Improving the Performance of CA-GMRES on Multicores with Multiple GPUs

Ichitaro Yamazaki*, Hartwig Anzt*, Stanimire Tomov*, Mark Hoemmen§, and Jack Dongarra*
*University of Tennessee, Knoxville, USA
§Sandia National Laboratory, New Mexico, USA

iyamazak@eecs.utk.edu, hanzt@icl.utk.edu, tomov@eecs.utk.edu, mhoemme@sandia.gov, and dongarra@eecs.utk.edu

***Abstract*—The Generalized Minimum Residual (GMRES) method is one of the most widely-used iterative methods for solving nonsymmetric linear systems of equations. In recent years, techniques to avoid communication in GMRES have gained attention because in comparison to floating-point operations, communication is becoming increasingly expensive on modern computers. Since graphics processing units (GPUs) are now becoming crucial component in computing, we investigate the effectiveness of these techniques on multicore CPUs with multiple GPUs. While we present the detailed performance studies of a matrix powers kernel on multiple GPUs, we particularly focus on orthogonalization strategies that have a great impact on both the numerical stability and performance of GMRES, especially as the matrix becomes sparser or ill-conditioned. We present the experimental results on two eight-core Intel Sandy Bridge CPUs with three NDIVIA Fermi GPUs and demonstrate that significant speedups can be obtained by avoiding communication, either on a GPU or between the GPUs. As part of our study, we investigate several optimization techniques for the GPU kernels that can also be used in other iterative solvers besides GMRES. Hence, our studies not only emphasize the importance of avoiding communication on GPUs, but they also provide insight about the effects of these optimization techniques on the performance of the sparse solvers, and may have greater impact beyond GMRES.**

I. Introduction

Many scientific and engineering simulations require solving sparse linear systems of equations. A direct method provides a numerically stable way to solve such a linear system with a predictable number of floating point operations (flops). However, for a large-scale linear system, the storage and/or computational costs of a direct factorization may be unfeasibly expensive. A parallel computer with a large aggregated memory and a high computing capacity may provide a remedy to this large cost of direct factorization, but the per-CPU memory requirement or the factorization time of a parallel direct solver may not scale due to the extensive amount of communication or the associated memory overhead for the message buffers. As a result, an iterative method may become more attractive or could be the only feasible alternative. Among the most widely-used iterative methods are *Krylov subspace methods* [1], [2], because of their smooth and well-studied convergence behaviors, and the Generalized Minimum Residual (GMRES) method [3] is one of the popular methods for a nonsymmetric linear system and converges with monotonically non-increasing residual norms.

On modern computers, in comparison to flops, communication is becoming increasingly expensive in terms of both required cycle time and energy consumption. To address this challenge, in recent years, several techniques to avoid communication in various algorithms, including GMRES [4], have gained attention. While graphics processing units (GPUs) have become crucial components in scientific and engineering computing, the same challenge exists on the GPUs, where the gap between the arithmetic and communication costs is growing. In this paper, we study the potential of using such communication-avoiding techniques for GMRES on multicore CPUs with multiple GPUs, providing the detailed performance studies of both a matrix powers kernel and several orthogonalization procedures that often dominate the GMRES iteration time. As part of our studies, we investigate several optimization techniques for the GPU kernels that are required for GMRES. Since these kernels are also needed for other sparse solvers, the current studies not only emphasize the importance of avoiding communication both on a GPU and between the GPUs, but they also provide insights on the effects of these optimization techniques on the performance of a sparse solver.

The rest of the paper is organized as follows: in Section II, we first survey related work. Then in Section III, we review Communication-Avoiding GMRES (CA-GMRES) and provide a high-level description of our implementation on multicore CPUs with multiple GPUs. Next, in Sections IV and V, we describe our implementations of the matrix powers kernel and of various orthogonalization procedures, and demonstrate their performance. Finally, in Section VI, we study the performance of CA-GMRES with multiple GPUs, and in Section VII, we provide final remarks. Throughout this paper, the i-th row and the j-th column of a matrix V are denoted by $\mathbf{v}_{i,:}$ and $\mathbf{v}_{:,j}$, respectively, while $V_{j:k}$ is the submatrix consisting of the j-th through the k-th columns of V, and $V(\mathbf{i},\mathbf{j})$ is the submatrix consisting of the rows and columns of V that are given by the row and column index sets $\mathbf{i}$ and $\mathbf{j}$, respectively. All of our experiments were conducted on a single compute node of the Keeneland system[1] at the Georgia Institute of Technology. It consists of two eight-core Intel Sandy Bridge (Xeon E5) CPUs and three NVIDIA M2090 GPUs.

[1]https://www.xsede.org/gatech-keeneland

$\widehat{\mathbf{x}} := \mathbf{0}$ and $\mathbf{v}_{:,1} := \mathbf{b}/\|\mathbf{b}\|_2$.
repeat `(restart-loop)`
  `Projection Subspace Generation (inner-loop):`
  for $j = 1, 2, \ldots, m$ **do**
    SpMV: Generate a new vector $\mathbf{v}_{:,j+1} := A\mathbf{v}_{:,j}$.
    Orth: Orthogonalize $\mathbf{v}_{:,j+1}$ against $\mathbf{v}_{:,1}, \mathbf{v}_{:,2}, \ldots, \mathbf{v}_{:,j}$.
  end for
  `Projected Subsystem Solution (restart):`
  Compute the solution $\widehat{\mathbf{x}}$ in the generated subspace,
    which minimizes its residual norm.
  Set $\mathbf{v}_{:,1} := \mathbf{r}/\|\mathbf{r}\|_2$, where $\mathbf{r} := \mathbf{b} - A\widehat{\mathbf{x}}$.
until solution convergence.

Fig. 1. Pseudocode of GMRES(m).

$\widehat{\mathbf{x}} := \mathbf{0}$ and $\mathbf{v}_{:,1} := \mathbf{b}/\|\mathbf{b}\|_2$.
repeat `(restart-loop)`
  `Projection Subspace Generation (inner-loop):`
  for $j = 1, s+1, 2s+1, \ldots, m$ **do**
    MPK: Generate new vectors $\mathbf{v}_{:,k+1} := A\mathbf{v}_{:,k}$
      for $k = j, j+1, \ldots, \min(j+s, m)$.
    BOrth: Orthogonalize $V_{j+1:j+s+1}$ against $V_{1:j}$.
    TSQR: Orthogonalize the vectors within $V_{j+1:j+s+1}$.
  end for
  `Projected Subsystem Solution (restart):`
  Compute the solution $\widehat{\mathbf{x}}$ in the generated subspace,
    which minimizes its residual norm.
  Set $\mathbf{v}_{:,1} := \mathbf{r}/\|\mathbf{r}\|_2$, where $\mathbf{r} := \mathbf{b} - A\widehat{\mathbf{x}}$.
until solution convergence.

Fig. 2. Pseudocode of CA-GMRES(s, m).

II. Related Work

Chapter 3 of Hoemmen's PhD dissertation [4] describes CA-GMRES in detail. Chapter 7 explains the importance of picking right basis in the matrix powers kernel (*MPK*). Chapter 2 gives an extensive overview of the computational kernels that CA-GMRES uses, including *MPK* with or without preconditioning, the tall-skinny QR (TSQR) factorization with an emphasis on the communication-avoiding QR (CAQR), and its associated block orthogonalization (*BOrth*). The work includes performance models that show how CAQR and *MPK* reduce the number of memory movements and the parallel communication latency. Demmel et al. [5] refine these models for general dense QR factorization algorithms.

Mohiyuddin et al. [6] provide shared-memory parallel performance results for CA-GMRES on a single compute node of multicore CPUs. The authors show the importance of the orthogonalization step for good CA-GMRES performance, while a similar study is conducted for a CA-Lanczos on a distributed-memory system in [7]. Anderson et al. [8] implement a version of a CAQR (used as the panel factorization of a general QR factorization) on a single GPU. They then apply it to solve optimization problems for image processing. Hoemmen [9] describes a hybrid-parallel (MPI+threads) implementation of CAQR and block Gram-Schmidt orthogonalization in the Trilinos software framework. In his work, he compares performance of CAQR and block Gram-Schmidt as a combined orthogonalization procedure, against both the classical and modified Gram-Schmidt procedures. Finally, Demmel et al. [10] recently describe a communication-avoiding rank-revealing QR factorization with column pivoting.

There are several different GMRES implementations on GPUs. CUSP [11] and PARALUTION [12] target a single NVIDIA GPU using CUDA, while ViennaCL [13] provides a more platform-independent support for a single GPU using OpenCL. PETSc [14] and Trilinos [15] have growing supports for multiple GPUs based on NDIVIA CUBLAS and CUSPARSE [16], in particular for iterative solvers.

