Stability and Performance of Various Singular Value QR Implementations on Multicore CPU with a GPU

ICHITARO YAMAZAKI, STANIMIRE TOMOV, and JACK DONGARRA,
Department of Electrical Engineering and Computer Science, University of Tennessee, Knoxville, TN

Singular Value QR (SVQR) can orthonormalize a set of dense vectors with the minimum communication (one global reduction between the parallel processing units, and BLAS-3 to perform most of its local computation). As a result, compared to other orthogonalization schemes, SVQR obtains superior performance on many of the current computers, where the communication has become significantly more expensive compared to the arithmetic operations. In this article, we study the stability and performance of various SVQR implementations on multicore CPUs with a GPU. Our focus is on the dense triangular solve, which performs half of the total floating-point operations of SVQR. As a part of this study, we examine an adaptive mixed-precision variant of SVQR, which decides if a lower-precision arithmetic can be used for the triangular solution at runtime without increasing the order of its orthogonality error (though its backward error is significantly greater). If the greater backward error can be tolerated, then our performance results with an NVIDIA Kepler GPU show that the mixed-precision SVQR can obtain a speedup of up to 1.36 over the standard SVQR.

Categories and Subject Descriptors: G.1.0 [**Mathematics of computing**]: Numerical Analysis

General Terms: Algorithms, Reliability, Performance

Additional Key Words and Phrases: Orthogonalization, GPU computation, mixed precision

ACM Reference Format:
Ichitaro Yamazaki, Stanimire Tomov, and Jack Dongarra. 2016. Stability and performance of various singular value QR implementations on multicore CPU with a GPU. ACM Trans. Math. Softw. 43, 2, Article 10 (September 2016), 18 pages.
DOI: http://dx.doi.org/10.1145/2898347

1. INTRODUCTION

Orthogonalizing a set of dense column vectors plays a salient part in many scientific and engineering computations. For example, it is a critical component in a software package that solves a large-scale linear system of equations or eigenvalue problem, having great impacts on both its numerical stability and performance. In many of these solvers, the input matrix to be orthogonalized is *tall-skinny*, having more rows the columns. This is true, for instance, in a subspace projection method that computes the orthonormal basis vectors of the generated projection subspace [Saad 2003, 2011]. Other applications

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."
Authors' address: Innovative Computing Laboratory, University of Tennessee, Department of Electrical Engineering and Computer Science, Suite 203 Claxton, 1122 Volunteer Boulevard, Knoxville, TN 37996; emails: {iyamazak, tomov, dongarra}@eecs.utk.edu.

DOI: http://dx.doi.org/10.1145/2898347

of the tall-skinny orthogonalization include the solution of an overdetermined least-squares problem [Golub and van Loan 2012]. Hence, an efficient and stable tall-skinny orthogonalization scheme is valuable in many applications, especially if the scheme is suited for current and emerging computer architectures.

On current computers, communication has become significantly more expensive compared to arithmetic operations, where the communication includes data movement or synchronization between parallel processing units, as well as data movement between the levels of the local memory hierarchy. This holds in terms of both time and energy consumption. It is critical to take this hardware trend into consideration when designing high-performance software on current and future computers. To orthogonalize a tall-skinny dense matrix, both Cholesky QR (CholQR) and Singular Value QR (SVQR) [Stathopoulos and Wu 2002] require only one global reduction between the parallel processing units and perform most of their local computation using BLAS-3. Hence, compared to other orthogonalization schemes, they obtain superior performance on many of the current computers. In addition, though the error bound of CholQR or SVQR depends quadratically on the condition number of the input matrix, with a careful implementation and recursive application of the procedure, CholQR and SVQR can stably orthogonalize many of the ill-conditioned matrices in practice. As a result, we found that CholQR is effective in many practical cases [Yamazaki et al. 2014a, 2015a, 2014b, 2015b].

Unfortunately, for some ill-conditioned matrices, CholQR may require many reorthogonalizations (see Section 5). In this article, we focus on SVQR, which avoids this numerical difficulty of CholQR and has three advantages over CholQR in our current context; first, the upper bound on the orthogonality error of CholQR assumes that the squared condition number of the input matrix is less than the reciprocal of the machine precision, while the upper bound of SVQR does not require this assumption; second, for an ill-conditioned matrix, SVQR may be able to quickly identify the ill-conditioned subspace, while CholQR may require multiple reorthogonalizations; third, SVQR computes the condition number of the Gram matrix, which may be used to adapt the orthogonalization scheme at runtime.

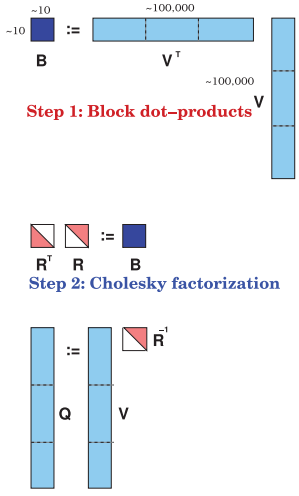
We study the various implementations of SVQR on multicore CPUs with multiple GPUs. Though BLAS-3 kernels are used to perform most of the required flops of SVQR, they must be carefully implemented to obtain stable and high performance. In this article, we focus on the dense triangular solve, which performs half of the total floating-point operations (flops) of SVQR.[1] On a modern computer, lower-precision arithmetic obtains higher peak performance. To take advantage of this hardware trend, as a part of the current study, we examine an adaptive mixed-precision scheme to improve the performance of SVQR. Our adaptive scheme uses the orthogonality error bound derived in Yamazaki et al. [2015a] and the computed condition number of the Gram matrix to adaptively decide if the lower precision can be used for the triangular solution without increasing its orthogonality error bound. The main tradeoff is that the backward error of the mixed-precision SVQR is much greater. If the greater backward error can be tolerated, then our performance results with an NVIDIA Kepler GPU show that the mixed-precision SVQR can obtain a speedup of up to 1.36 over the standard SVQR.

The rest of the article is organized as follows: First, in Section 2, we present our implementation of CholQR and SVQR on multicore CPUs with multiple GPUs and discuss the numerical stability of these two algorithms in practice. Then, in Section 3, we present two different dense triangular solvers: one based on forward substitution and the other based on matrix inversion. To overcome the potential numerical instability associated with the matrix inversion, our implementation adaptively adjusts the sizes

[1]The symmetric matrix-matrix multiplication, which performs the other half of the total flops, is studied in Yamazaki et al. [2015a].

```
Step 1: Form Gram matrix B on GPUs
  for p = 1, 2, ..., n_p do
    B^(p) := V^(p)T V^(p) on p-th GPU
  end for
  B := sum_{p=1}^{n_p} B^(p) (GPUs to CPU global reduction)

Step 2: Compute upper-triangular matrix R on CPU
  with CholQR                with SVQR
    R := chol(B)               [U, Σ, U] = svd(B)
                               [W, R] := qr(√Σ U^T)

Step 3: Compute orthonormal matrix Q on GPUs
  copy R to all the GPUs (CPU to local GPUs broadcast)
  for p = 1, 2, ..., n_p do
    Q^(p) := V^(p) R^-1 on p-th GPU
  end for
```

(a) Pseudocode; $\texttt{chol}(B)$, $\texttt{svd}(B)$, and $\texttt{qr}(B)$ compute the Cholesky, Singular Value, and QR factorization of a matrix B, respectively. When the the condition number of B is greater than the reciprocal of the machine epsilon, $\texttt{chol}(B)$ could fail.



(b) Illustration of CholQR; dashed lines show the 1D block row distribution of the matrix among the GPUs.

Fig. 1. CholQR and SVQR implementations on multicore CPU with multiple GPUs.

of the diagonal blocks to be inverted. Our implementation of the triangular solvers on a GPU and their performance are shown in Section 4. In Section 5, we introduce the adaptive mixed-precision SVQR, and in Sections 6 and 7, we show the numerical and performance results of the mixed-precision SVQR, respectively. Final remarks are listed in Section 8.

