SAND2013-8969C

Studying 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
[†]Oak Ridge National Laboratory, Oak Ridge, USA
[‡]University of Manchester, Manchester, UK
[§]Sandia National Laboratory, New Mexico, USA

iyamazak@eecs.utk.edu, hanzt@icl.utk.edu, tomov@cs.utk.edu, and dongarra@eecs.utk.edu

***Abstract*—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 GPUs are now becoming a crucial component in computing, in this paper, we investigate the effectiveness of these techniques on multicore CPUs with multiple GPUs. While we present the detailed performance studies of a matrix-power kernel on the GPUs, we particularly focus on the orthogonalization strategies which have a great impact not only on the numerical stability of GMRES but also on its performance, especially as the coefficient matrix becomes sparser or more ill-conditioned. We present the experimental results on two six-cores Intel Sandy Bridge CPUs with three NDIVIA Fermi GPUs and demonstrate that significant speedups can be obtained avoiding the communication both on a single GPU and between the GPUs. As a part of our studies, we investigate several optimization techniques for the GPU kernels that are also used in other sparse solvers beside GMRES. Hence, our studies not only demonstrate the importance of avoiding communication on the GPUs, but they also provide several insights about the effects of these optimization techniques on the performance of the sparse solvers.**

I. Introduction

Many scientific or engineering simulations require the solution of 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 required memory and/or the required flop count of the direct factorization of the coefficient matrix could become unfeasibly expensive. A parallel computer with a large aggregated memory and a high computing capacity may provide a remedy to this large cost of the 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 and of 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 the Krylov subspace iterative methods [1], [2], that as they usually provide smooth sequences of solution approximations at low computational cost. For unsymmetric linear systems, the Generalized Minimum Residual (GMRES) method [3] is often the method of choice as it combines high robustness with flexibility allowing for problem-specific optimization.

On modern computers, in comparison to flops, communication is becoming increasingly expensive in term of both required cycle time and energy consumption. To overcome this challenge, in recent years, several techniques to avoid communication in various algorithms including GMRES [4] have gained attention. While graphic processing units (GPUs) have often become crucial components in scientific and engineering computing, the gap between the arithmetic and communication costs is growing on these as well. In this paper, we study the potential of using communication-avoiding techniques on multicore CPUs with multiple GPUs, providing the detailed performance studies of both a matrix-power kernel and several orthogonalization procedures that often dominate the GMRES iteration time. As a 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 on both a single 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 the related works. Then in Section III, we review the communication-avoiding GMRES (CA-GMRES) and provide the 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-power kernel and of several orthogonalization procedures, respectively, and demonstrate their performance. Finally, in Section VI, we study the performance of CA-GMRES on the GPUs, and in Section VII, we conclude with 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

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

Fig. 1. Pseudocode of GMRES(m).

```
$\widehat{\mathbf{x}} := \mathbf{v}_{:,1}$ and $\mathbf{v}_{:,1} := \mathbf{b}/\|\mathbf{b}\|_2$.
repeat
    Projection Subspace Generation:
    for $j = 1, s+1, 2s+1, \dots, m$ do
        MPK: Generate new vectors $\mathbf{v}_{:,k+1} := A\mathbf{v}_{:,k}$
            for $k = j, j+1, \dots, \min(j+s, m)$.
        BOrt: 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:
    Compute the solution $\widehat{\mathbf{x}}$ in the generated subspace,
        which minimizes its residual norm.
    Set $\mathbf{v}_{:,1} := \mathbf{r}/\|\mathbf{r}\|_2$, where $\mathbf{r} := \mathbf{b} - A\widehat{\mathbf{x}}$.
until solution convergence do
```

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

experiments were conducted on a single compute node of the Keeneland System [5] at the Georgia Institute of Technology, which consists of two six-core Intel Sandy Bridge (Xeon E5) processors and three NVIDIA M2090 GPU accelerators.

II. Related Works

a) Communication-avoiding, CA-GMRES, in particular any work done on multiple GPUs?:

b) SpMV, MPK, on GPUs?: MPK: [6]..

GPU: different storage formats [7],

c) Orthgonalization, TSQR, on GPUs?:

III. Communication-Avoiding GMRES

Figure 1 shows a pseudocode of the standard restarted GMRES iterations. At each iteration, a new Krylov basis vector is generated through a sparse matrix-vector multiplication (SpMV), and the resulting vector $\mathbf{v}_{:,j+1}$ is orthonormalized (Orth) against the previously-generated orthonormal basis vectors $\mathbf{v}_{:,1}, \mathbf{v}_{:,2}, \dots, \mathbf{v}_{:,j}$. Both SpMV and Orth require communication since the vector $\mathbf{v}_{:,j}$ need to be loaded into the local memory. Figure 2 shows a pseudocode of CA-GMRES iterations that aims to reduce this communication. The main idea is that by having special kernels to generate and to orthogonalize a set of s vectors at once, it becomes possible not only to communicate the vector $\mathbf{v}_{:,j}$ once for every s steps but also to optimize the data-access to both A

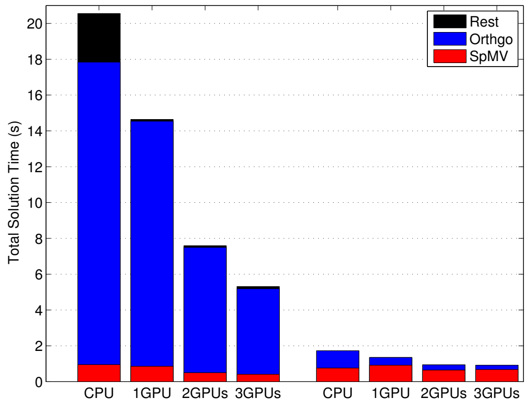


Fig. 3. Performance of GMRES on 16-core Sandy Bridge CPUs with up to three NDIVIA M2090 GPUs. CPU code is linked to MKL 2011_sp1.8.273, and SpMV uses the CSR format on the CPU, while ELLPACKT format is used on the GPUs. The detailed description of the GPU implementation is provided in the remaining of the paper.

and $\mathbf{v}_{:,1}, \mathbf{v}_{:,2}, \dots, \mathbf{v}_{:,j}$. In Sections IV and V, we discuss these two computational kernels SpMV and Orth in more details.

