QRP algorithm.

$$= \left(\sum_{\ell=1,2,\ldots,n-r} \left(\sigma_\ell \left(A^{[2,2]} \right) \left| v_{k,\ell} \right| \right)^2 \right)^{1/2}$$

$$\approx \sigma_1 \left(A^{[2,2]} \right) \left| v_{k,1} \right|. \tag{22}$$

Hence, at the $(r+1)$th step, the algorithm pivots the $(r+k)$th column, whose corresponding element $v_{k,1}$ of the most dominant right singular vector $\mathbf{v}_1$ has the largest module. Algorithm 3(b) shows the Chan-II algorithm which is the task-2 version of the Chan-I algorithm and pushes the column that minimizes $\sigma_{\max}(R^{[2,2]})$ into the trailing submatrix at each step.

Hybrid-I (R, r, n, m) to solve task-1
  `repeat`
    pivot "best" column among $\{\mathbf{r}_r, \mathbf{r}_{r+1}, \ldots, \mathbf{r}_n\}$ *to* $\mathbf{r}_r$
    Golub-I (R, r, n, m)
    pivot "worst" column among $\{\mathbf{r}_1, \mathbf{r}_2, \ldots, \mathbf{r}_r\}$ *to* $\mathbf{r}_r$
    Chan-II $(R, 1, r, m)$
  `until` no column pivot occurred

Hybrid-II (R, r, n, m) to solve task-2
  `repeat`
    pivot "best" column among $\{\mathbf{r}_1, \mathbf{r}_2, \ldots, \mathbf{r}_{r+1}\}$ *to* $\mathbf{r}_{r+1}$
    Chan-II $(R, r+1, n, m)$
    pivot "worst" column among $\{\mathbf{r}_{r+1}, \mathbf{r}_{r+2}, \ldots, \mathbf{r}_n\}$ *to* $\mathbf{r}_{r+1}$
    Golub-I $(R, 1, r+1, m)$
  `until` no column pivot occurred

Hybrid-III (R, r, n, m) to solve both task-1 and task-2
  Hybrid-I (R, r, n, m), to address task-1
  Hybrid-II (R, r, n, m), to address task-2

Algorithm 4: Hybrid algorithms.

Finally, Algorithm 4 shows a single iteration of three hybrid algorithms Hybrid-I, Hybrid-II, and Hybrid-III [9] that solve task-1, task-2, and both task-1 and task-2, respectively. The iteration is terminated when the rth column is not moved.

2.4. QPR Algorithm. Even when QP3 and QR3 performs about the same number of flops, QP3 is often slower. This is largely because QR3 performs most of its flops using BLAS-3, while QP3 performs about half of its flops using BLAS-2. Since this BLAS-2 is needed to select the pivot among all the remaining columns, QPR [3] tries to reduce the bottleneck by selecting the pivot among a fixed number, n_w, of the columns. Algorithm 5(a) shows the pseudocode of QPR. At each step of the panel factorization, while QP3 computes the matrix-vector product with the whole trailing submatrix, QPR updates the columns within this window using a Householder matrix (both by BLAS-2). Since the pivots are now selected only within the window, in order to ensure the rank-revealing properties, QPR uses a condition estimator and accepts only the pivots that satisfy (5). After all the columns are either accepted or rejected (Step 1 in Algorithm 5(a)), the QR factorization of the rejected columns is computed to obtain the upper-triangular matrix R (Steps 2 and 3). At the end, in comparison to QP3, QPR performs a fewer flops using BLAS-2 to select the pivots. However, the $n_w - n_b$ columns of the trailing submatrix are now updated using BLAS-2. In Sections 3 and 5.1, we compare the performance of QPR and QP3 on a CPU and on multicore CPUs with a GPU, respectively.

Since QPR selects the pivots only within the windows, it requires postprocessing to globally ensure the rank-revealing property. Though there are two postprocessing options [9, 15], here, we focus on the first [9] because it has shown to be more effective in our experiments. Algorithm 5(b) shows the pseudocode of this postprocessing scheme, which modifies the Hybrid-III algorithm of Algorithm 4 to improve its convergence rate [3].

3. Case Studies with StruMF on a CPU

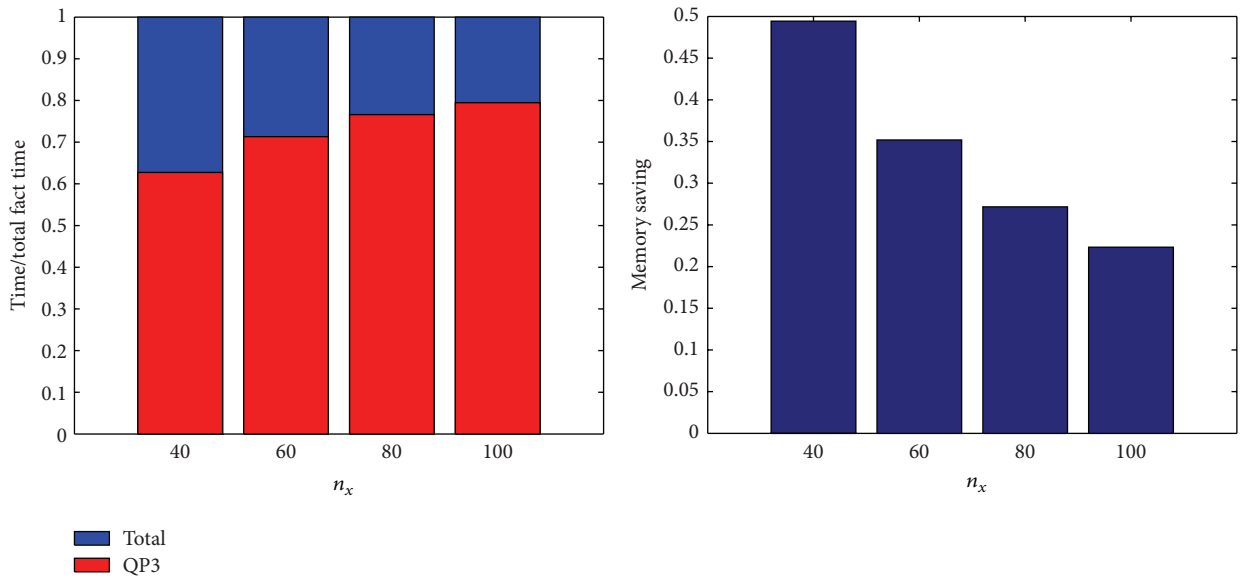
The performance of both QP3 and QRP depends on the input matrix A. To study their performance, in this section, we provide their case studies with StruMF for solving 3D seven-point Poisson's equations on one core Intel Sandy Bridge CPU. In our experiments, we use the same default parameters used for solving Poisson's equations with StruMF in [5] (e.g., the numerical rank tolerance is set to be $\tau = 5 \times 10^{-1}$, and FGMRES(30) is considered to be converged when the residual norm is reduced at least by the order of 10^{-6}), and for QPR, we used the default window size recommended in [3] (i.e., $n_w = n_b + \min(m, n, \max\{10, n_b/2 + 0.05n\})$).

3.1. Performance of Original StruMF and QRP. Figure 1(a) shows the breakdown of the StruMF time using QP3. It clearly shows that StruMF spends most of its preconditioner construction time in QP3. Moreover, the percentage of the time spent in QP3 often increases as the global matrix dimension increases. To analyze the performance of QP3, Figure 2 shows the dimensions of the off-diagonal blocks for which the low-rank approximations are computed. Most of these off-diagonal blocks are short and wide having a few hundreds rows and tens of thousands of columns. In addition, the dimensions of the off-diagonal blocks, especially, their numbers of columns, often increase with respect to the global dimension. Hence, if the compression time of these short-wide blocks can be reduced using another algorithm or a GPU, then the StruMF solution time may be significantly reduced.

