LAWRENCE
LIVERMORE
NATIONAL
LABORATORY

LLNL-TR-825909

Advances in Mixed Precision Algorithms: 2021 Edition

A. Abdelfattah, H. Anzt, A. Ayala, E. Boman, E. Carson, S. Cayrols, T. Cojean, J. Dongarra, R. Falgout, M. Gates, T. Gruetzmacher, N. Higham, S. Kruger, X. Li, N. Lindquist, Y. Liu, J. Loe, P. Luszczek, P. Nayak, D. Osei-Kuffuor, S. Pranesh, S. Rajamanickam, T. Ribizel, B. Smith, K. Swirydowicz, S. Thomas, S. Tomov, Y. Tsai, I. Yamazaki, U. M. Yang

August 18, 2021

This work performed under the auspices of the U.S. Department of Energy by Lawrence Livermore National Laboratory under Contract DE-AC52-07NA27344.

ECP-U-ST-RPT_2021_00239

Advances in Mixed Precision Algorithms: 2021 Edition

11/17/2021

Advances in Mixed Precision Algorithms: 2021 Edition

by the ECP Multiprecision Effort Team (Lead: Hartwig Anzt)

Ahmad Abdelfattah, Hartwig Anzt, Alan Ayala, Erik G. Boman, Erin Carson, Sebastien Cayrols, Terry Cojean, Jack Dongarra, Rob Falgout, Mark Gates, Thomas Grützmacher, Nicholas J. Higham, Scott E. Kruger, Sherry Li, Neil Lindquist, Yang Liu, Jennifer Loe, Piotr Luszczek, Pratik Nayak, Daniel Osei-Kuffuor, Sri Pranesh, Sivasankaran Rajamanickam, Tobias Ribizel, Barry Smith, Kasia Swirydowicz, Stephen Thomas, Stanimire Tomov, Yaohung M. Tsai, Ichi Yamazaki, Urike Meier Yang

LLNL-TR-825909

Preface

Over the last year, the ECP xSDK-multiprecision effort has made tremendous progress in developing and deploying new mixed precision technology and customizing the algorithms for the hardware deployed in the ECP flagship supercomputers. The effort also has succeeded in creating a cross-laboratory community of scientists interested in mixed precision technology and now working together in deploying this technology for ECP applications. In this report, we highlight some of the most promising and impactful achievements of the last year. Among the highlights we present are

- Mixed precision IR using a dense LU factorization and achieving a 1.8× speedup on Spock;
- Results and strategies for mixed precision IR using a sparse LU factorization;
- A mixed precision eigenvalue solver;
- Mixed Precision GMRES-IR being deployed in Trilinos, and achieving a speedup of 1.4× over standard GMRES;
- Compressed Basis (CB) GMRES being deployed in Ginkgo and achieving an average 1.4× speedup over standard GMRES;
- Preparing hypre for mixed precision execution;
- Mixed precision sparse approximate inverse preconditioners achieving an average speedup of 1.2×;
- Detailed description of the memory accessor separating the arithmetic precision from the memory precision, and enabling memory-bound low precision BLAS 1/2 operations to increase the accuracy by using high precision in the computations without degrading the performance;

We emphasize that many of the highlights presented here have also been submitted to peer-reviewed journals or established conferences, and are under peer-review or have already been published.

TABLE OF CONTENTS

1 Dense Linear Algebra **3**
- 1.1 The mix of precisions in a direct solver . . . 3
- 1.2 Details of implementation . . . 3
- 1.3 Experimental results . . . 4
- 1.4 Enabling Mixed Precision Iterative Refinement Solvers on Spock . . . 5

2 Eigen-Solvers **6**

3 Mixed Precision Sparse Factorizations **8**
- 3.1 Mixed Precision sparse LU and QR . . . 8
- 3.2 Mixed Precision sparse direct solvers . . . 9

4 Mixed Precision Krylov solvers **10**
- 4.1 Mixed Precision GMRES With Iterative Refinement . . . 10
  - 4.1.1 Convergence and Kernel Speedup for GMRES vs GMRES-IR . . . 10
  - 4.1.2 Convergence and Kernel Speedup for Preconditioned GMRES vs GMRES-IR . . . 11
- 4.2 Compressed Basis Krylov Solvers . . . 12
- 4.3 s-step Lanczos and CG . . . 16
- 4.4 Arnoldi-QR MGS-GMRES . . . 18
- 4.5 Alterative Approaches . . . 19

5 Mixed Precision Sparse Approximate Inverse Preconditioning **19**

6 Mixed Precision Strategies for Multigrid **20**
- 6.1 Mixed-precision algebraic multigrid . . . 20
- 6.2 Enabling Mixed-Precision capabilities in hypre . . . 23
- 6.3 Current status and future plans: . . . 24

7 Mixed Precision FFT **25**
- 7.1 Data compression to reduce communication . . . 25
- 7.2 Approximate FFTs with speed-to-accuracy trade-offs . . . 26
- 7.3 Towards mixed-precision MPI . . . 27

8 Memory Accessor **28**

9 Software featuring mixed- and multiprecision functionality **29**
- 9.1 Ginkgo . . . 29
- 9.2 Kokkos Core, Kokkos Kernels, and Trilinos Additions . . . 31
- 9.3 MAGMA . . . 31
- 9.4 heFFTe . . . 31
- 9.5 PLASMA . . . 31
- 9.6 PETSc . . . 31
- 9.7 hypre . . . 32

1. Dense Linear Algebra

Both the Cholesky and LU decompositions factor a square matrix A of dimension n in $O(n^3)$ operations in order to solve the system $AX = B$, where B is the concatenation of *nrhs* right-hand sides (RHS) and X is a matrix that contains the solution for each RHS. These two decompositions are well-known to be costly and many efforts focus on improving the performance while keeping the same accuracy. However, in some cases, we may experience a loss of accuracy. For that, an iterative method can be used to recover the solution. One such method is named *iterative refinement*. As presented in [1] Section 2.2, the iterative refinement is a long-standing method that Wilkinson was using to improve the accuracy of the solution. However, all computations were done using a single precision.

Current hardware offers the capability to compute using single-precision that can give two times speedup over the double-precision. Morever, on GPU, computation in half-precision can exceed four times speedup over the double-precision. Therefore, in this section, we describe how the use of multiple precisions, namely *mixed-precision*, in two direct solvers, posv (Cholesky) and gesv (LU), can improve the performance compared with their classical version while reaching the same accuracy.

1.1 THE MIX OF PRECISIONS IN A DIRECT SOLVER

Given a working-precision, the idea of using mixed-precision relies on the use of a lower precision to accelerate the computation at the key steps of an algorithm and use the iterative refinement method to reach the desired accuracy.

Since the mixed-precision LU case was presented in [1] Section 2.2, we focus our explanation on the Cholesky decomposition. The algorithm presented in algorithm 1 is almost the same, except that the decomposition of A is LL^T and so the call to gemm is replaced by hemm in Line 7. In substance, after computing the initial guess, X_0, the algorithm iterates until reaching the desired accuracy. This means, at iteration k, a correction term Z_k is computed based on the residual of the previous iteration.

Algorithm 1 posvMixed(A,B)

```
 1: Input: $A \in \mathbb{C}^{n \times n}$
 2: Input: $B \in \mathbb{C}^{n \times nrhs}$
 3: $LL^T \leftarrow A$                                  ▷ potrf $\epsilon_s$
 4: Solve $LY = B$                                      ▷ trsm $\epsilon_s$
 5: Solve $L^T X_0 = Y$                                 ▷ trsm $\epsilon_s$
 6: for k=1,2,... do
 7:     $R_k = B - AX_{k-1}$                            ▷ hemm $\epsilon_d$
 8:     Solve $LY = R_k$                                ▷ trsm $\epsilon_s$
 9:     Solve $L^T Z_k = Y$                             ▷ trsm $\epsilon_s$
10:     $X_k \leftarrow X_{k-1} + Z_k$                  ▷ $\epsilon_d$
11:     Check convergence
12: end for
```

We observe that this algorithm is using two working precisions: the double-precision and the single-precision, denoted ϵ_d and ϵ_s, respectively. Since the most costly part is the factorization of the matrix A, it must be performed in lower precision and so are the associated triangular solves. Only the computation of the residual and the update of the solution are done in higher precision.

1.2 DETAILS OF IMPLEMENTATION

We did a first distributed implementation of posvMixed in SLATE [2]. This library is using a 2D block cyclic tile distribution of the data and for simplicity, we consider in the following the case of square tiles of size nb. We compared the performance of posvMixed with the original posv also implemented in SLATE. This comparison revealed that posvMixed may be slower than posv, depending on the number of iterations needed to converge. Note that we also observed the same behavior for our implementation of gesvMixed.

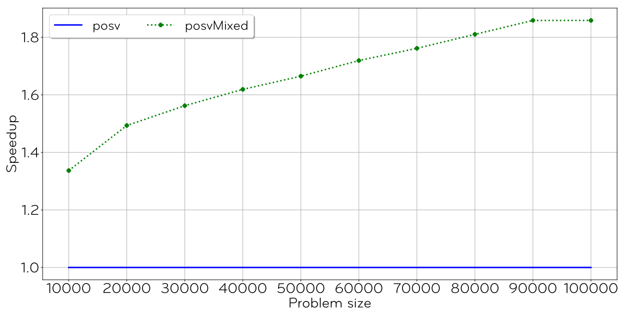
A breakdown of the execution showed that lots of time was spent in BLAS-2 calls for computing the solution (trsm) and residual (gemm or hemm). Each of these routines are implemented assuming that the computation is performed where the output data is located. In other words, if we consider the gemm operation, which is $C = \alpha A \times B + \beta C$ with $\alpha, \beta \in \mathbb{C}$, each contribution $A_{ik} \times B_{kj}$ is performed where the tile C_{ij} is located. Therefore, the tiles A_{ik} and B_{kj} are moved to the process that owns C_{ij}. Thus, at each iteration of the iterative refinement, the computation of the residual R_k and the correction term Z_k involve the communication of tiles of A and B, which corresponds to a volume of communication proportional to $nb^2 + nb \times nrhs$.