To reduce both the computational and storage requirements of computing a large projection subspace, the GMRES iteration is restarted after a fixed number $m+1$ of the basis vectors is computed. At restart, the approximate solution $\widehat{\mathbf{x}}$ is updated by solving a least-square 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 is not only a by-product of the orthogonalization procedure (see Section V) but is also in a Hessemberg form. Hence, the least-square 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 total of about $2m \cdot nnz(A)$ and $2mn^2$ flops over the m iterations, respectively ($n \gg m$).

To utilize the GPUs, we distribute the matrix A in a block row format among the GPUs. Then, the basis vectors $\mathbf{v}_{:,1}, \mathbf{v}_{:,2}, \dots, \mathbf{v}_{:,m+1}$ are generated entirely on the GPUs, while the Hessenberg matrix H is copied to the CPUs and the least-square problem is solved on the CPUs. Our main focus in this paper is to compare the performance of CA-GMRES on the GPUs with that of GMRES on the GPUs. Figure 3 compares the performance of GMRES on the GPUs with that of our GMRES implementation on CPUs that uses MKL routines for SpMV and Ortho. Clearly, this may not be a fair comparison since MKL routines may not be optimized for the matrices arising from GMRES. However, we provide the figure just to give a reference point to our GMRES performance on the GPUs.

IV. Matrix-Power Kernel

For the sparse matrix-vector multiplication on multiple GPUs, the communication of the distributed vector through the PCI-express could become a bottleneck. To reduce this bottleneck, given a starting vector $\mathbf{v}_{:,j}$, the matrix-power kernel

