Mixed-Precision Orthogonalization Scheme and Adaptive Step Size for CA-GMRES on GPUs

Ichitaro Yamazaki, Stanimire Tomov, Tingxing Dong, and Jack Dongarra
University of Tennessee, Knoxville, U.S.A.

Abstract. We propose a mixed-precision orthogonalization scheme that takes the input matrix in a standard 64-bit floating-point precision, but accumulates its intermediate results in the doubled-precision. When the target hardware does not support the desired higher precision, we use software emulation. Compared with the standard orthogonalization scheme, we require about 8.5× more computation but a much smaller increase in communication. Since the computation is becoming less expensive compared to the communication on new and emerging architectures, the relative cost of our mixed-precision scheme is decreasing. Our case studies with CA-GMRES on a GPU demonstrate that using mixed-precision for this small but critical segment of CA-GMRES can improve not only its overall numerical stability but also, in some cases, its performance. We also study an adaptive scheme to dynamically adjust the step size of the matrix powers kernel. Our experiments on multiple GPUs show that a near optimal step size can be chosen based on the performance measurements from the first restart loop of CA-GMRES.

1 Introduction

On modern computers, communication (e.g., data movement through memory hierarchies) is becoming expensive compared to computation (i.e., floating point operations). It is critical to take this hardware trend into consideration when designing high-performance software for new and emerging computers. To this end, we previously studied a communication-avoiding variant [3] of the Generalized Minimum Residual method [4] (CA-GMRES) on multicore CPUs with multiple GPUs [6], focusing on the orthogonalization (Orth) and matrix powers kernels (MPK). We found that the Cholesky QR (CholQR) orthogonalization [5] obtains a superior performance based on the optimized BLAS-3 GPU kernels. Unfortunately, CholQR can be numerically unstable, and CA-GMRES may not converge even with reorthogonalization. We also found that depending on the matrix sparsity pattern, MPK can be slower than the standard algorithm due to the computational and/or communication overheads traded for reducing the

---

0 We thank Maho Nakata, Daichi Mukunoki, and the members of DOE “Extreme-scale Algorithms & Solver Resilience (EASIR)” for helpful discussions.
\( \hat{x} := 0 \) and \( v_1 := b / \| b \|_2 \).

repeat (restart-loop)

**Projection Subspace Generation on GPUs (inner-loop):**

for \( j = 1, s + 1, 2s + 1, \ldots, m \) do

MPK: Generate new vectors \( v_{k+1} := A v_k \)
for \( k = j, j + 1, \ldots, \min(j + s, m) \).

BOrth: Orthogonalize \( V_{j+1:s+1} \) against \( V_1:j \).

TSQR: Orthogonalize the vectors within \( V_{j+1:s+1} \).

end for

**Projected Subsystem Solution on CPUs (restart):**

Compute the solution \( \hat{x} \) in the generated subspace, which minimizes its residual norm.
Set \( v_1 := r / \| r \|_2 \), where \( r := b - A \hat{x} \).

until solution convergence do

End...
Even on one GPU, CA-GMRES can obtain a speedup of up to two [6]. Figure 1 shows the pseudocode of CA-GMRES\((s, m)\), where \(v_j\) and \(V_{j:k}\) are the \(j\)-th column and the submatrix consisting of the \(j\)-th through the \(k\)-th columns of \(V\), respectively, and the iteration is restarted after \(m+1\) basis vectors are generated.

### 3 Cholesky QR factorization

In this section, we use \(V_{1:s+1}^{(d)}\) to denote the local matrix of \(V_{1:s+1}\) on the \(d\)-th GPU, and \(n_g\) is the number of available GPUs. To orthogonalize the \(s+1\) vectors \(V_{1:s+1}\) (as for TSQR), CholQR first forms the Gram matrix \(B := V_{1:s+1}^{T} V_{1:s+1}\) through the matrix-matrix product \(B^{(d)} := V_{1:s+1}^{(d)T} V_{1:s+1}^{(d)}\) on the GPU, followed by the reduction \(B := \sum_{d=1}^{n_g} B^{(d)}\) on the CPU. Next, the Cholesky factor \(R\) of \(B\) is computed on the CPU. Finally, the GPU orthogonalizes \(V_{1:s+1}\) by a triangular solve \(V_{1:s+1}^{(d)} := V_{1:s+1}^{(d)} R^{-1}\). Hence, all the required GPU-GPU communication is aggregated into a pair of messages, while the GPU computation is based on BLAS-3. Hence, both intra and inter GPU communication can be optimized. Figure 2 shows these three steps of CholQR. Unfortunately, the condition number \(\kappa(B)\) of \(B\) is the square of \(\kappa(V_{1:s+1})\) [5]. This often causes numerical instability, especially in CA-GMRES, where even using the Newton basis [1], the vector \(v_j\) can converge to the principal eigenvector of \(A\), and \(\kappa(V_{1:s+1})\) can be large.