We thus implemented a variant of each BLAS-2 operation where the tiles B_{kj} and C_{ij} are communicated where A_{ik} is located. By doing so, the volume of communication is reduced to be proportional to $2nb \times nrhs$ plus one extra communication to place the output data correctly ($nb \times nrhs$). In [3], authors show that for a small number of RHS, the reduction of the volume of communication during the computation of the triangular solve is beneficial and can lead to up to 12× speedup on large problem sizes. We extended this approach and we denote these variants as trsmA, hemmA, and gemmA. We next show the gain obtained by replacing all BLAS-2 calls with their new version.

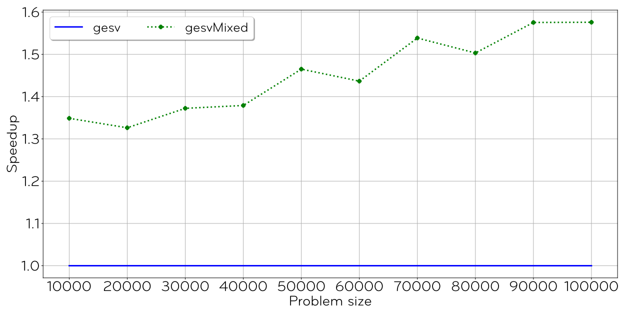
1.3 EXPERIMENTAL RESULTS

We did our experiments on Summit supercomputer using four nodes and six MPI processes per node. Each MPI process is using seven OpenMP threads. We generate problems ranging from 10 000 to 100 000 and measure the time it takes to compute the solution. We consider the classical algorithms as the references and we show in the following the speedup obtained by using mixed-precision approach in Cholesky and LU solver.

Figure 1a shows that using single-precision for the factorization of A and doing a few iterations gives us more than 1.8× speedup on the largest problems considered. We observe that the speedup tends to converge a 2× speedup as expected. However, due to the small number of iterations, this theoretical bounds cannot be reached. Figure 1b displays the same general behavior. The speedup obtained by using gesvMixed increases with the problem size. But it only reaches almost 1.6× speedup. Moreover, we see some variations in the performance. For example, a problem size of 70 000 gives us better performance than a problem size of 80 000. This is explained by the number of iterations needed to converge. In the former case, seven iterations are needed while for the latter, 10 iterations are necessary to reach the same accuracy. Note that for small problem size, the number of iterations was only three and this number increases with the size of the problem.



(a) posv



(b) gesv

Figure 1: Performance comparison between posvMixed and the reference posv (1a), and between gesvMixed and the reference gesv (1b), on four nodes, 24 processes, when the size of the generated problems increases.

This last observation emphases the need to focus our attention on the number of iterations. With very large problems, these iterations may become the issue. One alternating would be to port the code on GPU while another would be to consider a different approach like GMRES-IR as presented in [1] in Section 2.3.

1.4 ENABLING MIXED PRECISION ITERATIVE REFINEMENT SOLVERS ON SPOCK

The Spock system at ORNL is very similar to the Frontier exascale supercomputer planned for delivery in 2021, and therefore an important test-bed for porting and preparing software packages for exascale. Furthermore, the Spock nodes feature AMD MI100 GPUs, each with a peak performance of 11.5 TFLOPs in double-precision (FP64), 46.1 TFLOPs in single-precision (FP32), and 184.6 TFLOPs in half-precision (FP16), thus making the system a potential target to benefit from mixed precision solvers.

Indeed, under certain assumptions, mixed-precision iterative refinement techniques can solve $Ax = b$ in FP64 accuracy significantly faster than direct FP64 solvers. A speedup of up to 4× has been achieved using FP16/FP32 Tensor Cores on NVIDIA GPUs [4, 5]. To enable these types of solvers on Spock for AMD GPUs, we ported the entire MAGMA library to the HIP runtime through code generation, and further did some architecture-specific optimizations for BLAS [6] and the mixed-precision iterative refinement solvers.

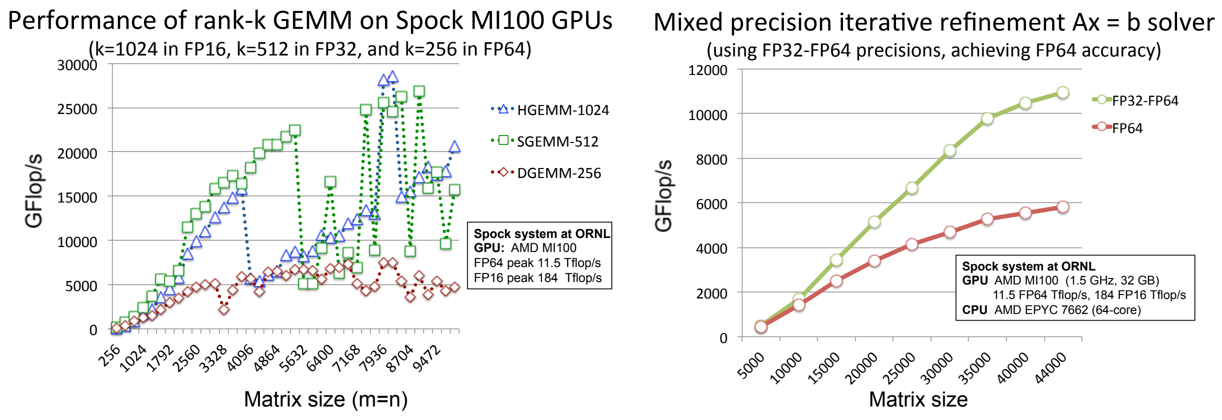


Figure 2: Left: Performance of the rank-k GEMMs typically used in the LU factorizations on GPUs. **Right:** Performance comparison (computed as $(\frac{2}{3}n^3 + \frac{3n^2}{2} - \frac{n}{6})$ / time in GFlop/s) of the FP64 solver and the mixed-precision FP32-FP64 solver in MAGMA, achieving FP64 accuracy. All experiments use ROCm 4.2.

Figure 2-left shows the performance from the MAGMA xGEMM benchmark in the FP64, FP32, and FP16 arithmetic, respectively, on the MI100 GPUs in Spock. These are the rank-k GEMM updates typically used in the MAGMA LU factorization (k=1024 in FP16, k=512 in FP32, and k=256 in FP64). The performance of the rank-k updates represents an upper-bound for the overall performance of the factorization, and hence of the resulting solvers. Note that the currently achievable performance on these rank-k GEMMs (using ROCm 4.2) is lower than the theoretical peaks. Also, there are a lot of variations in the performance, e.g., in the range of 10 to 30 TFLOPs for both FP32 and FP16. We observe that the rank-k FP16 GEMMs are about the same performance as the FP32 ones. Because of this, we developed the mixed-precision iterative refinement using just FP32-FP64 arithmetic.

Figure 2-right compares the performance of the FP64 solver and the mixed-precision FP32-FP64 solver. Both solvers achieve the same FP64 accuracy, but the mixed-precision one is approximately two times faster, as expected. The mixed-precision solver uses from 3 iterations (on the smallest problem) to 16 iterations (for the largest problem) to achieve FP64 accuracy. These results are on random matrices. The rate of execution is computed as $\frac{2n^3}{3} + \frac{3n^2}{2} - \frac{n}{6}$ over the execution time for both solvers, so higher execution rate illustrates how much faster is the mixed-precision solver (i.e., close to 2× in this case).

These results can be reproduced using the recent MAGMA 2.6 release. The only change required is to replace the hybrid factorization (that is using both CPUs and GPU) in the solver with the GPU-only factorization [7], both available in the MAGMA 2.6 release.

2. Eigen-Solvers

The original SICE algorithm by Dongarra el al. [8, 9, 10] starts with the eigenpair λ, x and an approximate eigenpair $\lambda + \mu$, $x + \tilde{y}$ both of each are suitably nearby. In the multi-precision algorithms' context, that distance between the eigenpairs is on the order of unit roundoff of the lower precision representation. Given the original formulation of the eigenproblem, we arrive at:

$$A(x + \tilde{y}) = (\lambda + \mu)(x + \tilde{y}) \tag{1}$$

By assuming that the vector x is normalized in the infinity norm: $|x|_\infty = 1 \equiv x_s$, we reduce the dimensionality of the solution space by one and require that $\tilde{y}_s = 0$. Rearranging terms in Eq. (1) gives us:

$$(A - \lambda I)\tilde{y} - \mu x = \lambda x - Ax - \mu\tilde{y} \tag{2}$$

Note that the last term is the second order term for the error as a function of λ and x. By simplifying the equation, we introduce a new vector y that we define as follows:

$$y^\intercal \triangleq (\tilde{y}_1, \tilde{y}_2, \ldots, \tilde{y}_{s-1}, \mu, \tilde{y}_{s+1}, \ldots, \tilde{y}_{n-1}, \tilde{y}_n) \tag{3}$$

This allows y to encode the information from both μ and $\tilde{y}$. By so doing, Eq. (2) now gets simplified into the following:

$$By = r + y_s\tilde{y} = r + \mu\tilde{y} \tag{4}$$

where $r = \lambda x - Ax$ is the residual vector of λ and x and B is the matrix $A - \lambda I$ with column s replaced by $-x$.