III. Communication-Avoiding GMRES

Figure 1 shows the pseudocode of a standard GMRES for computing an approximate solution $\widehat{\mathbf{x}}$ to a linear system of equations $Ax = b$. The j-th GMRES iteration first generates a new Krylov basis vector $\mathbf{v}_{:,j+1}$ through a sparse matrix-vector product (*SpMV*), followed by the orthonormalization (*Orth*) of $\mathbf{v}_{:,j+1}$ against the previously-generated orthonormal basis vectors $\mathbf{v}_{:,1}, \mathbf{v}_{:,2}, \ldots, \mathbf{v}_{:,j}$. To reduce both the computational and storage requirements of computing a large projection subspace, the iteration is restarted after computing a fixed number $m + 1$ of basis vectors. Before restart, the approximate solution $\widehat{\mathbf{x}}$ is updated by solving a least-squares problem $\mathbf{y} := \arg\min_{\mathbf{z}} \|\mathbf{c} - H\mathbf{z}\|$, where $\mathbf{c} := V_{1:m+1}^T\mathbf{r}$, $H := V_{1:m+1}^T A V_{1:m}$, and $\widehat{\mathbf{x}} := \widehat{\mathbf{x}} + V_{1:m}\mathbf{y}$. The matrix H, obtained as a by-product of the orthogonalization procedure (see Section V), is in an upper Hessenberg form. Hence, the least-squares problem can be efficiently solved, requiring only about $3(m+1)^2$ flops, while for an n-by-n matrix A with $nnz(A)$ nonzeros, *SpMV* and *Orth* require a total of about $2m{\cdot}nnz(A)$ and $2m^2n$ flops over the m iterations, respectively (i.e., $n \gg m$).

Both *SpMV* and *Orth* require communication. This includes point-to-point messages or neighborhood collectives for *SpMV*, and global all-reduces in *Orth*, as well as data movements between the levels of the local memory hierarchy (for reading the sparse matrix and for reading and writing vectors, assuming that they are not small enough to fit in cache). CA-GMRES (see Figure 2 for its pseudocode) aims to reduce this communication. It does so by redesigning the algorithm to replace *SpMV* and *Orth* with three new kernels – *MPK*, *BOrth*, and *TSQR* – that generate and orthogonalize a set of s basis vectors all at once. In theory, this communicates no more than a single GMRES iteration (plus a lower-order term), but accomplishes the work of s iterations. In Sections IV and V, we discuss these three computational kernels in detail.

To utilize GPUs, we distribute the matrix A and the basis vectors $\mathbf{v}_{:,1}, \mathbf{v}_{:,2}, \ldots, \mathbf{v}_{:,m+1}$ in a block row format (in Section IV, we discuss the distribution of A in more detail). We then generate the basis vectors on the GPUs, while the least-squares problem is solved on the CPUs. While our objective of this paper is to compare the performance of CA-GMRES with that of GMRES on the GPUs, Figure 3 compares the performance of GMRES on the GPUs with that of our CPU implementation of GMRES that uses a threaded version of

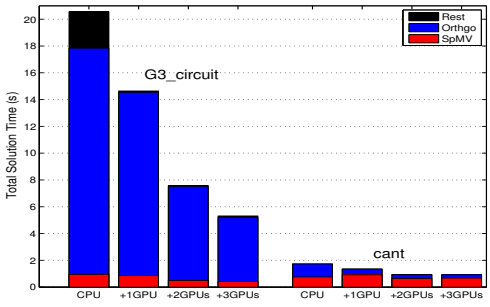


Fig. 3. Performance of GMRES on 16-core Sandy Bridge CPUs with up to three NDIVIA M2090 GPUs. CPU code is linked to MKL 2011_sp1.8.273, and *SpMV* uses the CSR or ELLPACK format on CPU or GPUs, respectively.

MKL for *SpMV* and *Orth*. Clearly, this may not be a fair comparison since MKL may not be optimized for the matrices arising from GMRES. However, the figure provides a reference point to our GMRES performance on the GPUs.

IV. Matrix Powers Kernel

For *SpMV* on multiple GPUs, the communication of the distributed vector elements through the PCI Express bus could become a bottleneck. To reduce this bottleneck, given a starting vector $\mathbf{v}_{:,j}$, *MPK* communicates all the required vector elements at once so that each GPU can independently compute the local components of the s matrix-vector products $A\mathbf{v}_{:,j}, A^2\mathbf{v}_{:,j}, \ldots, A^s\mathbf{v}_{:,j}$ without further communication. Here, in Section IV-A, we first describe our *MPK* implementation on multiple GPUs, and then in Section IV-B, we measure its performance for different test matrices. For our discussion, we use $A^{(d)}$ and $V^{(d)}$ to denote the local matrices on the d-th GPU, while n_g is the number of available GPUs.

A. Implementation

Figure 4 shows the *MPK* pseudocode. Here, $\mathbf{v}^{(d,k)} = \mathbf{v}_{\mathbf{i}^{(d,k)},k}$ and $\mathbf{i}^{(d,k)}$ is the index set of the rows of the k-th vector $\mathbf{v}_{:,k}$, which are required to compute $\mathbf{v}^{(d)}_{:,s+1}$ (for $k = 1, 2, \ldots, s$). The row index set $\mathbf{i}^{(d,k)}$ is the union of two disjoint sets $\mathbf{i}^{(d,k+1)}$ and $\boldsymbol{\delta}^{(d,k)}$, where $\mathbf{i}^{(d,s+1)}$ is the row index set of the d-th local submatrix $\big($i.e., $A^{(d)} = A(\mathbf{i}^{(d,s+1)},:)\big)$, and $\boldsymbol{\delta}^{(d,k)}$ contains the remaining row indexes in $\mathbf{i}^{(d,k)}$. In the adjacency graph of A, the set $\mathbf{i}^{(d,k)}$ is the set of the vertices that are reachable through at most $s-k+1$ edges from a vertex in $\mathbf{i}^{(d,s+1)}$, and $\boldsymbol{\delta}^{(d,k)}$ is the set of the vertices whose shortest path from a vertex in $\mathbf{i}^{(d,s+1)}$ is of length $s-k+1$ (see Figure 5 for an illustration). In our implementation, before the iteration begins, the k-th boundary set $\boldsymbol{\delta}^{(d,k)}$ is computed on the CPU based on the following recursion for $k = s, s-1, \ldots, 1$:

$$\boldsymbol{\delta}^{(d,k)} := \bigcup_{i \in \mathbf{i}^{(d,k+1)}} str\left(\mathbf{a}^{(d,k+1)}_{i,:}\right) \setminus \mathbf{i}^{(d,k+1)},$$

where $str(\mathbf{a}^{(d,k+1)}_{i,:})$ is the column index set of the nonzeros in the i-th row of the local submatrix $A^{(d,k+1)}$ which is

```
Setup: exchange elements of v_{:,1} to form v^{(d,1)}
for d = 1, 2, ..., n_g do
    compress elements of v^{(d)}_{:,1} needed by other GPUs into w^{(d)}
    asynchronously send w^{(d)} to CPU
end for
for d = 1, 2, ..., n_g do
    expand w^{(d)} into a full vector w on CPU
end for
for d = 1, 2, ..., n_g do
    compress elements of w required by d-th GPU into w^{(d)}
    asynchronously send w^{(d)} to d-th GPU
    copy the local vector v^{(d)}_{:,1} into z^{(d,1)}_{i^{(d,1)},:}
    expand w^{(d)} into a full vector z^{(d,1)}
end for

Matrix Powers: generate v^{(d)}_{:,2}, v^{(d)}_{:,3}, ..., v^{(d)}_{:,s+1}
for k = 1, 2, ..., s do
    for d = 1, 2, ..., n_g do
        SpMV: compute y^{(d)} := A^{(d,k)} z^{(d,k%2)}
        expand y^{(d)} into a full vector z^{(d,(k+1)%2)}
        copy the local part y^{(d)}_{i^{(d,1)}} of y^{(d)} into v^{(d)}_{:,k+1}
    end for
end for
```

Notations used for *MPK*:

$A^{(d)}$, $\mathbf{v}^{(d)}$	: local matrix/vector on d-th GPU
$\mathbf{i}^{(d,s+1)}$	: row index set of $A^{(d)}$, i.e., $A^{(d)} = A(\mathbf{i}^{(d,s+1)},:)$
$\mathbf{i}^{(d,k)}$	: row index set of $\mathbf{v}_{:,k}$ needed for *MPK*, i.e., $\mathbf{i}^{(d,k+1)} \bigcup \boldsymbol{\delta}^{(d,k)}$
$\boldsymbol{\delta}^{(d,k:s)}$	: k-th boundary set, i.e., $\bigcup_{k \le \ell \le s} \boldsymbol{\delta}^{(d,\ell)}$ or $\mathbf{i}^{(d,k)} \setminus \mathbf{i}^{(d,s+1)}$
$\mathbf{v}^{(d,k)}$	: rows of $\mathbf{v}_{:,k}$ required by *MPK*, i.e., $\mathbf{v}_{\mathbf{i}^{(d,k)},k}$

Fig. 4. Pseudocode of Matrix Powers Kernel, *MPK*(s, $\mathbf{v}_{:,1}$).