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

Matrix-power Generation: generate $\mathbf{v}^{(d)}_2, \mathbf{v}^{(d)}_3, \ldots, \mathbf{v}^{(d)}_{s+1}$
for $k = 1, 2, \ldots, s$ do
    for $d = 1, 2, \ldots, n_g$ do
        SpMV: compute $\mathbf{y}^{(d)} := A^{(d,k)}\mathbf{z}^{(d,k\%2)}$
        expand $\mathbf{y}^{(d)}$ into a full vector $\mathbf{z}^{(d,(k+1)\%2)}$
        copy the local part $\mathbf{y}^{(d)}_{\mathbf{i}^{(d,1)}}$ of $\mathbf{y}^{(d)}$ into $\mathbf{v}^{(d)}_{k+1}$
    end for
end for
```

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

communicates all the required vector elements at once so that each GPU can independently compute the local components of the s matrix vector multiplications $A\mathbf{v}_{:,j}, A^2\mathbf{v}_{:,j}, \ldots, A^s\mathbf{v}_{:,j}$. Here, in Section IV-A, we first describe this matrix-power kernel that we implemented on multiple GPUs, and then in Section IV-B, we analyze its performance using different test matrices. For our discussion, we use $A^{(d)}$ and $V^{(d)}$ to denote the local matrices on the d-th GPU and use n_g to denote the number of available GPUs.

A. Algorithm to Generate Matrix-Power

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

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

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

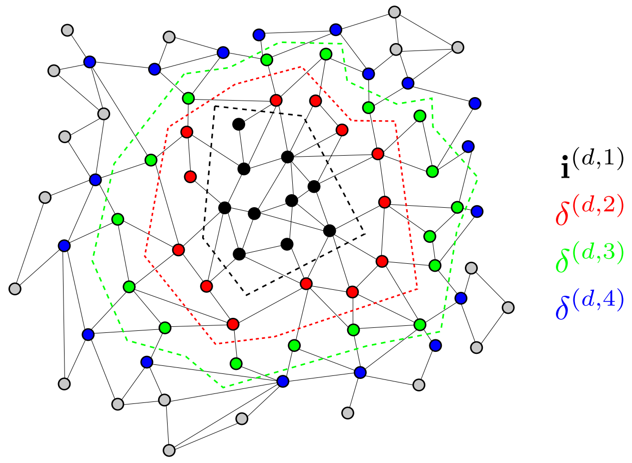


Fig. 5. Illustration of Surface-to-Volume Ratio.

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

Though this matrix-power kernel reduces the latency by the factor of s, the d-th GPU requires the additional memory to store the boundary submatrix $A(\boldsymbol{\delta}^{(d,1:s+1)},:)$, where $\boldsymbol{\delta}^{(d,1:k+1)} = \bigcup_{\ell \le k+1} \boldsymbol{\delta}^{(d,\ell)}$. Furthermore, at the k-th step of the matrix-power kernel, in addition to the matrix-multiplication with the local submatrix $A^{(d)}$, the d-th GPU performs the additional multiplication with the k-th boundary submatrix $A(\boldsymbol{\delta}^{(d,1:k+1)},:)$. Finally, to generate the m basis vectors over the GMRES restart-loop, the d-th GPU must gather the total of $O(\frac{m}{s}|\boldsymbol{\delta}^{(d,1:s+1)}|)$ vector elements, where $|\boldsymbol{\delta}^{(d,1:s+1)}|$ is the size of the index set $\boldsymbol{\delta}^{(d,1:s+1)}$. For $s > 1$, this total communication volume could be greater than that required by the standard algorithm to compute one sparse matrix-vector multiplication at a time over the restart-loop (i.e., $s = 1$). The amount of these storage, computational, and communication overheads depend on the sparsity structure of the matrix A, and we study these in Section IV-B for different test matrices.

Generating the monomial basis vectors based on the above matrix-power kernel is often numerically unstable, leading to a stochastic convergence of the CA-GMRES iterations. This is because the generated vectors converges to the eigenvector corresponding to the largest 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. To avoid this numerical instability, our matrix-power kernel 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 Hessemberg matrix H from the first restart and approximates an extreme eigenvalue of the matrix A [8]. To further improve the stability, these shifts are ordered in a Leja ordering such that the distance between two consecutive shifts is maximized. When we encounter a complex shift for a real-precision matrix A, we rearrange the procedure 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 for the first restart-loop.

B. Performance Studies of Matrix-Power Kernel

The performance of the matrix-power kernel strongly depends on the sparsity structure of the matrix A. In particular, the main factor determining the performance is a so-called surface-to-volume ratio [] which quantifies how the local diagonal block $A(\mathbf{i}^{(d,1)}, \mathbf{i}^{(d,1)})$ is connected to the other diagonal blocks through the off-diagonal submatrix $A(\mathbf{i}^{(d,1)}, \boldsymbol{\delta}^{(d,1:s+1)})$. In Figures 6(a) and 7(a), we study the increase in this surface-to-volume ratio $nnz(A(\mathbf{i}^{(d,s)}, :)/nnz(A(\boldsymbol{\delta}^{(d,1:s+1)}, :)$ with respect to the parameter s for two of our test matrices (see Figure 13 for matrix properties). Since the natural matrix ordering in some cases leads to the full index set $\mathbf{i}^{(d,s+1)}$ even for a small value of s, we tested using two matrix reordering algorithms, the reverse Cuthill-McKee (RCM) [9] from HSL[1] and a k-way graph partitioning (KWY) of METIS[2]. 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.

Figures 6(a) and 7(a) also illustrate the additional computation $W^{(d,k)}$ required by the matrix-power kernel; for a fixed value k of s, this is given by the area under the curve over $s = 1$ through k $\left(\text{i.e., } W^{(d,k)} = \sum_{s=1}^{k} 2nnz(A(\boldsymbol{\delta}^{(d,1:s)}, :))\right)$. Then, the total computational overhead over the m iterations is given by $\frac{m}{k}W^{(d,k)}$. 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.

In Figure 6(b) and 7(b), we show the required total communication volume for different value of s: i.e., $\frac{m}{s}(|\bigcup_d \boldsymbol{\delta}^{(d,1:s+1)}| + \sum_d |\boldsymbol{\delta}^{(d,1:s+1)}|)$, where the first term $|\bigcup_d \boldsymbol{\delta}^{(d,1:s+1)}|$ represents the communication to gather the required vector elements from the GPUs to the CPU, while the second term $\sum_d |\boldsymbol{\delta}^{(d,1:s+1)}|$ represents the communication to scatter the required elements to the GPUs. In particular, for cases where the index set size $|\boldsymbol{\delta}^{(d,1:s+1)}|$ increases linearly with s, the total communication volume will stay constant or even decrease with s. For both G3_circuit and cant, the increase of $|\boldsymbol{\delta}^{(d,1:s+1)}|$ is relatively fast for small s, and slows down for larger ones. Hence, in comparison to the standard algorithm ($s = 1$), the matrix-power kernel requires a greater communication volume ($s > 1$), but the communication volume grows slowly for a larger value of s (e.g., $s > 5$). For the naturally banded cant, KWY leads to greater communication volume than RCM. However, KWY computes a partitioning to minimize the edge cut while balancing the load among the GPUs, and it often renders smaller communication volume for other matrices.[3] For instance for G3_circuit, the communication volume using KMY is smaller than for RCM, especially for small values of s. However, with KWY, the communication volume increase is larger, and for large s, the two algorithms need about the same amount of communication.