2. SINGULAR VALUE QR FACTORIZATION WITH GPUS

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

$$V = QR, \tag{1}$$

where Q is an m-by-n matrix with orthonormal columns (i.e., $Q^T Q = I$), and R is an n-by-n upper-triangular matrix. In order to utilize multiple GPUs, we distribute the matrices V and Q in a 1D block row format. Then, our implementation of CholQR first forms the Gram matrix $B := V^T V$ through the local matrix-matrix product $B^{(p)} := V^{(p)T} V^{(p)}$ on the pth GPU, followed by the reduction $B := \sum_{p=1}^{n_p} B^{(p)}$ on the CPU, where we let n_p be the number of available GPUs, and $V^{(p)}$ is the block row of the matrix V distributed to the pth GPU. We then compute the upper-triangular matrix R based on the Cholesky factorization of B on the CPU (i.e., $R^T R := B$). Finally, each GPU independently orthogonalizes V by the triangular solve $Q^{(p)} := V^{(p)} R^{-1}$. Hence, CholQR requires only one global reduction among the GPUs, while most of the local computation is performed using BLAS-3 on each GPU. Figure 1 shows these three steps of CholQR. This implementation can be trivially extended to use the message-passing interface (MPI) on a hybrid CPU/GPU cluster, and CholQR obtains a great performance even on a distributed-memory system [Yamazaki et al. 2014b].

The condition number of the Gram matrix B is the square of the condition number of the input matrix V, and CholQR may cause numerical instability when V is ill-conditioned. In particular, when the condition number of B is greater than the reciprocal of the machine precision, the Cholesky factorization of the Gram matrix could

Iter.	$\|A - QR\|_2$		$\|I - Q^T Q\|_2$		$\kappa(Q)$	
	CholQR	SVQR	CholQR	SVQR	CholQR	SVQR
0	6.2×10^{1}	6.2×10^{1}	3.9×10^{3}	3.9×10^{3}	4.8×10^{16}	4.8×10^{16}
1	1.6×10^{-15} (f)	3.5×10^{-15} (t)	5.8×10^{2} (f)	1.0×10^{0} (t)	1.4×10^{16} (f)	6.8×10^{8} (t)
2	3.5×10^{-15} (f)	5.6×10^{-15}	2.8×10^{2} (f)	6.7×10^{-13}	7.1×10^{15} (f)	1.0×10^{0}
3	3.9×10^{-15} (f)	7.8×10^{-15}	1.9×10^{2} (f)	5.9×10^{-15}	6.6×10^{15} (f)	1.0×10^{0}
4	4.2×10^{-15}	1.2×10^{-14}	1.5×10^{2}	3.0×10^{-15}	5.4×10^{15}	1.0×10^{0}
5	4.4×10^{-15}	1.1×10^{-14}	1.0×10^{0}	2.8×10^{-15}	2.3×10^{7}	1.0×10^{0}
6	4.4×10^{-15}	1.2×10^{-14}	4.7×10^{-16}	2.1×10^{-15}	1.0×10^{0}	1.0×10^{0}

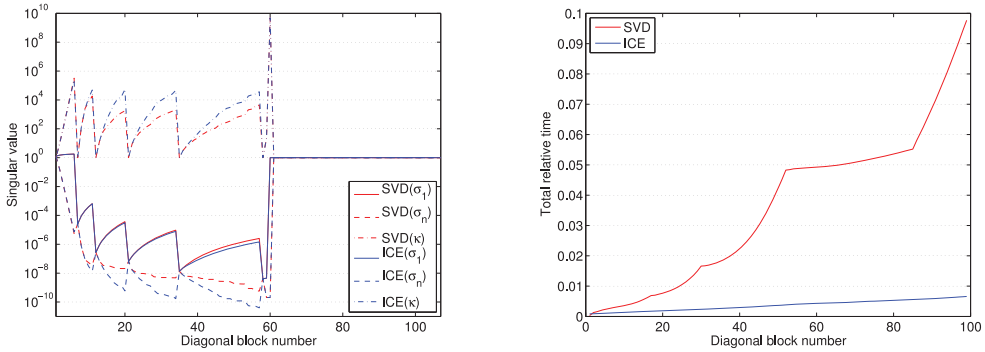
Fig. 2. Error norms $\|I - Q^T Q\|_2$ and $\|A - QR\|_2$, and condition number $\kappa(Q)$ of CholQR and SVQR. We initialized our test matrix as a $1,000$-by-15 matrix of double-precision random numbers from a uniform distribution on an open interval $(0, 1)$. We then scaled every third column with the machine epsilon and added the two previous columns. Hence, every third column of the matrix is nearly a linear combination of the two previous columns. In the table, "f" and "t" indicate that the Cholesky factorization failed or some of the singular values were truncated, respectively.

fail (i.e., encounter a nonpositive pivot). When the Cholesky factorization fails, our implementation sets the trailing submatrix of the Cholesky factor R to be identity. Hence, by recursively applying CholQR on the computed matrix Q, we implicitly block the columns of the input matrix V and orthonormalize the columns block by block such that the corresponding leading submatrix of the Gram matrix has a condition number less than the reciprocal of the machine epsilon, and its Cholesky factorization can be computed. This implementation of CholQR worked especially well for the matrix V whose column vectors become increasingly linearly dependent (e.g., the Krylov basis vectors from a communication-avoiding variant of a Krylov method [Yamazaki et al. 2015b]). Nevertheless, to orthogonalize an ill-conditioned matrix, CholQR may require several reorthogonalizations due to the repeated breakdowns of the Cholesky factorization.

Though SVQR has the same norm-wise upper bound on the orthogonality error as CholQR [Stathopoulos and Wu 2002], SVQR may reduce the number of reorthogonalizations needed by CholQR by identifying the ill-conditioned subspace of the input matrix V all at once. In addition, SVQR does not suffer from the numerical instability of CholQR associated with the Cholesky factorization of the Gram matrix. For this, SVQR 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, just like CholQR, the column vectors are orthonormalized through the triangular solve $Q^{(p)} := V^{(p)} R^{-1}$. Compared to the Cholesky factorization, computing 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 has only one global reduce. To compare the numerical behaviors of CholQR and SVQR in practice, Figure 2 shows the orthogonality errors and the condition numbers of the computed matrices Q for one of our test matrices. We see that the Cholesky factorization can fail multiple times (e.g., one for each linearly dependent column), while SVQR may directly identify the subspace spanned by these nearly dependent columns and quickly reduce the orthogonality error.

3. INVERSION-BASED TRIANGULAR SOLVE

When solving a lower-triangular system of equations for multiple right-hand sides, the performance of the triangular solve may be improved by exploiting the data reuse of the triangular matrix and the parallelism among the right-hand sides. However, the forward substitution computes one element of each solution vector at a time and can exploit only a limited amount of parallelism. The amount of the data parallelism may be increased by first explicitly computing the inverses of the diagonal blocks of



(a) Smallest and largest singular values of diagonal blocks, and corresponding condition numbers.

(b) Total estimation time, relative to orthogonalization time.

Fig. 3. Performance of incremental condition estimator during triangular solve for Hilbert matrix.

the triangular matrix R and then computing the solution based on the matrix-matrix multiplies. In addition, though the numerical error of inverting the diagonal block depends quadratically on the condition number of the diagonal block [Golub and van Loan 2012], this inversion-based solver may be stable for some linear algebra subroutines—for the third step of SVQR, because the first two steps of the factorization have already introduced the numerical errors that depend quadratically on the condition number of the input matrix V.[2] Hence, even if the triangular system is solved using the explicit inverse of R, the overall orthogonality error of SVQR would be in the same order as that using the substitution-based triangular solver (i.e., $\|I - Q^T Q\| = O(\epsilon_d \kappa(V)^2)$, where ϵ_d is the machine epsilon in the working double precision).

However, the inversion-based triangular solve may increase the overall numerical error of other subroutines (e.g., the mixed-precision SVQR in Section 5). To maintain the numerical accuracy of the inversion-based triangular solve, we adaptively block the upper-triangular matrix R such that the condition number of each diagonal block is less than a numerical threshold:

$$O\left(\kappa\left(R^{(i,i)}\right)\right) \leq O(\tau \kappa(R)), \tag{2}$$

where $R^{(i,i)}$ is the (i,i)-th diagonal block of R, and τ is a numerical threshold (see Figure 4(e) for the pseudocode of the blocked algorithm and Section 6 for our use of the adaptive block sizes in the mixed-precision SVQR). In our numerical experiments, to estimate the condition number of the diagonal blocks, we use the incremental condition estimator (ICE) developed for a triangular matrix [Bischof 1990]. Figure 3(a) compares the condition number estimated by ICE against the condition number computed by SVD, demonstrating the high accuracy of the estimation. In addition, Figure 3(b) shows the total time spent in ICE during the orthogonalization based on SVQR, demonstrating that ICE requires only a small overhead in the total orthogonalization time (e.g., less than 1% of the orthogonalization time).