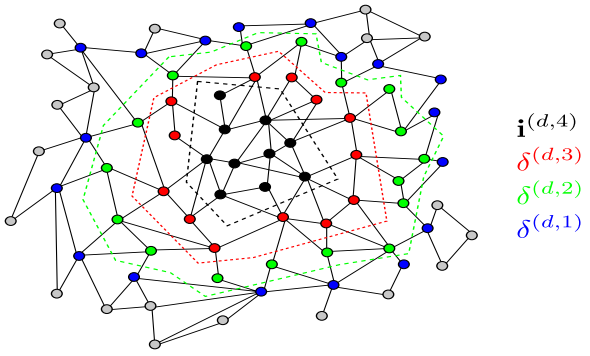
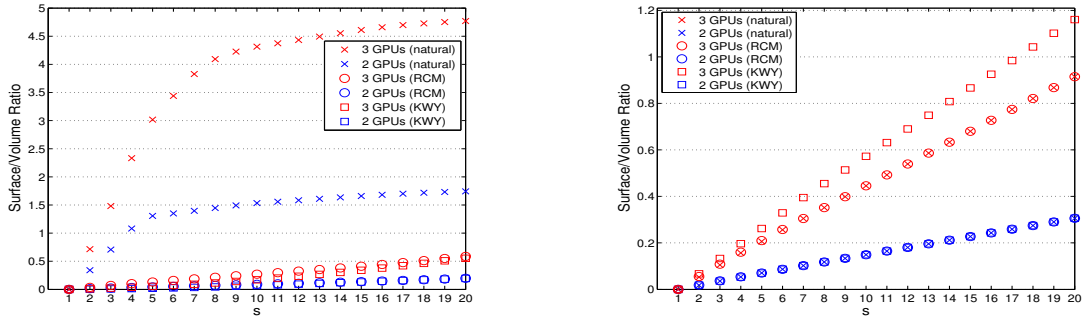


Fig. 5. Illustration of Surface-to-Volume Dependencies.

the submatrix of A consisting of the rows given by $\mathbf{i}^{(d,k+1)}$ $\big($i.e., $A^{(d,k+1)} = A(\mathbf{i}^{(d,k+1)},:)\big)$.

MPK trades additional storage and computation, and potentially greater communication volume, for communication latency. It reduces the number of communication phases between GPUs by a factor of s, but the d-th GPU requires additional memory to store the boundary submatrix $A(\boldsymbol{\delta}^{(d,1:s)},:)$, where $\boldsymbol{\delta}^{(d,k:s)} = \mathbf{i}^{(d,k)} \setminus \mathbf{i}^{(d,s+1)}$. Furthermore, at the k-th step of *MPK*, in addition to multiplying a vector with the local submatrix $A^{(d)}$, the d-th GPU must compute a multiplication



(a) `G3_circuit` matrix. (b) `cant` matrix.

Fig. 6. Surface-to-Volume Ratio in Matrix Powers Kernel.

with the k-th boundary submatrix $A(\boldsymbol{\delta}^{(d,k:s)},:)$. Finally, to generate the m basis vectors over the GMRES restart-loop, the d-th GPU must gather the total of $O(\frac{m}{s}|\boldsymbol{\delta}^{(d,1:s)}|)$ vector elements, where $|\boldsymbol{\delta}^{(d,1:s)}|$ is the size of the index set $\boldsymbol{\delta}^{(d,1:s)}$. For $s > 1$, this total communication volume could be greater than that required by *SpMV*. The amount of these storage, computational, and communication overheads depend on the sparsity structure of the matrix A. We study these in Section IV-B for different matrices.

Generating the monomial basis vectors based on the above *MPK* is often numerically unstable, leading to a stochastic convergence of the CA-GMRES iterations. This is because the generated basis vectors converge to the eigenvector corresponding to the most dominant eigenvalue of A with the ratio $|\lambda_2/\lambda_1|$, where λ_1 and λ_2 are the dominant and the second dominant eigenvalues of A, respectively. Hence, the condition number of the monomial basis $V_{1:k+1}$ increases exponentially. To avoid this numerical instability, our *MPK* can generate a Newton basis $\mathbf{v}_{:,k+1} = (A - \theta_k I)\mathbf{v}_{:,k}$, where the k-th shift θ_k is an eigenvalue of the Hessenberg matrix H from the first restart and approximates an extreme eigenvalue of the matrix A [17]. To further improve the stability, these shifts are ordered in a Leja ordering, such that the distance between two consecutive shifts is maximized. If we encounter a complex shift for a real-precision matrix A, we rearrange the arithmetics so that the complex arithmetic is avoided [4, Section 7.3.2]. Since these shifts are not available for the first restart-loop, we use the standard GMRES iterations.

B. Performance Studies

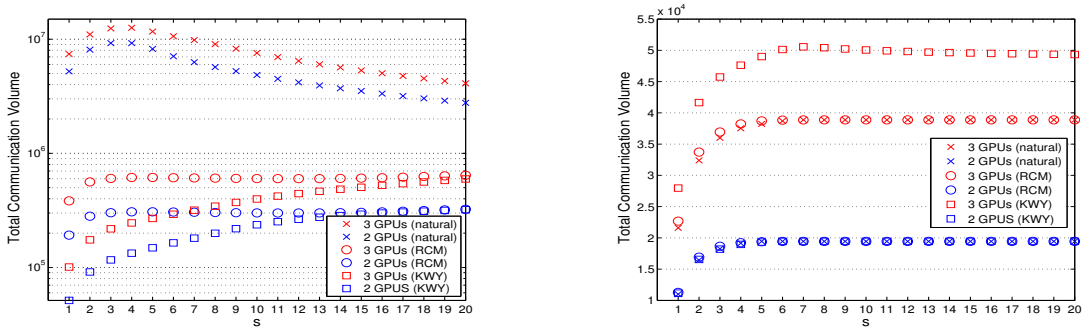
The performance of *MPK* strongly depends on the sparsity structure of the matrix A. One of the performance-impact factors is a so-called surface-to-volume ratio which quantifies how the local diagonal block $A(\mathbf{i}^{(d,s+1)}, \mathbf{i}^{(d,s+1)})$ is connected to the other diagonal blocks through the off-diagonal submatrix $A(\mathbf{i}^{(d,s+1)}, \boldsymbol{\delta}^{(d,s)})$. Figure 6 plots the ratio $nnz(A(\boldsymbol{\delta}^{(d,1:s)},:))/nnz(A^{(d)})$ to study the increase in this surface-to-volume ratio with respect to the parameter s (see Figure 12 for matrix properties). This ratio also quantifies the additional memory needed to store the boundary submatrix $A(\boldsymbol{\delta}^{(d,1:s)},:)$ in comparison to the memory needed to store the local matrix $A^{(d)}$. Since the natural matrix ordering in some cases leads to the full index set $\mathbf{i}^{(d,1)}$, even for a small value of s, we tested using two matrix reordering algorithms, the reverse Cuthill-McKee (RCM) [18] from HSL[2] and a k-way graph partitioning (KWY) of METIS[3]. We observe that for `G3_circuit`, the matrix reordering significantly reduces the surface-to-volume ratio, but the ratio still increases superlinearly with respect to s. On the other hand, `cant` is naturally banded, and the surface-to-volume ratio increases almost linearly with all the ordering schemes.

For given $A^{(d)}$, Figure 6 also shows the additional computation $W^{(d,s)}$ required by *MPK*, which is the area between the x-axis and plot $\left(\text{i.e., } W^{(d,s)} = 2\sum_{k=1}^{s} nnz(A(\boldsymbol{\delta}^{(d,k:s)},:))\right)$. Hence, the total computational overhead over the m iterations is given by $\frac{m}{s}W^{(d,s)}$. For instance, if the surface $nnz(A(\boldsymbol{\delta}^{(d,1:s)},:))$ increases linearly with s, then $W^{(d,s)}$ is a quadratic function of s and the total computational overhead over a restart-loop increases linearly with s.