Finally, Figure 8 shows the performance of our matrix-power kernel 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 (bashed line). As we discussed above (see Figures 6(a) and 7(a)), the flop count increased 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 from that with the standard algorithm ($s = 1$). This is because though the communication volume increases (see Figures 6(b) and 7(b)), the latency is reduced with the inversely proportional rate with 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 CA-GMRES than the bandwidth does, especially on a small number of GPUs.

V. Orthogonalization Kernels

Beside the sparse matrix-vector multiplication, the orthogonalization process often dominates the iteration time of GMRES; i.e., Bort to orthogonalize $V_{j+1:j+s+1}$ against $V_{1:j}$ and TSQR to orthogonalize the column vectors $V_{j+1:j+s+1}$ with each other. Here in Sections V-A through V-E, we first describe the four orhogonalization procedures that we have implemented for CA-GMRES on the GPUs. While for the matrix-power kernel in Section IV, we focused on reducing the communication between the GPUs, in this section we consider the communication both between the GPUs and between the computing cores on a single GPU. In Section V-F, we study the performance of these different orthogonalization procedures using random matrices (the numerical stability of these procedures within CA-GMRES for different test matrices are studied in Section VI).

A. Modified Gram-Schmidt Procedure

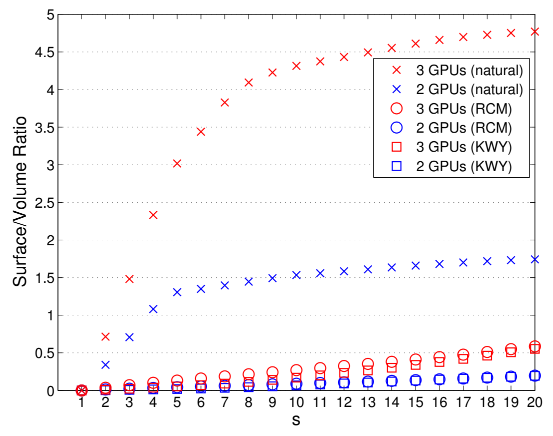
In the modified Gram-Schmidt (MGS) procedure to orthogonalize the $s+1$ columns $V_{j+1:j+s+1}$ against each other, each vector $\mathbf{v}_k$ is incrementally orthogonalized against the previously-orthogonalized columns $\mathbf{v}_{j+1}, \mathbf{v}_{j+2}, \ldots, \mathbf{v}_{k-1}$;

$$\mathbf{v}_k := \prod_{\ell=j+1}^{k-1} (I - \mathbf{v}_\ell \mathbf{v}_\ell^T)\mathbf{v}_k.$$

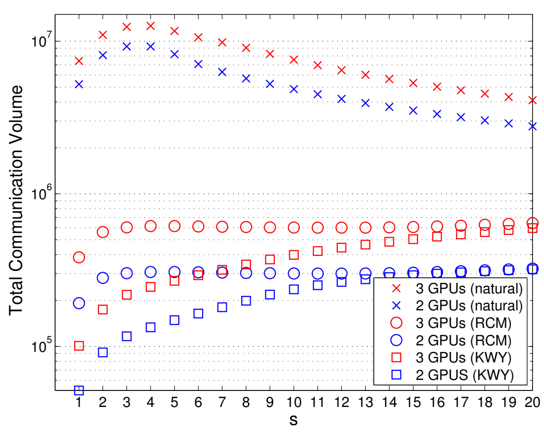
[1]http://www.hsl.rl.ac.uk/catalogue/mc60.xml

[2]http://glaros.dtc.umn.edu/gkhome/metis/metis/overview. We have also tested using recursive vertex and edge bisection algorithms, and observed the k-way partitioning algorithm that minimizes the edge-cut often gives smaller surfaces with better load balances.

[3]With either the natural or RCM ordering, the matrix is distributed such that each GPU has an equal number of rows.

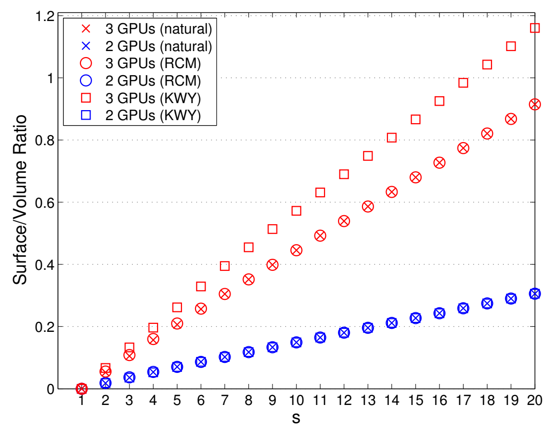


(a) Surface-to-Volume Ratio.

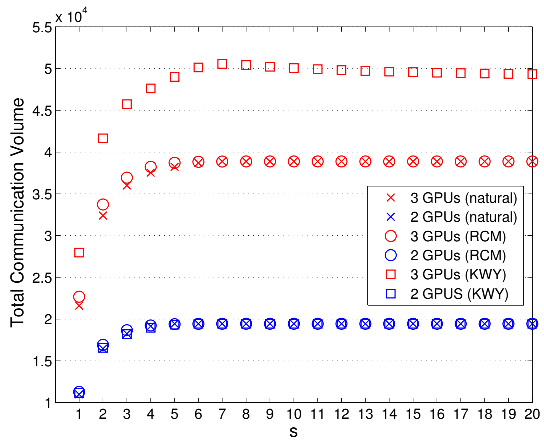


(b) Communication Volume.

Fig. 6. Costs of Matrix-Power Kernel for `G3_circuit` matrix.

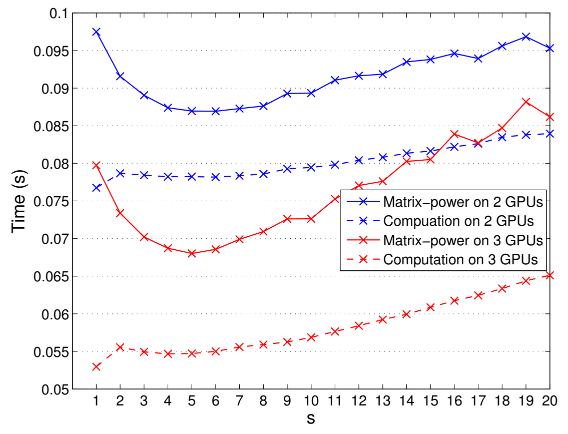


(a) Surface-to-Volume Ratio.

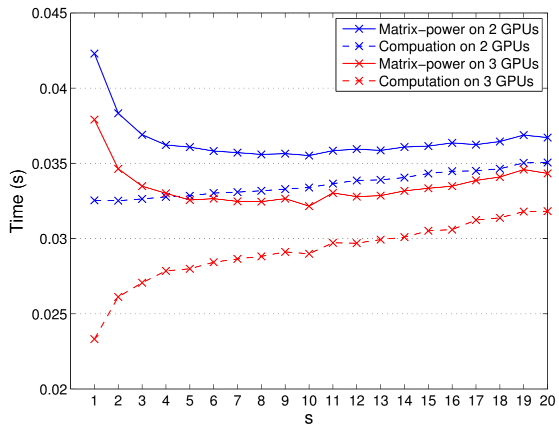


(b) Communication Volume.

Fig. 7. Costs of Matrix-Power Kernel for `cant` matrix.



(a) `G3_circuit` matrix with k-way partitioning.



(b) `cant` matrix with natural ordering.

Fig. 8. Peformance of Matrix-Power Kernel in ELPACKT format.This is on KIDS

We observed that this procedure is often stable in practice. However, the procedure requires $2(k-j-1)$ communication among the GPUs for computing the s dot-products $r_{\ell,k} := \mathbf{v}_\ell^T\mathbf{v}_k$; specifically, to orthogonalize $\mathbf{v}_k$ against each $\mathbf{v}_\ell$, 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 by summing the local products $r_{\ell,k} := \sum_{d=1}^{n_g} r_{\ell,k}^{(d)}$ and then copies the product back to the GPUs for the local orthogonalization $\mathbf{v}_k^{(d)} := \mathbf{v}_k^{(d)} - r_{\ell,k}\mathbf{v}_\ell^{(d)}$.

This procedure can be used for a block orthogonalization `Borh` to orthogonalize the set of vectors $V_{j+1:j+s+1}$ against the previous vectors $V_{1:j}$:

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

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

B. *Classical Gram-Schmidt Procedure*

To orthogonalizes the column vectors $V_{j+1:j+s+1}$ with each other, the classical Gram-Schmidt (CGS) procedure assumes the orthogonality of the previous basis vectors, and orthogonalizes the vector $\mathbf{v}_k$ against the previous basis vectors at once:

$$\mathbf{v}_k := (I - V_{1:k-1}V_{1:k-1}^T)\mathbf{v}_k.$$

Hence, all the required communication can be aggregated into a single message, and in comparison to MGS, CGS reduces the latency by a factor of k for orthogonalizing the k-th vector $\mathbf{v}_k$. Specifically, the GPU first independently computes its local matrix-vector multiplication $\mathbf{r}_{1:k-1,k} := V_{1:k-1}^{(d)T}\mathbf{v}_k^{(d)}$. Then, the CPU accumulate these local results, $\mathbf{r}_{1:k-1,k} := \sum_{d=1}^{n_g}\mathbf{r}_{1:k-1,k}^{(d)}$. Finally, the CPU broadcasts $\mathbf{r}_{1:k-1,k}$ to the GPUs, and each GPU independently orthogonalizes its local vector, $\mathbf{v}_k^{(d)} := \mathbf{v}_k^{(d)} - V_{1:k-1}^{(d)T}\mathbf{r}_{1:k-1,k}$. In comparison to MGS that uses BLAS-1 dot-products, CGS relies on a BLAS-2 matrix-vector multiplication. As a result, in comparison to MGS, CGS not only reduces the latency by a factor of k between the GPUs, but it also improves the data locality to access $\mathbf{v}_j^{(d)}$ on the distinct GPUs.[4]

Just like MGS, CGS can be used in `Bort`,

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

Unfortunately, in practical implementations, the orthogonality often suffers from rounding errors in the simultaneous orthogonalization procedure. Though restarting the iteration helps to maintain the orthogonality, a re-orthogonalization is often required to ensure the orthogonality of the vectors.