(a)
QPR factorization algorithm.
(1) **QR factorization with restricted pivoting:**
setup: $r := 0$ and $k := 0$.
`While` $r < m$ do
(1.1) **panel factorization:**
setup: initialize column norms within the window,
$w = r + n_w$ (last column in window).
`for` $j = 1, \ldots, n_b$ do
(1.1.1) **restricted pivoting:**
swap jth column with pivot column
of the largest norm within the window.
`if condest` $(R_{1:r+1,1:r+1}) > \tau$ `then`
$n_b := j$ and `break`.
$r := r + 1$ (update numerical rank).
(1.1.2) **Householder matrix computation:**
$H^{[k,j]} := I - \tau_j^{[k]} \mathbf{v}_j^{[k]} \mathbf{v}_j^{[k]T}$.
(1.1.3) **right-look update of window:**
$A_{r:m,r:w} := H^{[k,j]}_{i:m,i:m} A_{r:m,r:w}$.
`end for`
(1.2) **trailing submatrix update:** ($k := k + 1$).
compute matrix $T^{[k]}$.
update trailing submatrix right of current window.
(1.3) **swapping rejected columns to end:**
`end while`
(2) **QR factorization of rejected columns with column pivoting**
(3) **QR factorization of rejected columns with no pivoting**

(b)
QPR postprocessing algorithm.
setup: set r to be the maximum column index
such that `condest` $(R_{1:r,1:r}) \leq \tau$.
`repeat`
(1) **pivoting for task-1 and task-2**
`repeat`
Golub-I (R, r, n, m)
Golub-I $(R, r + 1, n, m)$
Chan-II $(R, r + 1, n, m)$
Chan-II (R, r, n, m)
`until` no column pivot occurred
(2) **convergence check**
$\alpha =$ `condest`$(R_{1:r,1:r})$
$\beta =$ `condest`$(R_{1:r+1,1:r+1})$
`if` $\alpha \leq \tau$ `and` $\beta > \tau$ `then`
break
`else if` $\alpha \leq \tau$ `then`
$r := r + 1$, move to right
`else`
$r := r - 1$, move to left
`end if`
`end repeat`

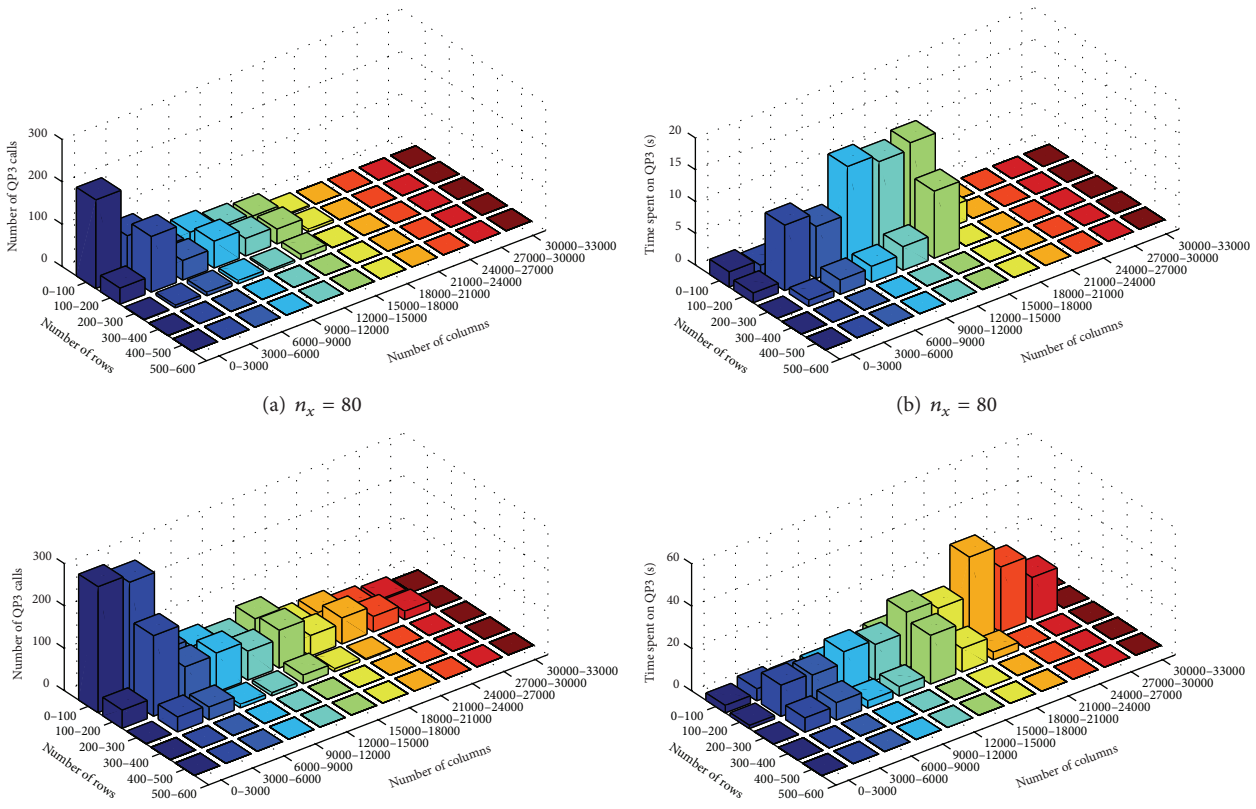
Algorithm 5: QRP factorization and postprocessing algorithms, where `condest` (R) computes an estimation of the condition number of R.



(a) Relative time spent in QP3

(b) Memory relative to direct factorization

Figure 1: Relative time and memory requirements of StruMF.



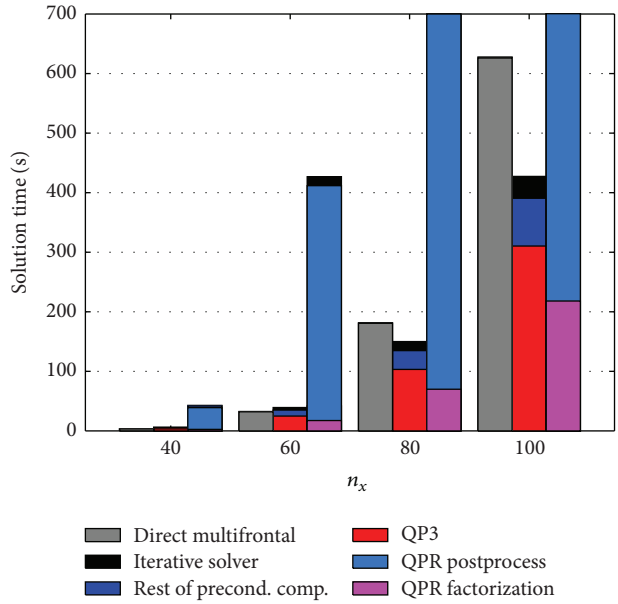
(a) $n_x = 80$

(b) $n_x = 80$

(c) $n_x = 100$

(d) $n_x = 100$

Figure 2: Statistics of off-diagonal blocks whose low-rank approximations are computed.



n_x	QP3	Prec	Sol	Total	Itrs
40	3.4	5.4	0.8	6.2	23
60	24.9	34.9	4.4	39.3	35
80	103.2	134.7	15.2	149.9	50
100	310.4	390.6	36.6	427.2	60

n_x	QPR		Prec	Sol	Total	Itrs
	Facto	Post				
40	2.5	36.6	41.9	0.9	42.8	25
60	17.3	394.4	422.1	4.6	426.7	37
80	69.9	4833.3	4921.2	15.0	4936.2	50
100	218.0	10779.2	11015.2	36.5	11051.7	60

FIGURE 3: Original StruMF solution time.