We can also view the iteration that refines the eigenpair from the lower to the higher precision as the Newton's method. In particular, by setting $v = \binom{x}{\lambda}$ we can be formulate the eigenvalue problem as:

$$f(v) \equiv \begin{bmatrix} Ax - \lambda x \\ e_s^\intercal x - 1 \end{bmatrix} = 0 \tag{5}$$

where e_s is the s-th column of the identity matrix of size n. The Newton's method then solves the linear system of the Jacobian J matrix:

$$J\begin{pmatrix} \tilde{y} \\ \mu \end{pmatrix} = \begin{bmatrix} A - \lambda I & -x \\ e_s^\intercal & 0 \end{bmatrix}\begin{bmatrix} \tilde{y} \\ \mu \end{bmatrix} = \begin{bmatrix} r \\ 0 \end{bmatrix} = f(v) \tag{6}$$

Expanding it, we arrive at Eq. (2) without the second-order term:

$$(A - \lambda I)\tilde{y} - \mu x = r \tag{7}$$

This is the basic idea of the SICE algorithm: by iteratively solving Eq. (4) we obtain the correction to both the eigenvalue and to the corresponding eigenvector. The original algorithm uses Schur decomposition and applies two steps of the Givens rotation in order to solve Eq. (4). For any real matrix A, there exists an orthogonal matrix Q and an upper quasi-triangular matrix U, such that

$$A = QUQ^\intercal \tag{8}$$

where U is upper quasi-triangular with some 2×2 diagonal blocks arising from the complex conjugate eignevalue pairs. Here, we define $Z_\lambda \equiv Z - \lambda I$ and $z_{\lambda s} \equiv Z_\lambda e_s = (Z - \lambda I)e_s$. By rewriting Eq. (4), we get:

$$[A_\lambda - (x + a_{\lambda s})e_s^\intercal]y = (A + ce_s^\intercal)y = r + y_s\tilde{y} \tag{9}$$

where $c = -x - a_{\lambda s}$. Using the Schur decomposition $A = QUQ^\intercal$ of Eq. (8), we have:

$$Q(U_\lambda + Q^\intercal ce_s^\intercal Q)Q^\intercal y = r + y_s\tilde{y} \tag{10}$$

$$(U_\lambda + df^\intercal)Q^\intercal y = Q^\intercal g \tag{11}$$

Algorithm 2 SICE algorithm

```
1: Input: Matrix $A \in \mathbb{R}^{n\times n}$. An approximate eigenvalue $\lambda$ and the corresponding eigenvector $x$. $\text{iter}_{\max}$ denotes the maximum number of iterations.
2: Output: Refined eigenvalue $\lambda$ and its eigenvector $x$.
3: function $[\lambda, x] \leftarrow$ SICE$(A, \lambda, x, iter)$
4:
5:     $[Q,U] \leftarrow \text{schur}(A)$          ▷ obtain Schur decomposition $A = QUQ^\intercal$, $QQ^\intercal = I$.
6:     $[m,s] \leftarrow max(abs(x)); x \leftarrow x/m$          ▷ Normalizing $x$ so that $\|x\|_\infty = s_x = 1$.
7:     for $i$ in $1 : \text{iter}_{\max}$ do
8:
9:             $r \leftarrow \lambda x - Ax$
10:            $c \leftarrow -x - a_{\lambda s}$
11:            $d \leftarrow Q^\intercal c$
12:            $f^\intercal \leftarrow Q(s,:) = e_s^\intercal Q$          ▷ $s$-th row of $Q$.
13:            $\bar{U}_\lambda \leftarrow Q_1(U - \lambda I); \bar{d} \leftarrow Q_1 d = \|d\|_2 e_1$          ▷ Givens rotations $Q_1$ from Eq. (12)
14:            $\bar{U}_\lambda \leftarrow \bar{U}_\lambda + \bar{d}(1) f^\intercal$
15:            $\bar{U}_\lambda \leftarrow Q_2 \bar{U}_\lambda$          ▷ Givens rotations $Q_2$ to introduce upper triangular form.
16:            Solve the triangular system $\bar{U}_\lambda z = Q_2 Q_1 Q^\intercal r$
17:            $y \leftarrow Qy$
18:            $\lambda \leftarrow \lambda + y(s)$          ▷ Update eigenvalue.
19:            $y(s) \leftarrow 0$          ▷ Set $y(s)$ to 0.
20:            $x \leftarrow x + y$          ▷ Update eigenvector.
21:            if desired accuracy is reached then
22:                break
23:            end if
24:        end
25:    end for
26: end
27: end function
```

where $d = Q^\intercal c$, $f^\intercal = e_s^\intercal Q$ and $g = r + y_s \tilde{y}$. Matrix $d \times f^\intercal$ constitutes a rank-1 update. Then two steps of Givens rotation are introduced: the first one Q_1 is constructed so that

$$Q_1 d = (P_2 P_3 \dots P_n) d = \gamma e_1 \quad \text{where } \gamma = \|d\|_2 \tag{12}$$

and P_i is the rotation in $(i-1, i)$ plane that eliminates the i-th component in $P_{i+1} \dots P_n d$. We also have:

$$Q_1(U_\lambda + d f^\intercal) = Q_1 U_\lambda + \gamma e_1 f^\intercal \tag{13}$$

The transformation Q_1 introduces one more nonzero element in the subdiagonal direction of U_λ. The new rank-one update $\gamma e_1 \times f^\intercal$ has nonzero elements only in the first row, which preserves the original structure. The second step of Givens rotation Q_2 can be applied subsequently in order to obtain the upper triangular form $\bar{U}_\lambda = Q_2 Q_1 (U_\lambda + d \times f^\intercal)$ in

$$\bar{U}_\lambda Q^\intercal y = Q_2 Q_1 Q^\intercal g \tag{14}$$

The triangular solve requires $O(n^2)$ operations while the remaining steps of the iteration are only $O(n)$. This procedure is shown in Algorithm 2.

Another issue is that treating each eigenpair independently, their orthogonality might not be maintained to a satisfiable level. The worst case scenario occurs when they all might converge to the same eigenpair which would occur for suitable clustering of the eigenvalues. However, it is easy to reorthogonalize with a number of techniques that fit the required accuracy and may depend on the eigenspecturm's properties. As a practical matter, we found that in our cases it is sufficient to reorthogonalize after the refinement process is finished. By so doing, each iteration would not speed up the convergence. The computation of orthogonality estimate $I - X^\intercal X$ also lets us detect if the pairs converged to the same eigenvector.

The system we used to test the eigensolver had two sockets of Intel(R) Xeon(R) CPU E5-2650v3. We show the profiling results from the PLASMA experiments in Figure 3. PLASMA was used in a CPU-only mode and no GPUs were used in the system and more details are provided in §9.5. The symmetric input matrix

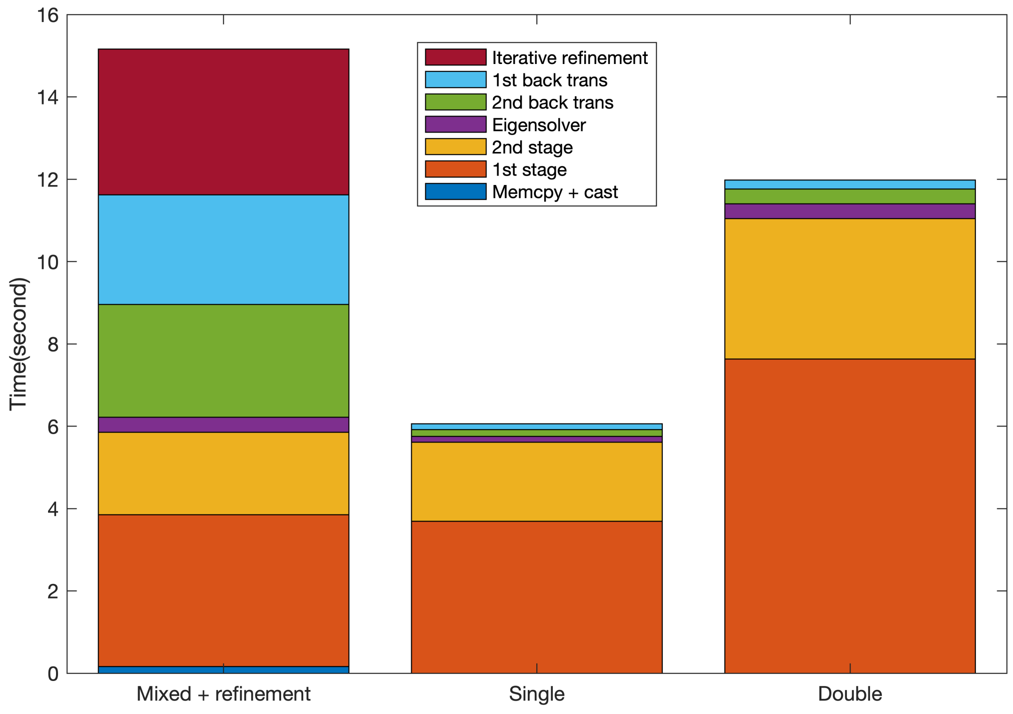


Figure 3: PLASMA CPU execution times and their detailed breakdowns for matrix of size $n = 10000$ and with 32 eigenpairs requested.

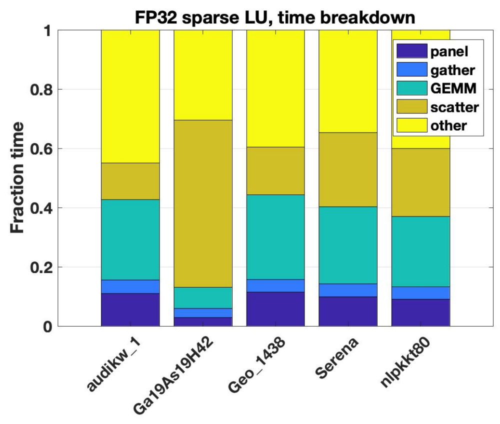
had size $n = 10000$. The three stacked bars represent the breakdown of time from mixed-precision with refinement, single-precision, and double-precision from the two-stage algorithm, respectively. The time for single precision is about half of that of double precision and each of the components take proportionally the same time for both precisions. The mixed-precision algorithm is slower than double precision in this setup because of the requirement of explicitly forming the transformation matrices from the first and second stages. They also take much more time compared to the double precision algorithm, which only applies transformations to the eigenvectors.