4. TRIANGULAR SOLVE ON A GPU

To solve the triangular system with multiple right-hand sides on a GPU, we first developed GPU kernels, each of which is specialized for a specific size of the upper-triangular matrix R (i.e., $n = 1, 2, \ldots, 40$). This is done using a parameterized GPU kernel shown in Figures 4(a) and 4(b) (a similar parameterization was previously used for auto-tuning GPU kernels [Kurzak et al. 2011]). In this kernel, as shown in

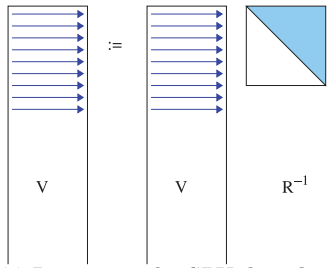
[2]The inversion-based triangular solve is also used for the LU factorization in the MAGMA library.

```
__global__ void
trsm_nb_kernel_name(int m, int n,
                    const FloatingPoint_t* __restrict__ dA, int ldda,
                          FloatingPoint_t*              dB, int lddb)
{
  __shared__ FloatingPoint_t rA[N*N];
  int id = threadIdx.x;
  int ind = blockIdx.x * NB + threadIdx.x;

  if (id < N) {
    #pragma unroll
    for (int j=0; j<N; j++) rA(id,j) = dA(id,j);
  }
  __syncthreads();

  if (ind < m) {
    #pragma unroll
    for (int j=0; j<N; j++) {
      FloatingPoint_t bij = dB(ind,j);
      for (int i=0; i<j; i++) bij -= rA(i,j) * dB(ind,i);
      dB(ind,j) = bij / rA(j,j);
    }
  }
}

void trsm_nb_name(int m, int n,
                  FloatingPoint_t* dA, int ldda,
                  FloatingPoint_t* dB, int lddb )
{
  dim3 grid( (m + NB - 1)/NB );
  dim3 threads( NB );
  trsm_nb_kernel_name <<<grid, threads, 0, stream>>>
                      (m, n, dA, ldda, dB, lddb);
}
```

(a) Base code, trsm_nb.cu.

```
typedef double FloatingPoint_t;
#define NB 128

////////////////////////////////////////////////////////////
#define N 1

#define trsm_nb_name \
        ztrsm_nb1
#define trsm_nb_kernel_name \
        ztrsm_nb1_kernel

#include "trsm_nb.cu"

#undef N
#undef trsm_nb_name
#undef trsm_nb_kernel_name
////////////////////////////////////////////////////////////
#define N 2
.
.
```

(b) Specialized kernel generation.



(c) Data access by GPU threads.

```
extern "C"
void dtrsm_(int m, int n,
            double* dA, int ldda,
                        double* dB, int lddb )
{
  switch(n) {
    case 1:
      dtrsm_nb1( m, n, alpha,
                       dA, ldda,
                       dB, lddb );
      break;
    case 2:
      .
      .
```

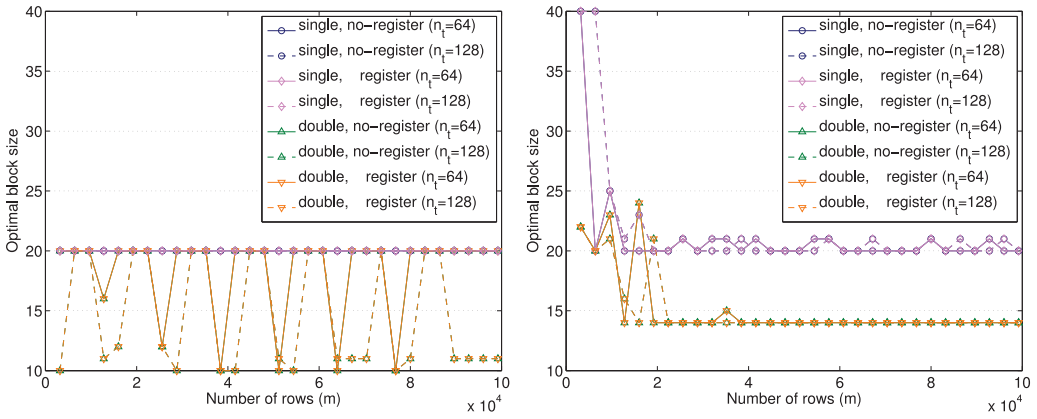
(d) Unblocked code.

```
for (int i=0; i<n; i+=nb) {
  int nbi = min(nb,n-i);
  dtrsm_(m, nbi,
    dA(i,i), ldda,
             dB(0,i), lddb );

  if (i+nb < n) {
    dgemm(NoTrans, NoTrans,
          m, n-i-nb, nb,
          -1.0, dB(0,i), lddb,
                dA(i,i+nb), ldda,
          1.0,  dB(0,i+nb), lddb );
  }
}
```

(e) Blocked code.

Fig. 4. Optimized GPU kernel for triangular solve.

Figure 4(c), each GPU thread independently solves the triangular system for a different right-hand side (i.e., a row of V). Since the matrix V is tall-skinny, containing a large number of right-hand sides, this implementation exploits the high level of parallelism. In addition, the matrix V is stored in column-major layout, and the memory access to read and write V is coalesced among the GPU threads. To reduce the data movement through the memory hierarchy of the GPU, we integrated a few standard optimization techniques; we fixed the loop-boundary by having a specialized kernel for the specific size of R, unrolled the loop using pragma, and loaded the upper-triangular matrix in the shared memory. Then, the top-level unblocked solver shown in Figure 4(d) calls the



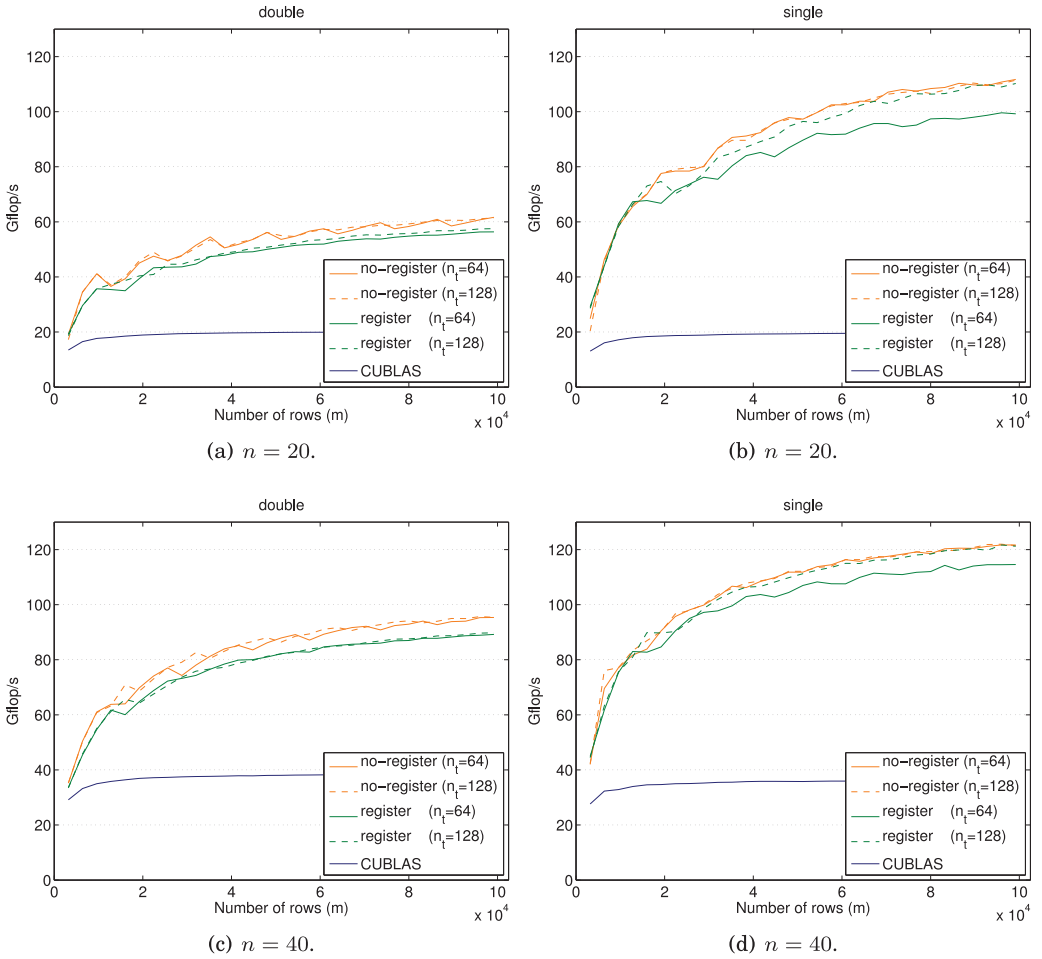
(a) $n = 20$. (b) $n = 40$.