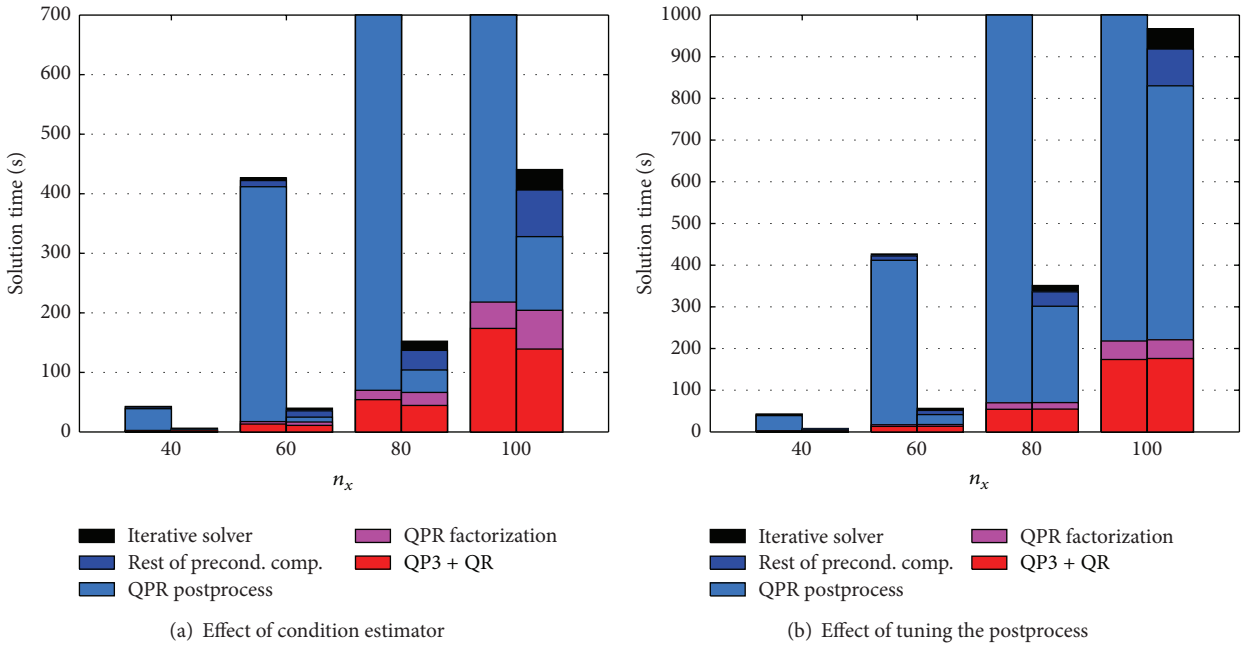
Figure 3 compares the StruMF solution times using QP3 and QPR. The figure clearly indicates that even though the factorization time is reduced using QPR, the postprocessing can be expensive. In the next subsection, we propose modifications to the QPR implementation, which often reduce the postprocessing time and make QPR more competitive. Just for reference, Figure 3 compares the StruMF solution time with that of a direct multifrontal factorization. We see that for a large enough system, StruMF can reduce not only the memory requirement (see Figure 1(b)) but also the solution time of the direct factorization.

3.2. Proposed Modifications to Original QPR Implementation. The performance of QPR depends strongly on the condition estimator used to evaluate Step 1.1.1 of Algorithm 5(a). In the original implementation, the smallest singular value of $R^{[1,1]}$ is estimated using an incremental condition estimator (ICE) [16, 17], while the largest singular value is estimated by the product of the largest column norm and the third root of the matrix dimension (i.e., $(r+1)^{1/3}\max_{j=1}^{r+1}\| \mathbf{r}_j \|_2$ where $R^{[1,1]}$ is of dimension $r+1$, and r_j is the jth column of $R^{[1,1]}$) [3]. This simple estimator is used for the largest singular value because though the estimator may lead to a greater estimation error, it requires a fewer flops (i.e., ICE requires $O(r)$ flops to update the estimation of $R^{[1,1]}$). During the postprocessing phase, the same estimators are used at Step 1 of Algorithm 3(b), but in order to obtain an accurate final factorization, both the largest and smallest singular values are estimated by ICE at Step 3 of Algorithm 5(b). We found that, in our numerical experiments using random matrices, this simple estimator underestimates the largest singular values. As a result, many components that should be rejected are accepted. In addition, during the preconditioner construction for solving Poisson's equation by StruMF, the simple estimator overestimates the singular values, rejecting the components which should be accepted. In either case, this estimation error could significantly increase the postprocessing cost. Figure 4(a) shows that when a more accurate condition estimator (i.e., ICE) is used, the postprocessing time can be dramatically reduced. In addition, ICE reduced the factorization time slightly because most of the components are accurately accepted by Step 1 of Algorithm 5(a), and Step 2 has less work.

During the postprocessing (Algorithm 5(b)), the Golub-I algorithm requires the column norms of the trailing submatrix $R_{r:m,r:n}$, and the Chan-II algorithm requires the column norms of the leading submatrix $R_{1:r+1,1:r+1}$. At the beginning of Step 1 to call the Chan-II algorithm, the original implementation computes the column norms of the leading submatrix $R_{1:r-1,1:r-1}$. Then, the norms $\|\mathbf{r}_r\|_2$ and $\|\mathbf{r}_{r+1}\|_2$ are computed as they are permuted into $R_{1:r+1,1:r+1}$. On the other hand, for each Golub-I call, the column norms of the trailing submatrix $R_{r:m,r:n}$ are recomputed. This is because when the Chan-II algorithm applies Given's rotation to the leading submatrix $R_{1:r,1:r}$, the column norms of the trailing submatrix $R_{r:m,r:n}$ changes. In our experiments, this norm computation could become expensive, especially when Step 1 requires many iterations or a large tolerance τ is used. To reduce this computational cost, we incrementally update or downdate the norms when Given's rotation is applied and when the submatrix size r changes at the outer iteration of Algorithm 5(b). In addition, we use the same criteria used in QP3 [14] to detect if the updated norms have diverged from the actual norms due to the round-off errors. When this happens, the column norms are recomputed. Figure 4(b) clearly indicates that the postprocessing time of StruMF can be significantly reduced by avoiding the norm computation.

For Step 2 of the factorization in Algorithm 5(a), the original implementation uses the column-wise QRP algorithm. On a modern computer, a blocked algorithm can obtain significant speedups. Hence, we replaced it with QP3. In addition, we use ICE to detect the numerical rank instead of the simple estimator used in the original implementation (i.e., QP3 terminates when the estimated condition number of the leading submatrix $R_{1:r+1,1:r+1}$ becomes greater than the threshold τ). Figure 5(a) shows that the blocked algorithm reduces the factorization time, but the overall improvement is not significant since after ICE is integrated, only a short time is spent at Step 2 for solving this Poisson's equation.

Finally, while QP3 aims to solve task-1, the QPR postprocessing tries to solve both task-1 and task-2. Hence, we replaced it with the postprocessing algorithms for solving



(a) Effect of condition estimator

(b) Effect of tuning the postprocess

Figure 4: Solution time of original StruMF (left bar) and after modification (right bars), using ICE and tuning the postprocess in left and right plots.