[4]We have also investigated a fused CGS where the computation of the norm $\|\mathbf{v}_k\|_2$ is fused with the matrix-vector multiplication $V_{1:k-1}^T\mathbf{v}_k$ [10]. Since it requires a check to ensure the numerical stability, this approach replaces the reduction with a synchronization on the GPU. We have not seen a significant performance improvement from this approach in our experiments. In addition, we have studied a pipelined GMRES [10] to overlap `SpMV` to compute $\mathbf{v}_{j+1}$ on the GPU with the matrix-vector multiplication to orthogonalize the previous vector $\mathbf{v}_j$ on the CPU. Since the matrix-vector multiplication with the tall-skinny matrices on the GPU was more efficient than that on CPU, using the CPU for the multiplication often slowed down the procedure in our experiments.

C. *Cholesky QR Factorization*

In the Cholesky QR (CholQR) factorization, a set of $s+1$ vectors $V_{j+1:j+s+1}$ are orthogonalized against each other in the following three steps; first the Gram matrix $B := V_{j+1:j+s+1}^T V_{j+1:j+s+1}$ is computed, then its Cholesky factorization $R^TR = B$ is derived, and finally $V_{j+1:j+s+1}$ is orthogonalized by a triangular solve: $V_{j+1:j+s+1} := V_{j+1:j+s+1}R^{-1}$. Hence, the set of $s{+}1$ vectors can be orthogonalized with a single pair of the GPU-to-CPU and CPU-to-GPU communication, while MGS and CGS would require $(s{+}1)(s{+}2)/2$ and $s+1$ reductions, respectively. Furthermore, the computation of B is based on the BLAS-3 matrix-matrix operation instead of the BLAS-1 vector-vector or the BLAS-2 matrix-vector operations in MGS or CGS, respectively. As a result, the data locality to access not only the single vector $\mathbf{v}_k^{(d)}$ but also all the previous vectors $V_{1:k-1}^{(d)}$ can be optimized on the GPU.

Unfortunately, the condition number of B is the square of the condition number of $V_{j+1:j+s+1}$. This often causes numerical problems, especially in CA-GMRES, where the condition number of $V_{j+1:j+s+1}$ can become large (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, in the Singular Value QR (SVQR) factorization [11], we compute the singular value decomposition (SVD) of the Gram matrix, $U\Sigma U^T := B$, and then compute the QR factorization of $\Sigma^{\frac{1}{2}}U^T$ to obtain the upper-triangular matrix R. Though computing the SVD and QR factorization is more expensive than computing the Cholesky factorization, the matrix dimension of the Gram matrix is much smaller compared to the original matrix A (i.e., $s \ll n$). Hence, most of the flops is computed by the BLAS-3 matrix-matrix multiplication to form the Gram matrix, and the factorization requires only a pair of the CPU-GPU communication, just like the CholQR factorization.

Unfortunately, we observed that CA-GMRES, based on SVQR factorization may suffer from a larger element-wise error than when using CholQR factorization, which may result in loss of convergence (see Section VI). The main reason for this is that the vectors $V_{j+1:j+s+1}$ generated from the matrix-power kernel become increasingly linearly-dependent and ill-conditioned as s 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 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. As a result, when the factorization is successful, we often observe that both norm-wise and element-wise backward errors of the factorization are relatively small. Furthermore the Gram matrix from the matrix-power kernel is scaled, 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 matrix, the errors from the bottom-right of the matrix propagate to the leading submatrix and at the end, though the norm-wise error is relatively small, we observe relatively large element-wise errors, especially on the top-right corner of the matrix. Fortunately, unlike CholQR, we observe that this numerical issue of SVQR is often resolved by scaling the Gram matrix such that its diagonals are one. However, were unable to identify a test case where CA-GMRES converges with SVQR but not with CholQR.

E. Communication-Avoiding QR Factorization

In the communication-avoiding QR (CAQR) algorithm, a set of vectors $V_{j+1:j+s+1}$ is orthogonalized against each other through a tree reduction of the local QR factorizations; in other words, each GPU first computes the QR factorization of the local matrix $V^{(d)}_{j+1:j+s+1}$, then the local R-factors are gathered on the CPU, and the final QR factorization is computed on the CPU. Just like CholQR, CAQR requires only a single pair of the GPU-to-CPU and CPU-to-GPU 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.[5]

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

F. Performance Studies of Orthogonalization Procedures

The performance of orthogonalization 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 `Bort` with CGS). We clearly see that the standard implementation (e.g., CUBLAS) is not optimized for the typical tall-skinny shape of the matrix 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 the performance of CholQR based on CUBLAS lower than that of 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 $V_{1:s+1}$ into h-by-$(s+1)$ submatrices, and then called the batched DGEMM of CUBLAS and performed a reduction operation to sum up the results of each DGEMM. To alight the memory access within each DGEMM of the batched DGEMM, we round up the number of rows, h, to be a multiple of 32. Furthermore, since the batched DGEMM

[5]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.

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
      $r^{(d)}_{\ell,k} := \mathbf{v}^{(d)T}_{\ell}\mathbf{v}^{(d)}_{k}$
    end for
    $r_{\ell,k} := \sum r^{(d)}_{\ell,k}$ (comm.)
    for $d = 1, 2, \ldots, n_g$ do
      $\mathbf{v}^{(d)}_{k} := \mathbf{v}^{(d)}_{k} - \mathbf{v}^{(d)}_{k} r_{\ell,k}$
    end for
  end for
  $r_{k,k} = \|\mathbf{v}_k\|_2$ (comm)
  for $d = 1, 2, \ldots, n_g$ do
    $\mathbf{v}^{(d)}_{k} := \mathbf{v}^{(d)}_{k} / r_{k,k}$
  end for
end for
```

