SIAM J. SCI. COMPUT.
Vol. 37, No. 3, pp. C307–C330

MIXED-PRECISION CHOLESKY QR FACTORIZATION AND ITS CASE STUDIES ON MULTICORE CPU WITH MULTIPLE GPUS*

ICHITARO YAMAZAKI†, STANIMIRE TOMOV†, AND JACK DONGARRA†

Abstract. To orthonormalize the columns of a dense matrix, the Cholesky QR (CholQR) requires only one global reduction between the parallel processing units and performs most of its computation using BLAS-3 kernels. As a result, compared to other orthogonalization algorithms, CholQR obtains superior performance on many of the current computer architectures, where the communication is becoming increasingly expensive compared to the arithmetic operations. This is especially true when the input matrix is tall-skinny. Unfortunately, the orthogonality error of CholQR depends quadratically on the condition number of the input matrix, and it is numerically unstable when the matrix is ill-conditioned. To enhance the stability of CholQR, we recently used mixed-precision arithmetic; the input and output matrices are in the working precision, but some of its intermediate results are accumulated in the doubled precision. In this paper, we analyze the numerical properties of this mixed-precision CholQR. Our analysis shows that by selectively using the doubled precision, the orthogonality error of the mixed-precision CholQR only depends linearly on the condition number of the input matrix. We provide numerical results to demonstrate the improved numerical stability of the mixed-precision CholQR in practice. We then study its performance. When the target hardware does not support the desired higher precision, software emulation is needed. For example, using software-emulated double-double precision for the working 64-bit double precision, the mixed-precision CholQR requires about 8.5× more floating-point instructions than that required by the standard CholQR. On the other hand, the increase in the communication cost using the double-double precision is less significant, and our performance results on multicore CPU with a different graphics processing unit (GPU) demonstrate that the overhead of using the double-double arithmetic is decreasing on a newer architecture, where the computation is becoming less expensive compared to the communication. As a result, with a latest NVIDIA GPU, the mixed-precision CholQR was only 1.4× slower than the standard CholQR. Finally, we present case studies of using the mixed-precision CholQR within communication-avoiding variants of Krylov subspace projection methods for solving a nonsymmetric linear system of equations and for solving a symmetric eigenvalue problem, on a multicore CPU with multiple GPUs. These case studies demonstrate that by using the higher precision for this small but critical segment of the Krylov methods, we can improve not only the overall numerical stability of the solvers but also, in some cases, their performance.

Key words. mixed-precision, orthogonalization, GPU computation

AMS subject classification. 65F25

DOI. 10.1137/14M0973773

1. Introduction.

Orthogonalizing a set of dense column vectors plays a salient part in many scientific and engineering applications, having great effects on both their numerical stability and performance. For example, it is an important kernel in a software package that solves a large-scale linear system of equations or eigenvalue

*Submitted to the journal's Software and High-Performance Computing section June 24, 2014; accepted for publication (in revised form) February 27, 2015; published electronically May 12, 2015. This research was supported in part by NSF SDCI - National Science Foundation Award OCI-1032815, "Collaborative Research: SDCI HPC Improvement: Improvement and Support of Community Based Dense Linear Algebra Software for Extreme Scale Computational Science," DOE grant DE-SC0010042: "Extreme-scale Algorithms & Solver Resilience (EASIR)," NSF Keeneland - Georgia Institute of Technology Subcontract RA241-G1 on NSF Prime Grant OCI-0910735, and "Matrix Algebra for GPU and Multicore Architectures (MAGMA) for Large Petascale Systems." This research used resources of the Keeneland Computing Facility at the Georgia Institute of Technology, which is supported by the National Science Foundation under contract OCI-0910735.
http://www.siam.org/journals/sisc/37-3/M97377.html

†Department of Electrical Engineering and Computer Science, University of Tennessee, Knoxville, TN 37996 (iyamazak@eecs.utk.edu, tomov@eecs.utk.edu, dongarra@eecs.utk.edu).

problem. In many of the solvers, the input matrix to be orthogonalized is *tall-skinny*, having more rows than columns. This is true, for instance, in a subspace projection method that computes the orthonormal basis vectors of the generated projection subspace [17, 18]. Other applications of the tall-skinny orthogonalization include the solution of an overdetermined least-squares problem [9] and data analysis based on random projection methods [11]. Hence, an efficient and stable tall-skinny orthogonalization scheme can have great impacts in many applications, especially if it is suited for current and emerging computer architectures.

On the current computer architectures, communication is becoming increasingly expensive compared to arithmetic operations, where communication includes data movement or synchronization between parallel processing units, as well as data movement between the levels of the local memory hierarchy [8, 10]. This holds in terms of both time and energy consumptions. It is critical to take this hardware trend into consideration when designing high-performance software for new and emerging computers. To orthogonalize a tall-skinny matrix, the Cholesky QR (CholQR) [21] requires only one global reduction between the parallel processing units, while performing most of its computation using BLAS-3 kernels. Hence, compared to other orthogonalization schemes, it obtains exceptional performance on many of the modern computers. However, the orthogonality error of CholQR is bounded by the squared condition number of the input matrix, and it is numerically unstable for an ill-conditioned input matrix.

To address the numerical deficiency of CholQR, we proposed a mixed-precision variant that reads and writes the input and output matrices, respectively, in the working precision but accumulates some of its intermediate results in the doubled precision [27]. When the doubled precision is not supported by the hardware, our mixed-precision scheme uses software-emulated arithmetic and may perform significantly more floating-point operations (flops) than the standard CholQR. However, the increase in its communication cost is often less significant. Namely, compared to the standard CholQR, our mixed-precision CholQR moves about the same amount of data between the levels of the local memory hierarchy. The communication cost between the parallel processing units is doubled in its volume. However, the amount of data communicated between the processing units is typically much less than that required on each processing unit, and the communication requires the same latency as the standard CholQR. To study the performance of the mixed-precision CholQR, in [27], we used the double-double arithmetic [12] to emulate the higher precision for the working 64-bit double precision on a multicore CPU with a different graphics processing unit (GPU). Using the double-double arithmetic, our mixed-precision CholQR performs about 8.5× more floating-point instructions than the standard CholQR, but our performance results demonstrated that the overhead of using the double-double arithmetic is decreasing on the newer architectures, where the computation is becoming less expensive compared to the communication. As a result, with the latest NVIDIA GPU, the mixed-precision CholQR was only 1.4× slower than the standard CholQR. We also provided case studies of using the mixed-precision CholQR within a communication-avoiding variant [13] of the generalized minimum residual (GMRES) method [19], called CA-GMRES, for solving a nonsymmetric linear system of equations on a multicore CPU with a GPU. The case studies demonstrated that this mixed-precision scheme can improve the overall numerical stability of the iterative solvers without a significant increase in the orthogonalization time. As a result, by using the higher precision for this small but critical segment of the Krylov methods, we were able to improve not only the stability of the iterative solver but also, in some

```
Step 1: Form Gram matrix B
    for d = 1, 2, ..., n_g do
        B^(d) := V^(d)T V^(d) on dth GPU
    end for
    B := sum_{d=1}^{n_g} B^(d) (GPUs to CPU reduction)

Step 2: Compute Cholesky factorization of B
    R := chol(B) on CPU

Step 3: Orthonormalize V
    copy R to all the GPUs (CPU to GPUs broadcast)
    for d = 1, 2, ..., n_g do
        Q^(d) := V^(d) R^{-1} on dth GPU
    end for
```

FIG. 1. *CholQR implementation on multicore CPU with multiple GPUs, where* $\texttt{chol}(B)$ *computes the Cholesky factorization of matrix* B.

cases, its performance.

The main contributions of this paper, over our initial reports on the mixed-precision CholQR [27], are (1) theoretical analysis of the mixed-precision CholQR, deriving upper bounds on the orthogonality error and the condition number of the computed orthogonal matrix; and (2) numerical results to study the stability of the mixed-precision CholQR in practice. In addition, we compliment our initial report with the following studies: (1) performance studies of the mixed-precision CholQR in the working 32-bit single precision with a GPU, where the higher precision is the 64-bit double precision and is supported by the hardware; (2) case studies with CA-GMRES on multiple GPUs (i.e., our initial study [27] used only one GPU); and (3) case studies with a communication-avoiding variant [13] of the Lanczos method [16], called CA-Lanczos, for solving a symmetric eigenvalue problem. The rest of the paper is organized as follows: First, in section 2, we discuss our implementations of several existing tall-skinny orthogonalization procedures, including CholQR, on a multicore CPU with multiple GPUs. Then, in section 3, we describe the implementation of our mixed-precision CholQR with the GPUs and analyze its numerical properties and performance. Finally, in section 4, we study the effects of the mixed-precision scheme on the performance of CA-GMRES and CA-Lanczos on a multicore CPU with multiple GPUs. We provide final remarks in section 5.

2. Cholesky QR factorization. We consider computing the QR factorization of an m-by-n tall-skinny dense matrix V (i.e., $m \gg n$),

$$V = QR,$$

where Q is an m-by-n matrix with orthonormal columns and R is an n-by-n upper-triangular matrix. To utilize multiple GPUs to compute the QR factorization using CholQR, we distribute the matrices V and Q in a one-dimensional (1D) block row format, where we let n_g be the number of available GPUs and $V^{(d)}$ be the block row of the matrix V distributed to the dth GPU. Then, we first form the Gram matrix $B := V^T V$ through the matrix-matrix product $B^{(d)} := V^{(d)T} V^{(d)}$ on the dth GPU, followed by the reduction $B := \sum_{d=1}^{n_g} B^{(d)}$ on the CPU. We then compute the

Modified Gram–Schmidt (MGS)
for $j = 1, 2, \ldots, n$ **do**
  for $i = 1, 2, \ldots, j-1$ **do**
    for $d = 1, 2, \ldots, n_g$ **do**
      $r_{i,j}^{(d)} := \mathbf{v}_i^{(d)T} \mathbf{v}_j^{(d)}$
    end for
    $r_{i,j} := \sum_{d=1}^{n_g} r_{i,j}^{(d)}$ from GPUs to CPU
    broadcast $r_{i,j}$ from CPU to all GPUs
    for $d = 1, 2, \ldots, n_g$ **do**
      $\mathbf{v}_j^{(d)} := \mathbf{v}_j^{(d)} - \mathbf{v}_i^{(d)} r_{i,j}$
    end for
  end for
  for $d = 1, 2, \ldots, n_g$ **do**
    $r_{j,j}^{(d)} := \mathbf{v}_j^{(d)T} \mathbf{v}_j^{(d)}$
  end for
  $r_{j,j} := \sqrt{\sum_{d=1}^{n_g} r_{j,j}^{(d)}}$ from GPUs to CPU
  broadcast $r_{j,j}$ from CPU to all GPUs
  for $d = 1, 2, \ldots, n_g$ **do**
    $\mathbf{q}_j^{(d)} := \mathbf{v}_j^{(d)} / r_{j,j}$
  end for
end for