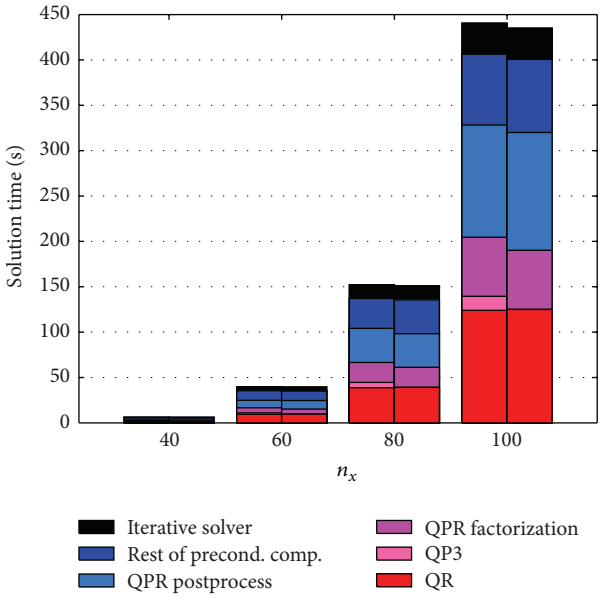
only either task-1 or task-2. Even though the postprocessing solved different tasks, StruMF converged in a similar number of iterations. However, the task-1 postprocessing converged slightly faster than the task-2 postprocessing, which was faster than the original postprocessing. We have also used the test matrices from the original paper [3] to evaluate the quality of the factorization after the postprocessing. Figure 5(c) shows that the quality of the factorization did not change significantly when different postprocessing schemes were used. We use the original postprocessing scheme in the remaining of the paper since it provides the most robust behavior, while the overhead is not significant after all the proposed modifications are integrated. Figure 5(d) compares the StruMF solution time using QP3 and QPR after all the modifications are made. Now, the solution time using QPR is shorter than that using QP3.

4. GPU Implementations

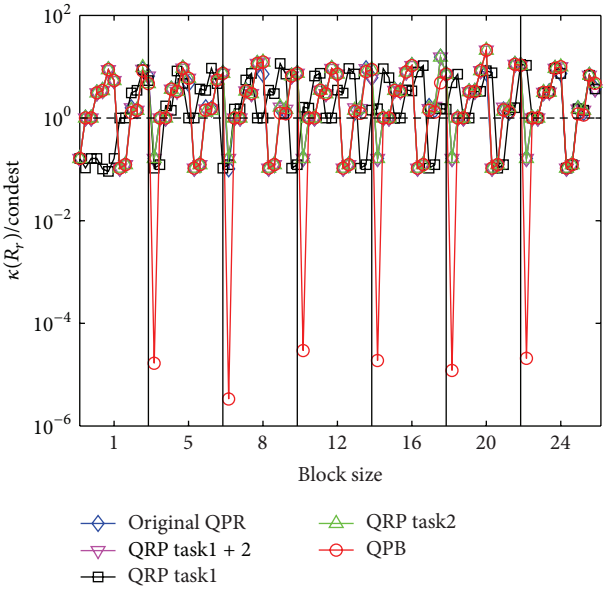
We now describe our QP3 and QPR implementations that utilizes a GPU.

4.1. QP3 Implementation. MAGMA (http://icl.utk.edu/magma/) extends LAPACK (http://www.netlib.org/lapack/) to heterogeneous architectures based on a hybrid programming and static scheduling. For instance, for the blocked QR factorization, the latency-limited BLAS-2 based panel factorization is scheduled on the CPUs, while the compute-intensive BLAS-3 based trailing submatrix update is scheduled on the GPU [18]. Furthermore, as soon as the next panel is updated on the GPU, it is copied to the CPU such that the panel factorization on the CPUs can be hidden behind the remaining submatrix update on the GPU (this is commonly referred to as a lookahead). As a result, for a large enough matrix, the QR factorization by MAGMA can obtain the performance of the BLAS-3, which exhibits a high-level of data parallelism and can be efficiently implemented on the GPU [19].

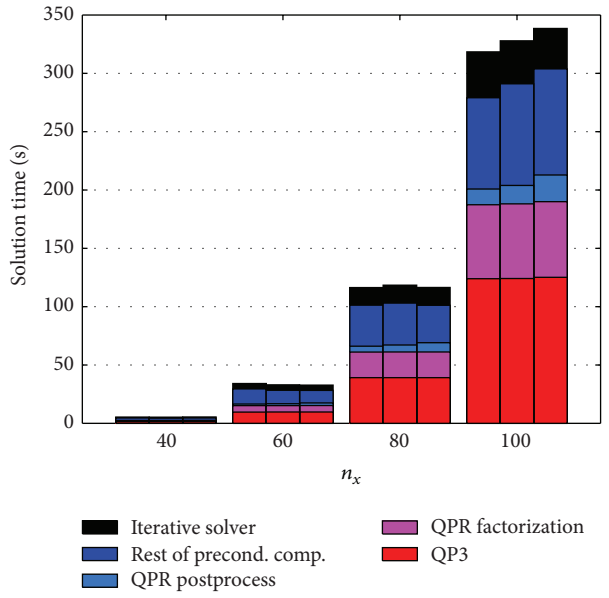
Our first QP3 implementation is based on the same hybrid paradigm, where the CPUs factorize the panel while the GPU updates the trailing submatrix. However, in contrast to the QR panel factorization that accesses only a panel, each step of the QP3 panel factorization accesses the whole trailing submatrix to look for the pivot. Hence, the entire trailing submatrix must be updated before the panel factorization starts. As a result, though the transfer of the panel to the CPU can be overlapped with the remaining submatrix update, the actual panel factorization cannot be overlapped with the update (the lookahead is not possible). In addition, the QP3 factorization time is often dominated by the BLAS-2 based panel factorization that performs about a half of the total flops. Hence, our second implementation accelerates this panel factorization using a GPU. Because the flop count of the panel factorization is dominated by the matrix-vector product with the trailing submatrix (Step 1.4 of Algorithm 2), the GPU is used to accelerate this BLAS-2 kernel. As shown in Figure 5(e), our implementation computes the matrix-vector product with the top block row of the trailing submatrix on the CPUs, while the rest of the product is computed on the GPU. This is because this top block row is needed for the column norm downdating (Step 1.6), which is performed on the CPUs. Hence, this hybrid paradigm avoids the transfer of a row from the GPU to the CPU at each step of the panel factorization. Figure 5(f) illustrates this hybrid paradigm.



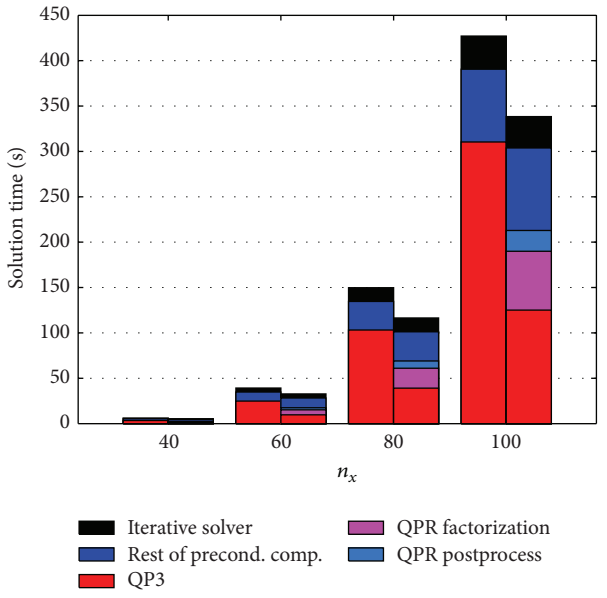
(a) Column-wise (left) and blocked (right bar) algorithm



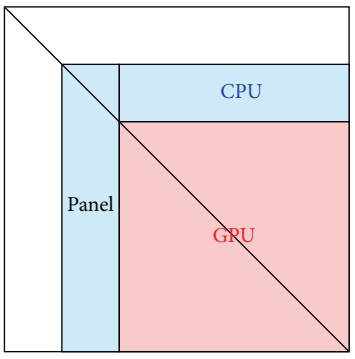
(b) Ratio of exact and estimated conditioner number of leading triangular factors after the QPR factorization