Classical Gram-Schmidt
```
for $k = 1, 2, \ldots, s+1$ do
  for $d = 1, 2, \ldots, n_g$ do
    $\mathbf{r}^{(d)}_{1:k-1,k} := V^{(d)T}_{1:k-1}\mathbf{v}^{(d)}_{k}$
  end for
  $\mathbf{r}_{1:k-1,k} := \sum \mathbf{r}^{(d)}_{1:k-1,k}$ (comm)
  end for
  $\mathbf{v}_k := \mathbf{v}_k - V_{1:k-1}\mathbf{r}_{1:k-1,k}$
  $r_{k,k} = \|\mathbf{v}_k\|_2$ (comm)
  for $d = 1, 2, \ldots, n_g$ do
    $\mathbf{v}^{(d)}_{k} := \mathbf{v}^{(d)}_{k} / r_{k,k}$
  end for
end for
```

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

Communication-Avoiding QR
```
for $d = 1, 2, \ldots, n_g$ do
    $[Q^{(d,1)}, R^{(d,1)}] =$ qr$(A^{(d)})$
    copy $R_g$ to CPU
end for
$[[Q^{(1,2)}; \ldots, Q^{(n_g,2)}], R_c] =$
    qr$([R^{(1,1)}; \ldots; R^{(n_g,1)}])$
    on CPU
for $d = 1, 2, \ldots, n_g$ do
    copy $Q^{(d,2)}$ to GPU$-d$
    $Q^{(d)} := Q^{(d,1)}Q^{(d,2)}$
end for
```

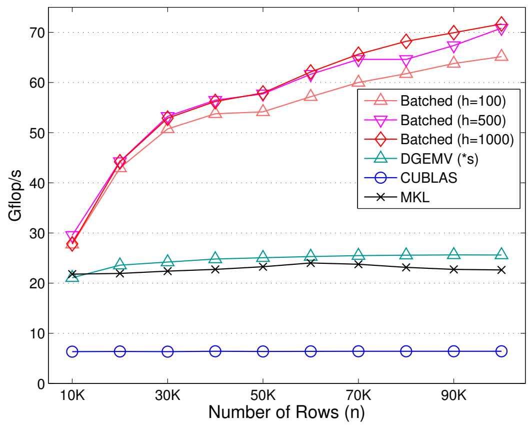
Fig. 9. Pseudocodes of TSQR algorithms.

	$\|I - Q^TQ\|$	# flops	# GPU-CPU comm.
MGS	$O(\epsilon\kappa)$	$2sn^2$, BLAS-1 xDOT	$(s+1)(s+2)$
CGS	$O(\epsilon\kappa^{s-1})$	$2sn^2$, BLAS-2 xGEMV	$2(s+1)$
CholQR	$O(\epsilon\kappa^2)$	$3sn^2$, BLAS-3 xGEMM	2
SVQR	$O(\epsilon\kappa^2)$	$3sn^2$, BLAS-3 xGEMM	2
CAQR	$O(\epsilon)$	$4sn^2$, BLAS-1,2 xGEQR2	2

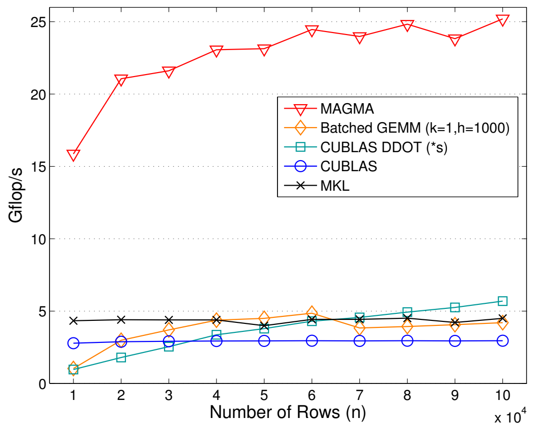
Fig. 10. TSQR$(V_{j+1:j+s+1})$, κ is the condition number of $V_{j+1:j+s+1}$.

expects the size of each DGEMM to be the same, we set the leading dimension to store $V_{1:s+1}$ to be a multiple of h and padded the bottom with zeros. This routine has the same interface as the standard DGEMM but internally uses an array of pointers that point to the beginnings of the submatrices for calling the batched DGEMM. We clearly see that this batched DGEMM outperforms the other implementations and is used for our orthogonalization procedures.

Figure 11(b) shows the performance of DGEMV that is used for `TSQR` based on CGS (and `Bort` based on MGS). Similar to DGEMM, the performance of the standard implementation (i.e., CUBLAS) was poor. For instance, the performance of CUBLAS DGEMV was lower than that of MKL or that of CUBLAS DDOT, making the performance of CGS lower than that of MGS when CUBLAS is used to implement these two algorithms. We also tried using the batched DGEMM to compute the matrix-vector multiplication 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 uses a thread block to perform a dot-product between a column of $V_{1:s+1}$ with a vector need to ask stan about the implementation. This improves the performance of DGEMV



(a) DGEMM to compute $V_{1:s+1}^T V_{1:s+1}$.



(b) DGEMV to compute $V_{1:s+1}^T \mathbf{v}_{:,s+2}$.