Classical Gram–Schmidt (CGS)
for $j = 1, 2, \ldots, n$ **do**
  for $d = 1, 2, \ldots, n_g$ **do**
    $\mathbf{r}_{1:j-1,j}^{(d)} := Q_{1:j-1}^{(d)T} \mathbf{v}_j^{(d)}$
  end for
  $\mathbf{r}_{1:j-1,j} := \sum_{d=1}^{n_g} \mathbf{r}_{1:j-1,j}^{(d)}$ from GPUs to CPU
  broadcast $\mathbf{r}_{1:j-1,j}$ from CPU to all GPUs
  for $d = 1, 2, \ldots, n_g$ **do**
    $\mathbf{v}_j^{(d)} := \mathbf{v}_j^{(d)} - Q_{1:j-1}^{(d)} \mathbf{r}_{1:j-1,j}$
    $r_{j,j}^{(d)} := \mathbf{v}_j^{(d)T} \mathbf{v}_j^{(d)}$
  end for
  $r_{j,j} := \sqrt{\sum_{d=1}^{n_g} r_{j,j}^{(d)}}$ from GPUs to CPU
  broadcast $r_{j,j}$ from CPU to all GPUs
  for $d = 1, 2, \ldots, n_g$ **do**
    $\mathbf{q}_j^{(d)} := \mathbf{v}_j^{(d)} / r_{j,j}$
  end for
end for

Singular Value QR (SVQR)
for $d = 1, 2, \ldots, n_g$ **do**
  $B^{(d)} := V^{(d)T} V^{(d)}$
end for
$B := \sum_{d=1}^{n_g} B^{(d)}$ from GPUs to CPU
$[U, \Sigma, U] := \texttt{svd}(B)$ on CPU
$[V, R] := \texttt{qr}(\sqrt{\Sigma} U^T)$ on CPU
broadcast R from CPU to all GPUs
for $d = 1, 2, \ldots, n_g$ **do**
  $Q^{(d)} := V^{(d)} R^{-1}$
end for

Communication-Avoiding QR (CAQR)
for $d = 1, 2, \ldots, n_g$ **do**
  $[X^{(d)}, R^{(d)}] := \texttt{qr}(V^{(d)})$
end for
$C := [R^{(1)}; R^{(2)}; \ldots; R^{(n_g)}]$ from GPUs to CPU
$[[Y^{(1)}; Y^{(2)}; \ldots, Y^{(n_g)}], R] = \texttt{qr}(C)$ on CPU
broadcast $Q^{(d)}$ from CPU to GPUs
for $d = 1, 2, \ldots, n_g$ **do**
  $Q^{(d)} := X^{(d)} Y^{(d)}$
end for

FIG. 2. *TSQR implementations on multicore CPU with multiple GPUs, where* $\texttt{svd}(B)$ *and* $\texttt{qr}(\sqrt{\Sigma} U^T)$ *compute the SVD and QR factorizations of the matrices* B *and* $\sqrt{\Sigma} U^T$*, respectively.*

		$\|I - Q^T Q\|_2$ upper bound	# flops	dominant kernel	# messages
MGS	[5]	$O(\epsilon\kappa(V))$	$2n^2m$	BLAS-1 xDOT	$O(n^2)$
CGS	[15]	$O(\epsilon\kappa(V)^{n-1})$	$2n^2m$	BLAS-2 xGEMV	$O(n)$
CholQR	[21]	$O(\epsilon\kappa(V)^2)$	$2n^2m$	BLAS-3 xGEMM	$O(1)$
SVQR	[21]	$O(\epsilon\kappa(V)^2)$	$2n^2m$	BLAS-3 xGEMM	$O(1)$
CAQR	[7]	$O(\epsilon)$	$4n^2m$	BLAS-1,2 xGEQF2	$O(1)$

FIG. 3. *Properties of standard* TSQR *implementations to orthonormalize an m-by-n matrix* V*, where* $\kappa(V)$ *is the condition number of* V*.*

Cholesky factor R of B on the CPU (i.e., $R^T R := B$). Finally, the GPU orthogonalizes V by the triangular solves $Q^{(d)} := V^{(d)} R^{-1}$. Hence, all the required inter-GPU communication is aggregated into one global reduce among the GPUs and a data copy from the CPU to its local GPU, while most of the computation is performed using BLAS-3 kernels on the GPUs. As a result, both intra- and inter-GPU communication can be optimized. Figure 1 shows these three steps of CholQR. Unfortunately, the condition number of the Gram matrix B is the square of the input matrix V, and CholQR causes numerical instability, when V is ill-conditioned.

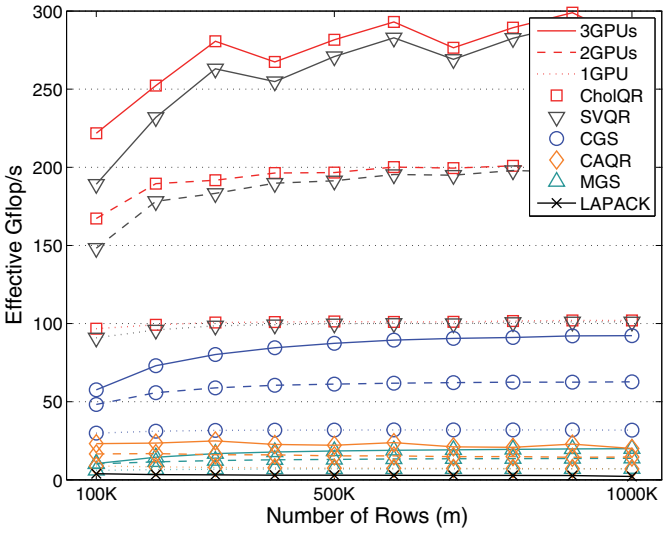


FIG. 4. *Performance of standard* TSQR *implementations in double precision with up to three NVIDIA Tesla* M2090 *GPUs. "LAPACK" uses DGEQRF and DORGQR of threaded MKL on* 16*-core Intel SandyBridge CPU, and "effective Gflop/s" is computed as the ratio of the total flops required by DGEQRF and DORGQR for the same input matrix over the orthogonalization time, in seconds. The number of columns is fixed at* 30 *(i.e.,* $n = 30$*).*

Besides CholQR, we considered several orthogonalization strategies on a multicore CPU with multiple GPUs. For instance, the Classical Gram–Schmidt (CGS) procedure [9] orthogonalizes the jth column $\mathbf{v}_j$ of V against the previously orthonormalized columns $Q_{1:j-1}$ based on BLAS-2, all at once for $j = 1, 2, \ldots, n$ (see Figure 2). Unfortunately, compared to CholQR, CGS has the greater upper-bound on its orthogonality error (see Figure 3), and BLAS-2 based CGS often obtains a much lower performance than BLAS-3 based CholQR does. The Modified Gram-Schmidt (MGS) procedure [9] significantly improves the numerical stability of CGS. However, the procedure is implemented using BLAS-1 when $\mathbf{v}_j$ is orthogonalized against $\mathbf{q}_1, \mathbf{q}_2, \ldots, \mathbf{q}_{j-1}$, one at a time in our implementation, or using BLAS-2 when all the remaining columns $V_{j:n}$ are orthogonalized against $\mathbf{q}_{j-1}$ at once (i.e., $V_{j:n} := V_{j:n} - \mathbf{q}_{j-1}(\mathbf{q}_{j-1}^T V_{j:n})$). Hence, it still obtains only a fraction of the CholQR performance. Communication-avoiding QR (CAQR) [7] orthogonalizes the set of n vectors V against each other through a tree reduction of the local QR factorizations. Just like CholQR, to orthogonalize V, CAQR requires only one global reduce among the GPUs. However, in our implementation, the local factorization is the BLAS-1 and BLAS-2 based Householder QR factorization on each GPU, and its performance is close to that of MGS on a single compute node, where the performance depends more on the intra-GPU communication than on the inter-GPU communication.[1] Finally, the Singular Value QR (SVQR) factorization [21] computes the upper-triangular matrix R by first computing the singular value decomposition (SVD) of the Gram matrix, $U\Sigma U^T := B$, followed by the QR factorization of $\Sigma^{\frac{1}{2}} U^T$. Then, the column vectors are orthonormalized through the triangular solve $Q^{(d)} := V^{(d)} R^{-1}$. Compared to the Cholesky factorization, computing

[1] We are looking to implement CAQR on each GPU using a batched QR kernel.

the SVD and QR factorization of the Gram matrix is computationally more expensive. However, the dimension of the Gram matrix is much smaller than that of the input matrix V (i.e., $n \ll m$). Hence, SVQR performs about the same number of flops as CholQR, using the same BLAS-3 kernels, and requires only a pair of the CPU-GPU communications. On the other hand, when the matrix V is ill-conditioned, or one of the column vectors $\mathbf{v}_j$ is a linear combination of the other columns, the Cholesky factorization of the Gram matrix may fail, while SVQR overcomes this numerical challenge. However, SVQR still requires computing the Gram matrix, leading to the same normwise upper-bound on the orthogonality error as CholQR. In this paper, we focus on CholQR since its implementation is simpler than that of SVQR, and in our previous studies, we did not identify a test case where CA-GMRES converges with SVQR but not with CholQR [25]. Nevertheless, most of the numerical analysis for CholQR can be trivially extended to SVQR [21]. As a summary, Figure 2 shows our implementations of various algorithms to compute tall-skinny QR (*TSQR*), and Figure 3 lists some of their properties. In addition, Figure 4 shows their performance on up to three NVIDIA Tesla M2090 GPUs. A more detailed description of our implementations and the performance of these standard orthogonalization schemes can be found in [25].[2]

3. Mixed-precision Cholesky QR factorization. In this section, we first analyze the numerical properties of the mixed-precision CholQR and present experimental results to study its numerical stability in practice (sections 3.1 and 3.2). We then describe our implementation on a multicore CPU with multiple GPUs and show its performance (sections 3.3 and 3.4).

3.1. Error analysis. Let us analyze the numerical errors of a mixed-precision CholQR in finite precision, where a different numerical precision is used at each step of the factorization. Following the standard error analysis, we let ϵ_i be the machine epsilon used at Step i of CholQR; E_i be the round-off errors introduced at Step i; c_i be a small constant scalar; and $\widehat{B}$ be the result of computing B in the finite precision. Then, we can express the finite precision operations at each step of CholQR as follows:

Step 1. Computation of Gram matrix (i.e., $B := V^T V$):
$$\widehat{B} = B + E_1, \qquad \text{where } \|E_1\| \le c_1\epsilon_1\|V\|^2.$$
Step 2. Cholesky factorization (i.e., $R^T R := \widehat{B}$):
$$\widehat{R}^T\widehat{R} = R^T R + E_2, \quad \text{where } \|E_2\| \le c_2\epsilon_2\|\widehat{B}\|.$$
Step 3. Forward-substitution (i.e., $Q := V\widehat{R}^{-1}$):
$$\widehat{Q} = Q + E_3, \qquad \text{where } \|E_3\| \le c_3\epsilon_3\|V\|\|\widehat{R}^{-1}\|.$$

For the bound on the numerical error at Step 2 [22, Theorem 23.2], the Cholesky factorization is assumed to succeed. In other words, we assume in this paper that the condition number of $\widehat{B}$ is less than the reciprocal of the machine precision used at Step 2 (i.e., $\kappa(\widehat{B}) \le \epsilon_2^{-1}$ or, equivalently, $\kappa(V)^2 \le \epsilon_2^{-1}$).