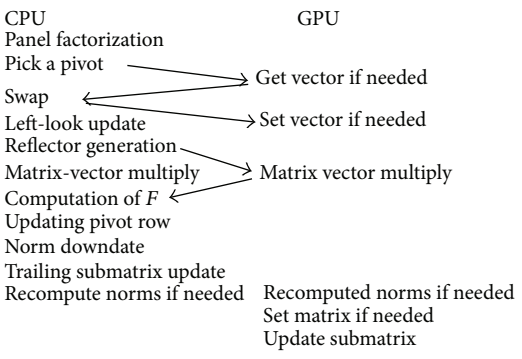
(c) StruMF solution time using task-1 (left), task-2 (middle), and task-1 and task-2 (right bar) postprocessing



(d) StruMF solution time with QP3 (left) and QPR (right bar)

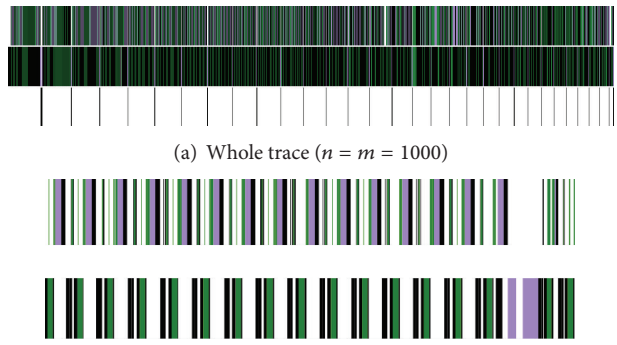


(e) Hybrid paradigm



(f) Algorithmic flow

FIGURE 5: Illustration of the hybrid QP3 implementation.



(a) Whole trace ($n = m = 1000$)

(b) Partial zoomed-in trace

Figure 6: Execution trace of the hybrid QP3 implementation. The top trace is on the CPU, while the remaining two traces are on the GPU with two GPU streams (matrix-vector multiply, matrix-matrix multiply, column swap, pivot selection, reflector generation, norm computation, and communication are in green, purple, orange, magenta, red, cyan, and black, respectively. Since the BLAS matrix-vector multiply routine does not support a vector-matrix multiply, a matrix-matrix multiply is used to compute $\mathbf{f}_j$ at Step 1.4 of Algorithm 2). The second GPU stream is used to transfer the next panel and top block row to the CPU.

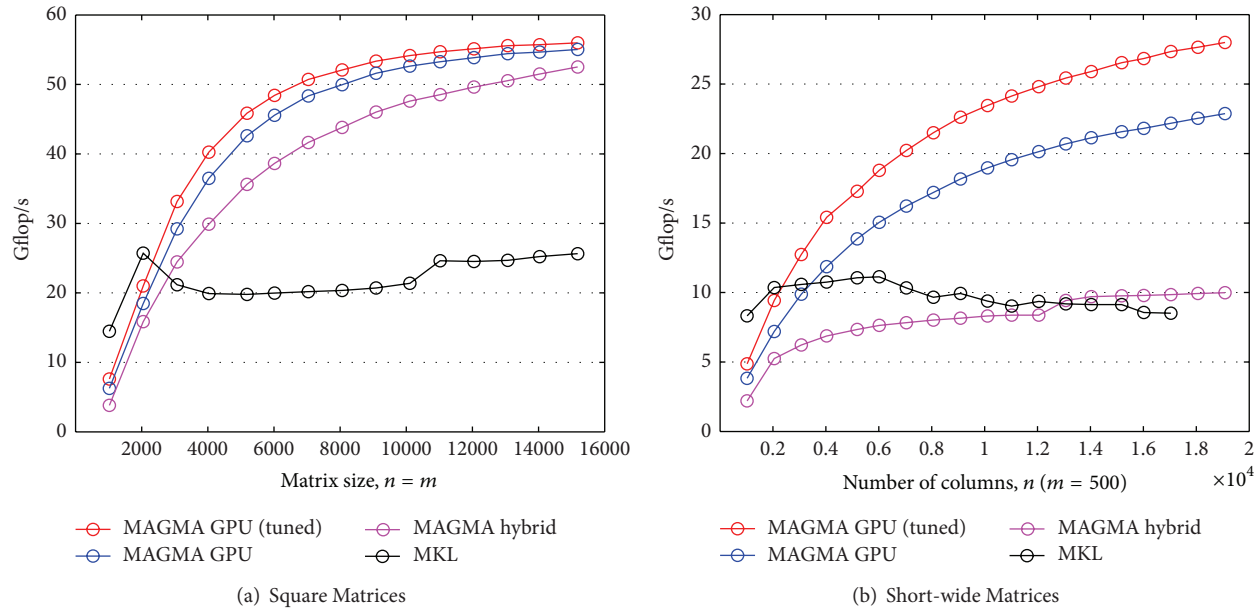
Unfortunately, in comparison to the QR factorization, this hybrid implementation of the QP3 factorization lead to much lower speedups. This is mainly because its performance is limited by the memory bandwidth to move a column between the CPU and GPU at each step of the panel factorization. This data transfer is needed for the column pivoting and matrix-vector multiply (see Figure 5(f)). As an execution trace in Figure 6 also shows, this CPU-GPU data transfer could become as expensive as the actual computation at each step of the panel factorization. To avoid this data transfer, our second implementation of QP3 performs all the computation on the GPU. At each step of the factorization, this GPU implementation still requires two synchronizations on the CPU; one to pick a pivot and the other one to check if the column norms must be recomputed (Steps 1.1 and 1.6, resp.). Furthermore, in many cases, the CPUs obtain higher performance of BLAS-1 (e.g., Householder vector generation) than the GPU. However, once the matrix is copied from the CPUs to the GPU, this GPU implementation does not transfer any vector or matrix between the CPUs and GPU and could obtain a higher performance than our hybrid implementation could (see Section 5).

Initially, our GPU implementations used an individual GPU kernel for each BLAS or LAPACK routine used for the panel factorization. However, many of these routines perform only small amounts of computation or require a scalar to be on the CPU. In order to improve the performance, we merged several computational kernels into one kernel in order to avoid the kernel launch overhead and unnecessary GPU-CPU communication. In addition, we have tuned these kernels (e.g., block size and thread grid) for the matrix dimensions typical for the QPR factorization. We will show the effects of these optimization techniques in Section 5. A similar GPU-implementation is proposed in [20].

4.2. QPR Implementation. In contrast to QP3, QPR allows lookaheads. Namely, once the GPU updates all the columns of the next window, the CPU can start the panel factorization while the GPU updates the remaining submatrix. However, in contrast to the QR panel factorization that only requires the panel, the QPR panel factorization updates all the columns within the window at each step of the panel factorization. Hence, when the window size is large in comparison to the trailing submatrix dimension, it becomes difficult to hide the BLAS-2 based panel factorization on the CPUs behind the BLAS-3 based trailing submatrix update on the GPU (we have investigated a hybrid panel factorization. However, in many cases, it was less efficient due to the CPU-GPU synchronization and communication, especially in our experiments with StruMF, where only a few columns were accepted at each panel factorization).

For Step 2 of the QPR factorization in Algorithm 5(a), we use the GPU implementation of QP3 (described in Section 4.1). Since at the end of Step 1, the coefficient matrix is on the GPU, Step 2 does not require any data transfer to the GPU. For Step 3, if the remaining submatrix is relatively small ($m < 300$ in our experiments), then we compute its QR factorization using LAPACK on the CPU. Otherwise, the QR factorization is computed using MAGMA on the GPU. We found that, in many cases, QP3 does not accept any of the rejected columns from Step 1. Hence, in order to hide the cost of copying the matrix from the GPU to the CPU for Step 3, the matrix is asynchronously copied to the CPU, while QP3 is performed on the GPU. Only when QP3 accepted a column, the matrix is resent to the CPU after the whole matrix is factorized. Finally, to generate the final orthogonal matrix Q, the Householder vectors from both QP3 and QR are accumulated on the GPU.