### 4 Mixed-Precision Orthogonalization Scheme

#### 4.1 Implementation

To improve the numerical stability of CholQR, we use the doubled-precision at the first two steps of CholQR, while the last step is in the standard precision. When the target hardware does not support a desired higher precision, software emulation is needed. For instance, double-double (dd) precision emulates the quadruple precision by representing each numerical value by an unevaluated sum of two double precision numbers, and is capable of representing the 106 bits precision, while the double-precision number is of 53 bits precision. There are two standard implementations [2] of adding two numerical values in dd-precision, \(a + b = \hat{c} + e\), where \(e\) is the round-off error; one satisfies the IEEE-style error bound \((e = \delta(a + b)\) with \(|\delta| \leq 2\epsilon_{dd}\) and \(\epsilon_{dd} = 2^{-105}\), and the other satisfies the weaker Cray-style error bound \((e = \delta_1 a + \delta_2 b\) with \(|\delta_1|, |\delta_2| \leq \epsilon_{dd}\). Table 3 lists the computational costs of the dd-arithmetics required by our mixed-precision dd-CholQR. The standard CholQR in double precision (d-CholQR) performs

<table>
<thead>
<tr>
<th>dd-multiply</th>
<th># of d-instructions</th>
<th>dd-addition</th>
<th># of d-instructions</th>
</tr>
</thead>
<tbody>
<tr>
<td>Add/Sub/Mul/FMA</td>
<td>Add/Sub/Mul/FMA</td>
<td>IEEE-style</td>
<td>Cray-style</td>
</tr>
<tr>
<td>dd-input</td>
<td>5</td>
<td>3</td>
<td>1</td>
</tr>
</tbody>
</table>

Fig. 3. Number of double-precision instructions in double-double operations.
about a half of its total flops at Step 1 and the other half at Step 3. Hence, using dd-precision for Steps 1 and 2, our dd-CholQR performs about 8.5× more computation than d-CholQR. On the other hand, the increase in communication is less significant; our intra-GPU communication is about the same, only writing the s-by-s output matrix in the dd-precision while reading the n-by-s input matrix in the d-precision (s ≪ n). We communicate twice more data between the GPUs (16a²Bytes with s ≈ 10), but with the same latency.

Since the Gram matrix is much smaller in its dimension than the coefficient matrix (s ≪ n), CholQR typically spends only a small portion of its orthogonalization time computing its Cholesky factor at Step 2. In addition, solving the triangular system with many right-hand-sides at Step 3 exhibits a high parallelism and can be implemented efficiently on a GPU. On the other hand, at Step 1, computing each element of the Gram matrix requires a reduction operation on n-length vectors. These inner-products (InnerProds) are communication-intensive and exhibit only limited parallelism. Hence, Step 1 often becomes the bottleneck, where standard implementations fail to obtain high-performance on the GPU. In our batched implementation of a matrix-matrix multiply (GEMM) to compute InnerProds, B := XᵀY, each thread block computes a partial product, B(i,j,k) := X(k,i)ᵀY(k,j), where X(k,i) and Y(k,j) are h-by-mₖ and h-by-nₖ blocks of X and Y, respectively.¹ Within the thread block, each of its nₜ threads computes its partial result in the local registries (see Figure 4 for an illustration, and Figure 5 for the pseudocode, where x(ℓ,j) is the (ℓ,j)-th element of X(k,i)). Then, each thread block performs the binary reduction of the partial results among its threads, summing n_r columns at a time using the shared memory to store nᵣ × (mₖ × n_r) numerical values. The final result is computed by another binary reduction among the thread blocks. Our implementation is designed to reduce the number of synchronizations among the threads while relying on the CUDA runtime and the parameter tuning to exploit the data locality. For the symmetric (SYRK) multiply, B := VᵀV, the thread blocks compute only a triangular part of B and reads V(k,j) once for computing a diagonal block.

¹ In the current implementation, the numbers of rows and columns in X and Y are a multiple of h, and multiples of mₖ and nₖ, respectively, where nₖ is a multiple of n_r.
4.2 Performance

Figure 6 compares the standard and mixed-precision InnerProds performance on different GPUs. Each GPU has a different relative cost of communication to computation, and on top of each plot, we show the ratio of the double-precision peak performance (Gflop/s) over the shared memory bandwidth (GB/s) (i.e., flop/B to obtain the peak). This ratio tends to increase on a newer architecture, indicating a greater relative communication cost. We tuned our kernel for each matrix dimension on each GPU in each precision (see the five tunable parameters $h$, $m_b$, $n_b$, $n_r$, and $n_t$ in Section 4.1), and the figure shows the optimal performance. Based on the memory bandwidth and the fixed number of columns in the figure, the respective peak performances of d-GEMM are 442, 625, and 720Gflop/s on M2090, K20c, and K40. Our d-GEMM obtained 29, 26, 28% of these peak performances and speedups of about 1.8, 1.7, and 1.7 over CUBLAS 5.5 on these GPUs. In addition, though it performs $16 \times$ more instructions, the gap between dd-InnerProds and d-InnerProds tends to decrease on a newer architecture, and dd-InnerProds is only less than four times slower on K20c.