The following lemma from [21] will be used to prove Theorem 3.2, which provides an upper-bound on the orthogonality error of the mixed-precision CholQR.

[2]Previously, the blocked variants of *TSQR* have been studied [1, 2, 4]. To generate $n+1$ orthonormal basis vectors, our CA-GMRES and CA-Lanczos [25] use block orthogonalization followed by TSQR with a step size of s, where the step size is equivalent to the block size in the blocked algorithm to orthogonalize $n+1$ vectors (e.g., $n = 60$ and $s = 15$ in our experiments). We present the experimental results with CA-GMERS and CA-Lanczos in section 4, but we have not studied the blocked algorithms for TSQR.

LEMMA 3.1. *The upper-triangular matrix $\widehat{R}$ computed by CholQR satisfies the following equalities:*

$$\|\widehat{R}\|_2 = c_{\max}\sigma_n(V) \quad \text{and} \quad \|\widehat{R}^{-1}\|_2 = c_{\min}\sigma_1(V)^{-1},$$

where $\sigma_i(V)$ is the ith smallest singular value of V (i.e., $\sigma_1 \le \sigma_2 \le \cdots \le \sigma_n$), and both $c_{\max}$ and $c_{\min}$ are small scalar constants.

Proof. For a symmetric matrix, the perturbation in its eigenvalue is bounded by a perturbation in the matrix [9, Corollary 8.1.6]. Hence, for the symmetric semidefinite matrix V^TV with $\|\widehat{B} - V^TV\|_2 \le c_1\epsilon_1\|V\|_2^2$, we have $\|\widehat{B}\|_2 \le (1 + c_1\epsilon_1)\|V\|_2^2$ and

$$\widehat{R}^T\widehat{R} = B + E_{1,2},$$

where $E_{1,2} = E_1 + E_2$ and

$$\begin{aligned}\|E_{1,2}\|_2 &\le c_1\epsilon_1\|V\|_2^2 + c_2\epsilon_2\|\widehat{B}\|_2,\\ &\le c_{1,2}\epsilon_{1,2}\|V\|_2^2 \quad (\text{with } c_{1,2} = c_1 + c_2 + O(\min(\epsilon_1,\epsilon_2)) \text{ and } \epsilon_{1,2} = \max(\epsilon_1,\epsilon_2)).\end{aligned}$$

As a result, for the symmetric semidefinite matrices B and $\widehat{R}^T\widehat{R}$, their corresponding singular values satisfy the relation

$$|\sigma_i(\widehat{R}^T\widehat{R}) - \sigma_i(B)| \le \|\widehat{R}^T\widehat{R} - B\|_2 \le c_{1,2}\epsilon_{1,2}\|B\|_2,$$

and

$$\sigma_i(\widehat{R}^T\widehat{R}) = \sigma_i(B) + c_{\sigma_i}\epsilon_{1,2}\|B\|_2, \text{ where } |c_{\sigma_i}| \le c_{1,2}.$$

In particular, with their largest singular values, we have

$$\|\widehat{R}\|_2 = \sqrt{1 + c_{\sigma_n}\epsilon_{1,2}}\,\sigma_n(V),$$

and with the smallest singular values, we have

$$\begin{aligned}\|\widehat{R}^{-1}\|_2 &= \frac{1}{\sqrt{\sigma_1(B) + c_{\sigma_1}\epsilon_{1,2}\sigma_n(B)}}\\ &= \frac{1}{\sqrt{1 + c_\sigma\epsilon_{1,2}\kappa(B)}}\,\sigma_1(V)^{-1}. \qquad \square\end{aligned}$$

THEOREM 3.2. *The orthogonality error norm of the matrix $\widehat{Q}$ computed by the mixed-precision CholQR is bounded by*

$$\|I - \widehat{Q}^T\widehat{Q}\|_2 \le O\left(\epsilon_{1,2}\kappa(V)^2 + \epsilon_3\|\widehat{Q}\|_2\kappa(V) + (\epsilon_3\kappa(V))^2\right).$$

Proof. The orthogonality error of CholQR is given by

$$\begin{aligned}I - \widehat{Q}^T\widehat{Q} &= I - (Q + E_3)^T(Q + E_3)\\ &= I - Q^TQ - Q^TE_3 - E_3^TQ - E_3^TE_3\\ &= I - \widehat{R}^{-T}V^TV\widehat{R}^{-1} - Q^TE_3 - E_3^TQ - E_3^TE_3\\ &= \widehat{R}^{-T}(\widehat{R}^T\widehat{R} - B)\widehat{R}^{-1} - Q^TE_3 - E_3^TQ - E_3^TE_3\\ &= \widehat{R}^{-T}E_{1,2}\widehat{R}^{-1} - Q^TE_3 - E_3^TQ - E_3^TE_3\\ &= \widehat{R}^{-T}E_{1,2}\widehat{R}^{-1} - \widehat{Q}^TE_3 - E_3^T\widehat{Q} + E_3^TE_3 \quad (\text{since } Q = \widehat{Q} - E_3).\end{aligned}$$

Hence, the error norm is bounded by

$$\|I - \widehat{Q}^T\widehat{Q}\|_2 \quad \leq \quad c_{1,2}\epsilon_{1,2}\|V\|_2^2\|\widehat{R}^{-1}\|_2^2 + 2c_3\epsilon_3\|Q\|_2\|V\|_2\|\widehat{R}^{-1}\|_2 + c_3^2\epsilon_3^2\|V\|_2^2\|\widehat{R}^{-1}\|_2^2,$$

where $\|V\|_2\|\widehat{R}^{-1}\|_2 = \sigma_n(V)/\sigma_1(\widehat{R}) = c_{\min}\sigma_n(V)/\sigma_1(V) = c_{\min}\kappa(V)$. ∎

For the remainder of the paper, we focus on the mixed-precision CholQR which uses the doubled precision at the first two steps of CholQR, while using the working precision at the last step. The following theorem specializes Theorem 3.2 for this particular implementation of the mixed-precision CholQR.

THEOREM 3.3. *If the doubled precision is used for the first two steps of CholQR, and ϵ_d is the machine epsilon in the working precision (i.e., $\epsilon_1 = \epsilon_2 = \epsilon_d^2$ and $\epsilon_3 = \epsilon_d$), then we have*

$$\|I - \widehat{Q}^T\widehat{Q}\|_2 \leq O\left(\epsilon_d\kappa(V) + (\epsilon_d\kappa(V))^2\right)$$

and

$$\|\widehat{Q}\|_2 = 1 + O\left((\epsilon_d\kappa(V))^{1/2} + \epsilon_d\kappa(V)\right).$$

In particular, with our assumption made at Step 2 of CholQR (i.e., $\kappa(V) \leq \epsilon_d^{-1}$), we have

$$\|I - \widehat{Q}^T\widehat{Q}\|_2 \leq O\left(\epsilon_d\kappa(V)\right) \text{ and } \|\widehat{Q}\|_2 = O(1).$$

Proof. To specialize Theorem 3.2 for our implementation, we replace ϵ_1 and ϵ_2 with the doubled precision ϵ_d^2, and ϵ_3 with the working precision ϵ_d. Then, we have

$$\|I - \widehat{Q}^T\widehat{Q}\|_2 \leq O\left(\epsilon_d\kappa(V)\|\widehat{Q}\|_2 + (\epsilon_d\kappa(V))^2\right). \tag{3.1}$$

Clearly, $\|\widehat{Q}\|_2$ is bounded as

$$\|\widehat{Q}\|_2 \leq \|V\|_2\|\widehat{R}^{-1}\|_2 = O(\kappa(V)).$$

This leads to the same bound on the orthogonality error as that of the standard CholQR:

$$\|I - \widehat{Q}^T\widehat{Q}\| \leq O\left(\epsilon_d\kappa(V)^2\right). \tag{3.2}$$

Now, according to [14, 21, Lemma 4.2], we have

$$\begin{aligned} &\text{if } \|I - \widehat{Q}^T\widehat{Q}\|_2 \leq \alpha, \quad &&\text{then } \|\widehat{Q}^T\widehat{Q}\|_2 = \|\widehat{Q}\|_2^2 \leq 1 + \alpha, \\ &&&\text{and } \|\widehat{Q}\|_2 \leq 1 + \sqrt{\alpha}, \text{ where } \alpha \geq 0. \end{aligned} \tag{3.3}$$

Therefore, for the matrix $\widehat{Q}$ satisfying (3.2), we at least have

$$\|\widehat{Q}\|_2 \leq 1 + O\left(\epsilon_d^{\frac{1}{2}}\kappa(V)\right).$$

This bound on $\|\widehat{Q}\|_2$ can be substituted back into our orthogonality error bound (3.1) to obtain

$$\|I - \widehat{Q}^T\widehat{Q}\|_2 \leq O\left(\epsilon_d\kappa(V) + \epsilon_d^{\frac{3}{2}}\kappa(V)^2 + (\epsilon_d\kappa(V))^2\right).$$

By substituting this new error bound into (3.3), we obtain a tighter bound on $\|\widehat{Q}\|_2$,

$$\|\widehat{Q}\|_2 \leq 1 + O\left((\epsilon_d\kappa(V))^{\frac{1}{2}} + \epsilon_d^{\frac{3}{4}}\kappa(V) + \epsilon_d\kappa(V)\right),$$

which can be substituted back into our orthogonality error bound (3.1) to obtain a tighter error bound,

$$\|I - \widehat{Q}^T\widehat{Q}\|_2 \leq O\left(\epsilon_d\kappa(V) + (\epsilon_d\kappa(V))^{\frac{3}{2}} + \epsilon_d^{\frac{7}{4}}\kappa(V)^2 + (\epsilon_d\kappa(V))^2\right).$$

This gives us an even tighter bound on $\|\widehat{Q}\|_2$,

$$\|\widehat{Q}\|_2 \leq 1 + O\left((\epsilon_d\kappa(V))^{\frac{1}{2}} + (\epsilon_d\kappa(V))^{\frac{3}{4}} + \epsilon_d^{\frac{7}{8}}\kappa(V) + \epsilon_d\kappa(V)\right),$$

which, in return, gives us a yet tighter error bound,

$$\|I - \widehat{Q}^T\widehat{Q}\|_2 \leq O\left(\epsilon_d\kappa(V) + (\epsilon_d\kappa(V))^{\frac{3}{2}} + (\epsilon_d\kappa(V))^{\frac{7}{4}} + \epsilon_d^{\frac{15}{8}}\kappa(V)^2 + (\epsilon_d\kappa(V))^2\right).$$

After recursively applying this process ℓ times, we have

$$\|\widehat{Q}\|_2 \leq 1 + O\left(\sum_{k=1}^{\ell}(\epsilon_d\kappa(V))^{1-(1/2)^k} + \left(1 + \epsilon_d^{-(1/2)^{\ell+1}}\right)\epsilon_d\kappa(V)\right)$$

and

$$\|I - \widehat{Q}^T\widehat{Q}\|_2 \leq O\left(\left(1 + \sum_{k=1}^{\ell}(\epsilon_d\kappa(V))^{1-(1/2)^k}\right)\epsilon_d\kappa(V) + \left(1 + \epsilon_d^{-(1/2)^{\ell+1}}\right)(\epsilon_d\kappa(V))^2\right).$$

