Mixed-precision Block Gram Schmidt Orthogonalization

Ichitaro Yamazaki*, Stanimire Tomov*, Jakub Kurzak*, Jack Dongarra*, and Jesse Barlow†
*Department of Electrical Engineering and Computer Science,
University of Tennessee, Knoxville, U.S.A.
†Department of Computer Science and Engineering,
Pennsylvania State University, Pennsylvania, U.S.A.

ABSTRACT

The mixed-precision Cholesky QR (CholQR) can orthogonalize the columns of a dense matrix with the minimum communication cost. Moreover, its orthogonality error depends only linearly to the condition number of the input matrix. However, when the desired higher-precision is not supported by the hardware, the software-emulated arithmetics are needed, which could significantly increase its computational cost. When there are a large number of columns to be orthogonalized, this computational overhead can have a dramatic impact on the orthogonalization time, and the mixed-precision CholQR can be much slower than the standard CholQR. In this paper, we examine several block variants of the algorithm, which reduce the computational overhead associated with the software-emulated arithmetics, while maintaining the same orthogonality error bound as the mixed-precision CholQR. Our numerical and performance results on multicore CPUs with a GPU, as well as a hybrid CPU/GPU cluster, demonstrate that compared to the mixed-precision CholQR, such a block variant can obtain speedups of up to 7.1× while maintaining about the same order of the numerical errors.

CCS Concepts

•**Mathematics of computing** → *Mathematical software performance; Computations on matrices;*

Keywords

Orthogonalization; GPU computation; Mixed precision.

1. INTRODUCTION

Orthogonalization of dense vectors plays a critical role in many scientific and engineering computation (in terms of numeric and performance). For example, subspace projection methods are widely-used methods for solving a large-scale linear system of equations [9] or solving a large-scale eigenvalue or singular value problem [10]. Both performance and numerical stability of these solvers depend critically on those of the orthogonalization procedure that generates the basis vectors of the projection subspace. Another important application of an orthogonalization procedure is a least-squares solution of an overdetermined system of equations [3].

The modified Gram Schmidt (MGS) [2] is a well-studied procedure for orthogonalizing the columns of a dense matrix. The norm of its orthogonality error depends only linearly to the condition number of the input matrix. However, to orthogonalize each column, it requires a couple of global all-reduces among the parallel processes, and each process performs its local computation based on BLAS-1 and BLAS-2. On the other hand, to orthogonalize all these columns, the Cholesky QR (CholQR) [11] requires only one global all-reduce, while performing most of its local computation using BLAS-3. On a modern computer, the communication has become significantly more expensive compared to the arithmetic operations, where the communication includes the data movement or synchronization between parallel processes, as well as data movement between the levels of the local memory hierarchy. As a result, compared to other orthogonalization procedures, CholQR obtains an excellent performance on such computers. Unfortunately, in a finite precision, the orthogonality error of CholQR depends quadratically to the condition number of the input matrix, potentially leading to a large numerical error for an ill-conditioned input matrix.

To reduce the numerical error of CholQR, we recently used the doubled-precision for a half of the arithmetic operations required by CholQR [14]. The orthogonality error of this mixed-precision CholQR (mCholQR) depends only linearly to the condition number of the input matrix. Unfortunately, when the desired higher precision is not supported by the hardware, the software-emulated arithmetics are needed, which can significantly increases its computational cost. This could become an intolerable overhead, especially when there are a large number of vectors to be orthogonalized, and the orthogonalization time depends greatly on the computational cost of the algorithm. Randomized algorithms to compute or update the low-rank approximation of a large-scale matrix are examples that often require to orthogonalize a large number of dense basis vectors of their projection subspaces [4, 8, 12].

To lower the computational overhead due to the software-emulated arithmetics of mCholQR, in this paper, we combine mCholQR with a block variant of MGS (BMGS), and use mCholQR to orthogonalize the columns within each block column. This variant of BMGS only requires a couple of global all-reduces to orthogonalize each block column, while most of the local computation can be performed using BLAS-3. In addition, though to maintain the numerical stability, BMGS requires to reorthogonalize each block column, compared to mCholQR, the computational overhead associated with the software-emulated arithmetics can be greatly reduced (i.e., by the factor of the number of block columns). Our numerical results show that when combined with mCholQR, BMGS

ScalA '15, November 16, 2015, Austin, TX, USA

DOI: http://dx.doi.org/10.1145/2832080.2832082

for $j = 1, 2, \ldots, n_t$ **do**
  $R_{1:j-1,j} := Q_{1:j-1}^T X_j$
  $X_j := X_j - Q_j R_{1:j-1,j}$
  $[Q_j, R_{j,j}] := \mathrm{TSQR}(X_j)$
end for

(a) Block CGS.

for $j = 1, 2, \ldots, n_t$ **do**
  $[Q_j, R_{j,j}] := \mathrm{TSQR}(X_j)$
  $R_{j,j+1:n_t} := Q_j^T X_{j+1:n_t}$
  $X_{j+1:n_t} := X_{j+1:n_t} - Q_j R_{j,j+1:n_t}$
end for

(b) Block MGS.

Figure 1: Block GS algorithms, where n_t is the number of block columns in the input matrix X, and $[Q_j, R_{j,j}] = \mathbf{TSQR}(X_j)$ returns the QR factorization of X_j such that $Q_j^T Q_j = I$, $R_{j,j}$ is upper-triangular, and $Q_j R_{j,j} := X_j$.

Name	Description
CholQR	standard CholQR
mCholQR	mixed-precision CholQR
BMGS	once CholQR for TSQR
B2MGS	twice CholQR for TSQR
mBMGS	once mCholQR for TSQR
mB2MGS	twice mCholQR for TSQR
mB1.5MGS	mCholQR followed by CholQR for TSQR

Figure 2: Orthogonalization algorithms studied in this paper.

for $j = 1, 2, \ldots, n_t$ **do**
  $[Q_j, R_{j,j}] = \mathrm{mCholQR}(X_j)$
  $B_j = Q_j^T Q_j$
  $\widehat{R}_{j,j} = \mathrm{chol}(B_j)$
  $R_{j,j} = \widehat{R}_{j,j} R_{j,j}$
  $R_{j,j+1:n_t} := Q_j^T X_{j+1:n_t}$
  $R_{j,j+1:n_t} := \widehat{R}_{j,j}^{-T} R_{j,j+1:n_t}$
  $T_{j,j+1:n_t} := \widehat{R}_{j,j}^{-1} R_{j,j+1:n_t}$
  $X_{j+1:n_t} := X_{j+1:n_t} - Q_j T_{j,j+1:n_t}$
end for

Figure 3: Block GS algorithms, where the matrix Q_j is kept in the implicit form of $Q_j \widehat{R}_j$.

can obtain similar numerical errors as mCholQR, while our performance results on multicore CPUs with a GPU, as well as a hybrid CPU/GPU cluster, demonstrate the speedups of up to 7.1×.