4.3. Integration into StruMF. To call our GPU kernels from StruMF, we copy the matrix to the GPU, compute the factorization, and then copy the result back to the CPU. Initially, we allocate a fixed amount of GPU memory for the workspace, and the workspace is reallocated when the current workspace is not large enough. Since the interfaces of most of the MAGMA routines are identical to those of LAPACK,



(a) Square Matrices

(b) Short-wide Matrices

FIGURE 7: Performance comparison of different QP3 implementations on random matrices.

replacing the LAPACK routine with the MAGMA routine is relatively easy.

5. Performance Studies with a GPU

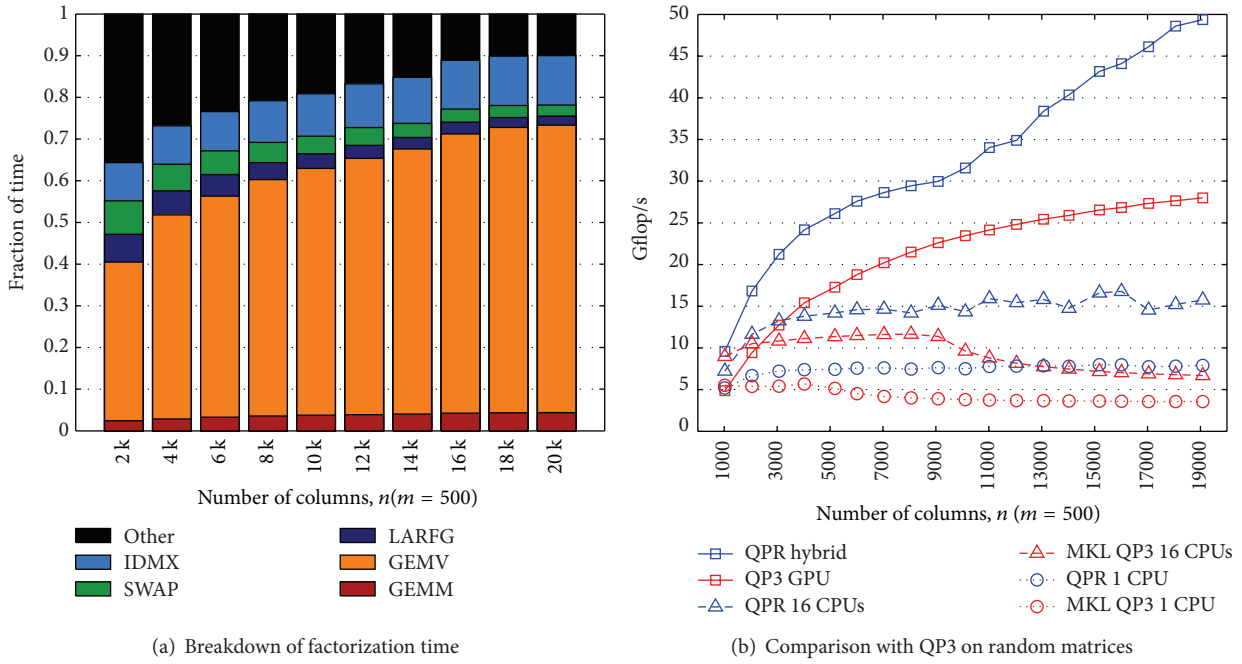
We now study the performance of our QP3 and QPR implementations with a GPU (Section 5.1) and its impacts on the performance of StruMF (Section 5.2). We compiled our codes using the C compiler gcc 4.4.6 and the CUDA compiler nvcc 5.0.35 and linked them to the threaded version of MLK 2013.4.183. We emphasize that our CPU codes have been optimized. Namely, our QPR code integrates all the modifications described in Section 3.2, and our QP3 code computes a partial factorization, where the factorization is terminated at the numerical rank specified by the tolerance τ. Computing a partial factorization obtains a significant speedup compared to the QP3 routine of MKL that computes the full factorization. This was especially true in our experiments, where a relatively large τ is used.

5.1. Kernel Performance. Figure 7(a) shows the performance of our QP3 implementations to factorize square random matrices with an NDIVIA Tesla K20c GPU ($\tau = 0.0$) and compares it with that of MKL on two eight-core Intel Sandy Bridge CPUs. Our GPU implementation improves the performance of our hybrid implementation because it avoids the data transfer between the CPU and GPU. Figure 7(b) demonstrates the advantage of the GPU implementation for the short-wide matrices. In addition, our optimized GPU implementation (Section 4.1) obtains significant speedups for both small and short-wide matrices, and this is used for the remaining of the experiments.

Figures 7(a) and 7(b) also show that all of our implementations obtain significant speedups over MKL. We observed that, for large matrices, our GPU implementation and MKL obtain similar performance relative to the practical peak performance on the corresponding hardware, where the practical peak performance is measured by computing the required matrix-vector products with the trailing submatrices and the matrix-matrix products for the submatrix update without any synchronization. In other words, our GPU implementation obtains the speedups over MKL because it effectively utilizes the higher memory and computational bandwidths of the GPU.

Figure 8(a) shows the breakdown of the QP3 factorization time. Up to 70% of the factorization time is spent in the BLAS-3 matrix-matrix and BLAS-2 matrix-vector multiplies. Even though these BLAS-3 and BLAS-2 perform about the same numbers of flops, the bandwidth-limited BLAS-2 dominates the factorization time. Figure 8(b) compares the performance of our QPR implementations with that of the QP3 implementations on random short-wide matrices. It shows that after the proposed modifications were integrated, the QPR implementation obtained the higher performance and greater scalability on the CPUs. Furthermore, our hybrid QPR implementation outperformed our QP3 GPU implementation due to the fewer BLAS-2 flops required to select the pivots. Since the performance of QP3 and QPR (especially that of QPR) depends greatly on the input matrix (e.g., the number of columns accepted at each panel factorization), in the next subsection, we study the performance of these two algorithms within StruMF.

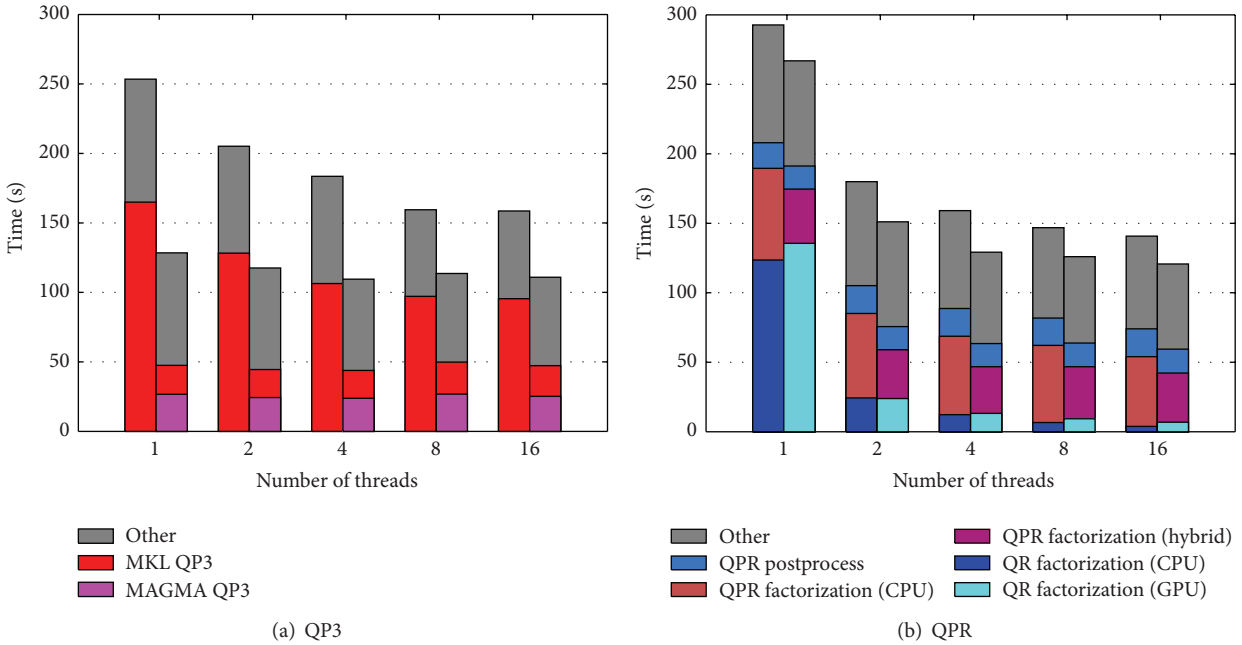
5.2. Performance of StruMF. Figure 9(a) shows the performances of StruMF using our QP3 implementations for solving a 3D Poisson's equation. First, by comparing the performance on a single CPU in the figure with that of Figure 5(a), we clearly see the advantage of computing the