Now if $\kappa(V) < \epsilon_d^{-1}$ (i.e., $\epsilon_d\kappa(V) < 1$), then for $k \geq 1$, we have $\epsilon_d\kappa(V) < (\epsilon_d\kappa(V))^{(1/2)^k}$, and thus, equivalently, $(\epsilon_d\kappa(V))^{1-(1/2)^k} < 1$. Therefore, we obtain the following upper-bound on the orthogonality error:

$$\|I - \widehat{Q}^T\widehat{Q}\|_2 < O\left((1+\ell)\epsilon_d\kappa(V) + \left(1 + \epsilon_d^{-(1/2)^{\ell+1}}\right)(\epsilon_d\kappa(V))^2\right).$$

Furthermore, for $\epsilon_d = c \cdot 2^{-p}$, when $\ell = \log_2(p)$, we have $\epsilon_d^{-(1/2)^{\ell+1}} = O(1)$ (e.g., for the IEEE standard 64-bit double precision, we have $p = 53$, and $\ell = 5$) and obtain

$$\begin{aligned}\|I - \widehat{Q}^T\widehat{Q}\|_2 &< O\left((1+\log_2(p))\epsilon_d\kappa(V) + \left(1 + \epsilon_d^{-(1/2)^{\log_2(p)+1}}\right)(\epsilon_d\kappa(V))^2\right)\\ &= O\left(\epsilon_d\kappa(V) + (\epsilon_d\kappa(V))^2\right).\end{aligned}$$

Finally, since $(\epsilon_d\kappa(V))^2 < \epsilon_d\kappa(V)$ for $\epsilon_d\kappa(V) < 1$, we arrive at the error bound,

$$\|I - \widehat{Q}^T\widehat{Q}\|_2 \leq O(\epsilon_d\kappa(V)),$$

and the bound

$$\|\widehat{Q}\|_2 \leq 1 + O\left((\epsilon_d\kappa(V))^{1/2}\right) = O(1). \qquad \square$$

MGS [5] is another popular orthogonalization procedure whose normwise orthogonality error is bounded by $O(\epsilon_d\kappa(V))$ (see Figure 3), and it was stable in our previous

numerical studies [25]. However, MGS is based on BLAS-1 and obtains only a fraction of the standard CholQR performance, as shown in Figure 4.

The following theorem provides an upper-bound on the condition number of the matrix $\widehat{Q}$ computed using the mixed-precision CholQR.

THEOREM 3.4. *When $\kappa(V) < \epsilon_d^{-1}$, the condition number of the matrix $\widehat{Q}$ computed using the mixed-precision CholQR is bounded by*

$$\kappa(\widehat{Q}) \leq 1 + O(\epsilon_d \kappa(V)).$$

Proof. According to the analysis in [21, Proposition 4.3], if $\|I - \widehat{Q}^T \widehat{Q}\|_2 \leq \alpha < 1$, then we have

$$\begin{aligned} \kappa(\widehat{Q}) &\leq \sqrt{\frac{1+\alpha}{1-\alpha}} = \sqrt{1 + \frac{2\alpha}{1-\alpha}} \\ &\leq 1 + \frac{\alpha}{1-\alpha}. \end{aligned}$$

We obtain the desired bound by substituting $\alpha = O(\epsilon_d \kappa(V))$ for the mixed-precision CholQR with the assumption that $\epsilon_d \kappa(V)$ is bounded away from one. ∎

Theorems 3.3 and 3.4 together imply that when $\kappa(V) < \epsilon_d^{-1}$, after one reorthogonalization, the orthogonality error $\|I - \widehat{Q}^T \widehat{Q}\|_2$ of the mixed-precision CholQR would be in the order of the working precision ϵ_d.

3.2. Numerical results. In this subsection, we present experimental results to compare the numerical behaviors of the standard and mixed-precision CholQR in the working 64-bit double precision, where we refer to the standard and mixed-precision CholQR as d-CholQR and dd-CholQR, respectively. The test matrices are those from [21] and are listed in Figure 5. In Figures 6 through 13, we show the orthogonality errors and the condition numbers of the matrix $\widehat{Q}$ computed by iteratively applying d-CholQR or dd-CholQR. With the standard d-CholQR, the first and the first two Cholesky factorizations failed orthogonalizing the 20 and 30 Krylov vectors of the two-dimensional (2D) Laplacian, respectively (Figures 6 and 8), while for the Hilbert and synthetic matrices, it failed four times and once, respectively (Figures 10 and 12). On the other hand, with the mixed-precision dd-CholQR, only the first and the first two Cholesky factorizations of the 2D Laplacian and Hilbert matrix failed, respectively. When the Cholesky factorization failed (i.e., encountered a nonpositive pivot), we set the trailing submatrix of $\widehat{R}$ to be identity. Hence, we implicitly blocked the columns of the input matrix V and orthonormalized the columns block by block such that the corresponding leading submatrix of its Gram matrix has a condition number less than the reciprocal of the machine epsilon used at Step 2 of CholQR, and its Cholesky factorization can be computed.

In Figures 6 and 7, the condition number of the input matrix is less than the reciprocal of the machine precision (i.e., $\kappa(V) < \epsilon_d^{-1}$), and the error norms and the condition numbers of $\widehat{Q}$ computed by dd-CholQR converged similarly to those of standard d-MGS, agreeing with our upper-bounds from section 3.1. On the other hand, the condition numbers of the matrices in Figures 8 through 13 are greater than the reciprocal of the machine precision, and our upper-bounds may not hold. However, by orthonormalizing the columns of V block by block, though dd-CholQR converged

Name	m	n	V	$\kappa(V)$
$\mathcal{K}_{20}(A, \mathbf{1})$ of 2D Laplacian A	1089	20	$[\mathbf{1}_m, A\mathbf{1}_m, A^2\mathbf{1}_m, \ldots, A^{20}\mathbf{1}_m]$	8.6×10^{13}
$\mathcal{K}_{30}(A, \mathbf{1})$ of 2D Laplacian A	1089	30	$[\mathbf{1}_m, A\mathbf{1}_m, A^2\mathbf{1}_m, \ldots, A^{30}\mathbf{1}_m]$	3.5×10^{19}
Hilbert matrix	100	100	$v_{i,j} = (i + j - 1)^{-1}$	6.6×10^{19}
Synthetic matrix	101	100	$[\mathbf{1}_n^T; diag(rand(m, 1) * \epsilon_d^3]$	7.8×10^{18}

FIG. 5. *Test matrices used for numerical experiments, where the Laplacian matrix A is of the dimension* 1089*-by-*1089*, $\mathbf{1}_m$ is the m-length vector of all ones, $v_{i,j}$ is the (i, j)th element of V, $rand(m, 1)$ is the m-length vector of random real numbers from a uniform distribution on the open interval $(0, 1)$, and the condition number $\kappa(V)$ is based on the singular values of V computed using MATLAB.*

Iteration	d-CGS	d-MGS	d-CAQR	d-SVQR	d-CholQR	dd-CholQR
1	9×10^{0}	2×10^{-4}	2×10^{-15}	7×10^{0}	6×10^{0} (f)	1×10^{-4}
2	4×10^{0}	1×10^{-15}	1×10^{-15}	9×10^{-7}	2×10^{0}	1×10^{-15}
3	2×10^{-6}			7×10^{-15}	6×10^{-15}	
4	1×10^{-15}					

FIG. 6. *Error norm $\|I - \widehat{Q}^T\widehat{Q}\|_2$ for $\mathcal{K}_{20}(A, \mathbf{1})$ of 2D Laplacian matrix A.*

Iteration	d-CGS	d-MGS	d-CAQR	d-SVQR	d-CholQR	dd-CholQR
1	3×10^{11}	1×10^{0}	1×10^{0}	9×10^{4}	2×10^{10} (f)	1×10^{0}
2	2×10^{6}	1×10^{0}	1×10^{0}	1×10^{0}	2×10^{0}	1×10^{0}
3	1×10^{0}			1×10^{0}	1×10^{0}	
4	1×10^{0}					

FIG. 7. *Condition number $\kappa(\widehat{Q})$ for $\mathcal{K}(A, 20)$ of 2D Laplacian matrix A.*

Iteration	d-CGS	d-MGS	d-CAQR	d-SVQR	d-CholQR	dd-CholQR
1	2×10^{2}	9×10^{-1}	2×10^{-16}	3×10^{0}	2×10^{9} (f)	1×10^{0} (f)
2	1×10^{1}	1×10^{-15}		1×10^{0}	2×10^{2} (f)	9×10^{-12}
3	9×10^{0}			8×10^{-13}	1×10^{0}	1×10^{-15}
4	3×10^{0}			7×10^{-15}	2×10^{-9}	
5	2×10^{-9}				5×10^{-15}	
6	1×10^{-15}					

FIG. 8. *Error norm $\|I - \widehat{Q}^T\widehat{Q}\|_2$ for $\mathcal{K}_{30}(A, \mathbf{1})$ of 2D Laplacian A.*

Iteration	d-CGS	d-MGS	d-CAQR	d-SVQR	d-CholQR	dd-CholQR
1	1×10^{16}	5×10^{0}	1×10^{0}	3×10^{9}	1×10^{18} (f)	5×10^{10} (f)
2	7×10^{13}	1×10^{0}		4×10^{1}	2×10^{13} (f)	1×10^{0}
3	2×10^{10}			1×10^{0}	1×10^{9}	1×10^{0}
4	7×10^{4}			1×10^{0}	1×10^{0}	
5	1×10^{0}				1×10^{0}	
6	1×10^{0}					

FIG. 9. *Condition number $\kappa(\widehat{Q})$ for $\mathcal{K}_{30}(A, \mathbf{1})$ of 2D Laplacian A.*

Iteration	d-CGS	d-MGS	d-CAQR	d-SVQR	d-CholQR	dd-CholQR
1	9×10^{1}	1×10^{-1}	2×10^{-15}	1×10^{0}	1×10^{0} (f)	1×10^{0} (f)
2	9×10^{1}	3×10^{-14}		4×10^{0}	1×10^{0} (f)	1×10^{0} (f)
3	8×10^{1}	9×10^{-16}		2×10^{-9}	1×10^{0} (f)	2×10^{-10}
4	8×10^{1}			1×10^{-14}	1×10^{0} (f)	1×10^{-15}
5	6×10^{1}			9×10^{-15}	5×10^{-4}	
6	4×10^{1}				1×10^{-15}	
7	8×10^{0}					
8	4×10^{-14}					
9	1×10^{-15}					

FIG. 10. *Error norm $\|I - \widehat{Q}^T\widehat{Q}\|_2$ for Hilbert matrix.*

Iteration	d-CGS	d-MGS	d-CAQR	d-SVQR	d-CholQR	dd-CholQR
1	2×10^{18}	2×10^{2}	1×10^{0}	2×10^{12}	4×10^{18} (f)	4×10^{18} (f)
2	4×10^{16}	1×10^{0}		6×10^{4}	1×10^{17} (f)	2×10^{0} (f)
3	3×10^{14}	1×10^{0}		1×10^{0}	1×10^{15} (f)	1×10^{0}
4	7×10^{12}			1×10^{0}	1×10^{0} (f)	1×10^{0}
5	8×10^{10}			1×10^{0}	1×10^{0}	
6	1×10^{9}				1×10^{0}	
7	2×10^{2}					
8	1×10^{0}					
9	1×10^{0}					

FIG. 11. *Condition number* $\kappa(\widehat{Q})$ *for Hilbert matrix.*

Iteration	d-CGS	d-MGS	d-CAQR	d-SVQR	d-CholQR	dd-CholQR
1	2×10^{1}	7×10^{-16}	4×10^{-15}	1×10^{0}	1×10^{0} (f)	3×10^{-15}
2	3×10^{-14}	5×10^{-16}		2×10^{-8}	7×10^{-15}	3×10^{-16}
3	6×10^{-16}			3×10^{-14}	5×10^{-16}	3×10^{-16}
4				9×10^{-15}	4×10^{-16}	

FIG. 12. *Error norm* $\|I - \widehat{Q}^T\widehat{Q}\|_2$ *for synthetic matrix.*

more slowly than d-MGS, it converged faster than d-CGS, d-CholQR, or d-SVQR.[3] This may be explained by the term $O((\epsilon_d \kappa(V))^2)$ in the upper-bound on the error norm $\|I - \widehat{Q}^T\widehat{Q}\|_2$ in Theorem 3.2. Because of this term, the error norm of dd-CholQR may be greater than that of d-MGS. However, it is smaller than those of the other standard schemes d-CGS, or d-CholQR and d-SVQR, due to the term $O(\epsilon_d \kappa(V)^{n-1})$ or $O(\epsilon_d \kappa(V)^2)$, respectively, in their error norm upper-bounds. Finally, for all the test matrices, d-CAQR converged in one iteration. This is because, for the results in this subsection, we conducted all the experiments with one GPU, and d-CAQR is equivalent to the Householder QR factorization.