The rest of the paper is organized as follows. In Section 2, we first describe the orthogonalization algorithms studied in this paper. Then, in Section 3, we compare the numerical errors of different algorithms. Next, in Section 4, we present the implementation and performance of the GPU kernels required to implement the orthogonalization algorithms on a hybrid CPU/GPU architecture. Finally, in Section 5, we show the performance of the orthogonalization algorithms on the hybrid architecture. The final remarks are listed in Section 6.

2. ALGORITHMS

We study various algorithms to orthogonalize the columns of an m-by-n dense matrix X based on its QR factorization,

$$QR := X,$$

where X is a tall-skinny matrix (i.e., $m \gg n$), Q is an m-by-n orthonormal matrix (i.e., $Q^T Q = I$), and R is an n-by-n upper-triangular matrix.

2.1 Classical and modified Gram Schmidt

The Gram Schmidt (GS) [3] is a widely-studied orthogonalization procedure. For example, the classical GS (CGS) orthonormalizes each column against the previously-orthognoalized columns, at once. Unfortunately, its local computation is based on BLAS-2 and BLAS-1, and it requires $O(n)$ all-reduces among the parallel processing units. In addition, compared to CholQR, it has a much greater bound on its orthogonalization error (i.e., $\|I - Q^T Q\| = O(\varepsilon \kappa(X)^n)$, where ε is the machine epsilon and $\kappa(X)$ is the condition number of X).

The modified GS (MGS) reduces the orthogonality error of CGS and has the same error bound as the mixed-precision CholQR (i.e., $\|I - Q^T Q\| = O(\varepsilon \kappa(X))$). However, it orthogonalizes each column against each of the previous columns at a time, and its local computation is still based on BLAS-2 and BLAS-1. Like CGS, MGS requires $O(n)$ all-reduces among the parallel processing units.

2.2 Mixed-precision Cholesky QR

For our implementation of CholQR on multicore CPUs with a GPU [11], we first compute the Gram matrix B of the input matrix X using the GPU (i.e., $B := X^T X$). Then, the Gram matrix is copied to the CPUs, where we compute its Cholesky factorization (i.e., $RR^T := B$). Finally, the upper-triangular factor R is copied back to the GPU, where we orthogonalize X through triangular solves (i.e., $Q := XR^{-1}$). Hence, CholQR performs most of its computation using BLAS-3, exploiting the high compute power and memory bandwidth of the GPU. The implementation can be easily extended to utilize a hybrid CPU/GPU cluster, where we would require only one global all-reduce among the MPI processes to form the Gram matrix, and each MPI process would redundantly compute the Cholesky factor on the CPU [13]. Though CholQR obtains an excellent performance on modern computers, the condition number of the Gram matrix B is the square of the condition number of X. As a result, the orthogonality error of CholQR depends quadratically to the condition number of X (i.e., $\|I - Q^T Q\|_2 < O(\varepsilon \kappa(X)^2)$). In fact, when the condition number of X is greater than the reciprocal of the square-root of the machine epsilon (i.e., $\kappa(X) > \varepsilon^{-1/2}$), the Cholesky factorization of the Gram matrix can fail.

To reduce the orthogonality error of CholQR in finite precision, we used the doubled-precision for computing and factorizing the Gram matrix [14]. When the condition number of X is less than the reciprocal of the machine epsilon, the orthogonality error of this mixed-precision CholQR (mCholQR) depends linearly to the condition number of X (i.e., $\|I - Q^T Q\|_2 < O(\varepsilon \kappa(X))$ when $\kappa(X) < \varepsilon^{-1}$). This is the same bound as that of MGS.

When the desired doubled-precision is not supported by the hardware, the implementation of mCholQR requires software emulation for the higher precision arithmetics. Since CholQR performs about a half of the total flops computing the Gram matrix, the use of the software-emulated arithmetics for computing the Gram matrix may significantly increase the total computational cost of the orthgonalization process. For instance, in our previous studies [14], we used the double-double arithmetics [5] to emulate the quadruple precision for the working double precision. This increases the required arithmetic instruction count by a factor of 8.5×. However, the input matrix X is still read in the working precision, while on average, CholQR performs n flops for each numerical value read. On a modern computer, the data movement is significantly more expensive

compared to the arithmetic operations. As a result, when the input matrix has only a small number of columns, the orthogonalization times of both CholQR and mCholQR are bounded by the memory bandwidth. Hence, mCholQR may not be significantly slower than CholQR. For instance, in our previous studies on two eight-core Intel SandyBridge CPUs with an NVIDIA Kepler GPU, mCholQR was only $1.7\times$ slower than CholQR when the input matrix has less than 20 columns (i.e., $n \leq 20$).

2.3 Mixed-precision Block Gram Schmidt

As the number of columns in X increases, the performance of CholQR starts to be bounded less by the memory bandwidth but more by the computation capacity of the hardware. As a result, with a large enough number of columns to be orthogonalized, mCholQR becomes significantly slower than CholQR since mCholQR performs $8.5\times$ more arithmetic instructions.

To reduce the computational overhead of mCholQR, we examine alternative orthogonalization procedures based on block variants of CGS or MGS (BCGS and BMGS, respectively). In these block algorithms, the j-th block column X_j of X is recursively orthogonalized by first orthogonalizing it against the previous block columns (for $j = 1, 2, \ldots, n_t$, where n_t is the number of the block columns in X),

$$\begin{aligned} X_j &:= (I - Q_{1:j-1} Q_{1:j-1}^T) X_j \quad \text{(in BCGS) or,} \\ &:= (I - Q_1 Q_1^T) \ldots (I - Q_k Q_k^T) X_j \quad \text{(in BMGS),} \end{aligned}$$

followed by the tall-skinny QR (TSQR) factorization of X_j,

$$Q_j R_j := X_j.$$

Figure 1 shows the pseudocodes of these block orthogonalization processes.

A few variants of BMGS and BCGS, that obtain the same orthogonality error bound as MGS, have been proposed [1, 7]. For instance, if the orthogonality error of TSQR is bounded by $O(\varepsilon)$, then BMGS can be as stable as MGS [7]. This bound may be obtained using CholQR or mCholQR. In particular, if the condition number of X_j is less than the reciprocal of the machine epsilon or less than the reciprocal of the square-root of the machine epsilon, then the respective orthogonality error of mCholQR or CholQR is in the order of one (i.e., $\|I - Q^T Q\| < O(1)$). Hence, after reorthogonalization using the standard CholQR, the orthogonality error becomes in the order of machine epsilon. As a result, by using CholQR or mCholQR followed by the reorthogonalization by CholQR for TSQR, BMGS may be able to obtain the same accuracy as MGS, while performing most of its local computation using BLAS-3 and reducing the number of all-reduces among the parallel processes by the factor of the block size n_b.