Fig. 5. Optimal block sizes for triangular solve among the block sizes among $n_b = 1, 2, \ldots, 40$, with n_t being the number of threads in each thread block.

specialized solver if available, or calls a default solver otherwise. Finally, Figure 4(e) shows a blocked triangular solver that is based on the unblocked solver. Though the performance of our blocked kernel may be improved by blocking within the CUDA kernel, we do not investigate that implementation in this article.

Figure 5 shows the block sizes n_b that obtained the best performance of triangular solution with respect to a different number of right-hand sides on an NVIDIA Tesla K20Xm GPU, where we used the CUDA `nvcc` version 6.0 compiler with the optimization flag `-O3`. These test matrices are extremely tall-skinny, having hundreds of thousands of rows, while having tens of columns. These are the typical dimensions of the matrices that appear in block or s-step Krylov methods [Saad 2003, 2011; van Rosendale 1983; Hoemmen 2010] and were used for our previous studies [Yamazaki et al. 2014a, 2015a]. We generated two types of specialized kernel; in the first type of kernel, each GPU thread first stores the computed solution in its local registers and then copies to the output matrix after all the elements of the solution are computed, while the other kernel does not use the registers. There are tradeoffs in using the shared memory or local registers; using the shared memory or local registers may improve the data locality but reduce the GPU occupancy, and its use must be tuned for each case. We also experimented using different numbers of threads per thread block (i.e., $n_t = 64$ and 128), but the performance did not change significantly. In the figure, we see that for the 20-by-20 upper-triangular matrix, blocking is mostly not needed, while for the 40-by-40 triangular matrix, the block sizes of 20 and 14 were good choices for single and double precisions, respectively.

Figure 6 compares the corresponding optimal performance of the triangular solve (TRSM). The NVIDIA Tesla K20Xm GPU has the memory bandwidth of 249.6GB/s, while the respective peak performances in double and single precision are 1311.7 and 3935.2Gflop/s. Hence, to obtain the peak performance, we need to at least perform about 42 or 126 flops for each double or single numerical value read, respectively. For our implementation of the triangular solve, each GPU thread reads at least its right-hand side with n numerical values and performs $n(n + 1)/2$ flops, hence, on average, performing about $(n + 1)/2$ flops for each numerical value read. For instance, with a 20-by-20 upper-triangular matrix (i.e., $n = 20$), on average, each GPU thread performs about 10 flops for each value read. Hence, in theory, we could obtain about 25% and 16% of the peak performance in double and single precision (i.e., about 327.6



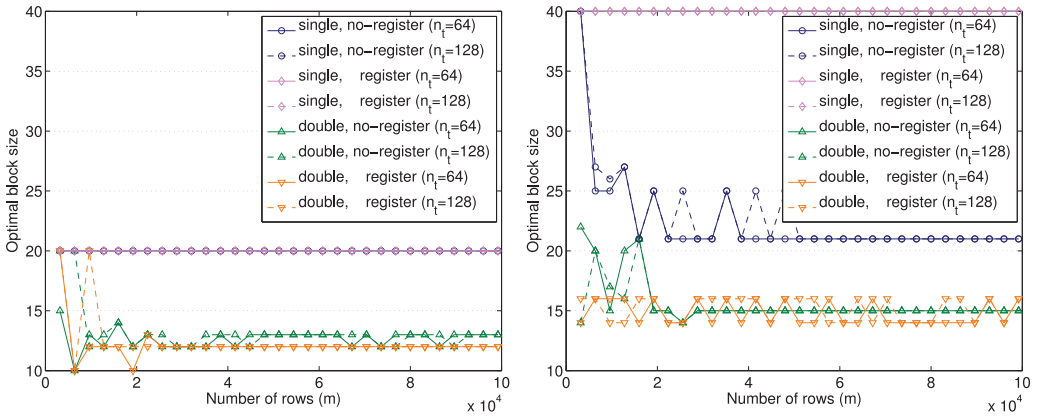
(a) $n = 20$. (b) $n = 20$.

(c) $n = 40$. (d) $n = 40$.

Fig. 6. Performance of triangular solve with R of different dimensions, n, with n_t being the number of threads in each thread block.

and 655.2Gflop/s), respectively. Though for the 20-by-20 upper-triangular matrix, we obtained only about 17.1% and 18.8% of the peak performance based on the memory bandwidth in double and single precision, respectively, and 18.6% and 29.1% for the 40-by-40 matrix, the performance of our kernel was still higher than that of CUBLAS. Also, compared to the double-precision kernel, the single-precision kernel obtained the higher performance, but the difference was smaller with the 40-by-40 matrix. We also see that for the triangular solve, the performance of the solver was degraded using the local registers.

We have also implemented a triangular matrix-matrix multiply (TRMM) in the same fashion, which we used for the inversion-based triangular solve. Figures 7 and 8 show the optimal block sizes and performance of TRMM in double and single precisions. The figure shows that while it degraded the performance of TRSM, explicitly storing the input matrix V in the local registers improved the performance of TRMM in single precision, and the performance of TRMM was higher than that of TRSM. In addition, when the registers were used to store V, blocking was not needed for both 20-by-20 and 40-by-40 matrices. The reason for this different behavior of TRSM and TRMM could



(a) $n = 20$. (b) $n = 40$.

Fig. 7. Optimal block sizes for triangular multiply among the block sizes of $n_b = 1, 2, \ldots, 40$, with n_t being the number of threads in each thread block.

be that TRMM accesses the upper-triangular matrix from the last column to the first column, while TRSM accesses it from the first column to the last column (see Figure 9(a) for the pseudocode, and Figures 9(b) and 9(c) for illustration). Compared to TRMM, a general matrix-matrix multiply (GEMM) may obtain even higher performance, taking advantage of more data parallelism. However, while TRMM implements an in-place multiplication, GEMM is based on an out-of-place multiplication, accumulating the results of the multiplication in another matrix. Hence, implementing the triangular solve using GEMM requires the m-by-n workspace. In addition, since we focus on a small dimension of R (e.g., $n = 20$), the performance of TRSM can be bounded by the memory bandwidth. As a result, in our experiments, compared with the substitution-based TRSM or the inversion-based TRSM with TRMM, TRSM based on CUBLAS GEMM was often slower due to the extra memory copy required by the out-of-place multiplication.

5. MIXED-PRECISION SINGULAR VALUE QR FACTORIZATION

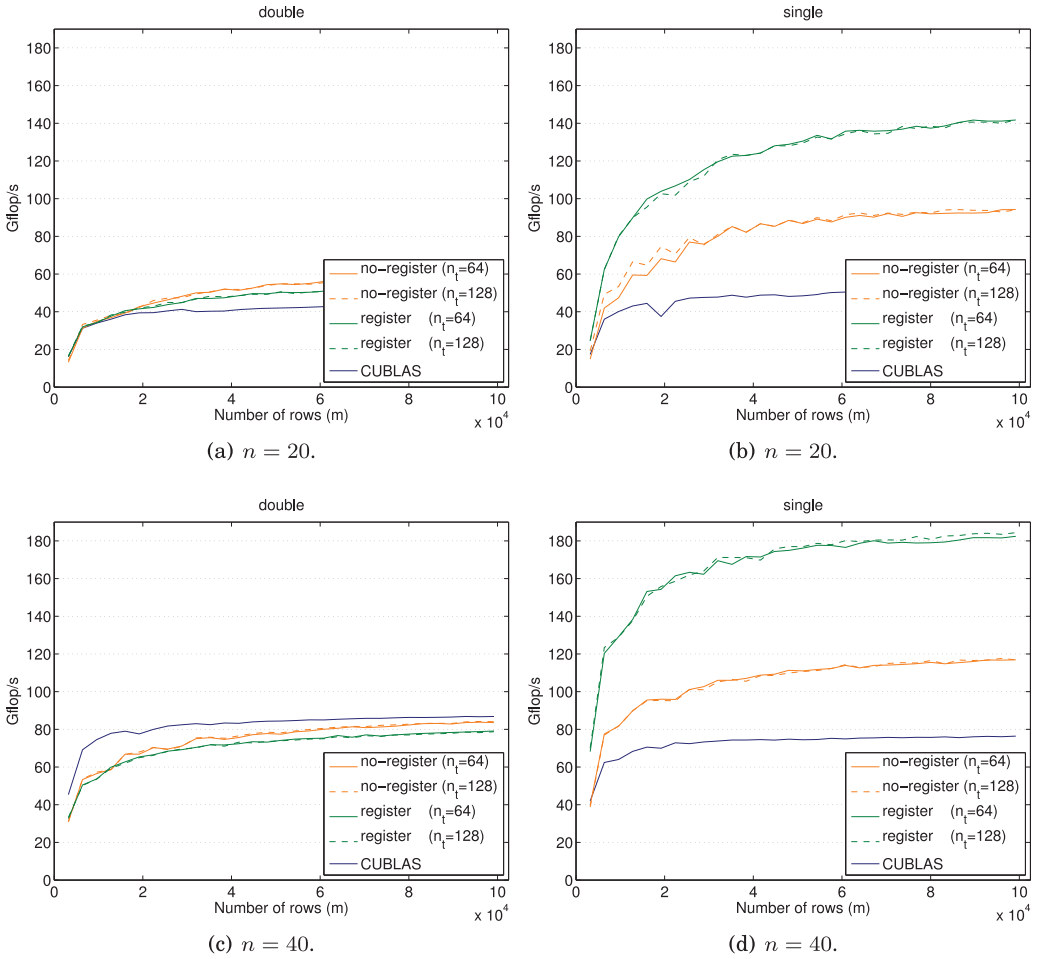
To improve the performance of SVQR, we examine the potential of using the halved precision at the third step of the factorization while using the working precision for the first two steps (the numerical error at the first two steps of SVQR depends quadratically on the condition number of the input matrix, while it depends linearly at the third step). For the rest of the article, to simplify our discussion, we focus on the 64-bit double working precision, and hence the halved precision is the 32-bit single precision. The following upper bound adapts the bound on the orthogonality error [Yamazaki et al. 2015b] to our mixed-precision SVQR.