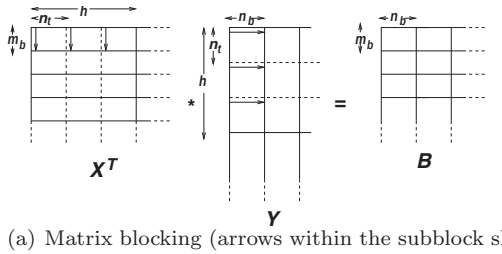
3.3. GPU implementation. Since the input matrix V is tall-skinny (i.e., $n \ll m$), CholQR spends only a small portion of its orthogonalization time computing the Cholesky factorization of the Gram matrix B at Step 2. In addition, solving the triangular system with many right-hand sides at Step 3 exhibits a high parallelism and can be implemented efficiently on a GPU. On the other hand, at Step 1, computing each element of the Gram matrix requires a reduction operation on two m-length vectors (i.e., the (i,j)th element of B is given by $b_{i,j} := \mathbf{v}_i^T \mathbf{v}_j$). These inner-products (*InnerProds*) exhibit only limited parallelism and are memory-bandwidth limited since they perform only at most n flops for each floating-point number read (e.g., $n = O(10)$). Hence, Step 1 often becomes the bottleneck, where standard implementations fail to obtain high-performance on the GPU.

In our GPU implementation of a matrix-matrix (GEMM) multiply, $B := X^T Y$, to compute *InnerProds*, the matrices X and Y are divided into h-by-m_b and h-by-n_b submatrices, where the (k,i)th block of X and the (k,j)th block of Y are denoted by $X(k,i)$ and $Y(k,j)$, respectively. Then, the (i,j,k)th thread block computes a partial result, $B(i,j)^{(k)} := X(k,i)^T Y(k,j)$, where each of the n_t threads in the thread block first independently computes a partial result in its local registers, as illustrated in

[3] Our implementation of SVQR implicitly works with the normalized input matrix V by symmetrically scaling the Gram matrix B such that $B := D^{-1/2} B D^{-1/2}$, where the diagonal entries of the diagonal matrix D are those of B. In addition, we set the singular values that are in the same order as or less than $O(\epsilon_d \sigma_n)$ to be $\epsilon_d \sigma_n$, where ϵ_d is the machine epsilon in the working precision and σ_n is the largest singular value of the Gram matrix. These two techniques from [21] significantly improved the stability of SVQR.

Iteration	d-CGS	d-MGS	d-CAQR	d-SVQR	d-CholQR	dd-CholQR
1	5×10^1	1×10^0	1×10^0	4×10^{10}	3×10^{17} (f)	1×10^0
2	1×10^0	1×10^0		1×10^0	1×10^0	1×10^0
3	1×10^0			1×10^0	1×10^0	1×10^0
4				1×10^0	1×10^0	

FIG. 13. *Condition number $\kappa(\widehat{Q})$ for synthetic matrix.*



(a) Matrix blocking (arrows within the subblock shows data access by a GPU thread).

```
double regB[m_b][n_b], regX[m_b], regY
for ℓ = 1, 2, ..., h/n_t do
    for j = 1, 2, ..., n_b do
      regX[i] = x_{ℓ·n_t, j}
    end for
    for j = 1, 2, ..., n_b do
        regY = y_{ℓ·n_t, j}
        for i = 1, 2, ..., m_b do
          regB[i][j] += regX[i] · regY
        end for
    end for
end for
```

(b) Pseudocode executed by (i, j, k)th thread to compute $X(k, i)^T Y(k, j)$; $\underline{x}_{\ell,j}$ is the (ℓ, j)th element of $X(k, i)$.

FIG. 14. *Implementation of InnerProds on a GPU.*

Figures 14(a) and 14(b). Then, the final partial result $B(i, j)^{(k)}$ is computed through the binary reduction of the partial results among its threads, summing n_r columns at a time using the shared memory to store $n_t \times (m_b \times n_r)$ numerical values.[4] The final result $B(i, j)$ is computed by launching another CUDA kernel to perform another binary reduction among the thread blocks. Our implementation is designed to reduce the number of synchronizations among the threads while relying on the CUDA runtime and the parameter tuning to exploit the data locality. For the symmetric (SYRK) multiply, $B := V^T V$, the thread blocks compute only a triangular part of B and read $V(k, j)$ once for computing a diagonal block (i.e., $k = j$).

When the target hardware does not support the desired higher precision, software emulation is needed. For instance, double-double (dd) precision emulates the quadruple precision by representing each numerical value by an unevaluated sum of two double precision numbers and is capable of representing the 106-bit precision, while a standard 64-bit double precision number is of 53-bit precision. There are two standard implementations [12] of adding two numerical values in double-double precision, $a + b = \widehat{c} + e$, where e is the round-off error; one satisfies the IEEE-style error bound ($e = \delta(a + b)$ with $|\delta| \leq 2\epsilon_{dd}$ and $\epsilon_{dd} = 2^{-105}$), and the other satisfies the weaker Cray-style error bound ($e = \delta_1 a + \delta_2 b$ with $|\delta_1|, |\delta_2| \leq \epsilon_{dd}$). Figure 15 lists the computational costs of the double-double arithmetic required by our mixed-precision *InnerProds* that reads the input matrices in the working double precision but accumulates the intermediate results and writes the output matrix in the double-double precision. By taking advantage of the working precision inputs, we not only reduce the amount of the intra-GPU communication (i.e., the input matrices are read into registers in double precision) but also reduce the number of floating-point instructions required for the double-double multiplication.

[4] In the current implementation, the numbers of rows and columns in X and Y are a multiple of h, and multiples of m_b and n_b, respectively, where n_b is a multiple of n_r.

Double-double operation	# of double precision instructions			
	Add/Substitution	Multiply	FMA	Total
Multiply (double-double input)	5	3	1	9
Multiply (double input)	3	1	1	5
Addition (IEEE-style)	20	0	0	20
Addition (Cray-style)	11	0	0	11

FIG. 15. *Number of required double precision instructions to perform double-double operations.*

CPU	NVIDIA GPUs	gcc	MKL	CUDA
2×8 Intel Xeon E5-2670 (2.60GHz)	$3 \times$ M2090	4.4.6	2011_sp1	5.0.35
2×8 Intel Genuine (2.60GHz)	$1 \times$ K20c	4.4.7	2013_sp1	5.5.22
2×8 Intel Genuine (2.60GHz)	$1 \times$ K40	4.4.7	2013_sp1	5.5.22

FIG. 16. *Experiment testbeds.*

The standard CholQR (d-CholQR in double precision) performs about half of its total flops at Step 1 and the other half at Step 3. Hence, using double-double precision with Cray-style error bound for Steps 1 and 2, our dd-CholQR performs about 8.5× more instructions than d-CholQR. On the other hand, the communication cost increases less significantly. Each thread moves about the same amount of data on each GPU, writing only the n-by-n output matrix in the double-double precision while reading the local part of the m-by-n input matrix in double precision (i.e., $n \ll m$). More specifically, our implementation moves $O(mn \cdot (n/\min(m_b, n_b)))$ data in double precision and $O(n^2 \cdot (m/h))$ data in double-double precision through the local memory hierarchy, where both $n/\min(m_b, n_b)$ and m/h are small scalars (e.g., $n/\min(m_b, n_b) \approx 5$, $m/h \approx 100$, and $m/n = O(10^5)$ in our experiments). The amount of data moved between the thread blocks is doubled, but with the same communication latency. In addition, the amount of the data reduction between the GPUs is much less than the total amount of the intra-GPU communication (i.e., $16 \cdot (n^2 n_g)$ bytes for inter-GPU reduction in comparison to $8(mn/n_g)$ bytes for intra-GPU communication, where $n \approx 10$ and $m = O(10^6)$ in our experiments). As a result, in comparison to the computational overheads of using the double-double arithmetic at the first two steps of CholQR, the increase in the communication is typically less significant. The performance results in the next section verify this.

3.4. Performance. We now study the performance of our mixed-precision computational kernels on the testbeds shown in Figure 16. First, Figure 17 compares the performance of our standard double-precision matrix-matrix multiply d-GEMM and mixed-precision dd-GEMM on different NVIDIA GPUs.[5] Each GPU has a different relative cost of communication to computation, and on top of each plot in parentheses, we show the ratio of the double-precision peak performance (Gflop/s) over the shared memory bandwidth (GB/s) (i.e., flop/B to obtain the peak). This ratio tends to increase on a newer architecture, indicating a greater relative cost of communication. We tuned our GPU kernels for each matrix dimension in each precision on each GPU (see the five tunable parameters h, m_b, n_b, n_r, and n_t in section 3.3), and the figure shows the optimal performance. Based on the fixed number of columns and the shared memory bandwidth in the figure, the respective peak performances of d-GEMM are

[5] Our standard implementation does everything in the working precision. On the other hand, our mixed-precision implementation reads the input matrices in the working precision, but it accumulates the intermediate results and writes the output matrix in the doubled precision.