When the standard CholQR is used for TSQR, both BCGS and BMGS perform the same number of flops as CGS or MGS (i.e., all the algorithms perform total of about $2mn^2$ flops). Among these flops, both BCGS and BMGS with the block size of n_b perform about $2mnn_b$ flops for TSQR (i.e., about $\frac{1}{n_t}$ of total flops are for TSQR). Hence, when we perform a full-reorthogonalization for TSQR (i.e., CholQR twice), BCGS and BMGS would require $(1 + \frac{1}{n_t})\times$ more flops. In addition since the mixed-precision mCholQR performs $8.5\times$ more instructions than the standard CholQR, when TSQR is based on mCholQR followed by a full-reorthogonalization based on either CholQR or mCholQR, compared to the standard block algorithm, BCGS or BMGS performs about $(1 + \frac{16}{n_t})\times$ and $(1 + \frac{8.5}{n_t})\times$ more instructions, respectively. Alternately stating, they reduce the computational cost of mCholQR by the factors of about $\frac{2.1}{n_t}$ and $\frac{1.1}{n_t}$, respectively, while maintaining the same orthogonality bound. Table 2 lists the block Gram Schmidt that uses different combinations of CholQR and mCholQR for TSQR. For example, we refer to BMGS combined with mCholQR with reorthogonalization using CholQR or mCholQR as mB1.5MGS or mB2MGS, respectively.

For the upper-bound of mCholQR to hold, the condition number of the input matrix X must be less than the reciprocal of the square-root of the machine epsilon. Otherwise, the Cholesky factorization of the Gram matrix in the double-double precision can fail. On the other hand, B1.5MGS only requires the condition number of each block column X_j to be the reciprocal of the machine epsilon.

In this paper, we also examine an algorithmic variant of the block Gram Schmidt, where the orthogonal block column Q_j is not explicitly computed, but kept in an implicit form given by two matrices X_j and $\widehat{R}_{j,j}$ such that $Q_j := X_j \widehat{R}_{j,j}^{-1}$. For example, to orthogonalize a new block column X_k against Q_j, we now compute

$$X_k - X_j(\widehat{R}_{j,j}^{-1}(\widehat{R}_{j,j}^{-T}(X_j^T X_k))).$$

When the condition number of X_j is $O(1)$, then this variant of the orthogonalization procedure can be as stable as the original procedure that explicitly computes orthogonal matrix Q_j. Figure 3 shows this algorithmic variant, where Q_j is kept in this implicit form after the reorthogonalization, and it should be as stable as explicitly forming Q_j when the condition number of X_j is less than the reciprocal of the machine epsilon. Moreover, besides CholQR, there is a variant of the orthogonalization procedure, called the Singular Value QR (SVQR) [11], which allows us to estimate the condition number of X_j as a by-product of the orthogonalization process. Specifically, SVQR computes the upper-triangular matrix $R_{j,j}$ by first computing the SVD of the Gram matrix, $U\Sigma U^T := B$, followed by the QR factorization of $\Sigma^{\frac{1}{2}} U^T$. Hence, using SVQR, it is possible to adaptively decide if we can safely keep each block column Q_j in its implicit form at run time.

By keeping the orthogonal block column Q_j in the implicit form $X_j \widehat{R}_{j,j}^{-1}$, the computational cost of CholQR to orthogonalize X_j is reduced in half (stable if $\kappa(X_j) = O(1)$), while it reduces the computational cost of CholQR with reorthogonalization by 25% (stable if $\kappa(X_j) < \varepsilon^{-1/2}$). In addition, by keeping the orthogonal columns in this implicit form, the computational cost of mCholQR with reorthogonalization by CholQR can be reduced by $\frac{1}{19}$ (stable when $\kappa(X_j) < \varepsilon^{-1}$). Hence, the computational costs of B2MGS and mB1.5MGS can be reduced by $\frac{1}{4n_t}$ and $\frac{1}{19n_t}$, respectively, when all the block columns are stored in this implicit form. This algorithmic variant is useful when the orthogonal matrix Q does not have to be explicitly formed. For instance, the matrix powers kernel [6] computes the matrix powers from the initial orthogonal block column, which is the last block column of the previously-generated basis vectors. Hence, only the last block column needs to be explicitly formed. Another potential application is the least-squares solution of the overdetermined system, where after the QR factorization of the coefficient matrix, we need to project the right-hand-sides onto the orthogonal space, which can be done in the implicit form.

3. NUMERICAL RESULTS

We now compare the numerical errors of the different variants of block Gram Schmidt. We performed all of our numerical experiments in the working 64-bit double precision (i.e., $\varepsilon = O(10^{-16})$).

Table 4 compares the numerical errors of block Gram Schmidt for a test matrix X defined as $X = (I + \alpha H_1)MH_2$, where H_1 and H_2 are 1024-by-1024 and 512-by-512 random matrices with uniformly distributed random numbers in the interval $(-1, 1)$, respectively,

n_b	BCGS	B2CGS	mBCGS	mB2CGS	mB1.5CGS
1	3.8×10^{-6}	3.9×10^{-6}	4.2×10^{-6}	3.9×10^{-6}	4.2×10^{-6}
32	4.0×10^{-7}	4.8×10^{-7}	4.7×10^{-7}	4.8×10^{-7}	4.8×10^{-7}
64	3.5×10^{-7}	2.8×10^{-7}	2.8×10^{-7}	2.8×10^{-7}	2.8×10^{-7}
128	2.1×10^{-7}	2.1×10^{-7}	2.5×10^{-7}	2.1×10^{-7}	2.1×10^{-7}
256	7.7×10^{-8}	3.5×10^{-11}	8.6×10^{-9}	3.8×10^{-11}	3.8×10^{-11}
512	1.6×10^{-5}	2.6×10^{-15}	2.8×10^{-12}	2.3×10^{-16}	2.3×10^{-15}

(a) Orthogonality error $\|I-Q^TQ\|_2$ of BCGS's.

n_b	BCGS	B2CGS	mBCGS	mB2CGS	mB1.5CGS
1	1.3×10^{-13}	1.1×10^{-13}	1.1×10^{-13}	1.2×10^{-13}	1.2×10^{-13}
32	1.2×10^{-13}	1.1×10^{-13}	1.3×10^{-13}	1.2×10^{-13}	1.2×10^{-13}
64	1.2×10^{-13}	1.4×10^{-13}	1.3×10^{-13}	1.4×10^{-13}	1.4×10^{-13}
128	1.1×10^{-13}	1.2×10^{-13}	1.1×10^{-13}	1.2×10^{-13}	1.2×10^{-13}
256	1.1×10^{-13}	1.2×10^{-13}	1.0×10^{-13}	1.0×10^{-13}	1.0×10^{-13}
512	2.1×10^{-13}	2.2×10^{-13}	2.1×10^{-13}	2.0×10^{-13}	2.0×10^{-13}

(b) Backward error $\|X-QR\|_2$ of BCGS's.