Next, in Figure 7, we show the total communication volume required by *MPK* for different values of s: i.e., $\frac{m}{s}(|\bigcup_d \boldsymbol{\delta}^{(d,1:s)}| + \sum_d |\boldsymbol{\delta}^{(d,1:s)}|)$, where the first term $|\bigcup_d \boldsymbol{\delta}^{(d,1:s)}|$ represents the communication to gather the required vector elements from the GPUs to the CPU, while the second term $\sum_d |\boldsymbol{\delta}^{(d,1:s)}|$ represents the communication to scatter the required elements to the GPUs. In particular, for cases where the index set size $|\boldsymbol{\delta}^{(d,1:s)}|$ increases linearly with s, the total communication volume will stay constant or even decrease with s. For both `G3_circuit` and `cant`, though the increase in $|\boldsymbol{\delta}^{(d,1:s)}|$ slowed down for larger s, it increased relatively fast for small s. Hence, for larger value of s (e.g., $s > 5$), the communication volume grew slowly, but in comparison to *SpMV*, *MPK* required a greater total communication volume over the m iterations. For the naturally

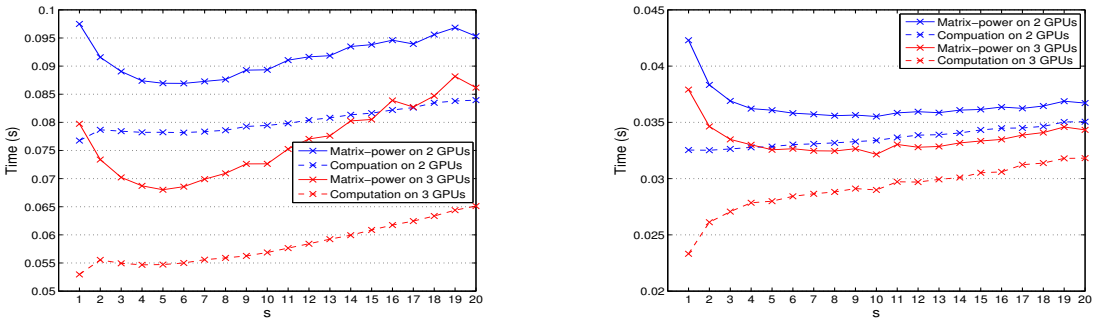
[2]http://www.hsl.rl.ac.uk/catalogue/mc60.xml With either the natural or RCM ordering, the matrix is distributed such that each GPU has about an equal number of rows.

[3]http://glaros.dtc.umn.edu/gkhome/metis/metis/overview. We also tested using recursive bisection algorithms, but the k-way partitioning that minimizes the edge-cut often gave smaller surfaces and better load balances.



(a) G3_circuit matrix. (b) cant matrix.

Fig. 7. Communication Volume in Matrix Powers Kernel.



(a) G3_circuit matrix with k-way partitioning. (b) cant matrix with natural ordering.

Fig. 8. Performance of Matrix Powers Kernel.

banded cant, KWY lead to greater communication volume than using RCM. However, KWY computes partitioning to minimize the edge cut and to balance the load among the GPUs, and it often rendered smaller communication volume for other matrices. For instance with G3_circuit, though KWY and RCM required about the same amount of communication for a large value of s, the communication volume using KWY was smaller for a small s.

Finally, Figure 8 shows the performance of our *MPK* to generate the total of one hundred vectors (i.e., $m = 100$). In addition to the total run time including the communication (solid line), we show the time spent in *SpMV* (dashed line). As we discussed above (see Figure 6), the flop count increases almost linearly with s for these two test matrices, and we see in Figure 8 that the computation time with *SpMV* also increases linearly. On the other hand, the communication time (the gap between the solid and dashed lines) decreases significantly compared to the standard algorithm ($s = 1$). This is because, though the communication volume increases (see Figure 7), the latency is reduced by a factor of s. As a result, the communication time decreases quickly with a small value of s, and then it starts to increase slightly as the communication bandwidth becomes dominant for a larger value of s. This indicates that the latency, together with the setups required for calling *MPK* (e.g., gathering and scattering of the vector elements) often has a greater impact on the performance of *MPK* than the bandwidth does, especially on a small number of GPUs.[4] In the end, *MKP* reduces the run time by up to 16% and 11% using $s > 1$ for the cant and G3_circuit matrices, respectively.

V. Orthogonalization Kernels

The orthogonalization process in GMRES may take as much or more time than the sparse matrix-vector products. While in Section IV, we focused on reducing communication of *SpMV* between GPUs, in this section we consider the communication of *TSQR* (and *BOrth*) both between the GPUs and on a GPU. In Sections V-A through V-E, we first describe the five orthogonalization procedures that we have implemented on the GPUs. Then, in Section V-F, we study their *TSQR* performance using random matrices. We defer investigation of their numerical stability within CA-GMRES to Section VI.

A. Modified Gram-Schmidt Procedure

For *TSQR*, Modified Gram-Schmidt (MGS) orthogonalizes each column $\mathbf{v}_{:,k}$ of $V_{j+1:j+s+1}$ against the previously orthogonalized columns $\mathbf{v}_{:,j+1}, \mathbf{v}_{:,j+2}, \ldots, \mathbf{v}_{:,k-1}$, one at a time: i.e.,

[4]Here, we compare the performance of *MPK* with that of *MPK* using $s = 1$, which in comparison to *SpMV*, performs one extra copying of the local vector $\mathbf{v}^{(d)}$ at each step on the GPU. In Section VI, we compare the performance of *SpMV* in GMRES with that of *MPK* in CA-GMRES.

for $\ell = j+1, j+2, \ldots, k-1$,

$$\mathbf{v}_{:,k} := \mathbf{v}_{:,k} - \mathbf{v}_{:,\ell}(\mathbf{v}_{:,\ell}^T \mathbf{v}_{:,k}).$$

MGS was numerically stable for *TSQR* in our experiments. However, to orthogonalize each $\mathbf{v}_{:,k}$, it requires the $k-j-1$ dot products $r_{\ell,k} := \mathbf{v}_{:,\ell}^T \mathbf{v}_{:,k}$, each of which requires a global reduction between the GPUs. Specifically, for our implementation to orthogonalize $\mathbf{v}_{:,k}$ against $\mathbf{v}_{:,\ell}$, the d-th GPU first forms its local dot product $r_{\ell,k}^{(d)} := \mathbf{v}_{:,\ell}^{(d)T} \mathbf{v}_{:,k}^{(d)}$ and asynchronously sends the result to the CPU. Then, the CPU computes the final product $r_{\ell,k} := \sum_{d=1}^{n_g} r_{\ell,k}^{(d)}$ and copies $r_{\ell,k}$ back to the GPUs for the local orthogonalization $\mathbf{v}_{:,k}^{(d)} := \mathbf{v}_{:,k}^{(d)} - r_{\ell,k}\mathbf{v}_{:,\ell}^{(d)}$.

MGS can be used for *BOrth* to orthogonalize the set of $s+1$ vectors $V_{j+1:j+s+1}$ against the previously orthogonalized vectors $V_{1:j}$: i.e., for $\ell = 1, 2, \ldots, j$,

$$V_{j+1:j+s+1} := V_{j+1:j+s+1} - \mathbf{v}_{:,\ell}(\mathbf{v}_{:,\ell}^T V_{j+1:j+s+1}).$$

Though the $s+1$ vectors $V_{j+1:j+s+1}$ are orthogonalized against $\mathbf{v}_{:,\ell}$ at once, *BOrth* still communicates j times.