THEOREM 5.1. *If single precision is used at the third step of SVQR to compute the orthonormal vectors $\widehat{Q}$, and ϵ_d and ϵ_s are the machine epsilons in the working double and single precision, respectively (i.e., $\epsilon_s^2 = \epsilon_d$), then we have*

$$\|I - \widehat{Q}^T \widehat{Q}\|_2 \leq O\big(\epsilon_s \kappa(V) + (\epsilon_s \kappa(V))^2\big),$$

and

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



(a) $n = 20$. (b) $n = 20$.

(c) $n = 40$. (d) $n = 40$.

Fig. 8. Performance of triangular multiply with R of different dimensions, n, with n_t being the number of threads in each thread block.

In other words, the norm of the orthogonality error is bounded by

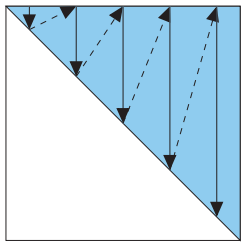
$$\begin{aligned}
&\textit{if } \kappa(V) \leq \epsilon_s^{-1} \textit{ then},\\
&\qquad \|I - \widehat{Q}^T \widehat{Q}\|_2 \leq O(\epsilon_s \kappa(V)), \textit{ and hence } \|\widehat{Q}\|_2 \leq 1 + O\big((\epsilon_s \kappa(V))^{1/2}\big);\\
&\textit{otherwise},\\
&\qquad \|I - \widehat{Q}^T \widehat{Q}\|_2 \leq O((\epsilon_s \kappa(V))^2), \textit{ and hence } \|\widehat{Q}\|_2 \leq 1 + O(\epsilon_s \kappa(V)).
\end{aligned}$$

Proof. This is the adaptation of the upper bound derived for the mixed-precision CholQR [Yamazaki et al. 2015b]. The only difference is that the upper bound of CholQR assumes that the Cholesky factorization of the Gram matrix can be computed at Step 2 (i.e., $\kappa(V)^2 < \epsilon_d$), while the upper bound of SVQR does not require this assumption.

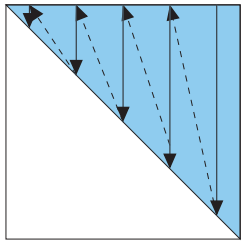
Theorem 5.1 implies that when the squared condition number of the input matrix V is greater than the reciprocal of the machine epsilon, then the mixed-precision SVQR has the same error bound as the standard SVQR. In other words, when the condition number of V is large enough, the numerical errors introduced at the first two steps

```
__shared__ FloatingPoint_t rA[N*N];
int id = threadIdx.x;
int ind = blockIdx.x * NB + threadIdx.x;

if (id < N) {
  #pragma unroll
  for (int j=0; j<N; j++) rA(id,j) = dA(id,j);
}
__syncthreads();

if (ind < m) {
  FloatingPoint_t rB[N];
  #pragma unroll
  for (int j=0; j<N; j++) rB(j) = dB(ind,j);

  #pragma unroll
  for (int j=N-1; j>=0; j--) {
    FloatingPoint_t bij = ZERO;

    for (int i=0; i<j; i++) bij += rA(i,j) * rB(i);
    rB(j) = bij + rA(j,j) * rB(j);
  }

  #pragma unroll
  for (int j=0; j<N; j++) dB(ind,j) = rB(j);
}
```

(a) Pseudocode using registers to store B.



(b) TRSM's access to R.



(c) TRMM's access to R.

Fig. 9. Data access pattern and GPU implementation of triangular matrix multiply.

```
Step 1: Form Gram matrix B
  B := V^T V in working precision

Step 2: Compute upper-triangular matrix R
  [U, Σ, U] = svd(B) and
  [W, R] := qr(√Σ U^T) in working precision

Step 3: Compute orthonormal matrix Q
  if σ_1/σ_n ≥ 1/ε_d then
    Q := V^(d) R^-1 in mixed-precision
  else
    Q := V^(d) R^-1 in working precision
  end if
```

(a) Pseudocode.

	$\|I - Q^T Q\|$	$\|V - QR\|$
d-SVQR	$O(\epsilon_d \kappa(V)^2)$	$O(\epsilon_d \kappa(V))$
ds-SVQR	$O(\epsilon_d \kappa(V)^2 + \epsilon_s \kappa(V))$	$O(\epsilon_s \kappa(V))$

(b) Error bounds.

Fig. 10. Pseudocode and error bounds of the mixed-precision SVQR.

become so large that we can use single precision at the third step without increasing the error bound. Hence, our adaptive mixed-precision SVQR uses single precision for the third step when the condition number of the Gram matrix computed at Step 2 is greater than the reciprocal of the working machine precision. Figure 10(a) shows the pseudocode of this adaptive mixed-precision SVQR. Another option could be to perform the QR factorization of $\sqrt{\Sigma}U^T$ in the mixed precision, but as the performance results in Sections 4 and 7 indicate, the time required for computing the QR factorization and copying the upper-triangular factor R to the GPU is often marginal, and hence, in this article, we focus on the adaptive scheme in Figure 10(a).

Figure 10(b) compares the error bounds of the standard and mixed-precision SVQR, where the standard SVQR is referred to as d-SVQR, and ds-SVQR is the mixed-precision SVQR. Though ds-SVQR may obtain performance improvement over

Name	m	n	V	$\kappa(V)$
$\mathcal{K}_{30}(A, \mathbf{1})$ 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	$[\mathbf{1}_n^T; diag(rand(n, 1) * \epsilon_d^3]$	7.8×10^{18}

Fig. 11. Test matrices used for numerical experiments, where $\mathcal{K}_n(A, \mathbf{v})$ is the Krylov basis vectors given the matrix A and the starting vector $\mathbf{v}$; 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.

d-SVQR while maintaining the same order of orthogonality error, the tradeoff is the greater backward error. It may be possible to selectively use single precision on the subset of the columns (e.g., based on the diagonal values of R) or to reduce the backward error based on iterative refinements or software-emulated higher precision. However, in this article, we focus on examining the accuracy and performance of ds-SVQR.

6. NUMERICAL RESULTS OF ADAPTIVE MIXED-PRECISION SVQR

In this section, we study the numerical behavior of the mixed-precision SVQR. Following Stathopoulos and Wu [2002], our SVQR implementation sets the singular values that are in the same order as or less than $O(\epsilon_d \sigma_1)$ to be $\epsilon_d \sigma_1$, where ϵ_d is the machine epsilon in the working precision and σ_1 is the largest singular value of the Gram matrix. In addition, our implementation 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 [Stathopoulos and Wu 2002]. With this implementation, the first step of SVQR could introduce a larger order of numerical error than the second step. Since our adaptive scheme is based on the condition number of the normalized input matrix, it may miss the cases where single precision can be used at the third step due to the large numerical error introduced at the first step. However, we found that scaling the Gram matrix often improves the overall numerical stability of SVQR, especially for ill-conditioned matrices.