n_b	BMGS	B2MGS	mBMGS	mB2MGS	mB1.5MGS
1	5.0×10^{-11}	5.8×10^{-11}	5.8×10^{-11}	5.8×10^{-11}	5.8×10^{-11}
32	7.8×10^{-9}	3.3×10^{-11}	3.1×10^{-9}	3.3×10^{-11}	3.3×10^{-11}
64	1.5×10^{-8}	1.8×10^{-11}	3.6×10^{-9}	1.8×10^{-11}	1.8×10^{-11}
128	3.8×10^{-8}	2.7×10^{-11}	6.3×10^{-9}	2.7×10^{-11}	2.7×10^{-11}
256	7.7×10^{-8}	3.5×10^{-11}	8.6×10^{-9}	3.8×10^{-11}	3.8×10^{-11}
512	1.6×10^{-5}	2.6×10^{-15}	2.8×10^{-12}	2.3×10^{-15}	2.3×10^{-15}

(c) Orthogonality error $\|I-Q^TQ\|_2$ of BMGS's.

n_b	BMGS	B2MGS	mBMGS	mB2MGS	mB1.5MGS
1	2.1×10^{-13}	2.0×10^{-13}	2.0×10^{-13}	2.0×10^{-13}	2.0×10^{-13}
32	1.9×10^{-13}	1.9×10^{-13}	2.0×10^{-13}	1.9×10^{-13}	1.9×10^{-13}
64	1.9×10^{-13}	1.9×10^{-13}	1.8×10^{-13}	1.9×10^{-13}	1.9×10^{-13}
128	1.6×10^{-13}	1.5×10^{-13}	1.5×10^{-13}	1.5×10^{-13}	1.5×10^{-13}
256	1.1×10^{-13}	1.2×10^{-13}	1.0×10^{-13}	1.0×10^{-13}	1.0×10^{-13}
512	2.1×10^{-13}	2.2×10^{-13}	2.1×10^{-13}	2.0×10^{-13}	2.0×10^{-13}

(d) Backward error $\|X-QR\|_2$ of BMGS's.

Figure 4: Numerical error of BGS's for 1024-by-512 matrix $X=(I+\alpha H_1)MH_2$ with $\kappa(X)=3.5\times10^6$.

n_b	BCGS	B2CGS	mBCGS	mB2CGS	mB1.5CGS
1	$4.9\times10^{+1}$	$5.1\times10^{+1}$	$4.6\times10^{+1}$	$5.4\times10^{+1}$	$5.5\times10^{+1}$
10	$1.0\times10^{+1}$	$1.0\times10^{+1}$	$1.0\times10^{+1}$	$1.0\times10^{+1}$	$1.0\times10^{+1}$
20	$4.1\times10^{+0}$	$4.0\times10^{+0}$	$4.0\times10^{+0}$	$4.0\times10^{+0}$	$4.1\times10^{+0}$
40	$2.0\times10^{+0}$	$2.0\times10^{+0}$	$2.0\times10^{+0}$	$2.0\times10^{+0}$	$2.0\times10^{+0}$
100	$4.5\times10^{+0}$	9.4×10^{-3}	$1.0\times10^{+0}$	1.0×10^{-2}	1.0×10^{-2}
200	–	–	5.2×10^{-3}	1.0×10^{-15}	1.0×10^{-15}

(a) Orthogonality error $\|I-Q^TQ\|_2$ of BCGS's.

n_b	BCGS	B2CGS	mBCGS	mB2CGS	mB1.5CGS
1	2.2×10^{-10}	2.3×10^{-10}	2.2×10^{-10}	2.2×10^{-10}	2.2×10^{-10}
10	2.8×10^{-10}	2.9×10^{-10}	2.9×10^{-10}	3.0×10^{-10}	2.8×10^{-10}
20	3.5×10^{-10}	3.4×10^{-10}	3.4×10^{-10}	3.5×10^{-10}	3.5×10^{-10}
40	4.5×10^{-10}	4.3×10^{-10}	4.4×10^{-10}	4.5×10^{-10}	4.5×10^{-10}
100	6.6×10^{-10}	6.9×10^{-10}	6.3×10^{-10}	6.5×10^{-10}	6.5×10^{-10}
200	–	–	8.8×10^{-10}	1.0×10^{-9}	1.0×10^{-9}

(b) Backward error $\|X-QR\|_2$ of BCGS's.

n_b	BMGS	B2MGS	mBMGS	mB2MGS	mB1.5MGS
1	6.1×10^{-3}	5.5×10^{-3}	6.9×10^{-3}	6.7×10^{-3}	6.9×10^{-3}
10	7.8×10^{-3}	9.4×10^{-3}	7.7×10^{-3}	5.6×10^{-3}	5.9×10^{-3}
20	6.8×10^{-3}	6.2×10^{-3}	7.4×10^{-3}	7.7×10^{-3}	7.4×10^{-3}
40	3.8×10^{-2}	6.9×10^{-3}	8.7×10^{-3}	7.8×10^{-3}	8.2×10^{-3}
100	4.5×10^{-0}	9.4×10^{-3}	1.0×10^{-0}	1.0×10^{-2}	1.0×10^{-2}
200	–	–	5.2×10^{-3}	9.6×10^{-16}	1.0×10^{-15}

(c) Orthogonality error $\|I-Q^TQ\|_2$ of BMGS's.

n_b	BMGS	B2MGS	mBMGS	mB2MGS	mB1.5MGS
1	8.8×10^{-10}	8.7×10^{-10}	9.3×10^{-10}	8.8×10^{-10}	9.6×10^{-10}
10	8.0×10^{-10}	7.8×10^{-10}	8.7×10^{-10}	9.3×10^{-10}	8.5×10^{-10}
20	8.8×10^{-10}	8.9×10^{-10}	8.6×10^{-10}	8.3×10^{-10}	7.8×10^{-10}
40	8.1×10^{-10}	8.1×10^{-10}	8.0×10^{-10}	7.9×10^{-10}	7.8×10^{-10}
100	6.6×10^{-10}	6.9×10^{-10}	6.3×10^{-10}	6.5×10^{-10}	6.5×10^{-10}
200	–	–	8.8×10^{-10}	1.0×10^{-9}	1.0×10^{-9}

(d) Backward error $\|X-QR\|_2$ of BMGS's.

Figure 5: Numerical error of BGS's for 1089-by-200 matrix $X=[X_1,AX_1,\ldots,A^{19}X_1]$, where A is 2D Laplacian and $\kappa(X)=7.5\times10^{15}$.

and M is a 1024-by-512 matrix such that

$$\begin{cases} m_{1,j}=1, & j=1,2,\ldots,512,\\ m_{i,j}=\beta\,\delta_{i-1,j}, & i=2,3,\ldots,1024,\end{cases}$$

where α and β are constant scalars, and $\delta_{i,j}$ is a delta function such that $\delta_{i,j}$ is one when $i=j$, and it is zero otherwise. In our experiments, we set the constant scalars such that $(\alpha,\beta)=(10^{-3},10^{-2})$. This matrix was previously used in [7] to study the numerical accuracies of a BMGS. The condition number of the matrix (computed using the SVD of LAPACK) is about 3.5×10^6. With this relatively small condition number of the input matrix (i.e., $\kappa(X)<\varepsilon^{-1/2}$), though CholQR has a greater orthogonality error bound than mCholQR, after reorthogonalization, they both obtain the same error bound (i.e., $\|I-Q^TQ\|_2<O(\varepsilon)$). We can see this in the table with $n_b=512$ (i.e., $n=512$), where B2MGS obtains the same error as both mB2MGS and mB1.5MGS.