B. Classical Gram-Schmidt Procedure

For *TSQR*, Classical Gram-Schmidt (CGS) orthogonalizes each column $\mathbf{v}_{:,k}$ of $V_{j+1:j+s+1}$ against the previously orthogonalized columns $\mathbf{v}_{:,j+1}$, $\mathbf{v}_{:,j+2}$, $\ldots$, $\mathbf{v}_{:,k-1}$, all at once: i.e.,

$$\mathbf{v}_{:,k} := \mathbf{v}_{:,k} - V_{j+1:k-1}(V_{j+1:k-1}^T \mathbf{v}_{:,k}).$$

Hence, CGS aggregates all the communication to orthogonalize each $\mathbf{v}_{:,k}$ into a single message, and in comparison to MGS, it reduces the communication latency by a factor of $k-j-1$. Namely, to orthogonalize $\mathbf{v}_{:,k}$, we first let the GPUs independently compute their local matrix-vector products, $\mathbf{r}_{j+1:k-1,k} := V_{j+1:k-1}^{(d)T} \mathbf{v}_{:,k}^{(d)}$. Then, the CPU accumulates these local products, $\mathbf{r}_{j+1:k-1,k} := \sum_{d=1}^{n_g} \mathbf{r}_{j+1:k-1,k}^{(d)}$. Finally, each GPU independently orthogonalizes its local vector, $\mathbf{v}_{:,k}^{(d)} := \mathbf{v}_{:,k}^{(d)} - V_{j+1:k-1}^{(d)T} \mathbf{r}_{j+1:k-1,k}$. CGS relies on BLAS-2 matrix-vector products, in contrast to MGS that relies on BLAS-1 dot products. As a result, in comparison to MGS, CGS not only reduces the latency, but also improves the data locality of accessing $\mathbf{v}_{:,\ell}^{(d)}$ on each GPU.[5]

Just like MGS, CGS can be used for *BOrth*:

$$V_{j+1:j+s+1} := V_{j+1:j+s+1} - V_{1:j}(V_{1:j}^T V_{j+1:j+s+1}).$$

Though it only requires a singe matrix-matrix product, in practice, the previous vectors are not completely orthogonal, and CGS results in a faster loss of orthogonality than when MGS is used. Though restarting the GMRES iteration helps to maintain orthogonality, a reorthogonalization is often required.

[5]We investigated a fused CGS that fuses the computation of the norm $\|\mathbf{v}_{:,k}\|_2$ with the matrix-vector product $V_{1:k-1}^T \mathbf{v}_{:,k}$ [19]. It replaces a reduction of CGS with a synchronization to check for the numerical stability on the GPU. We have not seen a significant performance improvement from this approach in our experiments. We have also studied a pipelined GMRES [19] to overlap *SpMV* to compute $\mathbf{v}_{:,j+1}$ on the GPU with the matrix-vector product to orthogonalize the previous vector $\mathbf{v}_{:,j}$ on the CPU. The matrix-vector product with tall skinny matrices on the GPU was more efficient than that on CPU, and using the CPU often slowed down the procedure in our experiments. MGS that computes $r_{k,:}$ at once would perform as well as CGS.

C. Cholesky QR Factorization

For *TSQR*, Cholesky QR (CholQR) orthogonalizes the set of $s+1$ vectors $V_{j+1:j+s+1}$ at once in three steps. It first forms the Gram matrix $B := V_{j+1:j+s+1}^T V_{j+1:j+s+1}$ on the CPU through the local matrix-matrix product

$$B^{(d)} := V_{j+1:j+s+1}^{(d)T} V_{j+1:j+s+1}^{(d)}$$

on the GPU, followed by the reduction $B := \sum_{d=1}^{n_g} B^{(d)}$ on the CPU. Then, the CPU computes its Cholesky factor R of B (i.e., $R^T R := B$). Finally, the GPU orthogonalizes $V_{j+1:j+s+1}$ by a triangular solve $V_{j+1:j+s+1}^{(d)} := V_{j+1:j+s+1}^{(d)} R^{-1}$. This orthogonalizes the set of $s+1$ vectors with a single pair of GPU-to-CPU and CPU-to-GPU communications, while MGS and CGS would require $(s+1)(s+2)/2$ and $s+1$ reductions, respectively. Furthermore, the computation of $B^{(d)}$ is based on a BLAS-3 matrix-matrix product instead of BLAS-1 or BLAS-2 products in MGS and CGS, respectively. Hence, the data locality of accessing the previous columns $V_{1:k-1}$ can be optimized not only to orthogonalize $\mathbf{v}_{:,k}^{(d)}$ (like CGS) but also to orthogonalize all the remaining columns $V_{k+1:s+1}^{(d)}$.

Unfortunately, the condition number of B is the square of the condition number of $V_{j+1:j+s+1}$. This often causes numerical instabilities, especially in CA-GMRES, where $V_{j+1:j+s+1}$ can be ill-conditioned (see Section VI).

D. Singular Value QR Factorization

When the matrix $V_{j+1:j+s+1}$ is ill-conditioned, or one of the column vectors is a linear combination of the other columns, the Cholesky factorization of its Gram matrix may fail. To overcome this numerical challenge, the Singular Value QR (SVQR) computes the upper-triangular matrix R by first computing the singular value decomposition (SVD) of the Gram matrix, $U\Sigma U^T := B$, and then the QR factorization $QR := \Sigma^{\frac{1}{2}} U^T$. Though computing the SVD and QR factorization is more expensive than computing the Cholesky factorization, the dimension of the Gram matrix is much smaller than that of the original matrix A (i.e., $s \ll n$). Hence, just like CholQR, SVQR performs most of its flops through the BLAS-3 matrix-matrix product to form the Gram matrix, and it requires only a pair of the CPU-GPU communications.

In some rare instances, CA-GMRES converged with CholQR but not with SVQR. A reason for this could be that the matrix $V_{j+1:j+k}$ generated by *MPK* becomes increasingly ill-conditioned as k increases, and the condition number of the leading matrix $B(1:k, 1:k)$ is the square of the condition number of $V_{j+1:j+k}$. In the Cholesky factorization, the matrix B is factorized from the top-left of the matrix to the bottom-right, and the error introduced during the Cholesky factorization of the trailing submatrix $B(k+1:s+1, k+1:s+1)$ is localized within itself. Furthermore, the Gram matrix from *MPK* is graded, and this property seems to help maintain the positive diagonals during the Cholesky factorization. Similarly, in the first step of SVD to bidiagonalize the Gram matrix through the Householder transformations, the numerical errors are localized. However, during the SVD of the bidiagonal

Modified Gram-Schmidt
for $k = 1, 2, \ldots, s+1$ **do**
  for $\ell = 1, 2, \ldots, k-1$ **do**
    for $d = 1, 2, \ldots, n_g$ **do**
      $r_{\ell,k}^{(d)} := \mathbf{v}_{:,\ell}^{(d)T} \mathbf{v}_{:,k}^{(d)}$
    end for
    $r_{\ell,k} := \sum r_{\ell,k}^{(d)}$
      on CPU (comm)
    for $d = 1, 2, \ldots, n_g$ **do**
      copy $r_{\ell,k}$ to GPU-d (comm)
      $\mathbf{v}_{:,k}^{(d)} := \mathbf{v}_{:,k}^{(d)} - \mathbf{v}_{:,k}^{(d)} r_{\ell,k}$
      $r_{k,k}^{(d)} := \mathbf{v}_{:,k}^{(d)T} \mathbf{v}_{:,k}^{(d)}$
    end for
  end for
  $r_{k,k} := \sqrt{\sum r_{k,k}^{(d)}}$
    on CPU (comm)
  for $d = 1, 2, \ldots, n_g$ **do**
    copy $r_{k,k}$ to GPU-d (comm)
    $\mathbf{v}_{:,k}^{(d)} := \mathbf{v}_{:,k}^{(d)} / r_{k,k}$
  end for
end for

Cholesky QR
for $d = 1, 2, \ldots, n_g$ **do**
    $B^{(d)} := V_{1:s+1}^{(d)T} V_{1:s+1}^{(d)}$
end for
$B := \sum B^{(d)}$ on CPU (comm)
$R := \texttt{chol}(B)$ on CPU
for $d = 1, 2, \ldots, n_g$ **do**
    copy R to GPU-d (comm)
    $V_{1:s+1}^{(d)} := V_{1:s+1}^{(d)} R^{-1}$
end for

Classical Gram-Schmidt
for $k = 1, 2, \ldots, s+1$ **do**
  for $d = 1, 2, \ldots, n_g$ **do**
    $\mathbf{r}_{1:k-1,k}^{(d)} := V_{1:k-1}^{(d)T} \mathbf{v}_{:,k}^{(d)}$
  end for
  $\mathbf{r}_{1:k-1,k} := \sum \mathbf{r}_{1:k-1,k}^{(d)}$
    on CPU (comm)
  for $d = 1, 2, \ldots, n_g$ **do**
    copy $\mathbf{r}_{1:k-1,k}$ to GPU-d (comm)
    $\mathbf{v}_{:,k}^{(d)} := \mathbf{v}_{:,k}^{(d)} - V_{1:k-1}^{(d)} \mathbf{r}_{1:k-1,k}$
    $r_{k,k}^{(d)} := \mathbf{v}_{:,k}^{(d)T} \mathbf{v}_{:,k}^{(d)}$
  end for
  $r_{k,k} := \sqrt{\sum r_{k,k}^{(d)}}$
    on CPU (comm)
  for $d = 1, 2, \ldots, n_g$ **do**
    copy $r_{k,k}$ to GPU-d (comm)
    $\mathbf{v}_{:,k}^{(d)} := \mathbf{v}_{:,k}^{(d)} / r_{k,k}$
  end for
end for