(a) Breakdown of factorization time

(b) Comparison with QP3 on random matrices

Figure 8: Performance of the QP3 GPU-implementation on random short-wide matrices.



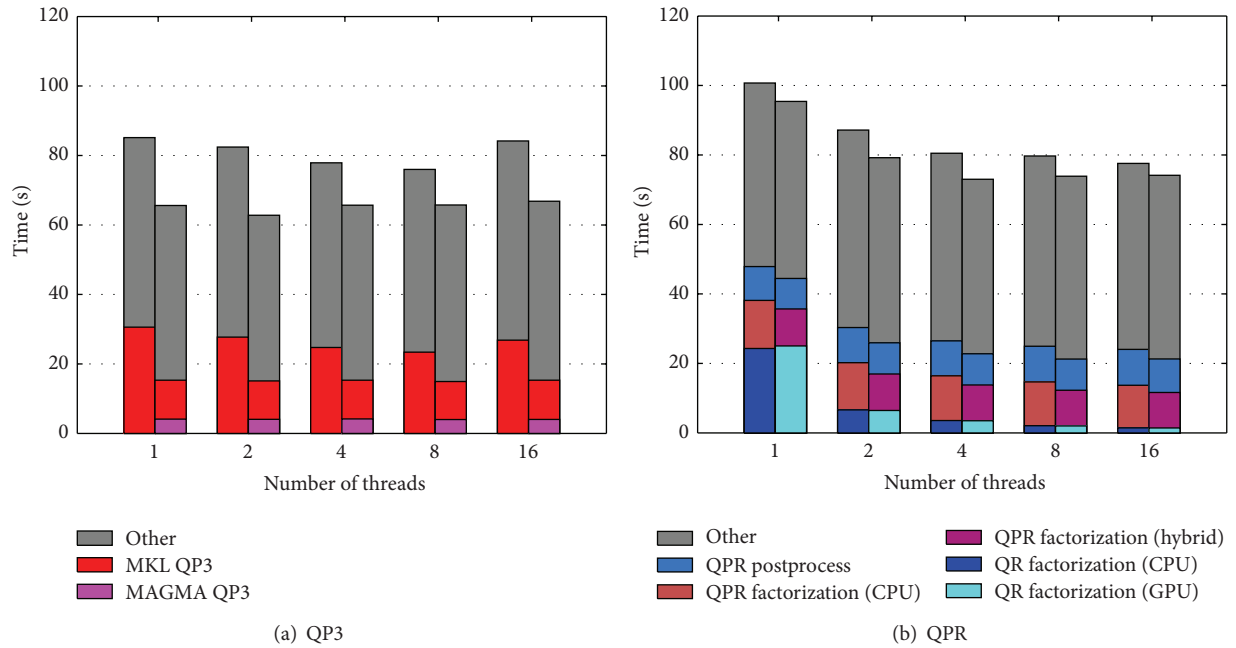
(a) QP3

(b) QPR

Figure 9: Performance of StruMF solving 3D Poisson equation using the GPU ($n_x = 100$).

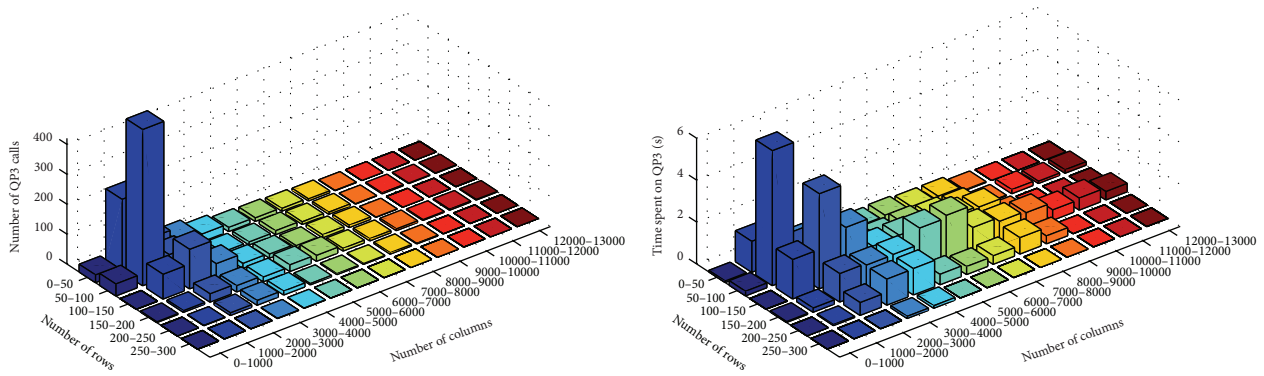
partial factorization by QP3 (the factorization is terminated at the numerical rank specified by the tolerance τ). Next, the preconditioner construction time of StruMF using QP3 is often dominated by BLAS-2 or BLAS-3 on small submatrices, and the construction time did not scale well on the CPUs. As a result, our GPU kernel was able to accelerate the construction time by 30%–50% over StruMF running on up to 16 CPUs.

Figure 9(b) then shows the performance of StruMF using QPR. Using QPR, the preconditioner construction slowed down on a single CPU, but it scaled better and was faster than using QP3 on multiple CPUs. Furthermore, the GPU reduced the construction time by 10%–20%. In comparison to QP3, the GPU acceleration was smaller in QPR mainly because the trailing submatrices were updated using smaller blocks. Most



(a) QP3 (b) QPR

Figure 10: Performance of StruMF solving 2D Poisson equations using the GPU ($n_x = 3000$).



(a) 2D problem ($n_x = 3000$) (b) 2D problem ($n_x = 3000$)

Figure 11: Statistics of off-diagonal blocks whose low-rank approximations are computed.

of the columns in each window are rejected, and significant time is spent swapping the rejected columns to the end of the matrix.

Figure 10 shows the results of solving a 2D Poisson equation, where the same tolerance τ from [5] was used (i.e., $\tau = 10^{-4}$). Though the sizes of the dense submatrices were significantly smaller in the 2D problem (see Figure 11), our GPU kernel, especially the QP3 implementation, still obtained significant speedups. Unfortunately, the compression time was less dominant in the 2D problem, and the reduction in the preconditioner construction time was less significant using the GPU.