Figures 12 through 14 show the orthogonality error and the condition number of the computed matrix Q in the working 64-bit double precision. In the figure, "f" for CholQR indicates that the Cholesky factorization failed due to the occurrence of a nonpositive diagonal, while for SVQR, "t" indicates that some of the computed singular values had their magnitudes less than $O(\epsilon_d \sigma_1)$, and "†" means that the mixed precision was used for the third step of ds-SVQR. We used the substitution-based triangular solve for these experiments. Figure 11 lists properties of our test matrices that were used to study CholQR and SVQR in Stathopoulos and Wu [2002]. In all the test cases, the orthogonality error of ds-SVQR was reduced at about the same or even faster rate than that of d-SVQR, indicating that our adaptive scheme obtains the desired numerical properties in practice. In addition, we see that as discussed in Section 2, for the ill-conditioned matrix, d-SVQR may more quickly identify the ill-conditioned subspace, requiring fewer iterations than d-CholQR.

Though the overall orthogonality error of the standard d-SVQR would be in the same order using either the substitution-based or inversion-based triangular solve, the inversion-based triangular solve increases the orthogonality error bound of the mixed-precision ds-SVQR, which uses the 32-bit single precision for the triangular solve. Figures 15 through 17 compare the orthogonality errors of d-SVQR and ds-SVQR using different implementations of the triangular solve. For the adaptive block size, we use the squared root of the computed condition number of the Gram matrix to estimate the condition number of R. We then set the numerical threshold in Equation (2) to be $\tau = \epsilon_s^{1/2}$ such that the numerical error introduced by each inversion-based triangular

Iter.	$\|I - Q^T Q\|_2$ d-CholQR	d-SVQR	ds-SVQR	$\kappa(Q)$ d-CholQR	d-SVQR	ds-SVQR
0	1.4×10^{12}	1.4×10^{12}	1.4×10^{12}	1.0×10^{19}	1.0×10^{19}	1.0×10^{19}
1	2.8×10^{8} (f)	1.0×10^{0} (t)	2.5×10^{0} (t) †	7.3×10^{17} (f)	2.1×10^{9} (t)	2.0×10^{1} (t) †
2	1.4×10^{2} (f)	1.0×10^{0} (t)	1.4×10^{-13}	3.8×10^{13} (f)	2.9×10^{1} (t)	1.0×10^{0}
3	1.0×10^{0} (f)	3.0×10^{-13}	2.3×10^{-14}	1.1×10^{7}	1.0×10^{0}	1.0×10^{0}
4	2.1×10^{-12}	2.2×10^{-14}	2.5×10^{-14}	1.0×10^{0}	1.0×10^{0}	1.0×10^{0}
5	2.0×10^{-14}	3.3×10^{-14}	1.8×10^{-14}	1.0×10^{0}	1.0×10^{0}	1.0×10^{0}

Fig. 12. Error norm $\|I - Q^T Q\|_2$ and condition number $\kappa(Q)$ for $\mathcal{K}_{30}(A, \mathbf{1})$ of 2D Laplacian matrix A.

Iter.	$\|I - Q^T Q\|_2$ d-CholQR	d-SVQR	ds-SVQR	$\kappa(Q)$ d-CholQR	d-SVQR	ds-SVQR
0	3.8×10^{0}	3.8×10^{0}	3.8×10^{0}	3.3×10^{19}	3.3×10^{19}	3.3×10^{19}
1	1.0×10^{0} (f)	1.0×10^{0} (t)	1.4×10^{0} (t) †	4.0×10^{18} (f)	2.1×10^{12} (t)	1.2×10^{3} (t) †
2	1.0×10^{0} (f)	1.0×10^{0} (t)	8.3×10^{-11}	1.4×10^{18} (f)	4.5×10^{4} (t)	1.0×10^{0}
3	1.5×10^{0} (f)	1.6×10^{-7}	1.4×10^{-14}	1.1×10^{18} (f)	1.0×10^{0}	1.0×10^{0}
4	3.8×10^{0} (f)	1.2×10^{-14}	1.0×10^{-14}	1.6×10^{18} (f)	1.0×10^{0}	1.0×10^{0}
5	4.5×10^{-4}	8.2×10^{-15}	9.5×10^{-15}	1.0×10^{0}	1.0×10^{0}	1.0×10^{0}
6	1.4×10^{-15}	8.6×10^{-15}	9.6×10^{-15}	1.0×10^{0}	1.0×10^{0}	1.0×10^{0}
7	1.0×10^{-15}	9.2×10^{-15}	9.6×10^{-15}	1.0×10^{0}	1.0×10^{0}	1.0×10^{0}

Fig. 13. Error norm $\|I - Q^T Q\|_2$ and condition number $\kappa(Q)$ for Hilbert matrix.

Iter.	$\|I - Q^T Q\|_2$ d-CholQR	d-SVQR	ds-SVQR	$\kappa(Q)$ d-CholQR	d-SVQR	ds-SVQR
0	9.9×10^{1}	9.9×10^{1}	9.9×10^{1}	2.9×10^{18}	2.9×10^{18}	2.9×10^{18}
1	1.0×10^{0} (f)	1.0×10^{0} (t)	1.0×10^{0} (t) †	2.9×10^{17} (f)	4.4×10^{10} (t)	4.4×10^{10} (t) †
2	6.5×10^{-15}	2.8×10^{-8}	1.0×10^{-13}	1.0×10^{0}	1.0×10^{0}	1.0×10^{0}
3	5.4×10^{-16}	1.6×10^{-14}	1.1×10^{-14}	1.0×10^{0}	1.0×10^{0}	1.0×10^{0}
4	4.4×10^{-16}	9.8×10^{-15}	8.9×10^{-15}	1.0×10^{0}	1.0×10^{0}	1.0×10^{0}
5	4.4×10^{-16}	9.3×10^{-15}	8.4×10^{-15}	1.0×10^{0}	1.0×10^{0}	1.0×10^{0}

Fig. 14. Error norm $\|I - Q^T Q\|_2$ and condition number $\kappa(Q)$ for synthetic matrix.

Iter.	d-substitution	d-inversion	ds-substitution	ds-inversion	ds-adaptive
1	1.0×10^{0} (t)	1.0×10^{0} (t)	2.5×10^{2} (t) †	2.4×10^{0} (t) †	1.8×10^{0} † (t)
2	1.0×10^{0} (t)	1.0×10^{0} (t)	1.4×10^{-13}	5.6×10^{-12}	3.3×10^{-13}
3	3.0×10^{-13}	1.4×10^{-12}	2.3×10^{-14}	2.1×10^{-14}	1.9×10^{-14}
4	2.2×10^{-14}	2.1×10^{-14}	2.1×10^{-14}	2.1×10^{-14}	2.1×10^{-14}
5	3.3×10^{-14}	3.3×10^{-14}	1.8×10^{-14}	1.9×10^{-14}	1.6×10^{-14}

Fig. 15. Orthogonality error norm $\|I - Q^T Q\|_2$ for $\mathcal{K}_{30}(A, \mathbf{1})$ of 2D Laplacian matrix A using standard d-SVQR or mixed-precision ds-SVQR in double precision with substitution-based or explicit and adaptive inversion-based TRSM. Initially, $\|I - Q^T Q\|_2 = 3.8 \times 10^{0}$.

Iter.	d-substitution	d-inversion	ds-substitution	ds-inversion	ds-adaptive
1	1.0×10^{0} (t)	1.0×10^{0} (t)	1.4×10^{0} (t) †	2.9×10^{0} (t) †	2.9×10^{0} (t) †
2	1.0×10^{0} (t)	1.4×10^{0} (t)	8.3×10^{-11}	1.2×10^{-10} †	7.5×10^{-10}
3	1.6×10^{-6}	2.7×10^{-7}	1.4×10^{-14}	1.3×10^{-14}	1.5×10^{-14}
4	1.2×10^{-14}	2.1×10^{-14}	1.0×10^{-14}	9.0×10^{-15}	1.1×10^{-14}
5	8.2×10^{-15}	9.9×10^{-15}	9.5×10^{-15}	9.3×10^{-15}	9.8×10^{-15}

Fig. 16. Orthogonality error norm $\|I - Q^T Q\|_2$ for Hilbert matrix using standard d-SVQR or mixed-precision ds-SVQR in double precision with substitution-based or explicit and adaptive inversion-based TRSM.