Communication-Avoiding QR
for $d = 1, 2, \ldots, n_g$ **do**
    $[V_{1:s+1}^{(d)}, R^{(d)}] := \texttt{qr}(V_{1:s+1}^{(d)})$
    copy $R^{(d)}$ to CPU (comm)
end for
$[[Q^{(1)}; Q^{(2)}; \ldots, Q^{(n_g)}], R] =$
    $\texttt{qr}([R^{(1)}; R^{(2)}; \ldots; R^{(n_g)}])$
    on CPU
for $d = 1, 2, \ldots, n_g$ **do**
    copy $Q^{(d)}$ to GPU$-d$ (comm)
    $V_{1:s+1}^{(d)} := V_{1:s+1}^{(d)} Q^{(d)}$
end for

Fig. 9. Pseudocodes of *TSQR* algorithms, where $\texttt{chol}(B)$ and $\texttt{qr}(V^{(d)})$ compute the Cholesky and QR factorization of B and $V^{(d)}$, respectively.

matrix, the errors from the bottom-right of the matrix may propagate to the leading submatrix. In the end, though the norm-wise errors of both SVQR and CholQR are relatively small, its element-wise errors could be greater in SVQR, especially on the top-right of the matrix. Fortunately, we observe that this numerical issue of SVQR is often resolved by scaling the Gram matrix such that its diagonals are one [20]. However, we have not identified a test case where CA-GMRES converges with SVQR but not with CholQR. We study the numerical behavior of CholQR and SVQR in Section VI.

E. Communication-Avoiding QR Factorization

Communication-avoiding QR (CAQR) orthogonalizes a set of vectors $V_{j+1:j+s+1}$ against each other through a tree reduction of the local QR factorizations. Namely, each GPU first computes the QR factorization of the local matrix $V_{j+1:j+s+1}^{(d)}$, then the local R-factors are gathered on the CPU, and the final QR factorization is computed on the CPU (see Figure 9 for a pseudocode). Just like CholQR, CAQR requires only a single pair of the GPU-CPU communication to orthogonalize $V_{j+1:j+s+1}$. However, the local QR factorizations are based on BLAS-1 and BLAS-2 operations, which often obtain only a fraction of the BLAS-3 performance in CholQR.[6]

[6]Currently, we explicitly form the orthogonal matrix Q. Though this makes the interfaces to the rest of the routines (e.g., reorthogonalization) simpler, it doubles the flop count. We plan to investigate the potential of storing Q as the set of Householder transformations. We will also investigate the effects of blocking [9] and a potential of using batched QRs on a GPU.

	$\|I - Q^T Q\|$	# flops	# GPU-CPU comm.
MGS [21]	$O(\epsilon\kappa)$	$2ns^2$, BLAS-1 xDOT	$(s+1)(s+2)$
CGS [22]	$O(\epsilon\kappa^s)$	$2ns^2$, BLAS-2 xGEMV	$2(s+1)$
CholQR [20]	$O(\epsilon\kappa^2)$	$2ns^2$, BLAS-3 xGEMM	2
SVQR [20]	$O(\epsilon\kappa^2)$	$2ns^2$, BLAS-3 xGEMM	2
CAQR [5]	$O(\epsilon)$	$4ns^2$, BLAS-1,2 xGEQR2	2

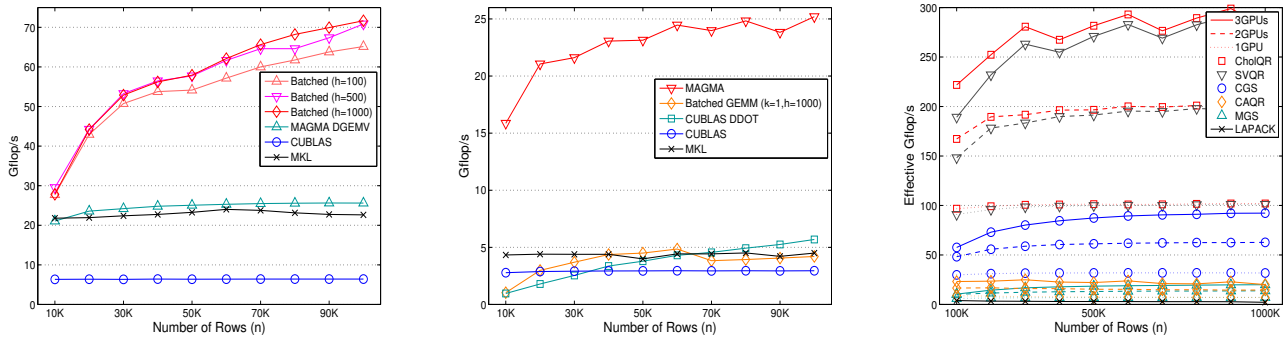
Fig. 10. *TSQR*($V_{j+1:j+s+1}$), κ is the condition number of $V_{j+1:j+s+1}$.

To summarize this subsection, Figure 9 shows the pseudocodes of our *TSQR* implementations, and Figure 10 lists some of their properties.

F. Performance Studies

The performance of the orthogonalization procedures depends strongly on the performance of the BLAS kernels (see Figure 10). Figure 11(a) shows the performance of DGEMM that is used for *TSQR* with CholQR and SVQR (and for *BOrth* with CGS). Clearly, the standard implementation (i.e., CUBLAS 4.2) is not optimized for the typical tall-skinny matrices in CA-GMRES (i.e., hundreds of thousands of rows, n, and tens of columns, s). In fact, the performance of CUBLAS DGEMM was lower than that of MKL or that of MAGMA DGEMV, making CholQR based on CUBLAS slower than CGS based on MAGMA. To improve the performance of CholQR and SVQR, we investigated the performance of a batched DGEMM, where we first divided the n-by-$(s+1)$ matrix $V_{j+1:j+s+1}$ into h-by-$(s+1)$ submatrices, and then called the CUBLAS batched DGEMM and performed a reduction operation to sum up the results of all the DGEMMs. To align the memory access within each DGEMM, we rounded up the number of rows, h, to be a multiple of 32. Furthermore, since the batched kernel assumes the sizes of all the DGEMMs to be the same, we set the leading dimension to store $V_{j+1:j+s+1}$ to be a multiple of h and padded the bottom with zeros. This routine has the same interface as the standard DGEMM. To internally call the batched kernel, our routine uses an array of pointers that point to the beginnings of the submatrices. We clearly see that this batched DGEMM outperforms the other implementations and we used it for our implementation of the orthogonalization procedures.

Figure 11(b) shows the performance of DGEMV that is used for *TSQR* based on CGS (and *BOrth* based on MGS). Similar to DGEMM, the performance of the standard implementation (i.e., CUBLAS 4.2) was poor. For instance, the performance of CUBLAS DGEMV was lower than that of MKL or that of CUBLAS DDOT, making CGS slower than MGS when CUBLAS is used to implement these two procedures. We also tried using the batched DGEMM to compute the matrix-vector product with this tall-skinny matrix, but the performance was improved only slightly. To improve the performance of CGS, we developed an optimized MAGMA DGEMV kernel for tall-skinny matrices, which computes $V_{j+1:k-1}^T \mathbf{v}_{:,k}$ based on dot-products – each thread block in the new GPU kernel computes a dot-product between a column of $V_{j+1:k-1}$ and $\mathbf{v}_{:,k}$. This improves the performance of DGEMV by a factor of about five over the other implementations and is used to implement



(a) DGEMM to compute $V_{1:s+1}^T V_{1:s+1}$. (b) DGEMV to compute $V_{1:s+1}^T \mathbf{v}$. (c) Performance of *TSQR*($V_{1:s+1}$).

Fig. 11. Performance of DGEMM, DGEMV, and *TSQR* for a tall-skinny matrix $V_{1:s+1}$ ($s+1=30$).

our orthogonalization procedures.[7]

Finally, Figure 11(c) shows the performance of *TSQR* on up to three GPUs, where "LAPACK" uses DGEQRF and DORGQR of threaded MKL on 16-core SandyBridge, and the effective Gflop/s is computed as the ratio of the total flops required by DGEQRF and DORGQR over the orthogonalization time in second. On a single GPU, our routines obtain the performance of the optimized BLAS kernels; i.e., MGS, CGS, and CholQR/SVQR obtain the performance of DDOT, DGEMV, and DGEMM, respectively. The performance of CAQR is close to that of MGS because *TSQR* on each GPU is based on BLAS-1 and BLAS-2 operations. The figure also shows that each routine scales well over the three GPUs.