6. Conclusion

We studied the performance of QP3 and QPR for computing the low-rank approximation of dense submatrices. We first proposed several modifications to the original QPR implementation to improve its performance on the CPUs. We then investigated the potential of using a GPU to accelerate the factorization time. Our performance results demonstrated that the proposed modifications could significantly improve the performance of the QPR factorization on the CPU, and the factorization time can be further reduced using a GPU. In addition, we provided the case studies with an hierarchically semiseparable linear solver StruMF, which showed that the preconditioner construction time of StruMF can be reduced by 30%–50% and 10%–20% using the GPU for the QP3 and QPR factorizations, respectively. Though we only show the results of solving Poisson's equations in this paper, the performance is representative of many cases and good indications for other cases. We emphasize that our focus is to study the performance of computing low-rank approximations, and our aim is not on improving the numerical performance of

StruMF. The techniques discussed in this paper are applicable to other software which computes low-rank approximations or numerical ranks of dense matrices, including those on distributed-memory systems.

We did not study the impact of the input parameters on the performance of our implementations. For instance, we used all the default parameters of StruMF and QPR (e.g., compression rate τ, iteration stopping and restarting criteria for StruMF, and window size for QPR). In particular for StruMF, a smaller compression rate would lead to larger dense blocks increasing the effectiveness of our GPU kernels, while it would also reduce the iteration count, potentially reducing the total solution time. For QPR, the smaller window size would improve the effectiveness of the lookahead and may improve the performance of our hybrid implementation. On the other hand, the larger window size could improve the performance of the trailing submatrix updates and may also lead to more accurate factorization, potentially reducing the postprocessing time. We are currently studying the effects of these parameters on the performance of StruMF. In addition, we mentioned that, in comparison to QP3, QPR obtained smaller speedups on the GPU, mainly because it uses a smaller block to update the trailing submatrix. We are currently studying prepossessing techniques that would increase the block size and improve the performance of the QPR implementation (e.g., before the factorization, reorder the matrix columns in the descending order of their norms). We also discussed that the performance of the QPR implementation depends on the performance of the condition number estimator. We are investigating if more accurate estimators are needed for other test matrices. Finally, we are studying the performance of the randomization algorithm [21] to compute the low-rank approximation on a GPU [22] and would like to investigate the performance of our QP3 implementation in a communication-avoiding version of the algorithm [23] on multiple GPUs (e.g., distributed-memory system).

Conflict of Interests

The authors declare that there is no conflict of interests regarding the publication of this paper.

Acknowledgments

The authors thank Artem Napov, Xiaoye Li, and Ming Gu for sharing StruMF software and exchanging helpful discussion with us. This research was supported in part by DOD and NSF SDCI, National Science Foundation Award no. OCI-1032815, "Collaborative Research: SDCI HPC Improvement: Improvement and Support of Community Based Dense Linear Algebra Software for Extreme Scale Computational Science," and Russian Scientific Fund, Agreement N14-11-00190.

References

[1] P. Businger and G. H. Golub, "Linear least squares solutions by Householder transformations," *Numerische Mathematik*, vol. 7, pp. 269–276, 1965.

[2] G. Quintana-Ortí, X. Sun, and C. H. Bischof, "A BLAS-3 version of the QR factorization with column pivoting," *SIAM Journal on Scientific Computing*, vol. 19, no. 5, pp. 1486–1494, 1998.

[3] C. H. Bischof and G. Quintana-Ortí, "Computing rank-revealing QR factorizations of dense matrices," *Association for Computing Machinery. Transactions on Mathematical Software*, vol. 24, no. 2, pp. 226–253, 1998.

[4] G. Quintana-Ortí and E. S. Quintana-Ortí, "Parallel code for computing the numerical rank," *Linear Algebra and its Applications*, vol. 275-276, pp. 451–470, 1998.

[5] A. Napov, X. Li, and M. Gu, "A parallel black box multifrontal preconditioner that exploits a low-rank structure," in *Proceedings of the SIAM Conference on Parallel Processing for Scientific Computing*, 2012, http://homepages.ulb.ac.be/~anapov/Talks/LBL12.pdf.

[6] A. Napov and X. S. Li, "An algebraic multifrontalpreconditioner that exploits the low-rank property," Tech. Rep., UniversiteLibre de Bruxelles, Brussel, Belgium, 2014, http://homepages.ulb.ac.be/~anapov/pub/strumf.pdf.

[7] S. Chandrasekaran, M. Gu, and T. Pals, "A fast ULV decomposition solver for hierarchically semiseparable representations," *SIAM Journal on Matrix Analysis and Applications*, vol. 28, no. 3, pp. 603–622, 2006.

[8] G. H. Golub and C. F. van Loan, *Matrix Computations*, The Johns Hopkins University Press, Baltimore, Md, USA, 4th edition, 2013.

[9] S. Chandrasekaran and I. C. Ipsen, "On rank-revealing factorisations," *SIAM Journal on Matrix Analysis and Applications*, vol. 15, no. 2, pp. 592–622, 1994.

[10] E. Anderson, Z. Bai, C. Bischof et al., *LAPACK Users' Guide*, Society for Industrial and Applied Mathematics, 3rd edition, 1999.

[11] C. Bischof and C. van Loan, "The WY representation for products of Householder matrices," *Society for Industrial and Applied Mathematics*, vol. 8, no. 1, pp. S2–S13, 1987.

[12] R. Schreiber and C. van Loan, "A storage-efficient WY representation for products of Householder transformations," *Society for Industrial and Applied Mathematics*, vol. 10, no. 1, pp. 53–57, 1989.

[13] G. Stewart, "Incremental condition calculation and column selection," Tech. Rep. UMIACS- TR 90-87, Department of Computer Science and Institute for Advanced Computer Studies, University of Maryland, College Park, Md, USA, 1990.

[14] Z. Drmač and Z. Bujanović, "On the failure of rank-revealing QR factorization software—a case study," *ACM Transactions on Mathematical Software*, vol. 35, no. 2, article 12, 2008.

[15] C.-T. Pan and P.-T. Tang, "Bounds on singular values revealed by QR factorizations," *BIT Numerical Mathematics*, vol. 39, no. 4, pp. 740–756, 1999.

[16] C. H. Bischof, "Incremental condition estimation," *SIAM Journal on Matrix Analysis and Applications*, vol. 11, no. 2, pp. 312–322, 1990.

[17] C. Bischof and P. Tang, "Robust incremental condition estimation," Tech. Rep. CS-91-133, Mathematics and Computer Science Division, Argonne National Laboratory, 1991, LAPACK Working Note 33.

[18] S. Tomov, J. Dongarra, and M. Baboulin, "Towards dense linear algebra for hybrid GPU accelerated manycore systems," *Parallel Computing*, vol. 36, no. 5-6, pp. 232–240, 2010.

[19] R. Nath, S. Tomov, and J. Dongarra, "An improved MAGMA GEMM for Fermi graphics processing units," *International*

Journal of High Performance Computing Applications, vol. 24, no. 4, pp. 511–515, 2010.

[20] A. Tomás, Z. Bai, and V. Hernández, "Parallelization of the QR decomposition with column pivoting using column cyclic distribution on multicore and GPU processors," in *High Performance Computing for Computational Science—VECPAR 2012: Proceedings of the 10th International Conference, Kope, Japan, July 17–20, 2012, Revised Selected Papers*, vol. 7851 of *Lecture Notes in Computer Science*, pp. 50–58, Springer, Berlin, Germany, 2013.

[21] F. Woolfe, E. Liberty, V. Rokhlin, and M. Tygert, "A fast randomized algorithm for the approximation of matrices," *Applied and Computational Harmonic Analysis: Time-Frequency and Time-Scale Analysis, Wavelets, Numerical Algorithms, and Applications*, vol. 25, no. 3, pp. 335–366, 2008.

[22] T. Mary, I. Yamazaki, J. Kurzak, P. Luszczek, S. Tomov, and J. Dongarra, "Performance of randomsampling for computing low-rank approximations of a dense matrix on GPUs," in press.

[23] J. Demmel, L. Grigori, M. Gu, and H. Xiang, "Communication avoiding rank revealing QR factorization with column pivoting," LAPACK Working Note 276, 2013.