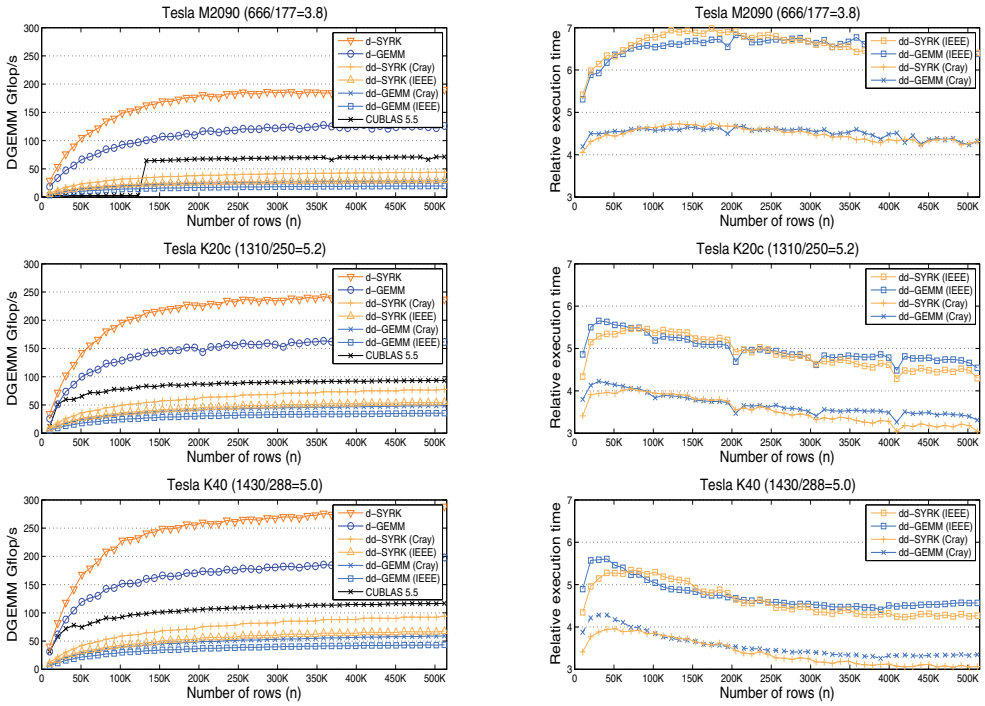


FIG. 17. *Performance of standard and mixed-precision InnerProds in double precision,* $n = 20$. *Gflop/s is computed as the ratio of the number of flops required by the standard algorithm over time in seconds.*

442, 625, and 720Gflop/s on M2090, K20c, and K40 GPUs. Our d-GEMM obtained 29, 26, and 28% of these peak performances and speedups of about 1.8, 1.7, and 1.7 over CUBLAS on M2090, K20c, and K40 GPUs, respectively.[6] In addition, though it performs 16× more instructions, the gap between dd-GEMM and d-GEMM tends to decrease on a newer architecture with a lower computational cost, and dd-GEMM is only about 3× slower on K20c. The figure also shows that by taking advantage of the symmetry, both d-SYRK and dd-SYRK obtain significant speedups over d-GEMM and dd-GEMM, respectively.

Figure 18 shows the performance of the standard matrix-matrix multiply s-GEMM and mixed-precision ss-GEMM for the working 32-bit single precision. For ss-GEMM, the input matrix is in single precision, but the intermediate results are computed in double precision. Hence, the higher-precision is now supported by the hardware. As a result, even on M2090, the performance of the mixed-precision ss-GEMM is much closer to that of standard s-GEMM, compared to the performance of dd-GEMM to that of d-GEMM in double precision, where the software emulation is needed. In

[6]Compared to a previous version, the performance of standard xGEMM has been significantly improved for these tall-skinny matrices in CUBLAS 5.5 (see the performance of CUBLAS 4.2 in our previous paper [25]). In comparison, the CUBLAS xSYRK still obtains significantly lower performance for our purpose (the latest version of CUBLAS includes an implementation of "batched" xGEMM but not of "batched" xSYRK).

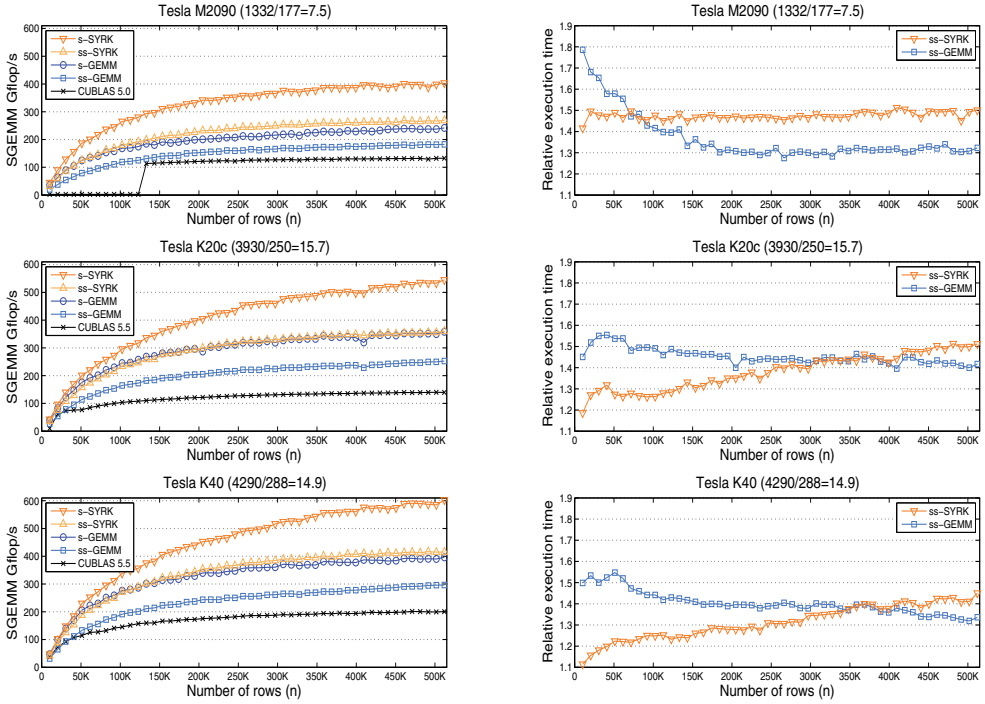


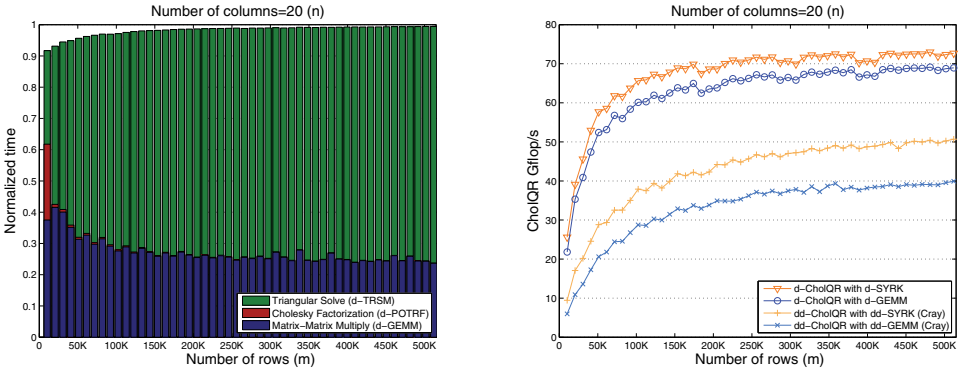
FIG. 18. *Performance of standard and mixed-precision InnerProds in single precision,* $n = 20$.

addition, by taking advantage of the single-precision input, ss-GEMM is significantly more efficient, obtaining over 300Gflop/s, in comparison to d-GEMM which obtains just over 150Gflop/s.

Finally, we compare the performance of the standard and mixed-precision CholQR in the working 64-bit double precision. For mixed-precision dd-CholQR, the Cholesky factorization in the double-double precision is computed using MPACK[7] on the CPU, while for standard d-CholQR, the factorization is computed using threaded MKL. Figure 19(a) shows the breakdown of d-CholQR orthogonalization time. Because of our efficient implementation of *InnerProds*, only about 30% of the orthogonalization time is now spent in d-GEMM. As a result, with the Cray-style error bound, while *dd-InnerProds* was about 3× slower than *d-InnerProds*, Figure 19(b) shows that dd-CholQR is only about 1.7× or 1.4× slower than d-CholQR when GEMM or SYRK is used for *InnerProds*, respectively.

4. Case studies with CA-GMRES and CA-Lanczos. We now study the effects of the mixed-precision dd-CholQR on the performance of CA-GMRES and CA-Lanczos. We first give the brief motivation and description of the solvers in section 4.1 and then present the experimental results on a multicore CPU with multiple GPUs in section 4.2.

[7] http://mplapack.sourceforge.net



(a) d-CholQR time breakdown. (b) d/dd-CholQR performance.

FIG. 19. *Performance comparison with K20c. Gflop/s is based on the flop count of d-CholQR.*

$\widehat{\mathbf{x}} := \mathbf{0}$ and $\mathbf{q}_1 := \mathbf{b}/\|\mathbf{b}\|_2$.
repeat `(restart-loop)`
  `1.Generate Projection Subspace` $O(n \cdot nnz(|A|) + n^2 m)$ flops on GPUs:
  for $j = 1, 2, \ldots, n$ **do**
    SpMV: Generate a new vector $\mathbf{v}_{j+1} := A\mathbf{v}_j$.
    Orth: Orthonormalize $\mathbf{v}_{j+1}$ against $\mathbf{q}_1, \mathbf{q}_2, \ldots, \mathbf{q}_j$
      to generate the next basis vector $\mathbf{q}_{j+1}$.
  end for
  `2.Solve Projected Subsystem` $O(n^2)$ flops on CPU, and
    `Restart Iteration` $O(nm)$ flops on GPUs:

`GMRES:`	`Lanczos:`
Compute the solution $\widehat{\mathbf{x}}$, in generated subspace, which minimizes its residual norm.	Compute eigenpairs $(\widehat{\lambda}_i, \widehat{\mathbf{x}}_i)$ in generated subspace, which minimizes their residual norms.
Set $\mathbf{q}_1 := \mathbf{r}/\|\mathbf{r}\|_2$, where $\mathbf{r} := \mathbf{b} - A\widehat{\mathbf{x}}$.	**if** necessary **then** prepare to thick-restart.

until solution convergence.

FIG. 20. *Pseudocode of GMRES(n) and Lanczos(n), where A is the m-by-m coefficient matrix of the linear system or the eigenvalue problem.*

4.1. Algorithms. When solving a large-scale linear system of equations or eigenvalue problem, an iterative method is often preferred over a direct method. This is because though a direct method computes the solution with a fixed number of flops in a numerically stable manner, its memory and/or computational costs of direct factorization may be unfeasibly expensive. A parallel computer with a large aggregated memory and a high computing capacity may provide a remedy to this large cost of factorization, but the per-core memory requirement or the total solution time of a parallel direct solver may not scale due to the extensive amount of communication or the associated memory overhead for the message buffers. As a result, an iterative method with a lower memory and/or computational costs may become more attractive or could be the only feasible alternative. *Krylov subspace projection methods* [17, 23] are a popular class of iterative methods for solving such large-scale problems. For our case studies in this paper, we used two well-known Krylov methods: the generalized minimum residual (GMRES) method [19] for solving nonsymmetric linear systems of

$\widehat{\mathbf{x}} := \mathbf{0}$ and $\mathbf{v}_1 := \mathbf{b}/\|\mathbf{b}\|_2$.
repeat (restart-loop)
  1.Generate Projection Subspace on GPUs:
  for $j = 1, s+1, 2s+1, \ldots, n$ **do**
    MPK: Generate new vectors $\mathbf{v}_{k+1} := A\mathbf{v}_k$
      for $k = j, j+1, \ldots, \min(j+s, n)$.
    BOrth: Orthogonalize $V_{j+1:j+s+1}$ against $V_{1:j}$.
    TSQR: Generate $Q_{j+1:j+s+1}$
      by orthonormalizing $V_{j+1:j+s+1}$ within.
  end for

  2.Solve Projected Subsystem on CPU, and
    Restart Iteration on GPUs:
  Compute the solution $\widehat{\mathbf{x}}$ in the generated subspace,
    which minimizes its residual norm.
  Set $\mathbf{v}_1 := \mathbf{r}/\|\mathbf{r}\|_2$, where $\mathbf{r} := \mathbf{b} - A\widehat{\mathbf{x}}$.
until solution convergence **do**

FIG. 21. *Pseudocode of CA-GMRES(s,n).*

equations and the Lanczos method [16] for solving symmetric eigenvalue problems.