VI. Experimental Results of CA-GMRES

Finally, in this section, we study the numerical behavior of the different orthogonalization procedures within CA-GMRES, and the performance of CA-GMRES on multiple GPUs. One of the parameters that affects the performance of GMRES is the number of iterations before each restart, m (a small value of m helps maintain the orthogonality of the basis vectors and reduces the cost of generating a larger projection subspace, while too small m leads to slow convergence or stagnation). Hence, for each test matrix, we use the parameter m that obtained the shortest solution time on a single GPU among the values of $m = 30, 60, 90, \ldots, 180$. The computed solution is considered to have converged when the ℓ_2-norm of the initial residual is reduced by at least four orders of magnitude. To improve the stability and the convergence, before the iteration starts, the matrix is balanced; namely, the rows are first scaled by their norms, and then the columns are scaled by their norms. Our code was compiled using the GNU **gcc** 4.4.6 compiler and CUDA **nvcc** 4.2 compiler with the optimization flag **-O3**, and linked with MKL 2011_sp1.8.273.

[7]We are investigating other batched kernels (e.g., GEMV, SYRK, and GEQRF) and the potential of using an auto-tuner to improve the performance (see [23]). The performance of CholQR/SVQR also depends on the triangular-solve on a tall-skinny matrix, where we use MAGMA DTRSM that is developed for the Cholesky or LU factorization.

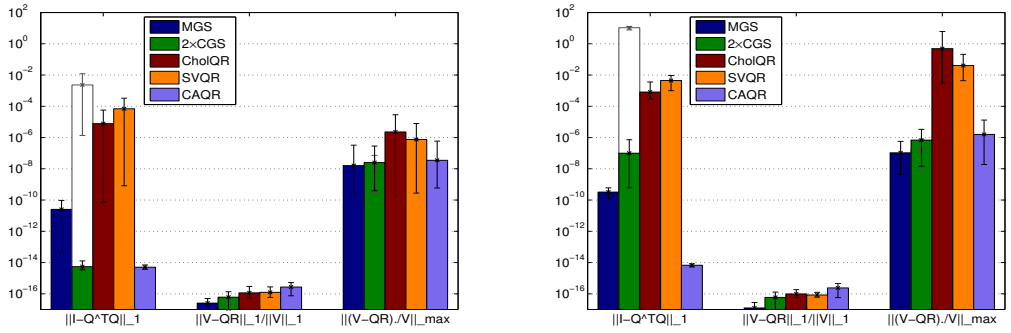
Name	Source	$n/1000$	nnz/n	θ_1/θ_2	$\kappa(B)$
cant	FEM Cantilever	62	64.2	$\frac{7.5685}{7.5682}$	$3.26e^{16}$
G3_circuit	Circuit simulation	1,585	4.8	$\frac{1.9964}{1.9829}$	$8.54e^{9}$
dielFilterV2real	FEM in EM	1,157	41.9	$\frac{5.2766}{5.1892}$	$5.81e^{11}$
nlpkkt120	KKT optimization	3,542	26.9	$\frac{3.6554}{3.6127}$	$2.42e^{7}$

Fig. 12. Test Matrices, $\kappa(B)$ is the condition number of the last Gram matrix from the first restart-loop with the setups in Figure 14.

Figure 12 lists the test matrices from the University of Florida Matrix Collection that were used for our experiments.

A. Numerical Studies of Orthogonalization Procedures

The bar graph in Figure 13(a) shows the average *TSQR* error norms using different orthogonalization procedures in CA-GMRES$(20, 30)$, where each *TSQR* computes $QR := V$, and $E./A$ is the element-wise division (i.e., $(E./V)_{i,j}$ is $e_{i,j}/v_{i,j}$). The error bars show the minimum and maximum errors. For this particular matrix, CA-GMRES with CGS required reorthogonalization to converge, which is indicated by "2×" in front of CGS in the figure, and the white bars show the error norms after the first orthogonalization. All the procedures obtained about the same factorization errors and residual norm convergence. In term of the orthogonality errors $\|I - Q^T Q\|$, CholQR and SVQR had greater errors than MGS due to the squared condition number of the Gram matrix, while MGS had greater errors than CAQR because the errors could be amplified by the condition number of the basis vectors generated by *MPK* (see Table 10). Figure 13(b) shows the same error norms in CA-GMRES$(30, 30)$. The results were similar to those in CA-GMRES$(20, 30)$, except that the orthogonality errors of CGS were greater than those of MGS even after reorthogonalization and that the element-wise errors $\|(A - QR)./A\|$ were significantly greater using CholQR and SVQR, illustrating the effects of the greater condition number. Some error bars were longer in CA-GMRES$(20, 30)$ than those in CA-GMRES$(30, 30)$. This is because with $(s, m) = (20, 30)$, *MPK* generates 20 and then 10 basis vectors, and the condition number of the basis vectors is much greater when 20 vectors are generated.



(a) CA-GMRES(20, 30). (b) CA-GMRES(30, 30).

Fig. 13. Average *TSQR* Errors in CA-GMRES, `G3_circuit` (1 GPU).

ng	*TSQR*	Rest.	Time (ms) Ortho/Res Total	*TSQR*	SpMV/Res	Total/Res	SpdUp
cant, natural ordering							
GMRES(60)							
1	MGS	7	167.1	–	36.3	204.3	
1	CGS	7	25.7	–	35.7	62.9	
2	CGS	7	17.1	–	22.9	40.0	
3	CGS	7	14.3	–	21.4	37.1	
CA-GMRES(1, 60)							
1	–	7	76.3	14.2	36.8	114.7	
CA-GMRES(15, 60)							
1	CGS	7	20.2	11.2	35.2	58.2	
1	2×CholQR	7	12.9	7.1	30.7	44.9	1.40
2	2×CholQR	7	8.2	4.3	21.0	30.3	1.32
3	2×CholQR	7	7.1	3.7	16.0	24.4	1.52
G3_circuit, k-way partitioning							
GMRES(30)							
1	MGS	16	855.0	–	54.8	931.9	
1	CGS	16	193.8	–	56.3	256.3	
2	CGS	16	100.0	–	31.3	143.8	
3	CGS	16	68.8	–	25.0	100.0	
CA-GMRES(1, 30)							
1	–	16	568.8	151.9	55.4	631.3	
CA-GMRES(15, 30)							
1	2×CGS	16	296.3	253.1	49.9	352.5	
1	CholQR	16	70.0	45.2	49.8	126.2	2.03
2	CholQR	16	38.4	24.9	30.6	75.0	1.92
3	CholQR	16	25.2	16.3	24.6	56.9	1.76
dielFilterV2real, k-way partitioning							
GMRES(180)							
1	MGS	176	20964.0	–	3437.5	24420.0	
1	CGS	181	3765.2	–	3419.9	7202.8	
2	CGS	168	1797.0	–	1732.2	3543.4	
3	CGS	199	1223.2	–	1130.7	2366.4	
CA-GMRES(1, 180)							
1	–	202	9440.6	715.3	3455.0	12921.0	
CA-GMRES(15, 180)							
1	2×CGS	148	1891.9	1247.3	3386.5	5301.4	
1	2×CholQR	181	1047.0	398.6	3391.2	4460.8	1.61
2	2×CholQR	139	431.8	176.8	2129.4	2688.2	1.31
3	2×CholQR	160	369.3	151.2	1989.3	2374.2	0.62

Fig. 14. CA-GMRES performance, where "Rest." is the number of restarts, "Ortho/Res" and "SpMV/Res" are the average *Orth* and *SpMV* time per restart-loop, respectively, "Total/Res" is the average restart-loop time, and "SpdUp" is the speedup over GMRES. *BOrth* is based on CGS.

B. Performance Studies of CA-GMRES

Finally, Figure 14 compares the CA-GMRES performance with that of GMRES, both of which use the optimized GPU kernels from Section V. Though CA-GMRES and GMRES needed about the same number of restarts on one GPU, for an ill-conditioned A, the round-off errors could lead to a different restart counts on a different number of GPUs. Hence, in the table, we show both the average time per restart and the restart counts. The first observation is that the CA-GMRES performance using $s = 1$ is much lower than that of GMRES. This is because CA-GMRES relies on computational kernels to orthogonalize multiple vectors at a time, and these kernels are not optimized for orthogonalizing one vector at a time.[8] For instance, with *BOrth* based on MGS or CGS, when $s = 1$, *BOrth* computes the dot-product $\mathbf{v}_{:,\ell}^T \mathbf{v}_{:,k}$ or the matrix-vector product $V_{j+1:j+s+1}^T \mathbf{v}_{:,\ell}$ using a matrix-vector or matrix-matrix multiplication routines, respectively. However, as soon as s becomes larger (e.g., $s = 10$), the combination of *BOrth* and *TSQR* reduces the communication both on a GPU and between the GPUs, and shortens the orthogonalization time, obtaining speedups of between 1.99 and 4.16 over *Orth* of GMRES.