3. Mixed Precision Sparse Factorizations

3.1 MIXED PRECISION SPARSE LU AND QR

Similar to dense LU and QR factorizations, a large fraction of the computation lies in the Schur complement updates throughout the elimination steps. In the dense case, much of the work in the Schur complement update can be realized in terms of GEMM operations. However, in the sparse case, each Schur complement update usually follows three steps: 1) gather the values from sparse data structures into contiguous memory, 2) perform GEMM operation, 3) scatter the output of GEMM into destination sparse data structures.

For the dense case the main benefit comes from accelerated GEMM speed. But in the sparse case, GEMM is only one part of the three steps above. Furthermore, the dimensions of the GEMM kernel calls is generally smaller and of non-uniform size throughout factorization. Therefore, the speed gain from GEMM alone is limited. We will need to design new schemes to enhance overlap of GEMM computation with gather/scatter operations. In Figure 4 we show the time breakdown of various steps of sparse LU factorization in SuperLU_DIST and the time comparision of using FP32 vs. FP64. These are measured times for five real matrices of dimension on the order of 1 million or so. As can be seen, depending on the matrix sparsity structure, the fraction of time in GEMM varies, and usually is less than 50% (left plot). Because of this, the Tensor Core version of GEMM calls led to only less than 5% speedup for the whole sparse LU. When comparing FP32 with the FP64 versions, we observed about 50% speedup with the FP32 version (right plot).



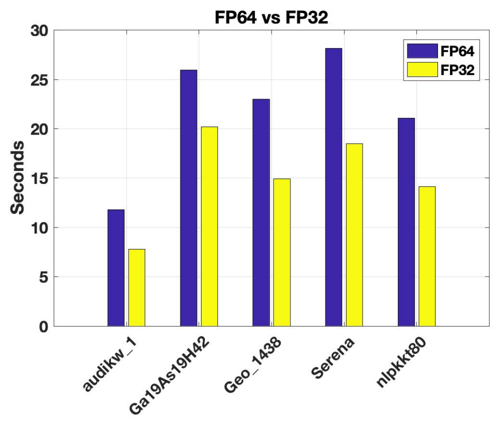


Figure 4: Left: tme breakdown of various steps of FP32 sparse LU in SuperLU_DIST, "Other" mostly consists of MPI communication. **Right:** Comparision of the sparse LU time between the FP32 and FP64 versions. All are measured on 10 Summit nodes with 6 MPI tasks and 6 GPUs per node.

3.2 MIXED PRECISION SPARSE DIRECT SOLVERS

The simplest mixed precision sparse direct solver is to use lower precision for the expensive LU and QR factorizations, and higher precision in the cheap residual and solution update in iterative refinement (IR). We recall the IR algorithm using three precisions in Algorithm 3 [11, 12]. This algorithm is already available in **x**GERFSX functions in LAPACK.

The following three precisions are used:

- ε_w is the working precision used to store the input data A and b. It is the lowest precision used in the solver, and is the desired precision for the output.
- ε_x is the precision used to store the computed solution $x^{(i)}$. We require $\varepsilon_x \leq \varepsilon_w$, possibly $\varepsilon_x \leq \varepsilon_w^2$ if necessary for componentwise convergence.
- ε_r is the precision used to compute the residuals $r^{(i)}$. We usually have $\varepsilon_r \ll \varepsilon_w$, typically being at least twice the working precision ($\varepsilon_r \leq \varepsilon_w^2$).

Algorithm 3 Three-precisions Iterative Refinement for Direct Solvers

```
1: Solve $Ax^{(1)} = b$ using the basic solution method (e.g., LU or QR)    ▷ ($\varepsilon_w$)
2: $i = 1$
3: repeat
4:     $r^{(i)} \leftarrow b - Ax^{(i)}$    ▷ ($\varepsilon_r$)
5:     Solve $A\,dx^{(i+1)} = r^{(i)}$ using the basic solution method    ▷ ($\varepsilon_w$)
6:     Update $x^{(i+1)} \leftarrow x^{(i)} + dx^{(i+1)}$    ▷ ($\varepsilon_x$)
7:     $i \leftarrow i + 1$
8: until $x^{(i)}$ is "accurate enough"
9: return $x^{(i)}$ and error bounds
```

With the above setup and adaptive adjustment of ε_x and ε_r, the algorithm converges with small normwise error and error bound if the normwise condition number of A does not exceed $1/(\gamma(n)\varepsilon_w)$. Similarly, the algorithm converges with small componentwise error and error bound if the componentwise condition number of A does not exceed $1/(\gamma(n)\varepsilon_w)$. Moreover, this IR procedure can return to the user the reliable error bounds both normwise and componentwise. The error analysis in [11] should all carry through to the sparse cases.

We implemented Algorithm 3 in SuperLU_DIST. The following two precisions are used:

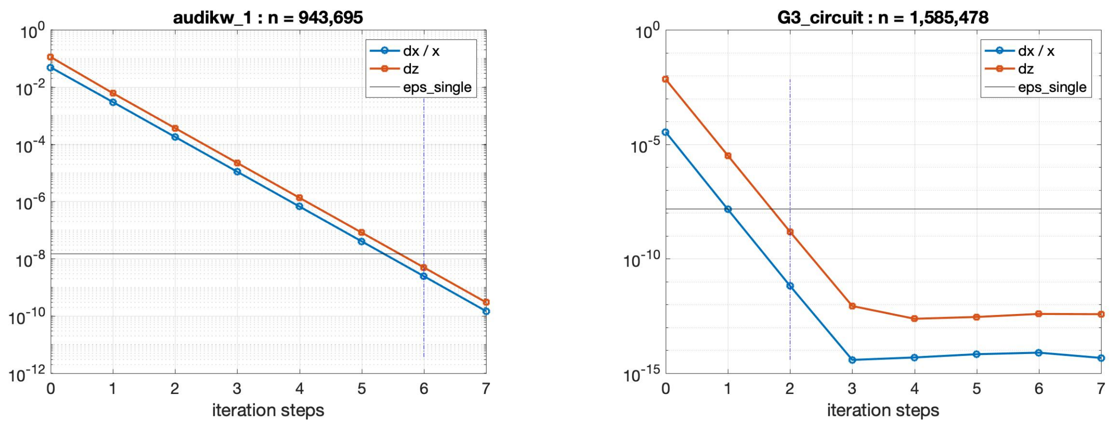


Figure 5: Convergence history of Algorithm 3 when applied to two sparse linear systems. "dx/x" is the normwise error term, "dz" is the componentwise error term. In each plot, the vertical line corresponds to the IRR steps taken when using our stopping criteria.

- $\varepsilon_w = 2^{-24}$ (IEEE-754 single precision), $\varepsilon_x = \varepsilon_r = 2^{-53}$ (IEEE-754 double precision)

In Figure 5, we show the convergence history of two matrices, in both normwise and componentwise corrections. The iterative refinement time is usually under 10% of the factorization time. So, overall, the mixed-precision speed is still faster than using FP64 all around. The experimental code is already available in the github branch: `https://github.com/xiaoyeli/superlu_dist/tree/Mixed-precision`. Our future work is to develop the reliable error bounds to be returned to the users, and investigate the use of even lower precision, such as bfloat16, in factorization.

4. Mixed Precision Krylov solvers

4.1 MIXED PRECISION GMRES WITH ITERATIVE REFINEMENT

Following are highlights from the paper "Experimental Evaluation of Multiprecision Strategies for GMRES on GPUs" [13], from the Sandia Labs team. We consider the algorithm GMRES with iterative refinement (GMRES-IR) for sparse nonsymmetric linear systems $Ax = b$. (See [14, 15, 16] for related work.) For GMRES-IR (Algorithm 4), we run GMRES in *single precision* (fp32) and then "refine" the algorithm at each restart by seeding the next GMRES run with a right-hand-side vector that has been computed in *double precision* (fp64). To do so, we maintain both double and single precision copies of the matrix A in memory. (Reported solve times do not include time to make the single precision copy of A.) Our experiments choose b to be a vector of ones and x_0 to be all zeros. We implement the solver in the Belos [17] linear solvers package of the Trilinos [18] software library. The solvers' linear algebra backend employs the Kokkos [19] and Kokkos Kernels libraries, which provide portable, optimized linear algebra operations for GPUs. For full implementation details, please see [13], Section IV. *The key contribution of this work is to evaluate this algorithm that shows promise in theory on a hardware that is designed to do well when using lower precision computation.* All experiments that follow are run on a node equipped with a Power 9 CPU and a Tesla V100 GPU. For GMRES-IR, the inner fp32 solver is always run until it has completed $m = 50$ iterations. We restart both double precision GMRES(m) and GMRES-IR after each run of $m = 50$ iterations. (At this point in GMRES-IR, we check for convergence.) Solvers are run to a relative residual convergence tolerance of 1e−10.