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

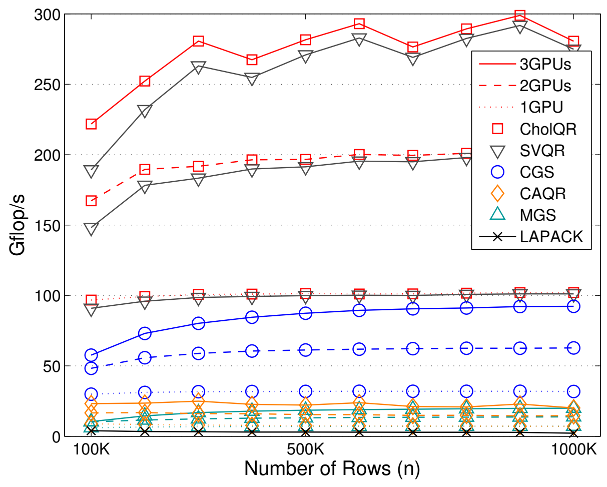


Fig. 12. Performance of TSQR ($s+1=30$).

by a factor of about five over the other implementations and is used for our orthogonalization procedures.[6]

Finally, Figure 12 shows the performance of our orthogonalization routines on up to three GPUs. On a single GPU, these routine obtain the performance of the optimized BLAS kernels; i.e., MGS, CGS, and CholQR and 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.

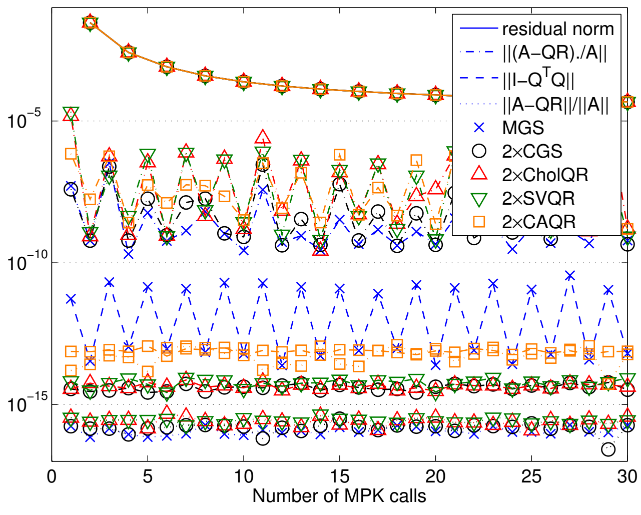
[6]We are investigating other batched kernels (e.g., GEMV, SYRK, and GEQRF) and potential of using auto-tuner to improve the performance of the dense kernels.

Name	Source	$n/1000$	nnz/n	θ_1/θ_m	θ_1/θ_2
cant	FEM Cantilever	62	64.2		
kkt_power	KKT optimization	$2,063$	6.2		
G3_circuit	Circuit simulation	$1,585$	4.8		
StocF-1465	Fluid dynamics	$1,465$	14.3		
dielFilterV2real	FEM in EM	$1,157$	41.9		

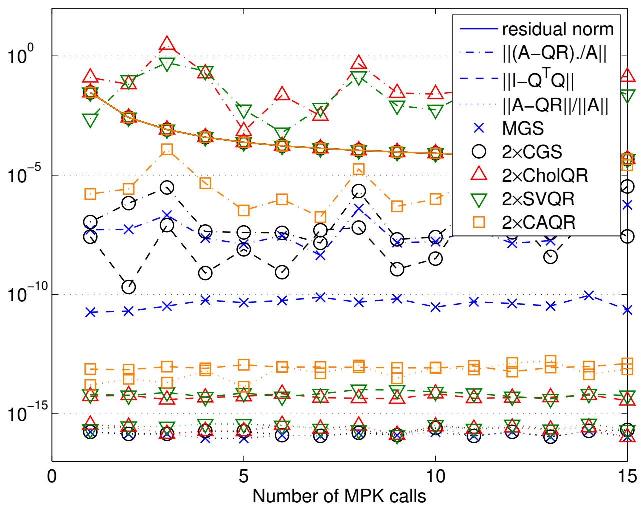
Fig. 13. Test Matrices.

VI. Performance Studies of CA-GMRES

Finally, in this section, we first study the numerical behavior of the different orthogonalization procedures within CA-GMRES. We then study the performance of CA-GMRES on multiple GPUs. One of the parameters that affect 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 if m is too small, then GMRES could suffer from slow convergence or stagnation). For each test matrix in this section, 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 optimal value of m may differ on a different number of GPUs). The computed solution is considered to have converged when the ℓ_2-norm of the initial residual is reduced by at least six orders of magnitude. To improve the stability and the convergence rate of the iteration, before the iteration starts, the matrix is equilibrated; namely, the rows of the matrix 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 then linked with MKL 2011_sp1.8.273. Figure 13 lists the test matrices used for our experiments. These matrices are from a wide range of applications and available from the University of Florida Matrix Collection[7].



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



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

Fig. 14. Effects of orthogonalization on 1 GPU, `G3_circuit`.

A. *Numerical Studies of Orthogonalization*

Figure 14(a) shows the errors of the TSQR factorization using different orthogonalization procedures in CA-GMRES(20, 30) on a single GPU, where $\|\cdot\|$ denotes the max-norm (i.e., $\|E\| = \max_{i,j} |e_{i,j}|$) and $E./A$ denotes the element-wise division (i.e., $(E./A)_{i,j}$ is $e_{i,j}/a_{i,j}$). All the orthogonalization procedures obtained obtained almost the same residual norm convergence and most of the factorization errors in the similar ranges. The only exceptions were $\|I - Q^T Q\|$ of MGS due to the lack of the reorthogonalization, and $\|I - Q^T Q\|$ and $\|A - QR\|/\|A\|$ of CAQR which was one and two order of magnitude greater than those of the other procedures even with the reorthogonalization. Figure 14(b) shows the same results in CA-GMRES(30, 30). The results were similar to those in CA-GMRES(20, 30), except that the element-wise errors $\|(A-QR)./A\|$ were significantly greater using CholQR and SVQR. For these particular setups, CA-GMRES with CGS or CholQR failed to converge without reorthogonalization, and it failed to converge with a larger value of s (i.e., $s = 40$) using all the orthogonalization procedures even with reorthogonalization, except for using MGS and using CAQR with reorthogonalization.