On the other hand, due to the overheads associated with *MPK* (see Section IV), obtaining the speedups in the sparse-matrix vector product was more challenging. Depending on the sparsity patterns, *MPK* could obtain a speedup of up to 1.33 over *SpMV*, but *MPK* can be slower. Figure 15 summarizes the performance of CA-GMRES by showing the time per restart-loop that is normalized by that of GMRES on one GPU. Here, if *SpMV* is faster than *MPK*, then CA-GMRES uses *SpMV*. By reducing the communication, CA-GMRES obtained speedups of between 1.32 and 2.06 over GMRES.

VII. Conclusion

We surveyed the numerical behavior of different orthogonalization (*Orth*) procedures, and of their blocked variants (*BOrth*) and tall-skinny QR (*TSQR*), in combination with a sparse matrix-vector product (*SpMV*) and a matrix powers kernel (*MPK*). We also showed that new optimizations, especially for tall skinny matrices, are needed to make *Orth*, *BOrth*, and *TSQR*, and hence CA-GMRES or GMRES, perform well on the GPUs. Since many existing GPU implementations of GMRES rely on standard techniques (e.g., CUBLAS), these optimizations may improve their performance. In addition, such tall-skinny matrices appear in other sparse solvers (e.g., sparse factorization), and both *SpMV* and *Orth* are needed in many solvers (e.g., subspace projection methods for linear and eigenvalue problems). Hence, our studies may have greater impact beyond GMRES. In the end, our performance results on

[8] We are investigating if an auto-tuner can reduce this performance gap.

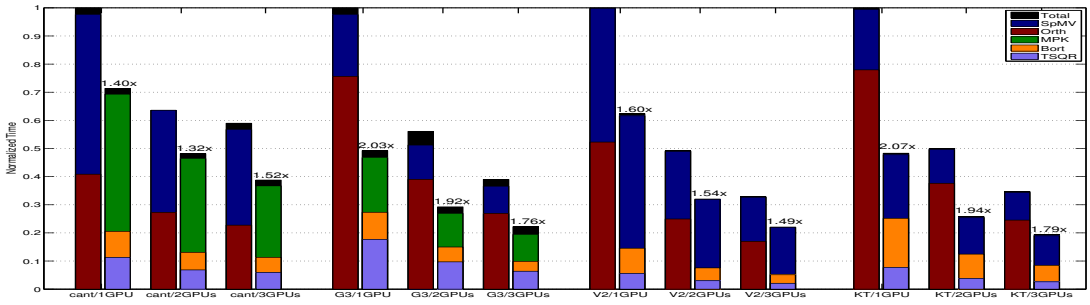


Fig. 15. Performance of GMRES and CA-GMRES, `nlpkkt120` required 746 GMRES(120) iterations and about 90 minutes on one GPU for the solution convergence. For CA-GMRES, we used $s = 10$. For the timing results of other matrices, see Figure 14. CA-GMRES bars show the speedups over GMRES.

16-core Intel Sandy Bridge CPUs with three NDIVIA Fermi GPUs showed that CA-GMRES can obtain a speedup of up to 2.0 over GMRES.

Since the performance of CA-GMRES depends critically on the performance of the GPU kernels, we are looking to further optimize these kernels. We also plan to study other partitioning algorithms (e.g., hypergraph partitioning), other orthogonalization strategies (e.g., rank-revealing QR with column pivoting [10] or the use of a mixed-precision arithmetic [23]), and adaptive schemes to select or switch orthogonalization strategies or to adjust input parameters (e.g., m and s [23]). Finally, our performance results demonstrated that though *MPK* could obtain a speedup of up to 1.3 over *SpMV*, it can be slower due to the overheads traded for reducing the communication latency. We would like to study the potential of reducing communication of *MPK* on a single GPU, and the performance of CA-GMRES on a larger number of GPUs, in particular, the GPUs distributed over multiple compute nodes, where the communication is more expensive.

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)," NSF Keeneland - Georgia Institute of Technology Subcontract #RA241-G1 on NSF Prime Grant #OCI-0910735, and Sandia National Laboratories is a multiprogram laboratory managed and operated by Sandia Corporation, a wholly owned subsidiary of Lockheed Martin Corporation, for the U.S. Department of Energy's National Nuclear Security Administration under contract DE-AC04-94AL85000.

REFERENCES

[1] Y. Saad, Iterative methods for sparse linear systems, 3rd Edition, the Society for Industrial and Applied Mathematics, Philadelphia, PA, 2003.
[2] H. van der Vorst, Iterative Krylov methods for large linear systems, Cambridge University Press, Cambridge, MA, 2003.
[3] Y. Saad, M. Schultz, GMRES: A generalized minimal residual algorithm for solving nonsymmetric linear systems, SIAM J. Sci. Stat. Comput. 7 (1986) 856–869.
[4] M. Hoemmen, Communication-avoiding Krylov subspace methods, Ph.D. thesis, University of California, Berkeley (2010).
[5] J. Demmel, L. Grigori, M. Hoemmen, J. Langou, Communication-optimal parallel and sequential QR and LU factorizations, SIAM Journal on Scientific Computing 34 (1).
[6] M. Mohiyuddin, M. Hoemmen, J. Demmel, K. Yelick, Minimizing communication in sparse matrix solvers, in: the proceedings of the Conference on High Performance Computing Networking, Storage and Analysis (SC), New York, NY, USA, 2009, pp. 36:1–36:12.
[7] I. Yamazaki, K. Wu, A communication-avoiding thick-restart Lanczos method on a distributed-memory system, in: the proceedings of Euro-Par Workshops, 2011, pp. 345–354.
[8] M. Anderson, G. Ballard, J. Demmel, K. Keutzer, Communication-avoiding QR decomposition for GPUs, Tech. Rep. UCB/EECS-2010-131, University of California Berkeley (Oct 2010).
[9] M. Hoemmen, A communication-avoiding, hybrid-parallel, rank-revealing orthogonalization method, in: the proceedings of IEEE International Parallel and Distributed Processing Symposium (IPDPS), 2011, pp. 966–977.
[10] J. Demmel, L. Grigori, M. Gu, H. Xiang, Communication avoiding rank revealing QR factorization with column pivoting, LAPACK Working Note 276 (May 2013).
[11] CUSP library, available at https://github.com/cusplibrary.
[12] D. Lukarski, PARALUTION - the library for iterative sparse methods on CPU and GPU, available online: http://www.paralution.com/ (2013).
[13] P. Tillet, K. Rupp, S. Selberherr, C.-T. Lin, Towards performance-portable, scalable, and convenient linear algebra, Talk: HotPar (2013).
[14] V. Minden, B. Smith, M. Knepley, Preliminary implementation of PETSc using GPUs, the proceedings of the 2010 International Workshop of GPU Solutions to Multiscale Problems in Science and Engineering.
[15] C. G. Baker, M. A. Heroux, Tpetra, and the use of generic programming in scientific computing, Scientific Programming 20 (2) (2012) 115–128.
[16] NVIDIA CUBLAS library, https://developer.nvidia.com/cublas.
[17] Z. Bai, D. Hu, L. Reichel, A Newton basis GMRES implementation, IMA Journal of Numerical Analysis 14 (1994) 563–581.
[18] E. Cuthill, J. McKee, Reducing the bandwidth of sparse symmetric matrices, in: the proceedings of the 24th National Conference, 1969, pp. 157–172.
[19] P. Ghysels, T. Ashby, K. Meerbergen, W. Vanroose, Hiding global communication latency in the GMRES algorithm on massively parallel machines, SIAM J. Scientific Computing 35.
[20] A. Stathopoulos, K. Wu, A block orthogonalization procedure with constant synchronization requirements, SIAM J. Sci. Comput. 23 (2002) 2165–2182.
[21] A. Bjorck, Solving linear least squares problems by Gram-Schmidt orthogonalization, BIT Numerical Mathematics 7 (1967) 1–21.
[22] N. Abdelmalek, Round off error analysis for Gram-Schmidt method and solution of linear least squares problems, BIT Numerical Mathematics 11 (1971) 345–368.
[23] I. Yamazaki, S. Tomov, T. Dong, J. Dongarra, Mixed-precision orthogonalization scheme and adaptive step size for CA-GMRES on GPUs, 2014, submitted to the 11th international meeting on high-performance computing for computational science (VECPAR).