4.1.1 *Convergence and Kernel Speedup for GMRES vs GMRES-IR*

We consider BentPipe2D1500, a 2D convection-diffusion problem with n = 2,250,000. The underlying PDE is strongly convection-dominated, so the matrix is ill-conditioned and highly non-symmetric. We compare GMRES in all single precision, GMRES in all double precision, and GMRES-IR. Convergence plots are in

Algorithm 4 GMRES-IR

```
1: $r_0 = b - Ax_0$ [double]
2: for $i = 1, 2, \ldots$ until convergence: do
3:     GMRES($m$) solves $Au_i = r_i$ for correction $u_i$ [single]
4:     $x_{i+1} = x_i + u_i$ [double]
5:     $r_{i+1} = b - Ax_{i+1}$ [double]
6: end for
```

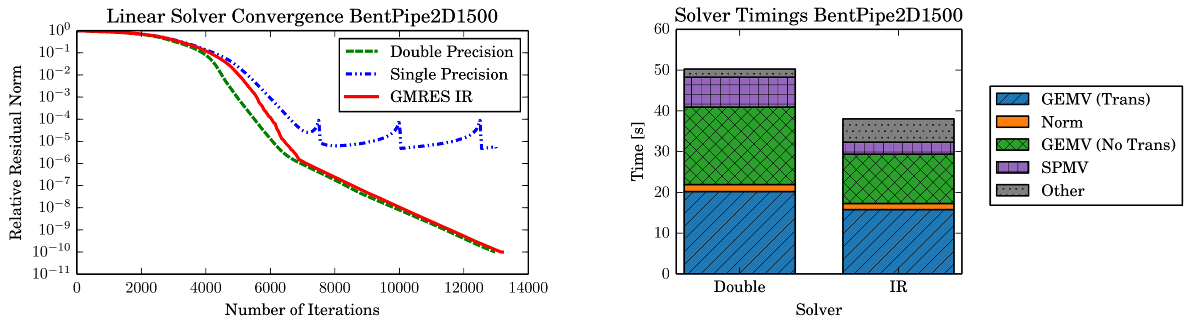


Figure 6: Convergence (left) and solve times (right) for a convection-diffusion problem (no preconditioning). We test GMRES in single precision, GMRES in double precision, and GMRES-IR (Alg. 4). Solve times on the right are broken down into specific kernels.

Figure 6 (left). The fp32 solver reaches a minimum relative residual norm of about 4.7e−6, and the fp64 solver needs 12,967 iterations to converge to 1e−10. GMRES-IR needs 13,150 total iterations to converge. Notice that *the convergence of the mixed precision version of the solver follows the double precision version closely*.

Figure 6 (right) shows the solve times and speedup of the GMRES double and IR solvers, split over different kernels. The bar segment labeled "other" indicates time solving the least squares problems and performing other non-GPU operations in GMRES. For GMRES-IR, it also includes computation of the new residual in double precision. GMRES-IR gives 1.32× speedup over the solve time of GMRES double. The two GEMV kernels give 1.28 to 1.57× speedup, but the SpMV gives a spectacular 2.48× speedup! This occurs due to near-perfect L2 cache reuse for the right-hand side vector with SpMV float, while there is a high L2 cache miss rate for SpMV double. We develop a model for SpMV speedup in [13], Section V-D.

4.1.2 Convergence and Kernel Speedup for Preconditioned GMRES vs GMRES-IR

Next we compare three preconditioning options. The matrix is a 2D Laplacian over a stretched grid with 1500 grid points in each direction. We apply a polynomial preconditioner [20] of degree 40, using a) GMRES-fp64 with fp64 preconditioning, b) GMRES-fp64 with fp32 preconditioning, and c) GMRES-IR with fp32 preconditioning. Here "fp32 preconditioning" indicates that the polynomial is both computed and applied in single precision. Figure 7 (left) demonstrates that, just as before, the problems with fp32 preconditioning converge very similarly to GMRES in all fp64. Figure 7 (right) shows solve times for all three configurations. The "other" portion of each bar indicates time spent in dense matrix operations, vector additions for the polynomial, and computation of double-precision residuals in GMRES-IR. Unlike the previous example where solve time was dominated by orthogonalization, polynomial preconditioning shifts the cost toward the sparse matrix-vector product. For this problem, the SpMV comprises 64% of the total solve time in fp64, so the improvement in SpMV time provides 32% of the ultimate speedup in GMRES-IR. Ultimately, GMRES-IR gives 1.58× speedup over GMRES double and 1.08× speedup over simply preconditioning in float. For further results on a set of test problems from SuiteSparse, please see [13], Section V-G.

We believe that the following concepts will extend to many other preconditioners: a) The convergence of an fp64 GMRES solver does not necessarily suffer from using an fp32 preconditioner instead of an fp64 preconditioner; b) If using an fp32 preconditioner does not degrade the convergence of the GMRES, it will

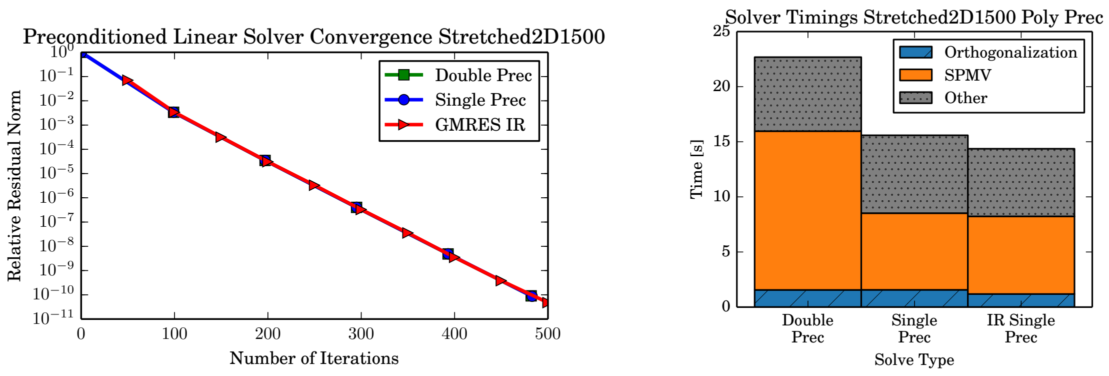


Figure 7: Convergence (left) and solve times (right) for a Laplacian on a stretched grid with a degree 40 polynomial preconditioner. We tested i) fp64 GMRES with fp64 polynomial [Double Prec], ii) fp64 GMRES with fp32 polynomial [Single Prec], and iii) GMRES-IR with an fp32 polynomial [IR Single Prec].

typically improve solve time over using the same preconditioner in fp64; and c) Preconditioning allows users to take advantage of kernels that have large speedup in lower precisions. We believe that GMRES-IR will become a key iterative linear solvers algorithm for using low precision hardware while maintaining the double precision accuracy required by applications.

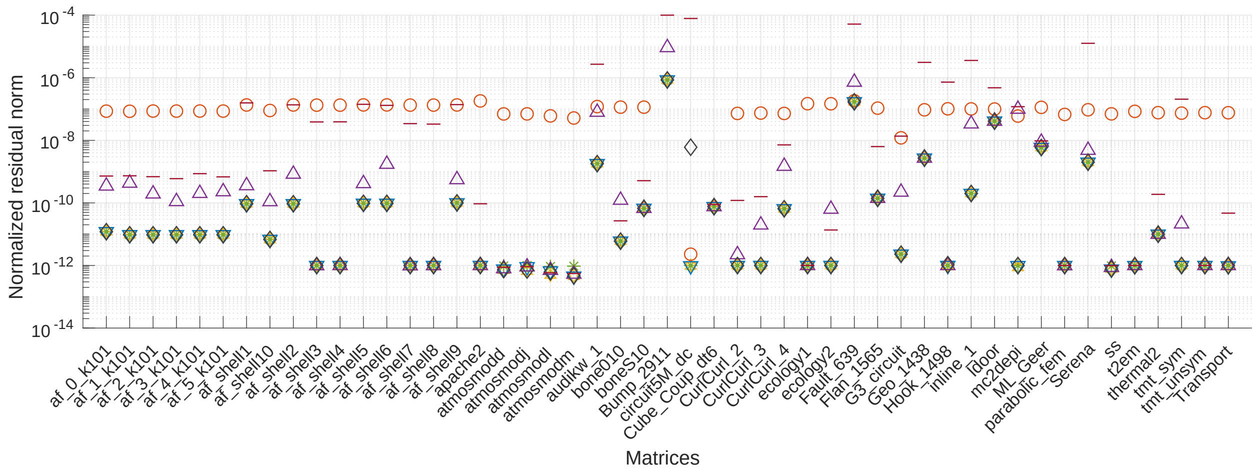
4.2 COMPRESSED BASIS KRYLOV SOLVERS

The motivation for compressing the basis vectors in iterative Krylov solvers comes from the observation that the performance of Krylov solvers is on virtually all hardware architectures limited by the memory access speed and the communication. A strategy to mitigate this problem is to use the memory accessor presented in Section 8 to compress the Krylov basis vectors for main memory operations. The simplest compression strategy in this context is to convert the double precision values in the Krylov basis vectors to a lower precision format. We have to accept that this conversion introduces perturbations to the Krylov basis vectors, and the methods may suffer in terms of convergence. The hope is that the convergence delay introduced by the perturbations can be compensated by the faster memory access, thereby accelerating the time-to-solution. At this point it is important to note that running the complete Krylov method in lower precision would not provide the same solution accuracy. The Compressed Basis Krylov methods preserve the accuracy in the final solution approximation, the performance depends on the trade-off between convergence delay and faster iterations.

We implemented a Compressed Basis GMRES (CB-GMRES) solver that uses the memory accessor for compressing the Krylov basis vectors [21].

To assess the solution accuracy, in Figure 8 we report the normalized explicitly-computed residual $||Ax^* - b||_2/||b||_2$ for the solution approximations generated with the distinct CB-GMRES versions either executed as plain algorithm (top) or enhanced with a scalar Jacobi preconditioner (bottom). All GMRES versions are based on the same code stack, use the same initial guess, and a restart parameter of 50. For comprehensiveness, we report the final residual norm also for those cases where the accuracy target cannot be reached. In these initial results, we observe that the standard DP GMRES based on CGS with re-orthogonalization (`GMRES<fp64,fp64>`), the standard DP GMRES based on modified Gram-Schmidt (`MGS-GMRES<fp64,fp64>`), and the CB-GMRES variants storing the basis vectors in 32-bit floating-point precision (`GMRES<fp64,fp32>`) or 32-bit fixed-point precision (`GMRES<fp64,int32>`) generally achieve the same residual accuracy for both the non-preconditioned application and the Jacobi-preconditioned case. We furthermore observe that a SP GMRES (`MGS-GMRES<fp32,fp32>`) fails to provide solution approximations of the same accuracy level, and may therefore be disregarded as a valid option when aiming for high accuracy. Storing the Krylov basis in `fp16` or `int16`, the CB-GMRES algorithm converges to a solution of lower accuracy, however, often still achieving a residual accuracy better than a SP GMRES (`MGS-GMRES<fp32,fp32>`).