The block size of one (i.e., $n_b=1$) with BCGS or BMGS corresponds to the column-wise CGS or MGS, respectively. As expected, in Table 4, we see that CGS has greater orthogonality errors than MGS. In addition, mBCGS or mBMGS with $n_b=n$ corresponds to mCholQR, which obtains about the same orthogonality errors as that of MGS. With BCGS's, a larger block size improved the accuracy since CholQR used for TSQR has a lower orthogonality error bound than CGS. For BCGS, using the mixed-precision or reorthogonalization for TSQR did not improve the numerical errors. At the end, the orthogonality errors of all the BCGS-based procedures were greater than those of MGS. For all the BMGS's, the error could increase with a larger block size, but both mB2MGS and mB1.5MGS obtained the same order of errors as the column-wise MGS. All the procedures were backward stable.

n_b	B2MGS	mB1.5MGS
1	8.6×10^{-11}	6.8×10^{-11}
32	9.5×10^{-11}	1.5×10^{-10}
64	3.1×10^{-11}	1.6×10^{-10}
128	1.2×10^{-10}	1.5×10^{-10}
256	1.5×10^{-10}	2.4×10^{-10}
512	3.9×10^{-15}	3.0×10^{-15}

(a) Orthogonal error $\|I-Q^TQ\|_2$.

n_b	B2MGS	mB1.5MGS
1	2.2×10^{-13}	2.0×10^{-13}
32	2.0×10^{-13}	2.0×10^{-13}
64	2.2×10^{-13}	1.9×10^{-13}
128	2.2×10^{-13}	1.7×10^{-13}
256	2.3×10^{-13}	1.4×10^{-13}
512	2.1×10^{-13}	1.9×10^{-13}

(b) Backward error $\|X-QR\|_2$.

Figure 7: Numerical error with implicit Q for 1024-by-512 matrix $X=(I+\alpha H_1)MH_2$ with $\kappa(X)=3.5\times10^6$.

Tables 5 and 6 show the numerical errors for test matrices whose condition numbers are greater than that of the matrix in Table 4. The test matrices consist of block Krylov basis vectors, $\mathscr{K}_{20}(A,X_1)$, where the coefficient matrix A is a 2D Laplacian matrix, and the starting block X_1 is 1089-by-10 columns with uniformly distributed random number in the interval $(-1,1)$. For Table 5, the input matrix is $X=[X_1,AX_1,\ldots,A^{19}X_1]$, while for Table 6, we reorder the columns such that $X_j=[\mathbf{x}_j,A\mathbf{x}_j,\ldots,A^{19}\mathbf{x}_j]$, and $\mathbf{x}_j$ is the j-th col-

n_b	BCGS	B2CGS	mBCGS	mB2CGS	mB1.5CGS
1	$1.3\times10^{+2}$	$1.2\times10^{+2}$	$1.5\times10^{+2}$	$1.4\times10^{+2}$	$1.4\times10^{+2}$
10	–	$1.1\times10^{+1}$	$1.2\times10^{+1}$	$1.1\times10^{+1}$	$1.1\times10^{+1}$
20	–	$2.8\times10^{+0}$	4.7×10^{-2}	4.1×10^{-2}	5.0×10^{-2}
40	–	–	2.8×10^{-2}	1.8×10^{-2}	2.4×10^{-2}
100	–	–	1.7×10^{-2}	7.7×10^{-3}	8.2×10^{-3}
200	–	–	1.1×10^{-2}	9.3×10^{-16}	1.2×10^{-15}

(a) Orthogonality error $\|I-Q^TQ\|_2$ of BCGS's.

n_b	BCGS	B2CGS	mBCGS	mB2CGS	mB1.5CGS
1	1.2×10^{-9}	1.4×10^{-9}	1.3×10^{-9}	1.6×10^{-9}	2.1×10^{-9}
10	–	5.6×10^{-10}	5.4×10^{-10}	6.1×10^{-10}	5.8×10^{-10}
20	–	5.0×10^{-10}	3.9×10^{-10}	4.2×10^{-10}	4.3×10^{-10}
40	–	–	5.1×10^{-10}	5.6×10^{-10}	5.1×10^{-10}
100	–	–	5.5×10^{-10}	7.5×10^{-10}	7.6×10^{-10}
200	–	–	7.7×10^{-10}	1.2×10^{-9}	1.2×10^{-9}

(b) Backward error $\|X-QR\|_2$ of BCGS's.

n_b	BMGS	B2MGS	mBMGS	mB2MGS	mB1.5MGS
1	8.6×10^{-3}	9.6×10^{-3}	8.6×10^{-3}	8.4×10^{-3}	9.3×10^{-3}
10	–	8.6×10^{-3}	1.0×10^{-0}	8.5×10^{-3}	8.8×10^{-3}
20	–	2.3×10^{-0}	8.0×10^{-3}	7.2×10^{-3}	7.6×10^{-3}
40	–	–	8.5×10^{-3}	7.3×10^{-3}	7.7×10^{-3}
100	–	–	1.7×10^{-2}	7.7×10^{-3}	8.2×10^{-3}
200	–	–	1.1×10^{-2}	9.3×10^{-16}	1.2×10^{-9}

(c) Orthogonality error $\|I-Q^TQ\|_2$ of BMGS's.

n_b	BMGS	B2MGS	mBMGS	mB2MGS	mB1.5MGS
1	7.9×10^{-10}	7.8×10^{-10}	7.9×10^{-10}	7.8×10^{-10}	7.9×10^{-10}
10	–	5.8×10^{-10}	6.1×10^{-10}	6.1×10^{-10}	6.3×10^{-10}
20	–	6.1×10^{-10}	5.7×10^{-10}	5.9×10^{-10}	6.0×10^{-10}
40	–	–	5.5×10^{-10}	6.3×10^{-10}	6.3×10^{-10}
100	–	–	5.5×10^{-10}	7.5×10^{-10}	7.6×10^{-10}
200	–	–	7.7×10^{-10}	1.2×10^{-9}	1.2×10^{-9}

(d) Backward error $\|X-QR\|_2$ of BMGS's.