[7]http://www.cise.ufl.edu/research/sparse/matrices/

s	n_g	Basis	Schemes	Restarts	Ortho/Itr Total	Ortho/Itr TSQR	SpMV/Itr	Total/Itr
				cant, natural ordering				
GMRES(60)								
−−	1	−−	CGS	7	25.7	−−	35.7	62.9
−−	2	−−	CGS	7	17.1	−−	22.9	40.0
−−	3	−−	CGS	7	14.3	−−	12.4	37.1
CA-GMRES(60)								
1	1	Newt	CGS	7	75.7	14.3	37.1	115.7
10	1	Newt	CGS	7	20.0	11.4	35.7	58.6
10	1	Newt	2×CHO	7	20.0	10.0	35.7	58.6
10	2	Newt	2×CHO	7	12.9	7.1	22.9	40.0
10	3	Newt	2×CHO	7	11.4	5.7	18.6	32.9
				G3_circuit, k-way partitioning				
GMRES(30)								
−−	1	−−	CGS	15	206.7	−−	60.0	273.3
−−	2	−−	CGS	15	106.7	−−	33.3	153.3
−−	3	−−	CGS	15	73.3	−−	26.7	106.7
CA-GMRES(30)								
1	1	Mono	CGS	15	606.7	160.0	66.7	680.0
10	1	Mono	2×CGS	15	320.0	273.3	66.7	386.7
10	1	Newt	2×CHO	15	173.3	126.7	66.7	3.7
10	2	Newt	2×CHO	15	93.3	66.7	40.0	140.0
10	3	Newt	2×CHO	15	60.0	46.7	26.7	93.3
				dielFilterV2real, k-way partitioning				
GMRES(180)								
10	1	−−	MGS					
10	1	−−	CGS	179	3807.8	−−	3458.1	7284.4
10	1	−−	2×CGS	166	7347.6	−−	3439.8	10806.0
10	2	−−	2×CGS	177	3500.6	−−	1744.1	5258.2
10	3	−−	2×CGS	137	2321.9	−−	1132.8	3467.9
CA-GMRES(180)								
10	1	Mono	CGS					
10	10	Newt	CGS					
10	10	Newt	2×CGS	187	2114.4	1179.1	3472.2	5610.7
10	1	Newt	2×CHO					
10	2	Newt	2×CHO	108	763.0	283.3	2046.3	2825.9
10	3	Newt	2×CHO	135	508.9	194.1	1743.7	2268.9

Fig. 15. Performance of CA-GMRES, times are in milliseconds.

B. *Performance Studies of CA-GMRES*

VII. Conclusion

In this paper, we compared the performance of CA-GMRES with that of GMRES on multiple GPUs. Our performance results on two six-cores Intel Sandy Bridge CPUs with three NDIVIA Fermi GPUs demonstrated the significant speedups can be obtained avoiding the communication both on a single GPU and between the GPUs. As a part of this study, we investigated the performance of the GPU kernels required for the sparse matrix-vector multiplication `SpMV` and for the orthogonalization procedures `Orth`, and discovered new optimization techniques are needed especially for tall-skinny dense matrices to obtain the high-performance of GMRES on the GPUs. Since tall-skinny dense matrices appears in many other sparse solvers (e.g., sparse factorization), and both `SpMV` and `Orth` are needed in many other sparse solvers (e.g., subspace projection methods for linear and eigenvalue problems). the current studies have greater impact beyond GMRES. We also surveyed the numerical behavior of different orthogonalization strategies, in the combination with a matrix-power kernel.

Currently, we studying the performance of CA-GMRES on a larger number of GPUs, in particular, on distributed GPUs, where the CPU-GPU communication becomes more dominant. Since the performance of the GPU kernels are critical to obtain the high-performance of CA-GMRES, we are also looking to further optimize these GPU kernels. We also plan to study the potential of reducing communication of `MPK` on a single GPU, other partitioning algorithms (e.g., hypergraph partitioning) or orthogonalization strategies (e.g., QR with column pivoting or use of mixed precisions) to improve the performance of CA-GMRES, and adaptive schemes to select or switch orthogonalization strategies or to adjust input parameters (e.g., m and s).

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 DOE MAGMA - Department of Energy Office of Science Grant #DE-SC0004983, "Matrix Algebra for GPU and Multicore Architectures (MAGMA) for Large Petascale Systems." This research used resources of the Keeneland Computing Facility at the Georgia Institute of Technology, which is supported by the National Foundation under Contract OCI-0910735.

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. Vetter, R. Glassbrook, J. Dongarra, K. Schwan, B. Loftis, S. McNally, J. Meredith, J. Rogers, P. Roth, K. Spafford, S. Yalamanchili, Keeneland: Bringing heterogeneous GPU computing to the computational science community, IEEE Computing in Science and Engineering 13 (2011) 90–95.

[6] M. Mohiyuddin, M. Hoemmen, J. Demmel, K. Yelick, Minimizing communication in sparse matrix solvers, in: Proceedings of the Conference on High Performance Computing Networking, Storage and Analysis (SC), New York, NY, USA, 2009, pp. 36:1–36:12.

[7] R. Barrett, M. Berry, T. Chan, J. Demmel, J. Donato, J. Dongarra, V. Eijkhout, R. Pozo, C. Romine, H. V. der Vorst, Templates for the Solution of Linear Systems: Building Blocks for Iterative Methods, 2nd Edition, SIAM, Philadelphia, PA, 1994.

[8] Z. Bai, D. Hu, L. Reichel, A Newton basis GMRES implementation, IMA Journal of Numerical Analysis 14 (1994) 563–581.

[9] E. Cuthill, J. McKee, Reducing the bandwidth of sparse symmetric matrices, in: Proceedings of the 1969 24th National Conference, New York, NY, USA, 1969, pp. 157–172.

[10] 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.

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