Iter.	d-substitution	d-inversion	ds-substitution	ds-inversion	ds-adaptive
1	1.0×10^{0} (t)	1.0×10^{0} (t)	1.0×10^{0} (t) †	6.3×10^{2} (t) †	1.0×10^{0} (t) †
2	2.8×10^{-8}	2.8×10^{-8}	1.0×10^{-13}	4.2×10^{1} (t) †	1.5×10^{-13}
3	1.6×10^{-14}	1.5×10^{-14}	1.8×10^{-6}	1.8×10^{-6}	9.0×10^{-15}
4	9.8×10^{-15}	9.5×10^{-15}	8.9×10^{-15}	2.3×10^{-14}	9.2×10^{-15}

Fig. 17. Orthogonality error norm $\|I - Q^T Q\|_2$ for synthetic matrix using standard d-SVQR or mixed-precision ds-SVQR in double precision with substitution-based or explicit and adaptive inversion-based TRSM. Initially, $\|I - Q^T Q\|_2 = 9.9 \times 10^{1}$.

Matrix	d-sub	d-inv	ds-sub	ds-inv	ds-adapt
$\mathcal{K}_{30}(A, \mathbf{1})$	2.5×10^{-10}	3.4×10^{-3}	7.2×10^{-2}	1.5×10^{4}	6.5×10^{-2}
Hilbert	1.2×10^{-16}	2.4×10^{-11}	9.1×10^{-8}	2.6×10^{-3}	1.5×10^{-7}
Synthetic	3.2×10^{-15}	3.5×10^{-15}	1.2×10^{-13}	3.7×10^{-6}	1.2×10^{-13}

Fig. 18. Backward error norm $\|V - QR\|_2$ using d-SVQR or ds-SVQR.

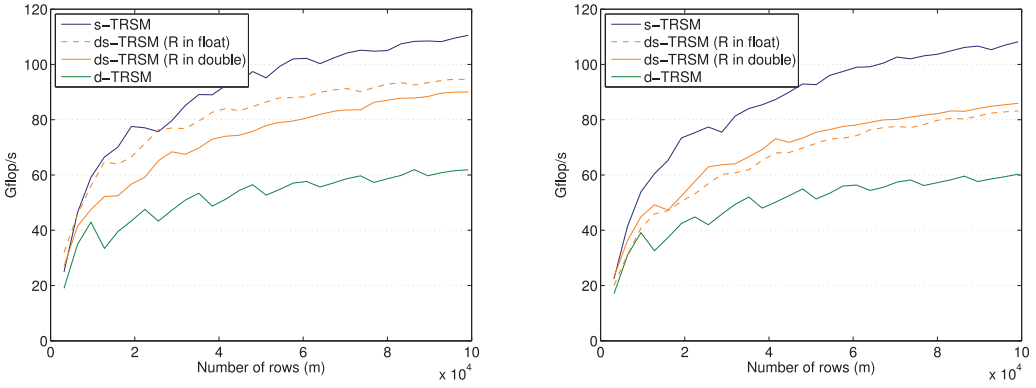
solve with the diagonal block in single precision is in the same order as that of the inversion-based triangular solve in double precision (i.e., $\epsilon_s \kappa(R^{(i,i)})^2 \leq \epsilon_d \kappa(R)^2$). As expected, the standard SVQR obtained similar error norms using either the substitution-based or the inversion-based triangular solve. Though the orthogonality errors of the mixed-precision SVQR increased using the inversion-based triangular solve in some cases (e.g., `synthetic` matrix), by adaptively adjusting the diagonal block sizes, the numerical error stayed at the same order as that using the substitution-based triangular solve. We have also tried computing the inverse of R in double precision and then performing the matrix multiplication in single precision, but this was not as stable as the inverse-based triangular solve with the adaptive block size.

These different implementations of the triangular solve lead to different orders of backward errors in the computed QR factorization, $\|V - QR\|/\|V\|$, which are shown in Figure 18. In particular, since single precision is used for the third step, the backward error of the mixed-precision SVQR is significantly greater than that of the standard SVQR. We are studying the effects of the backward errors on the solution convergence of several subspace projection solvers.

7. PERFORMANCE OF MIXED-PRECISION SVQR WITH A GPU

We now study the performance of the mixed-precision triangular solve ds-TRSM, where the input and output matrices are in double precision, but the arithmetic operations are internally performed in single precision. To reduce the data movement through the memory hierarchy of the GPU, our implementation of ds-TRSM first reads the upper-triangular matrix into the shared memory in single precision, and then internally computes its solutions in single precision to improve its performance. However, if the performance of ds-TRSM is bounded by the memory bandwidth, its performance can be lower than that of the standard d-TRSM. This is because both the input and output matrices are stored in the working double precision, and even if we only read the single precision of the input matrix through the local memory hierarchy, we suffer from the noncontiguous memory access. In addition, we incur the overhead of type-casting the matrices to single precision. Hence, the mixed-precision kernel obtains most of its performance gain not from the reduction in the intra-GPU communication volume, but mainly from the higher-peak performance of the lower-precision arithmetic (e.g., the peak performance of an NVIDIA K20Xm GPU is 3,935.2 and 1,311.7Gflop/s in single and double precisions, respectively).

Figure 19(a) shows the performance of ds-TRSM for the 20-by-20 upper-triangular matrix without blocking. We see that for the mixed-precision solve, explicitly storing the



(a) without cast/copy of R. (b) with cast/copy of R.

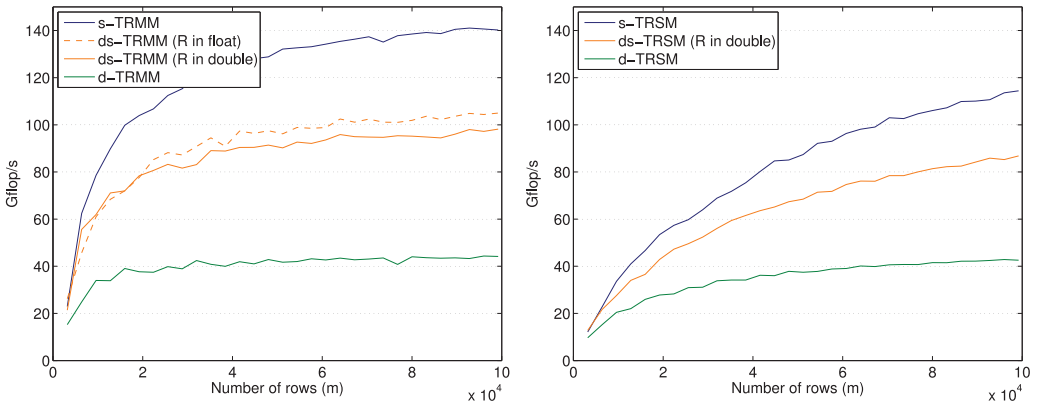
Fig. 19. Performance of mixed-precision triangular solve, including the time required for casting R on the CPU and copying the matrix to the GPU, $n = 20$, $n_b = 20$, $n_t = 64$.

solution vector in single precision in the register, which reduces the cost of type-casting, improved the performance. By internally using single precision, ds-TRSM obtained speedups of up to 1.56 and an average speedup of 1.41 over the standard d-TRSM, demonstrating that ds-TRSM can obtain a substantial speedup over the standard algorithm when there are enough columns in the matrix. We also see that the single-precision s-TRSM obtained speedups of up to 1.99 over d-TRSM.

In Figure 19(a), we also show the performance of the mixed-precision ds-TRSM when the upper-triangular matrix R is stored in single precision (e.g., the QR factorization of $\sqrt{\Sigma}U^T$ is computed in single precision). This may improve the performance by avoiding the type-cast on the GPU and halving the amount of data copied to the GPU. However, Figure 19(b) shows that at least on our particular testbed, this approach did not improve the performance of ds-TRSM. The performance difference between ds-TRSM and s-TRSM is due to the cost of type-casting and reading the right-hand sides in double precision. Finally, instead of d-TRSM at the third step of the standard d-SVQR, a mixed-precision iterative refinement with s-TRSM may be an option [Buttari et al. 2007]. However, the residual norms must be computed in double precision using d-TRMM, and by looking at Figures 6 and 8, on our particular testbed, one substitution with d-TRSM may be the preferred choice over the mixed-precision iterative refinements. Similarly, using an iterative refinement for reducing the backward error of the mixed-precision SVQR is likely to be more expensive than the standard triangular solve of the standard SVQR.