Figure 6: Numerical error of BGS's for 1089-by-200 matrix $X=[X_1,X_2,\ldots,X_{20}]$, where $X_j=[\mathbf{x}_j,A\mathbf{x}_j,\ldots,A^{19}\mathbf{x}_j]$, A is 2D Laplacian, and $\kappa(X)=7.5\times10^{15}$.

n_b	B2MGS	mB1.5MGS
1	8.4×10^{-3}	1.0×10^{-2}
10	1.5×10^{-0}	1.1×10^{-2}
20	2.8×10^{-0}	9.4×10^{-3}
40	––	1.1×10^{-2}
100	––	1.6×10^{-2}
200	––	2.4×10^{-15}

(a) Orthogonal error $\|I-Q^TQ\|_2$.

n_b	B2MGS	mB1.5MGS
1	7.8×10^{-10}	8.1×10^{-10}
10	1.7×10^{-7}	6.5×10^{-10}
20	4.3×10^{-4}	6.6×10^{-10}
40	––	7.0×10^{-10}
100	––	1.9×10^{-10}
200	––	1.1×10^{-9}

(b) Backward error $\|X-QR\|_2$.

Figure 8: Numerical error with implicit Q for 1089-by-200 matrix $X=[X_1,X_2,\ldots,X_{20}]$, where $X_j=[\mathbf{x}_j,A\mathbf{x}_j,\ldots,A^{19}\mathbf{x}_j]$, A is 2D Laplacian, and $\kappa(X)=7.5\times10^{15}$.

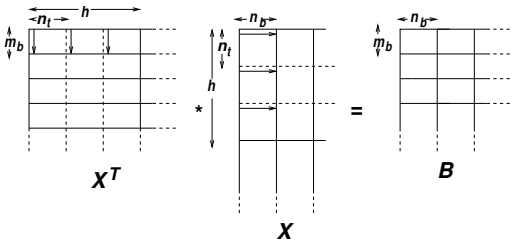


Figure 9: Batched SYRK kernel, where each thread block, which consists of n_t threads, multiplies m_b-by-h and h-by-n_b submatrices of X^T and X, respectively (arrows within the sub-block shows data access by a thread).

umn of X_1 used in Table 5. Hence, these two matrices are the same matrix with different column permutations and have the same condition number. However, the block columns of the matrix in Table 6 have greater condition numbers than that in Table 5. We see the benefits of using the mixed-precision arithmetic and reorthogonalization in Table 6, but not in Table 5.

Figures 7 and 8 show the numerical errors of BMGS-based orthogonalization procedures when the orthogonal matrix Q_j is kept in the implicit form of $Q_j\widehat{R}_{j,j}^{-1}$ after the reorthogonalization. In both tables, mB1.5MGS obtains the numerical errors similar to those that explicitly form Q_j in Figures 4 and 6. On the other hand, for the ill-conditioned matrix in Figure 8, the numerical errors of B2MGS were slightly greater because the condition numbers of the block columns after the initial orthogonalization by CholQR can be greater than $O(1)$.

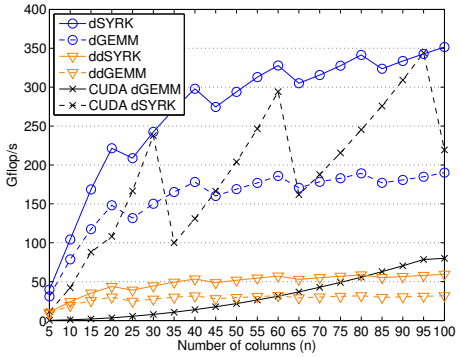


Figure 10: Performance of matrix-matrix multiply on Tesla K40c using CUDA 7.0 ($m=100,000$). Gflop/s is computed as the ratio of the flops needed for general matrix-matrix multiply over the time required in seconds.

4. GPU KERNELS

Our implementations of the orthogonalization algorithms on a hybrid CPU/GPU architecture require the following BLAS kernels on a GPU:

- Symmetric matrix-matrix multiply either in the standard double precision (dSYRK) or in the mixed-precision (mSYRK) that reads the input matrix X in the standard double precision but internally accumulates the resulting matrix B in the double-double precision. These two kernels are needed either in CholQR or mCholQR, respectively.
- Triangular solves in the standard double precision (dTRSM), which is needed in both CholQR and mCholQR.
- General matrix-matrix multiply in the standard double precision (dGEMM) which is needed to orthogonalize the remaining block columns with the current block column based on block Gram Schmidt.

Figure 9 illustrates our GPU implementation of SYRK which is based on a batched SYRK kernel [14]. In this implementation, each thread block first computes a partial result of an m_b-

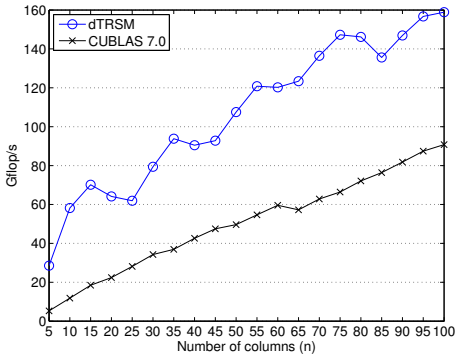


Figure 11: Performance of triangular solves on Tesla K40c using CUDA 7.0 ($m = 100,000$).

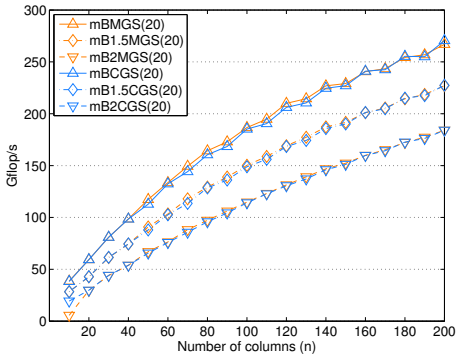


Figure 12: Performance comparison of BMGS and BCGS on two eight-core Intel SandyBridge CPUs with an NVIDIA Kepler GPU ($m = 100,000$).

by-n_b submatrix of the resulting matrix B by multiplying m_b-by-h and h-by-n_b submatrices of X^T and X, respectively. Then, the final result B is computed through a binary reduction among the thread blocks. Figure 10 compares the performance of our kernels with that of CUDA 7.0. Our dSYRK improves the performance of dGEMM by taking advantage of the symmetry of the matrices, while our dGEMM obtains about the same performance as that of CUBLAS. The performance of dSYRK in CUBLAS was significantly lower than that of dGEMM for these particular shapes of the matrices[1]. Compared to dSYRK (or dGEMM), when $n = 20$ or $n = 200$, ddSYRK (or ddGEMM) was only about 3.5× or 5.9× slower, respectively, though performing 16× more floating-point instructions. This is because even with a relatively large number of columns (e.g., $n = 200$), the performance of dSYRK can be still largely influenced by the memory bandwidth, and it is a challenge for dSYRK to obtain the double-precision peak performance of the GPU (both dSYRK and ddSYRK read the input matrix X in the double precision). However, with a larger number of columns, the relative overhead of ddSYRK became greater.