To quantify the convergence delay introduced by storing the Krylov basis vectors in lower precision, in



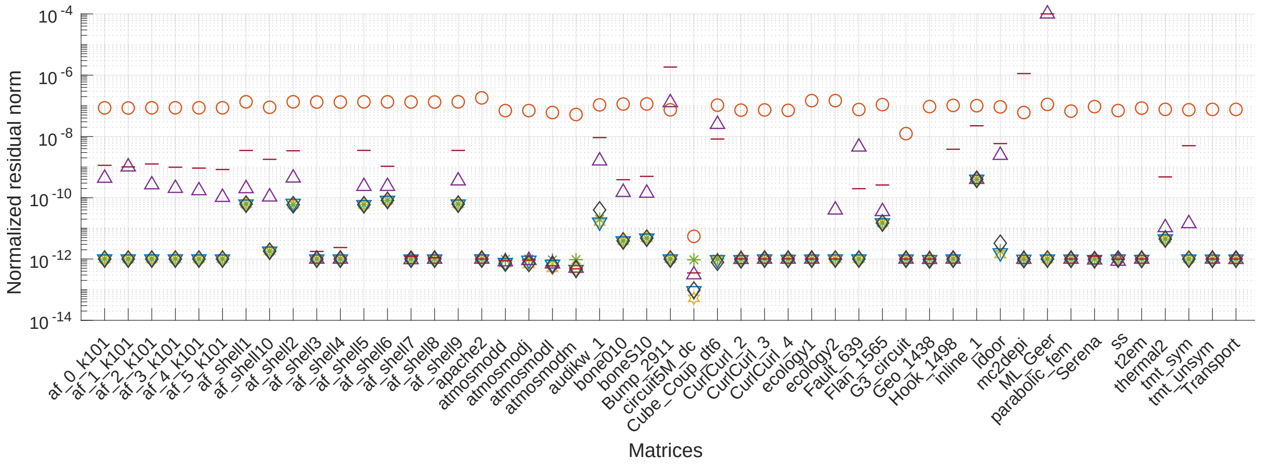


Figure 8: Normalized residual of plain GMRES (top) and Jacobi-preconditioned GMRES (bottom) using different precision for arithmetic and memory operations.

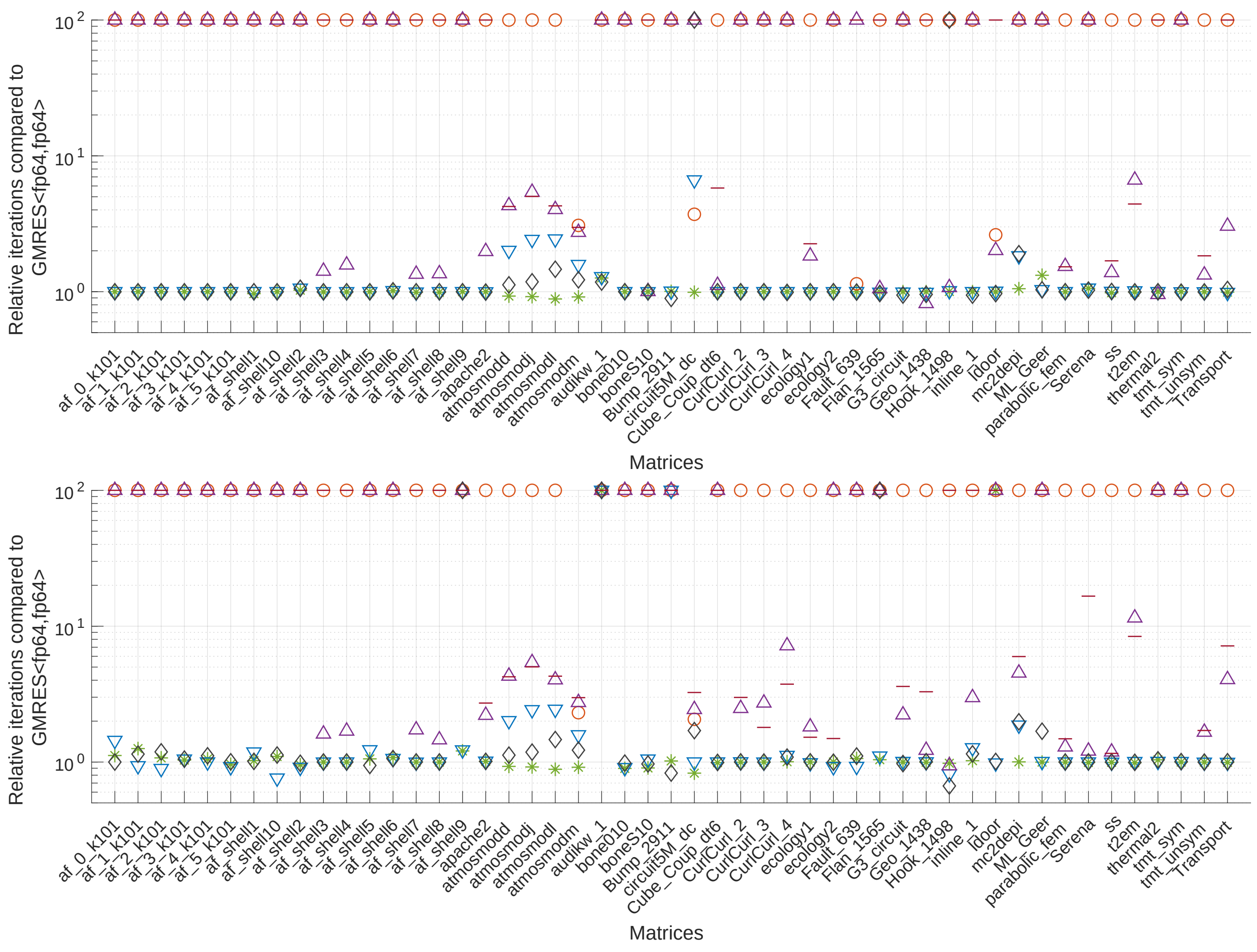
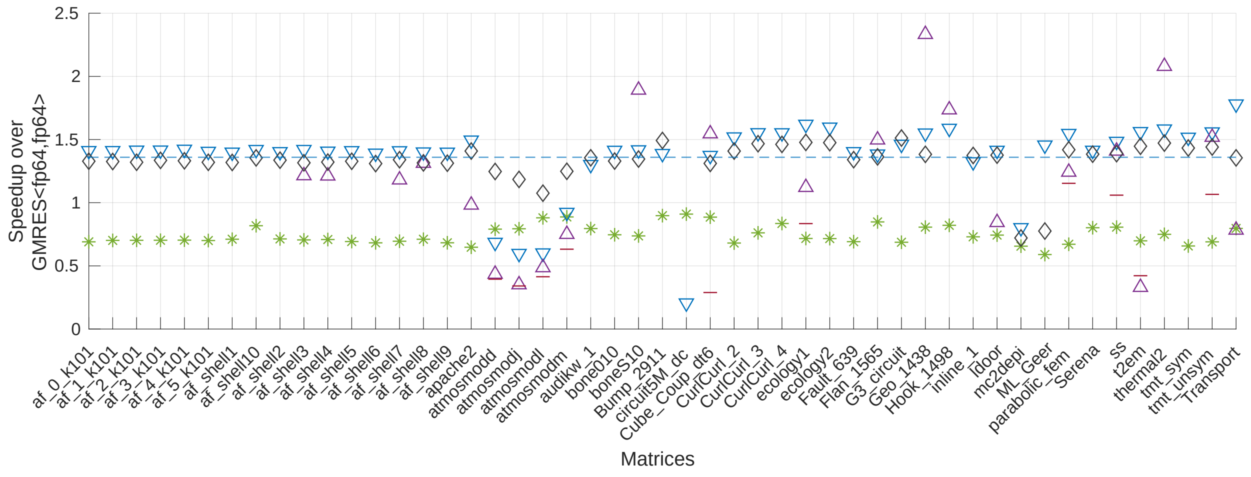


Figure 9: Iteration overhead of the CB-GMRES variants relative to the DP GMRES iteration count. The upper graph represents the results for the plain solver runs, the lower graph for the Jacobi-preconditioned solver runs.



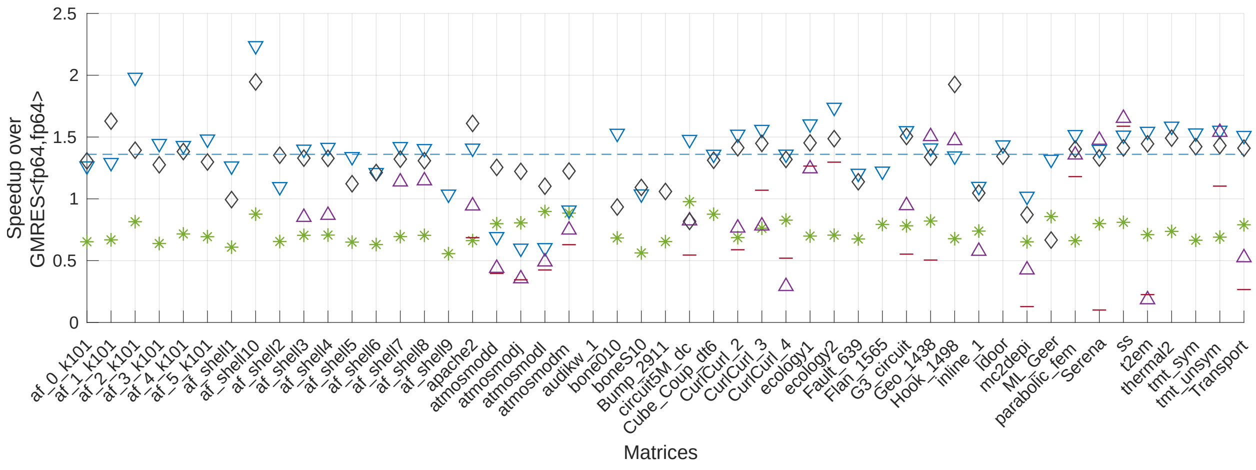


Figure 10: Speed-up of the non-preconditioned CB-GMRES (top) and Jacobi-preconditioned CB-GMRES (bottom) over the respective DP GMRES variants.