Figure 7 shows the breakdown of d-CholQR orthogonalization time. Because of our efficient implementation of InnerProds, only about 30% of the orthogonalization time is now spent in d-InnerProds. As a result, while dd-InnerProds was about four times more expensive than d-InnerProds, Figure 8 shows that dd-CholQR is only about 1.7 or 1.4 times more expensive than d-CholQR when GEMM or SYRK is used for InnerProds, respectively. For dd-CholQR, the Cholesky factorization in dd-precision is computed using MPACK\(^2\) on the CPU.

\(^2\) http://mplapack.sourceforge.net
4.3 Case Studies with CA-GMRES

Figure 9 shows the normalized solution time of CA-GMRES with a single K20c for two sparse matrices from University of Florida Sparse Matrix collection\(^3\). Using dd-CholQR, even with the computationally expensive software emulation, the solution time was reduced not only because the reorthogonalization was avoided but also because CA-GMRES converged in fewer iterations.

5 Adaptive Step Size for Matrix Powers Kernel

Most of CA-GMRES implementations including ours [6] use the same step size \(s\) for MPK, BOrth, and TSQR, while the optimal \(s\) for MPK is typically smaller than that of BOrth or TSQR due to the computational and/or communication overheads associated with MPK. Instead of having a different \(s\) for MPK as an input, we design an adaptive scheme to dynamically adjust the step size \(\hat{s}\) of MPK based on the static input (i.e., the sparsity pattern of the coefficient matrix \(A\)) and the performance measurements from the first restart-loop of CA-GMRES.\(^4\) For this, we use the following performance model:

\[
MPK \text{ time} = \text{Inter-communication time} + \text{Kernel time},
\]

where we let

\[
\text{Inter-communication time} = \text{Latency} + \frac{\text{Communication volume}}{\text{Bandwidth}}, \quad \text{and}
\]

\[
\text{Kernel time} = \frac{\text{Flop count}}{\text{flop/s}} + \# \text{ of random data accesses } \times \text{Data access time},
\]

and “Kernel time” consists of the computation and inter-communication time. In our experiments, “Communication volume” and “Flop count” are computed

\(^3\) [http://www.cise.ufl.edu/research/sparse/matrices/](http://www.cise.ufl.edu/research/sparse/matrices/)

\(^4\) Our first loop is GMRES since the shifts for the Newton basis are not available.
based on the sparsity pattern of the coefficient matrix, while “# of random data accesses” is approximated by the aggregated number of non-local vector elements accessed by $MPK$. On the other hand, we computed “Latency,” “Bandwidth,” “flop/s,” and “Data access time” based on the measured time of the reduction for the dot-products, point-to-point communication for $SpMV$, $SpMV$, and data copy on the GPU, respectively. To generate the $s$ basis vectors, we invoke $MPK$ $s/\hat{s}$ times using the step size $\hat{s}$ before calling $BOrth$ and $TSQR$. Figure 10 shows the results for two sparse matrices from University of Florida Sparse Matrix collection on three Tesla M2090 GPUs.

6 Conclusion

We proposed a mixed-precision orthogonalization scheme to improve the numerical stability of CA-GMRES. Our case studies demonstrated that though it requires about $8.5\times$ more computation, using a higher-precision for this small but critical segment of CA-GMRES can improve not only its overall stability but also, in some cases, its performance. We also showed that the overhead of a higher-precision is decreasing on a newer architecture with an increasingly lower cost of computation. In this paper, we only studied the effects of a higher-precision on a single GPU. On multiple GPUs of a compute node, the performance of CA-GMRES depends more on the performance of the GPU kernels (i.e., intra-GPU communication) than the inter-GPU communication [6]. Hence, similar benefits of using a higher-precision are expected on the multiple GPUs. We will study its effects on a system with a greater communication latency (e.g., distributed GPUs or CPUs) where the improvement may be greater. We are also
studying the use of mixed-precision in eigensolvers where the orthogonality can be more crucial, and are writing an extended paper focusing on the numerical properties of our mixed-precision scheme [7]. Finally, it is of our interest to apply or extend recent mixed precision efforts (e.g., reproducible BLAS\(^5\) and precision tuning\(^6\)) for our studies. In this paper, we also studied an adaptive scheme to adjust the step size of MPK on multiple GPUs. Our MPK is currently optimized only for the inter-GPU communication which is relatively inexpensive on a node. We expect a greater benefit of the adaptive scheme (in term of time or memory) on a larger system with greater communication cost (e.g., a distributed system).

**References**


---

\(^5\) [http://www.eecs.berkeley.edu/~hdnguyen/rblas](http://www.eecs.berkeley.edu/~hdnguyen/rblas)