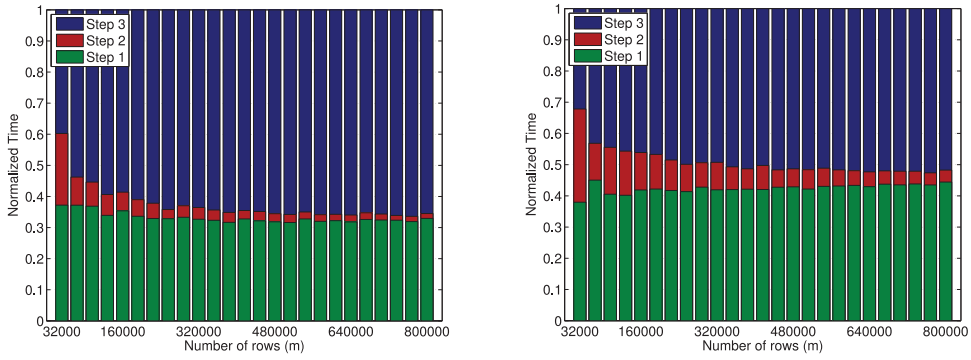
Figure 20(a) shows the performance of the mixed-precision multiply ds-TRMM. Like ds-TRSM, ds-TRMM's input and output matrices are in double precision, but the internal arithmetics are performed in single precision. ds-TRMM obtained speedups of up to 2.33 over the standard d-TRMM. Then, Figure 20(b) shows the performance of the inversion-based triangular solve. For the mixed-precision ds-TRSM, we computed the inverse of R in double precision, and then used the mixed-precision ds-TRMM that takes both input matrices R and V in double precision. We see that though the performance of the mixed-precision ds-TRMM was higher than that of the substitution-based ds-TRSM, due to the overhead of computing the matrix inverse, the inversion-based ds-TRSM was slower than the substitution-based ds-TRSM for this small size of R (i.e., $n = 20$).

Finally, in Figures 21 and 22, we compare the performance of the standard d-SVQR and mixed-precision ds-SVQR on two 6-core Intel Xeon CPU with an NVIDIA Tesla



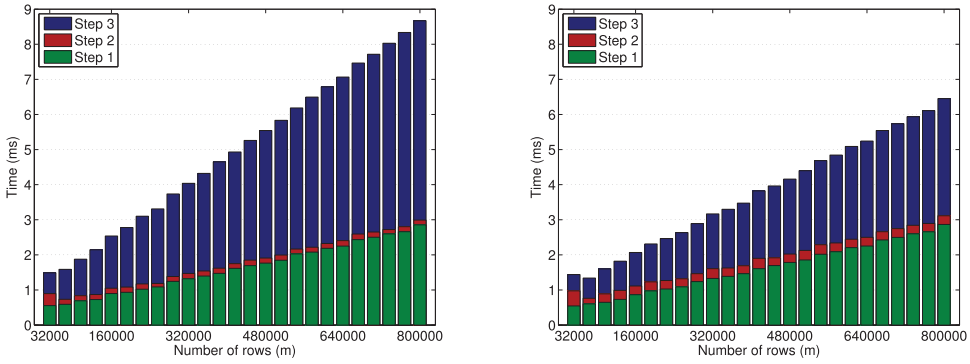
(a) TRMM performance. (b) Inversion-based TRSM.

Fig. 20. Performance of mixed-precision triangular multiply, $n = 20$, $n_b = 20$, $n_t = 128$.



(a) d-SVQR. (b) ds-SVQR.

Fig. 21. Normalized time of standard and mixed-precision SVQR ($n = 20$).



(a) d-SVQR. (b) ds-SVQR.

Fig. 22. Performance of standard and mixed-precision SVQR ($n = 20$).

K20Xm GPU. Our mixed-precision SVQR uses the batched symmetric matrix multiplication (d-SYRK) developed in Yamazaki et al. [2015b] and the Cholesky factorization routine of threaded MKL version xe2013.1.046 (d-POTRF) for the first two steps of the mixed-precision SVQR, while the substitution-based mixed-precision triangular solve (ds-TRSM) is used for the third step. First, Figure 21 shows the breakdown of the factorization time for orthogonalizing 20 vectors of different lengths (i.e., $n = 20$ while $m = 32{,}000 - 80{,}000$). For this particular case with long enough vectors, the standard d-SVQR spends about 65% of the factorization time in the triangular solve. Now, Figure 22 compares the performance of d-SVQR with the mixed-precision ds-SVQR for orthogonalizing the 20 vectors. By internally using single precision for the triangular solve, the mixed-precision SVQR obtained speedups of up to 1.36 and an average speedup of 1.28.

8. CONCLUSION

To orthogonalize a set of dense column vectors, we studied the numerical stability and performance of different Singular Value QR (SVQR) implementations. We focused on the triangular solver, which performs about half of the total floating-point operations of the orthogonalization process. Though we have integrated several optimization techniques into our GPU kernels, to reduce their communication overheads and increase the benefit of using the mixed-precision kernels, we are looking to further tune their performance (e.g., using BEAST[3]). All the BLAS kernels developed for this study will be released through the MAGMA library.[4] We are also working on the implementation of a communication-avoiding variant of Householder QR factorization (CAQR) [Demmel et al. 2012] on a GPU. Our implementation is based on a batched QR and GEMM, while our CholQR implementation was based on a batched SYRK and TRSM. Though performing twice more flops, we are investigating how closely CAQR would perform compared to CholQR on a GPU, especially for the tall-skinny matrix when the performance can be limited by the memory bandwidth of the GPU.

As a part of the studies, we examined a mixed-precision variant that uses the halved precision for the triangular solve. Our performance results on multicore CPUs with a GPU illustrated that though the intra-GPU communication volume may not be reduced, the mixed-precision variant can improve the performance by taking advantage of the higher peak performance of the lower-precision arithmetic. Then, we described an adaptive mixed-precision scheme to decide if the lower precision can be used for the triangular solve at runtime. Compared to the standard SVQR, this adaptive mixed-precision SVQR maintains the same order of the orthogonality error, but significantly increases its backward error. We are currently studying several techniques to reduce the backward errors, but on the current GPU architecture, the overhead of reducing the errors tends be greater than the performance improvement gained using the single precision. We are also studying the effects of the backward errors on various linear and eigenvalue solvers.

REFERENCES

C. Bischof. 1990. Incremental condition estimation. *SIAM J. Matrix Anal. Appl.* 11 (1990), 312–322.

A. Buttari, J. Dongarra, J. Langou, J. Langou, P. Luszczek, and J. Kurzak. 2007. Mixed precision iterative refinement techniques for the solution of dense linear systems. *Int. J. High Perform. Comput. Appl.* 21, 4 (2007), 457–466.

J. Demmel, L. Grigori, M. Hoemmen, and J. Langou. 2012. Communication-optimal parallel and sequential QR and LU factorizations. *SIAM J. Sci. Comput.* 34, 1 (2012), A206–A239.

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

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

G. Golub and C. van Loan. 2012. *Matrix Computations* (4th ed.). Johns Hopkins University Press, Baltimore, MD.

M. Hoemmen. 2010. *Communication-Avoiding Krylov Subspace Methods*. Ph.D. Dissertation. EECS Department, University of California, Berkeley.

J. Kurzak, S. Tomov, and J. Dongarra. 2011. *Autotuning GEMMs for Fermi*. Technical Report UT-CS-11-671. Computer Science Department, University of Tennessee.

Y. Saad. 2003. *Iterative Methods for Sparse Linear Systems* (3rd ed.). Society for Industrial and Applied Mathematics, Philadelphia, PA.

Y. Saad. 2011. *Numerical Methods for Large Eigenvalue Problems* (2nd ed.). Society for Industrial and Applied Mathematics, Philadelphia, PA.

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

J. van Rosendale. 1983. Minimizing inner product data dependence in conjugate gradient iteration. In *Proceedings of the IEEE International Conference on Parallel Processing*.

I. Yamazaki, H. Anzt, S. Tomov, M. Hoemmen, and J. Dongarra. 2014a. Improving the performance of CA-GMRES on multicores with multiple GPUs. In *Proceedings of the IEEE International Parallel and Distributed Symposium (IPDPS'14)*. 382–391.

I. Yamazaki, J. Kurzak, P. Luszczek, and J. Dongarra. 2015a. Randomized algorithms to update partial singular value decomposition on a hybrid CPU/GPU cluster. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis (SC'15)*. 345–354.

I. Yamazaki, S. Rajamanickam, E. Boman, M. Hoemmen, M. Heroux, and S. Tomov. 2014b. Domain decomposition preconditioners for communication-avoiding Krylov methods on a hybrid CPU/GPU cluster. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis (SC'14)*. 933–944.

I. Yamazaki, S. Tomov, and J. Dongarra. 2015b. Mixed-precision Cholesky QR factorization and its case studies on multicore CPUs with multiple GPUs. *SIAM J. Sci. Comput.* 37 (2015), C307–330.

Received February 2015; revised October 2015; accepted February 2016