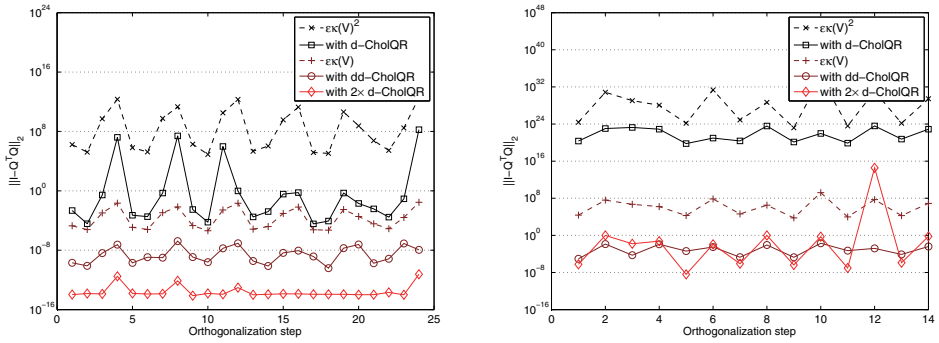
Starting with an initial input vector $\mathbf{q}_1$, the jth iteration of GMRES or Lanczos generates the $(j+1)$th Krylov basis vector $\mathbf{q}_{j+1}$ by multiplying the previously generated vector $\mathbf{q}_j$ with the sparse coefficient matrix A (*SpMV*), followed by its orthonormalization (*Orth*) against all the previously orthonormalized basis vectors $\mathbf{q}_1, \mathbf{q}_2, \ldots, \mathbf{q}_j$. As the iteration continues, the computational and memory requirements of *Orth* becomes increasingly expensive. To limit the cost of generating the basis vectors, the iteration is restarted after computing $n+1$ basis vectors, using the best approximate solution in the generated Krylov subspace as a new initial vector. Typically, the computational cost of this restarted Krylov method is dominated by the combined cost of *SpMV* and *Orth*. Hence, to accelerate the solution process, we generate these basis vectors on the GPU, while solving the projected subsystem on the CPU.[8] To utilize multiple GPUs, we use a matrix reordering or graph partitioning algorithm to distribute both the coefficient matrix A and the basis vectors $\mathbf{q}_1, \mathbf{q}_2, \ldots, \mathbf{q}_{n+1}$ among the GPUs in a 1D block row format. Figure 20 shows the pseudocodes of GMRES(n) and Lanczos(n), where $Q_{j:k}$ is the matrix consisting of the jth through the kth column vectors $\mathbf{q}_j, \mathbf{q}_{j+1}, \ldots, \mathbf{q}_k$. Our Lanczos implementation uses the three-term recurrence followed by the full reorthogonalization to maintain the orthogonality among the basis vectors, and *thick restart* [24] to improve its convergence rate. A more detailed description of our implementation can be found in [25].

Both *SpMV* and *Orth* require communication. This includes the point-to-point messages or the neighborhood collectives between the GPUs for *SpMV*, the global all-reduces for *Orth*, and the data movements through the local memory hierarchy of the GPU (for reading the sparse matrix and for reading and writing vectors). The communication-avoiding variants of GMRES and Lanczos [13] (called CA-GMRES and CA-Lanczos, respectively) aim to reduce this communication by redesigning the

[8] After solving the projected subsystem, the approximate solution is projected back to the projection subspace on the GPUs. This requires the dense matrix-vector or matrix-matrix multiply between the generated subspace and the solution of the projected subsystem to update the computed solution for GMRES or to compute the Ritz vectors for Lanczos, respectively.

Name	Source	$m/1000$	nnz/m
cant	FEM cantilever	62.4	64.2
shipsec1	FEM ship section	140.8	87.3
G3_circuit	Circuit simulation	1585.4	4.8

FIG. 22. *Test Matrices used for test cases with CA-GMRES.*



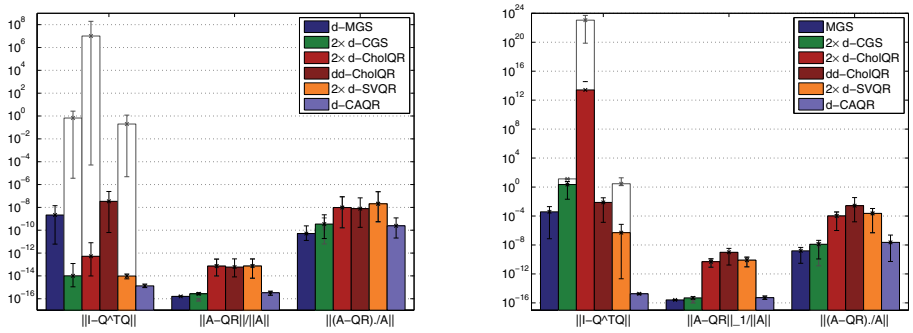
(a) $(s, n) = (15, 60)$. (b) $(s, n) = (30, 60)$.

FIG. 23. *Orthogonality error norm and bound at each step of CA-GMRES with cant matrix.*

algorithms and replacing *SpMV* and *Orth* with three new kernels—matrix-powers kernel (*MPK*), block orthogonalization (*BOrth*), and tall-skinny QR (*TSQR*)—that generate and orthogonalize a set of s basis vectors at once. Figure 21 shows the pseudocode of CA-GMRES(s, n). In theory, CA-GMRES and CA-Lanczos generate the s basis vectors with the communication cost of a single standard Krylov iteration (plus a lower-order term). Our previous performance studies of CA-GMRES on a multicore CPU with multiple GPUs [25] demonstrated that by avoiding the communication, CA-GMRES could obtain a speedup of up to two. In that previous study, we also found that an efficient and numerically stable orthogonalization scheme is crucial to achieve the high performance of CA-GMRES, and CholQR obtained the superior performance based on the optimized BLAS-3 GPU kernels. Unfortunately, CholQR may cause numerical instability. This is because even though using the Newton basis improves the numerical stability [3], the vector $\mathbf{v}_j$ may still converge to the principal eigenvector of A, and $V_{1:s+1}$ can be ill-conditioned. As a result, in some cases, CA-GMRES did not converge even with reorthogonalization.

In the next section, we study the effects of using the mixed-precision CholQR to orthogonalize the $s + 1$ Krylov vectors at a time to generate the total of $n + 1$ orthonormal basis vectors over each restart cycle.

4.2. Experimental results. We now study the effects of the mixed-precision dd-CholQR on the performance of CA-GMRES and CA-Lanczos. Figure 22 shows the properties of our test matrices from the University of Florida Sparse Matrix Col-



(a) $(s, n) = (15, 60)$. (b) $(s, n) = (30, 60)$.

FIG. 24. *Average TSQR error* 1*-norms in CA-GMRES with cant matrix.*

lection.[9] First, Figure 23 shows the orthogonality error norm and its upper-bound after each orthogonalization step of CA-GMRES$(15, 60)$ for the `cant` matrix. The figure shows that the orthogonality errors of dd-CholQR and d-CholQR depend linearly and quadratically on $\kappa(V)$, respectively, agreeing with their upper-bounds in section 3.1. Next, for the same matrix, Figure 24 shows the average error norms over the CA-GMRES iterations using different orthogonalization schemes (see Figure 3 for the description of the orthogonalization schemes). For this particular matrix, CGS, CholQR, and SVQR require reorthogonalization for CA-GMRES to converge, and the white bars show the error norms after the first orthogonalization. Though the orthogonality error of dd-CholQR is slightly greater than that of MGS, CA-GMRES converges with the same number of iterations without the reorthogonalization.

Finally, Figure 25 shows the normalized solution time of CA-GMRES on a 16-core Intel CPU with one NVIDIA K20c GPU. The computed solution is considered to have converged when the ℓ_2-norm of the initial residual is reduced by at least six orders of magnitude for the double precision and five orders of magnitude for the single precision. Using dd-CholQR, in these particular cases, the solution time was reduced not only because the reorthogonalization was avoided but also because CA-GMRES converged in fewer iterations. In addition, using ss-CholQR, which does not require the software emulation, the solution time was reduced even with the same number of iterations. Finally, Figure 26 shows the performance of CA-GMRES using up to three M2090 GPUs for the `G3_Circuit` matrix. The matrix is distributed among the GPUs such that each GPU has a similar number of rows after the reverse Cuthill–McKee (RCM) matrix reordering [6] from HSL[10] is applied. We see similar performance improvements using the mixed precision on the multiple GPUs.

Figure 27 shows the performance of CA-Lanczos using d-CholQR and dd-CholQR to compute the smallest 100 eigenvalues and the corresponding eigenvectors of a diagonal matrix $A_1(m) = \mathrm{diag}(1, 2, \ldots, m)$. This test matrix was previously used in [20, 26].

[9] http://www.cise.ufl.edu/research/sparse/matrices/. Clearly, the effects of dd-CholQR depend on the input matrix. Here, to demonstrate the potential performance benefit, we show only the test cases where dd-CholQR improved the performance of the solvers. However, for other test cases (e.g., well-conditioned matrices), dd-CholQR may degrade the performance. We are investigating an adaptive scheme to decide when to use the mixed-precision or when to perform reorthogonalization at run time based on the result of the Cholesky factorization.