For the block Gram Schmidt to orthogonalize X_j against $Q_{1:(j-1)}$, we call dGEMM twice (e.g., for BCGS, the first dGEMM computes $R_{1:j-1,j} := Q_{1:(j-1)}^T X_j$, and the second dGEMM computes $X_j := X_j - Q_{1:(j-1)} R_{1:(j-1),j}$). In our experiments, we used our batched GPU kernel for the first dGEMM, while CUBLAS dGEMM was used for the second. These GPU kernels obtain excellent performance for the shapes of the input matrices

Our implementation of dTRSM lets each thread independently solve for a different right-hand-side [15]. To read the triangular matrix only once for all the threads in the same thread block, the triangular matrix is first loaded into shared memory. Figure 11 shows the performance of the triangular solves. We see that our implementation obtained significant speedups over CUBLAS.

5. PERFORMANCE RESULTS

Figure 12 compares the performance of BCGS with that of BMGS. While the *left-looking* BCGS has a good data locality to update each block columns, *right-looking* BMGS can exploit the high data locality during the update with each block column. In our experiments, BMGS obtained about the same performance as BCGS, while obtaining the lower orthogonality errors.

[1]We suspect CUBLAS dGEMM calls its batched kernel for this particular shapes of the input matrices, while CUBLAS does not support the batched dSYRK.

Figure 13 shows the breakdown of the orthogonalization time by B1.5MGS, where we varied the numbers of columns to be orthogonalized while fixing the block size to be twenty (i.e., $n = 10 \sim 200$ and $n_b = 20$). As the number of columns increases, B1.5MGS spent more time in the working-precision triangular solve and matrix-matrix multiply (i.e., dTRSM and dGEMM), indicating the decreasing overhead associated with the double-double arithmetics (i.e., ddSYRK and ddPOTRF).

Figure 14 compares the performance of BMGS with the performance of CholQR and mCholQR. First, at $n = 20$ and $n = 200$, CholQR was about 2.7× and 7.1× faster than mCholQR respectively. This is because though in Figure 10, ddSYRK was only about 5.9× slower than dSYRK, for the large number of columns, the sequential ddPOTRF of the Gram matrix on the CPU can dominate the orthogonalization time (e.g., about 50% of the time when $n = 200$). On the other hand, at $n = 200$, CholQR is only about 1.7× and 2.1× faster than B1.5MGS and B2MGS, which perform about 1.9× and 2.6× more floating-point instructions, respectively (see Section 2.3 for their complexity analysis). In other words, these BMGS's reduce the computational overhead associated with the computation and factorization of the Gram matrix in the double-double precision. For instance, the time for ddPOTRF is insignificant in Figure 13 because with $(n, n_d) = (200, 20)$, mB1.5MGS computes 20-by-20 ddPOTRF ten times, which requires much less computation than one 200-by-200 ddPOTRF required by mCholQR. At the end, the total orthogonalization time by B1.5MGS can be shorter than that by CholQR if CholQR needs to reorthogonalize to obtain the desired orthogonality error, while B1.5MGS does not.

Figure 15 compare the performance of different variants of BMGS when the orthogonal matrix Q is either implicitly or explicitly formed. When $n = 200$, by storing Q in the implicit form, the computational costs were reduced only by 5%, 0.5%, and 0.3% in BMGS, B2MGS, and mB1.5MGS, respectively. On the other hand, their respective reductions in the execution times were 12.8%, 10.8%, and 8.1%. This is because the triangular solve often obtain lower performance compared to matrix-matrix multiply and can be more dominant in the orthogonalization time.

We now studies the performance of BMGS on the Tsubame Computer at the Tokyo Institute of Technology[2]. Each of its compute nodes consists of two six-core Intel Xeon CPUs and three NVIDIA Tesla K20Xm GPUs. Figure 16 shows the breakdown

[2]http://tsubame.gsic.titech.ac.jp

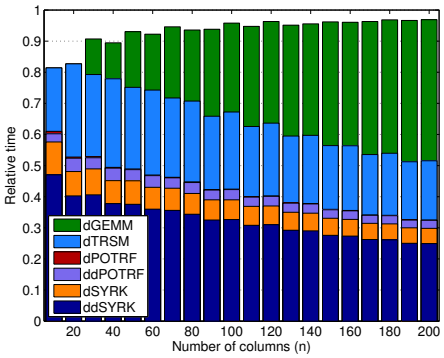


Figure 13: Breakdown of orthogonalization time by mB1.5MGS(20) with $m = 100,000$, where dGEMM is the double-precision matrix-matrix multiply to orthogonalize against the previous blocks; dTRSM is the double-precision triangular solve in mCholQR and CholQR; dPOTRF and ddPOTRF are the Cholesky factorization of the Gram matrix in the double-double and double precisions, respectively; and dSYRK and ddSYTRK are the matrix-matrix multiply in the double and double-double precisions, respectively, to form the Gram matrix.

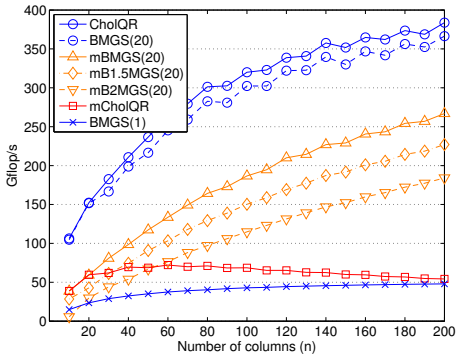


Figure 14:] Performance comparison of BMGS and CholQR on two eight-core Intel SandyBridge CPUs with an NVIDIA Kepler GPU ($m = 100,000$). In the legend, the number in the parenthesis is the block size n_b.

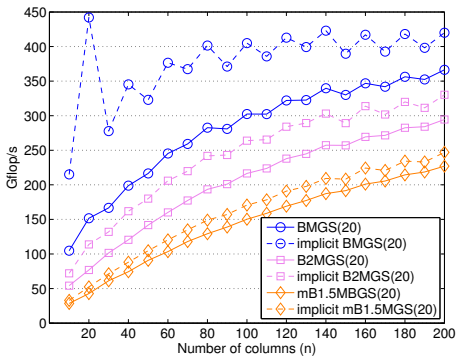


Figure 15: Performance of block GS with implicit Q ($m = 100,000$).

of the orthogonalization time based on CholQR or mCholQR. We clearly see that due to the computational overhead associated with the software-emulated arithmetics, the time needed for ddSYRK and ddPOTRF of mCholQR was much longer than that needed for dSYRK and dPOTRF of CholQR, respectively. In addition, the serial bottleneck of ddPOTRF became significant as the number of GPUs increased. Finally, Figure 17 compares the parallel strong scaling of CholQR, mCholQR, and BMGS. We clearly see that though it has a greater communication latency overhead, BMGS reduces the computational overhead associated with the software-emulated arithmetics of mCholQR, and obtains the notable speedups over mCholQR on the hybrid CPU/GPU cluster.