Figure 9 we display the iteration count of the CB-GMRES variants relative to the DP GMRES iteration count if they reach the same residual accuracy (even if that accuracy does not fulfill the residual norm stopping criteria). Again, we include results for the non-preconditioned solvers (top) and the Jacobi-preconditioned solvers (bottom). If a solver does not succeed in reaching the same residual accuracy, we set the iteration overhead marker to "100" to clearly indicate that this solver is not a valid option.

This experiment shows that the CB-GMRES realizations `GMRES<fp64,fp32>` and `GMRES<fp64,int32>` generally match the iteration count of DP GMRES. An exception are the ATMOSMOD problems where the CB-GMRES variants using 32-bit memory precision need a few additional iterations. In contrast, when the orthogonal basis is stored using the 16-bit formats, even if the residual accuracy can be achieved, the overhead often increases dramatically.

Finally, we want to answer the question whether the CB-GMRES algorithm is outperforming the standard GMRES algorithm in the most relevant metric: the time-to-solution. For that, in Figure 10 we provide a detailed performance evaluation by visualizing the speed-up for the distinct test problems, distinguishing between the non-preconditioned GMRES (top graph) and the Jacobi-preconditioned GMRES (bottom graph). Here we notice a rather uniform picture concerning the speed-ups for `GMRES<fp64,fp32>` and `GMRES<fp64,int32>`, with the exception of the ATMOSMOD problems where the iteration overhead cannot be compensated with faster memory access. Overall, `GMRES<fp64,fp32>` is slightly superior over `GMRES<fp64,int32>` which is likely due to the overhead of the scaling process and the additional scaling factors needed when storing the basis vectors in `int32`. From this experiment, we conclude that the `GMRES<fp64,fp32>` is an appropriate choice for a wide range of problems. We also note that GMRES based on modified Gram-Schmidt orthogonalization (`MGS-GMRES<fp64,fp64>`) is consequently slower than GMRES based on classical Gram-Schmidt orthogonalization with reorthogonalization (`GMRES<fp64,fp64>`). This may be expected as MGS requires the use of less efficient BLAS-1 operations.

On average, `GMRES<fp64,fp32>` is 1.4× faster than the standard DP GMRES without impacting the final accuracy of the result.

4.3 *S*-STEP LANCZOS AND CG

Communication-avoiding or s-step Krylov subspace methods are variants which reduce communication (parallel latency and/or sequential latency and bandwidth) by a factor of $O(s)$ over a fixed number of iterations. The main idea is to expand the Krylov subspace $O(s)$ dimensions at the time and then perform a block orthogonalization or inner product. For an overview, see [22] and [23]. While these variants can reduce communication, they can be much less stable than their classical counterparts; although equivalent in exact arithmetic, the s-step variants are much more vulnerable to the effects of finite precision arithmetic, often exhibiting a greater convergence delay and a decrease in attainable accuracy.

In [24], we have recently developed mixed precision variants of s-step Lanczos and CG which can help regain what is lost in terms of convergence at minimal overhead cost. We have proved that if, in s-step Lanczos, the Gram matrix is computed and applied in double the working precision, the orthogonality among Lanczos basis vectors is improved by a factor related to the condition numbers of the s-step bases.

The resulting bound on the loss of orthogonality has the same structure as the uniform precision s-step Lanczos appearing in [23], but with the notable exception that the bound now contains only a factor of $\bar{\Gamma}_k$ rather than $\bar{\Gamma}_k^2$, where $\bar{\Gamma}_k = \max_{i\in\{0,\dots,k\}} \||\hat{\mathcal{Y}}_i^+|\|_2 \||\hat{\mathcal{Y}}_i|\|_2$. As Γ_k can potentially grow very quickly with s, this is a significant improvement, and indicates that, among other things, the Lanczos basis vectors will maintain significantly better orthogonality and normality due to the selective use of higher precision.

We have extended the results of Paige [25] to this case, which allows for results on the accuracy and convergence of eigenvalues. In the uniform precision s-step Lanczos, these results are applicable as long as

$$\bar{\Gamma}_k < \left(24u(n+11s+15)\right)^{-1/2} = O(1/\sqrt{nu}).$$

Due to the use of extended precision in the mixed precision case, the constraints are now relaxed, requiring that

$$\bar{\Gamma}_k < \left(2u(6s+11)\right)^{-1} = O(1/u),$$

under the assumption that $n\varepsilon\Gamma_k < 1$ for all k. This is significant improvement. For example, if the working precision is double, then in the uniform precision case, we can only expect predictable behavior as long as the s-step bases have condition number bounded by $\approx 10^8$, whereas this becomes 10^{16} in the mixed precision case.

The analyses rely on the definition of a quantity ε_2, which is $O(\varepsilon n)$ in the classical Lanczos case, $O(\varepsilon n\bar{\Gamma}_k^2)$ in the uniform precision s-step Lanczos case, and $O(\varepsilon\bar{\Gamma}_k^2)$ in the mixed precision s-step Lanczos case. The main result is that, assuming no breakdown occurs and the size of $\bar{\Gamma}_k$ satisfies the respective constraints on $\bar{\Gamma}_k$, these results say the same thing for the mixed precision s-step Lanczos case as in the uniform precision s-step Lanczos and classical Lanczos cases: *until an eigenvalue has stabilized, the mixed precision s-step Lanczos algorithm behaves very much like the error-free Lanczos process, or the Lanczos algorithm with reorthogonalization*.

The primary difference among these three Lanczos variants is how tight the constraints are by which we consider an eigenvalue to be "stabilized". The larger the value of ε_2, the looser the constraint on stabilization becomes, and thus the sooner an eigenvalue is considered to be "stabilized". Thus, somewhat counterintuitively, for the uniform precision s-step Lanczos process where ε_2 is expected to be largest (as it contains a factor $\bar{\Gamma}_k^2$), we expect "stabilization" to happen sooner than in the other methods (but again, to within a larger interval around the true eigenvalues of A), *and thus we expect faster deviation from the exact Lanczos process*. In the classical Lanczos method, the smaller value of ε_2 means that we are more discriminating in what we consider to be a stabilized eigenvalue, and thus stabilization will take longer, which means we follow the exact Lanczos process for more iterations. For the mixed precision s-step Lanczos case, we expect the value of ε_2 to fall somewhere in the middle of the other two variants.

In the classical Lanczos case, the results in [25] say that we have at least one eigenvalue of A with high accuracy by iteration $m = n$. In both uniform and mixed precision s-step Lanczos algorithms, it is still true

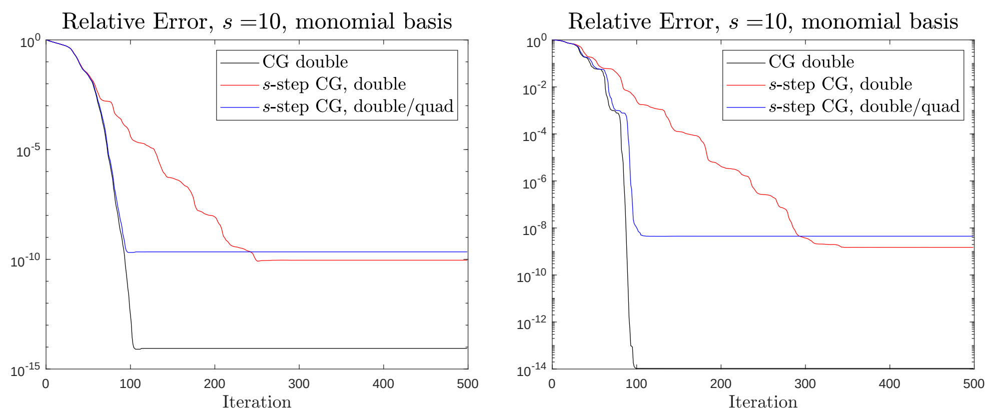


Figure 11: Convergence of classical CG in double (black), uniform precision s-step CG in double (red), and the new mixed precision s-step CG in double and quad (blue), for `nos4` (left) and `bcsstk02` (right) matrices from SuiteSparse.

that we will find at least one eigenvalue with some degree of accuracy by iteration $m = n$ as long as the respective constraints on $\bar{\Gamma}_k$ hold, but here the limit on accuracy is determined by the size of $\bar{\Gamma}^2_{\lceil n/s \rceil}$ in the uniform precision case and $\bar{\Gamma}_{\lceil n/s \rceil}$ in the mixed precision case. Thus we can expect in general, eigenvalue estimates will be about a factor $\bar{\Gamma}_{\lceil n/s \rceil}$ more accurate in the mixed precision case versus the uniform precision case.

As the s-step CG algorithm is based on an underlying s-step Lanczos algorithm, we expect that the improved Ritz value accuracy obtained by the use of the mixed precision approach in the s-step Lanczos algorithm will lead to improvements in convergence behavior in a corresponding mixed precision s-step CG algorithm. We note that we do not expect that the use of extended precision in the Gram matrix computation will improve the maximum attainable accuracy in s-step CG, as this is primarily dependent on the precision used for SpMVs. The bounds on the maximum attainable accuracy for s-step CG are discussed elsewhere [23].