[10] http://www.hsl.rl.ac.uk/catalogue/mc60.xml

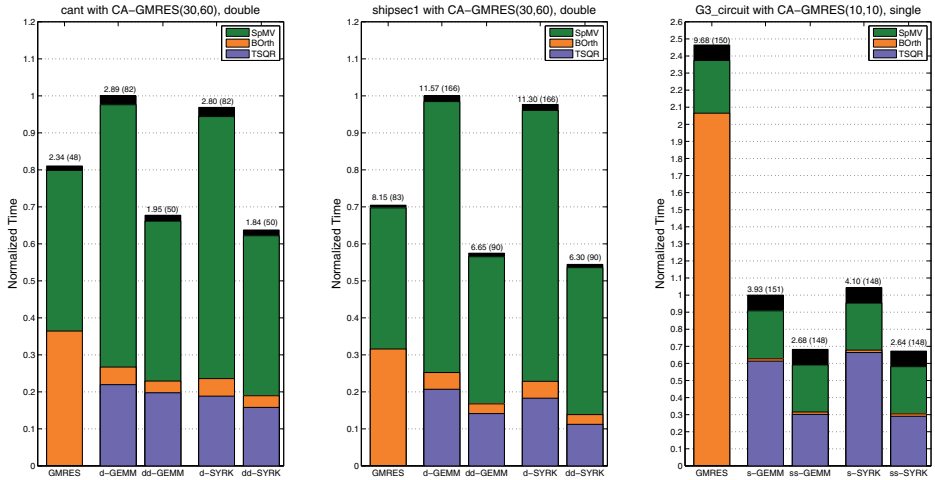


FIG. 25. *CA-GMRES performance with K*20*c: The total time in seconds and the restart count are shown at the top of each bar. CA-GMRES with nonoptimal s got speedups over GMRES using CGS.*

n_g	Type	Restart	Time (s) *TSQR*	*Orth*	*SpMV*	Total	Speedup
1	d	159	3.58	4.31	13.06	17.58	
1	dd	85	2.85	3.26	6.98	10.38	1.69
2	d	194	2.63	3.10	10.50	13.91	
2	dd	74	1.60	1.80	4.01	5.94	2.34
3	d	185	1.93	2.27	8.63	11.28	
3	dd	81	1.42	1.58	3.75	5.50	2.05

FIG. 26. *Parallel performance of CA-GMRES(*30, 60*) using d-CholQR and dd-CholQR for* `G3_Circuit` *matrix on up to three Tesla M*2090 *GPUs. In the table, "restart" is the number of restarts, "TSQR," "Orth," and "SpMV" are the times spent for TSQR, Orth, and SpMV, respectively, and "Total" is the total solution time in seconds.*

Our implementation of the Lanczos method is based on the thick restart Lanczos [26], and its correctness has been verified using the test matrices in [29]. In these tests, we found that dd-CholQR enables CA-Lanczos to converge using a larger step size than was possible using d-CholQR. The GPU kernels obtained higher performance using the larger step size, and the solution time of CA-Lanczos was reduced using dd-CholQR.

5. Conclusion. We analyzed the numerical properties of a recently proposed variant of the CholQR orthogonalization scheme, which uses mixed-precision arithmetic to improve the numerical stability. Our analysis showed that the orthogonality error of the mixed-precision variant depends linearly on the condition number of the input matrix, while the original scheme depends quadratically. Though the higher precision may require software emulation and increase the computational cost, the increase in the communication cost is less significant. As a result, our performance results showed that the overhead of using the mixed-precision arithmetic is decreasing on a newer architecture where the relative cost of the computation is decreasing, com-

Type	s	Restart	*SpMV*	Time (s) init	Restart	*TSQR*	*Orth*	*SpMV*	Total
				$-A_1(600K)-$					
d	20	94	5,985	0.1	0.9	1.2	4.4	0.2	6.1
dd	20	92	5,924	0.1	0.9	0.7	3.8	0.2	5.6
dd	30	106	6,794	0.1	1.0	1.0	3.7	0.2	5.6
				$-A_1(800K)-$					
d	20	108	6,912	0.1	1.1	1.6	6.4	0.2	8.7
dd	20	111	7,095	0.1	1.1	1.0	5.9	0.2	8.3
dd	30	108	6,905	0.1	1.1	1.2	4.7	0.2	7.0
				$-A_1(1,00K)-$					
d	20	122	7,698	0.2	1.3	2.0	8.9	0.3	10.3
dd	20	122	7,759	0.2	1.2	1.2	7.2	0.3	10.9
dd	30	121	7,698	0.2	1.3	1.4	5.8	0.3	8.6

FIG. 27. *Performance of CA-Lanczos($n = 200$) computing the smallest eigenvalues of $A_1 = diag(1, 2, \ldots, m)$ on Tesla K20c. In this table, "SpMV" is the number of SpMV operations, and "init" and "restart" are the times needed for the first iteration and restart, respectively. For the description of the other measurements, see Figure* 26.

pared to the communication. Our case studies of using double-double arithmetic for the working 64-bit double precision with CA-GMRES on a multicore CPU with multiple GPUs demonstrated that, though the mixed-precision CholQR requires about 8.5× more computation, the use of higher precision for this small but critical segment of CA-GMRES can improve not only its overall stability but also, in some cases, its performance by avoiding the reorthogonalization, improving the convergence rate, and/or allowing a larger step size. This is especially the case when CA-GMRES suffers from numerical instability using the standard orthogonalization scheme, even with a small step size and with reorthogonalization (more so if the higher precision is supported by the target hardware). We observed similar benefits for CA-Lanczos.

In this paper, we studied the performance of CA-GMRES on multiple GPUs of a single compute node, where the performance of CA-GMRES depends more on the performance of the GPU kernels (i.e., the intra-GPU communication) than on the inter-GPU communication [25]. We plan to study the performance of the mixed-precision orthogonalization scheme on systems where the communication becomes more expensive (e.g., distributed GPUs, or CPUs), and where the scheme, therefore, may lead to a greater performance improvement. Another great interest of ours is the performance comparison of the mixed-precision CholQR against the communication-avoiding QR (CAQR) [7], and extending SVQR to use mixed precision, which may remove our assumption on the condition number to ensure the successful completion of the Cholesky factorization. In addition, instead of using a higher precision to improve the stability, we are also studying a mixed-precision scheme that selectively uses a lower precision to improve the performance [28]. Finally, to improve the numerical stability of the Krylov methods, we are studying adaptive schemes to adjust the block size and/or to select the orthogonalization procedure at runtime. The GPU kernels developed for this study will be released through the MAGMA library.[11]

[11] http://icl.utk.edu/magma/

Acknowledgments. We would like to thank the editor and anonymous reviewers for their great suggestions and comments.

REFERENCES

[1] M. Anderson, G. Ballard, J. Demmel, and K. Keutzer, *Communication-avoiding QR decomposition for GPUs*, in Proceedings of the IEEE International Parallel and Distributed Processing Symposium (IPDPS), IEEE, Washington, DC, 2011, pp. 48–58.
[2] T. Auckenthaler, T. Huckle, and R. Wittmann, *A blocked QR-decomposition for the parallel symmetric eigenvalue problem*, Parallel Comput., 40 (2014), pp. 186–194.
[3] Z. Bai, D. Hu, and L. Reichel, *A Newton basis GMRES implementation*, IMA J. Numer. Anal., 14 (1994), pp. 563–581.
[4] J. Barlow and A. Smoktunowicz, *Reorthogonalized block classical Gram-Schmidt*, Numer. Math., 123 (2013), pp. 395–423.
[5] A. Björck, *Solving linear least squares problems by Gram-Schmidt orthogonalization*, BIT, 7 (1967), pp. 1–21.
[6] E. Cuthill and J. McKee, *Reducing the bandwidth of sparse symmetric matrices*, in Proceedings of the 24th National ACM Conference (ACM '69), ACM, New York, 1969, pp. 157–172.
[7] J. Demmel, L. Grigori, M. Hoemmen, and J. Langou, *Communication-optimal parallel and sequential QR and LU factorizations*, SIAM J. Sci. Comput., 34 (2012), pp. A206–A239.
[8] S. Fuller and L. Millett, *Future of Computing Performance: Game Over or Next Level?*, The National Academies Press, Washington, DC, 2011.
[9] G. Golub and C. van Loan, *Matrix Computations*, 3rd ed., The Johns Hopkins University Press, Baltimore, MD, 1996.
[10] S. Graham, M. Snir, and C. Patterson, *Getting Up to Speed: The Future of Supercomputing*, The National Academies Press, Washington, DC, 2004.
[11] N. Halko, P. G. Martinsson, and J. A. Tropp, *Finding structure with randomness: Probabilistic algorithms for constructing approximate matrix decompositions*, SIAM Rev., 53 (2011), pp. 217–288.
[12] Y. Hida, X. Li, and D. Bailey, *Quad-double arithmetic: Algorithms, implementation, and application*, Tech. report LBNL-46996, Lawrence Berkeley National Laboratory, Berkeley, CA, 2000.
[13] M. Hoemmen, *Communication-Avoiding Krylov Subspace Methods*, Ph.D. thesis, University of California-Berkeley, Berkeley, CA, 2010.
[14] W. Hoffman, *Iterative algorithms for Gram-Schmidt orthogonalization*, Computing, 41 (1989), pp. 335–348.
[15] A. Kielbasiński, *Analiza numeryczna algorytmu ortogonlizacji Grama-Schmidta*, Seria III: Matematyka Stosowana II, 1974 (1974), pp. 15–35.
[16] C. Lanczos, *An iteration method for the solution of the eigenvalue problem of linear differential and integral operators*, J. Res. Natl. Bur. Standards, 45 (1950), pp. 255–281.
[17] Y. Saad, *Iterative Methods for Sparse Linear Systems*, 2nd ed., SIAM, Philadelphia, 2003.
[18] Y. Saad, *Numerical Methods for Large Eigenvalue Problems*, revised ed., SIAM, Philadelphia, 2011.
[19] Y. Saad and M. Schultz, *GMRES: A generalized minimal residual algorithm for solving nonsymmetric linear systems*, SIAM J. Sci. Statist. Comput., 7 (1986), pp. 856–869.
[20] A. Stathopoulos and K. Orginos, *Computing and deflating eigenvalues while solving multiple right-hand side linear systems with an application to quantum chromodynamics*, SIAM J. Sci. Comput., 32 (2010), pp. 439–462.
[21] A. Stathopoulos and K. Wu, *A block orthogonalization procedure with constant synchronization requirements*, SIAM J. Sci. Comput., 23 (2002), pp. 2165–2182.
[22] L. N. Trefethen and D. Bau, III, *Numerical Linear Algebra*, SIAM, Philadelphia, 1997.
[23] H. van der Vorst, *Iterative Krylov Methods for Large Linear Systems*, Cambridge University Press, Cambridge, UK, 2003.
[24] K. Wu and H. Simon, *Thick-restart Lanczos method for large symmetric eigenvalue problems*, SIAM J. Matrix Anal. Appl., 22 (2000), pp. 602–616.
[25] I. Yamazaki, H. Anzt, S. Tomov, M. Hoemmen, and J. Dongarra, *Improving the performance of CA-GMRES on multicores with multiple GPUs*, in Proceedings of the IEEE International Parallel and Distributed Processing Symposium (IPDPS), IEEE, Washington, DC, 2014, pp. 382–391.

[26] I. Yamazaki, Z. Bai, H. Simon, L.-W. Wang, and K. Wu, *Adaptive projection subspace dimension for the thick-restart Lanczos method*, ACM Trans. Math. Software, 37 (2010), 27.

[27] I. Yamazaki, S. Tomov, T. Dong, and J. Dongarra, *Mixed-precision orthogonalization scheme and adaptive step size for CA-GMRES on GPUs*, Tech. report UT-EECS-14-730, University of Tennessee, Knoxville; in High-Performance Computing for Computational Science (VECPAR 2014), Springer, New York, to appear.

[28] I. Yamazaki, S. Tomov, and J. Dongarra, *Stability and performance of various singular value QR implementations and their case-studies with adaptive mixed-precision on multicore CPU with GPUs*, 2015.

[29] I. Yamazaki and K. Wu, *A communication-avoiding thick-restart Lanczos method on a distributed-memory system*, in Proceedings of the Workshop on Algorithms and Programming Tools for Next-Generation High-Performance Scientific and Software (HPCC), 2011.