6. CONCLUSION

We studied the block orthogonalization procedure that combines the block modified Gram Schmidt (BMGS) with the mixed-precision Cholesky QR factorization (mCholQR). Compared to mCholQR, this variant of BMGS significantly reduces the computational overhead associated with the higher-precision arithmetics, while maintaining the same numerical error bound. As the number of columns to be orthogonalized increases, the orthogonalization time becomes dominated more by the computation time. Hence, this BMGS can obtain a significant speedup over mCholQR even though BMGS has a greater communication latency overhead. Our numerical and performance results on multicore CPUs with a GPU, as well as a hybrid CPU/GPU cluster, demonstrated that the speedup $7.07\times$ can be obtained, while maintaining the same order of numerical errors.

We are working to further optimize the GPU kernels required by BMGS (e.g., using Bench-testing Environment for Automated Software Tuning (BEAST)[3]). These optimized kernels not only would improve the performance of the orthogonalization processes, but they also help us understand their performance differences through a fair performance comparison. We are also working to integrate these orthogonalization procedures into the higher-level solver such as the randomized algorithm to compute or update the partial singular value decomposition [12]. Finally, we are performing more extensive numerical studies of the mixed-precision algorithms with the theoretical upper-bounds of their numerical errors.

[3] http://icl.utk.edu/beast/.

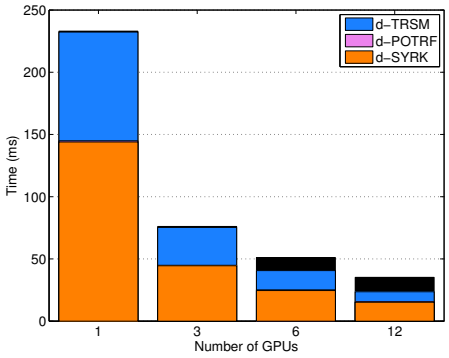
Acknowledgments

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)," Russian Scientific Fund, Agreement N14-11-00190, and "Matrix Algebra for GPU and Multicore Architectures (MAGMA) for Large Petascale Systems."

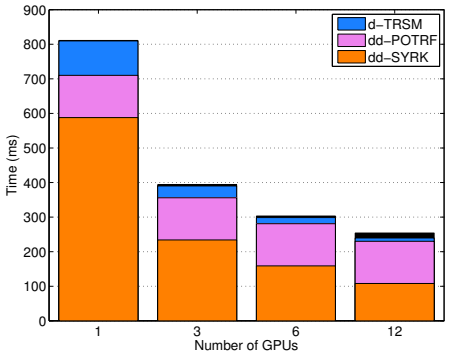
7. REFERENCES

[1] J. Barlow and A. Smoktunowicz. Reorthogonalized block classical Gram-Schmidt. *Numerische Mathematik*, 123:395–423, 2013.

[2] A. Björck. Solving linear least squares problems by Gram-Schmidt orthogonalization. *BIT Numerical Mathematics*, 7:1–âĂŞ21, 1967.

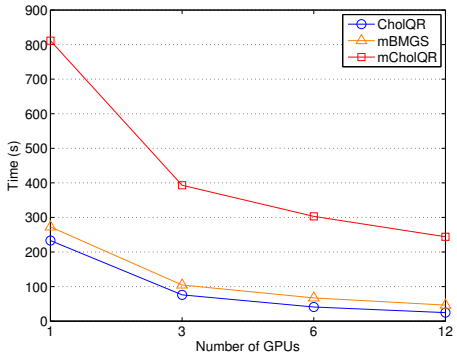


(a) CholQR.

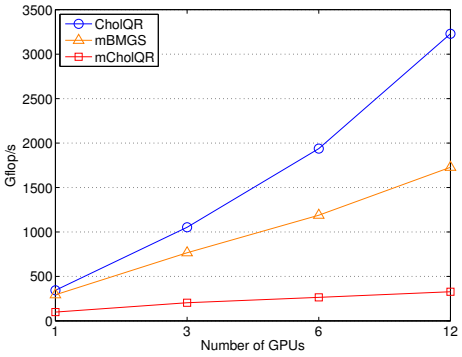


(b) mCholQR.

Figure 16: Breakdown of orthogonalization time on different number of GPUs with $m = 500,000$ and $n = 200$.



(a) Time in seconds.



(b) Performance in Gflop/s

Figure 17: Strong parallel scaling results with $m = 500,000$ and $n = 200$.

[3] G. Golub and C. van Loan. *Matrix Computations*. The Johns Hopkins University Press, Baltimore, MD, 4th edition, 2012.

[4] N. Halko, P. Martinsson, and J. Tropp. Finding structure with randomness: Probabilistic algorithms for constructing approximate matrix decompositions. *SIAM Rev.*, 53:217–288, 2011.

[5] Y. Hida, X. Li, and D. Bailey. Quad-double arithmetic: Algorithms, implementation, and application. Technical Report LBNL-46996, 2000.

[6] M. Hoemmen. *Communication-avoiding Krylov subspace methods*. PhD thesis, EECS Department, University of California, Berkeley, 2010.

[7] W. Jalby and B. Philippe. Stability analysis and improvement of the block Gram-Schmidt algorithm. *SIAM J. Sci. Stat. Comput.*, 12:1058–1073, 1991.

[8] M. Mahoney. Randomized algorithms for matrices and data. *Found. Trends Mach. Learn.*, 3:123–224, 2011.

[9] Y. Saad. *Iterative methods for sparse linear systems*. The Society for Industrial and Applied Mathematics, Philadelphia, PA, 3rd edition, 2003.

[10] Y. Saad. *Numerical methods for large eigenvalue problems*. The Society for Industrial and Applied Mathematics, Philadelphia, PA, 2nd edition, 2011.

[11] A. Stathopoulos and K. Wu. A block orthogonalization procedure with constant synchronization requirements. *SIAM J. Sci. Comput.*, 23:2165–2182, 2002.

[12] I. Yamazaki, J. Kurzak, P. Luszczek, and J. Dongarra. Randomized algorithms to update partial singular value decomposition on a CPU/GPU cluster. In *the proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis (SC)*, 2015.

[13] I. Yamazaki, S. Rajamanickam, E. G. Boman, M. Hoemmen, M. A. Heroux, and S. Tomov. Domain decomposition preconditioners for communication-avoiding Krylov methods on a hybrid CPU/GPU cluster. In *the proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis (SC)*, pages 933–944, 2014.

[14] I. Yamazaki, S. Tomov, and J. Dongarra. Mixed-precision Cholesky QR factorization and its case studies on multicore CPUs with multiple GPUs. *SIAM J. Sci. Comput.*, 37:C307–C330, 2015.

[15] I. Yamazaki, S. Tomov, and J. Dongarra. Stability and performance of various singular value QR implementations on multicore CPU with GPUs. *Submitted to ACM Trans. Math. Softw.*, 2015.