We present an example which demonstrates this, performed in MATLAB (R2020a). We compare classical CG (the 2-term recurrence variant) in double precision to s-step CG in double precision and to mixed precision s-step CG in double/quad. The mixed precision variant follows the same principle as the mixed precision s-step Lanczos variant. Namely, that the Gram matrix is computed and applied in double the working precision, and everything else is done in the working precision. For double precision, we use the built-in MATLAB datatype and for quadruple precision we use the Advanpix MATLAB Toolbox with 34 decimal digits. We use right-hand sides generated to have equal components in the eigenbasis of A and unit 2-norm, which represents a difficult case for CG (all components must be found). In each experiment, we measure the relative error measured in the A-norm, where the true solution is computed using MATLAB backslash in quadruple precision via Advanpix. The results for the matrices `nos4` and `bcsstk02` from SuiteSparse [26] using $s = 10$ and a monomial basis are shown in Figure 11, where the improved convergence rate is clearly observable.

We have completed an initial performance evaluation on a 3D Laplace model problem on NVIDIA V100 GPUs. The initial results indicate that the overhead of the extra precision is minimal when restricted to using hardware precisions, and thus in this case we expect that the reduced convergence rate will provide significant improvement in time-to-solution. When using software-implemented precisions, however, the overhead can be much higher, although this effect is diminished when scaling to larger numbers of GPUs as the relative latency cost grows. A more thorough performance study and evaluation is needed to gauge the practical applicability of this technique. Future work also includes the investigation of the mixed precision s-step CG algorithm in combination with a residual replacement technique so that attainable accuracy can be simultaneously improved.

4.4 ARNOLDI-*QR* MGS-GMRES

For MGS-GMRES the mixed precision work by Gratton et. al. [27] is the most recent and appropriate - and in particular the loss-of-orthogonality relations due to Björck [28] and Paige [29], later refined by Paige, Rozložnik and Strakoš [30], are employed in order to provide tolerances for mixed single–double computations. MGS-GMRES convergence stalls (the norm-wise relative backward error approaches ε) when linear independence of the Krylov vectors is lost, and this is signaled by Paige's S matrix norm $\|S\|_2 = 1$. The S matrix [31] is derived from the lower triangular T matrix appearing in the rounding error analyses by Giraud et. al. [32].

To summarize, the Gratton et. al. [27] paper postulates starting from the Arnoldi-QR algorithm using the modified Gram-Schmidt algorithm and employing exact arithmetic in the MGS-GMRES iterative solver. The Arnoldi-QR algorithm applied to a non-symmetric matrix A produces the matrix factorization, with loss of orthogonality F_k

$$A V_k = V_{k+1} H_k, \quad V_{k+1}^T V_{k+1} = I + F_k \tag{15}$$

They next introduce inexact (e.g. single precision) inner products - this directly relates to the loss-of-orthogonality relations for the $A = QR$ factorization produced by MGS. The resulting loss of orthogonality, as measured by $\|I - Q^T Q\|_2$, grows as $\mathcal{O}(\varepsilon)\kappa(A)$ as was derived by Björck [28] and $\mathcal{O}(\varepsilon)\kappa([\ r_0,\ AV_k\ ])$ for Arnoldi-QR - which is described by Paige, Rozložník and Strakoš [33, 30] and related work. The inexact inner products are given by

$$h_{ij} = v_i^T w_j + \eta_{ij} \tag{16}$$

where h_{ij} are elements of the Hessenberg matrix H_k, and the Arnoldi-QR algorithm produces a QR factorization of the matrix

$$\left[\ r_0, \quad AV_k\ \right] = V_{k+1} \left[\ \beta e_1, \quad H_k\ \right], \tag{17}$$

The loss of orthogonality relations for F_k are given below, where the matrix U is strictly upper triangular

$$F_k = \bar{U}_k + \bar{U}_k^T, \quad U_k = \begin{bmatrix} v_1^T v_2 & \cdots & v_1^T v_{k+1} \\ & \ddots & \\ & & v_k^T v_{k+1} \end{bmatrix} \tag{18}$$

Define the matrices,

$$N_k = \begin{bmatrix} \eta_{11} & \cdots & \eta_{1k} \\ & \ddots & \\ & & \eta_{kk} \end{bmatrix}, \quad R_k = \begin{bmatrix} h_{21} & \cdots & h_{2k} \\ & \ddots & \\ & & h_{k+1,k} \end{bmatrix} \tag{19}$$

The loss of orthogonality relation derived by Björck [28], for the $A = QR$ factorization via the modified Gram-Schmidt algorithm can be applied to the Arnoldi-QR algorithm to obtain

$$N_k = -\left[\ 0, \quad U_k\ \right] H_k = -U_k R_k \tag{20}$$

The complete loss of orthogonality (linear independence) of the Krylov vectors in MGS-GMRES signals the minimum error is achieved and GMRES then stalls or really can go no further than when the norm-wise relative backward error reaches $\mathcal{O}(\varepsilon)$. Gratton et al. [27] show how to maintain sufficient orthogonality in order to achieve a desired relative residual error level - by switching the inner products from double to single at certain tolerance levels and combine this with inexact matrix-vector products as in van den Eshof and Sleijpen [34] and Simoncini and Szyld [35].

In practice, the restarted variant of GMRES is often employed to reduce memory requirements. The algorithm produces both implicit and explicit residuals. Thus, we might ask whether either can be performed in reduced precision. The work described herein on iterative refinement by Nick Higham and Erin Carson for mixed precision can be applied to analyse the convergence of restarted GMRES(m), assuming a fixed number of iterations - because restarted GMRES is just iterative refinement with GMRES as the solver for the correction term. However, a more detailed analysis with experiments has yet to be performed. We are fairly certain that the residual computations must be performed in higher precision in order to achieve a norm-wise backward error close to double precision machine round-off.

4.5 ALTERATIVE APPROACHES

Although somewhat outside the scope of this review, we can demonstrate that it is possible to modify the Gratton et al. [27] analysis based on the inverse compact WY form of the MGS algorithm, introduced by Świrydowicz et al. [36]. Rather than treat all of the inner products in the MGS-GMRES algorithm equally, consider the strictly upper triangular matrix $U = L^T$ from the loss of orthogonality relations. We introduce single precision $L_{:,j-1} = Q_{j-1}^T q_{j-1}$ and double precision triangular solve $r = (I + L_{j-1})^{-1} Q_{j-1}^T a$ to update R - as this would directly employ the forward error analysis of Higham [37]. The former affects the loss of orthogonality, whereas the latter affects the representation error for QR - but then also for Arnoldi-QR. This could allow more (or most) of the inner products to be computed in single precision.

Barlow [38] contains similar if not the same algorithm formulations in block form. His work is related to Björck's 1994 paper [39, Section 7] which derives the triangular matrix T using a recursive form for MGS, and which is referred to as a "compact WY" representation in the literature. While Björck used a lower triangular matrix for the compact WY form of MGS, Malard and Paige [40] derived the upper triangular form, also employed by Barlow, which reverses the order of elementary projectors. The latter is unstable in that a backward recurrence leads to $\mathcal{O}(\varepsilon)\kappa^2(A)$ loss of orthogonality. An interesting observation from Julien Langou is that the upper triangular form is less stable than the lower triangular (even though the backward-forward algorithm results in re-orthogonalization; see the algorithm in Leon, Björck, Gander [41]).

Barlow [38] employs the Householder compact WY representation of reflectors and also refers to the work of Chiara Puglisi [42] – discussed in Joffrain et al. [43] – and this is referred to as the "inverse compact WY" representation of Householder; this originally comes from Walker's work on Householder GMRES [44]. Barlow then extends this approach to the block compact WY form of MGS; see also the technical report by Sun [45]. The contribution by Świrydowicz et al. [36] was to note that there exists an inverse compact WY representation for MGS - having the projector

$$P^{IM} = I - Q_{j-1} T^{IM} Q_{j-1}^T = I - Q_{j-1}(I + L_{j-1})^{-1} Q_{j-1}^T$$

and to "lag" the norm $\|q_{j-1}\|_2$ so that these can be computed in one global reduction. Barlow [38] makes this connection for blocks (and in effect this is given in his equation (3.10)) and references Puglisi [42].

Björck and Paige [46] made the link between Householder and MGS based on the observation made by Sheffield. Paige defines this to be augmentation and Gratton et al. [27] also references this work. Paige has also recently extended these augmentation ideas to Lanczos. The T matrix appears in Paige's work with Wülling [47] and then later in [31] to derive the loss of orthogonality matrix $S = (I + L_{j-1}^T)^{-1} L_{j-1}^T$. This also appears in the work of Giraud, Gratton and Langou [32]; Langou also worked with Barlow and Smoktunowicz [48] on the Pythagorean trick to reduce cancellation error in the computation of vector norms and a Cholesky-like form of classical Gram-Schmidt (CGS).

In order to combine single-double floating-point operations in MGS-GMRES, at first it appears that we could store the T matrix in single precision - but then we would still have to form $Q_{j-1}^T a$, and store Q_{j-1} in double precision. By examining the cost trade-offs a bit further, we can instead use a new form of T based on a truncated Neumann series

$$T = I - L_{j-1} - L_{j-1}^T + L_{j-1}^T\, L_{j-1} + L_{j-1}\, L_{j-1}^T$$

and our initial computational results demonstrate this works well, driving the relative residual to $\mathcal{O}(\varepsilon)$ in double, with orthogonality maintained to $\mathcal{O}(\varepsilon)$ in single.

The representation error (backwards error) for $A + E = QR$ computed by MGS, is not affected by single precision inner products - and remains $\mathcal{O}(\varepsilon)$. We are not aware of whether or not this was previously known.

5. Mixed Precision Sparse Approximate Inverse Preconditioning

Preconditioners are by design approximate linear operators, and therefore natural candidates for mixed precision approaches. An important aspect however is that using a low precision preconditioner in a high precision iterative solver results in a non-constant preconditioner, therewith requiring a flexible Krylov solver. An attractive workaround is to do all arithmetic in high precision, and only store the preconditioner in a low precision format. This way, the preconditioner remains a constant operator while still reducing

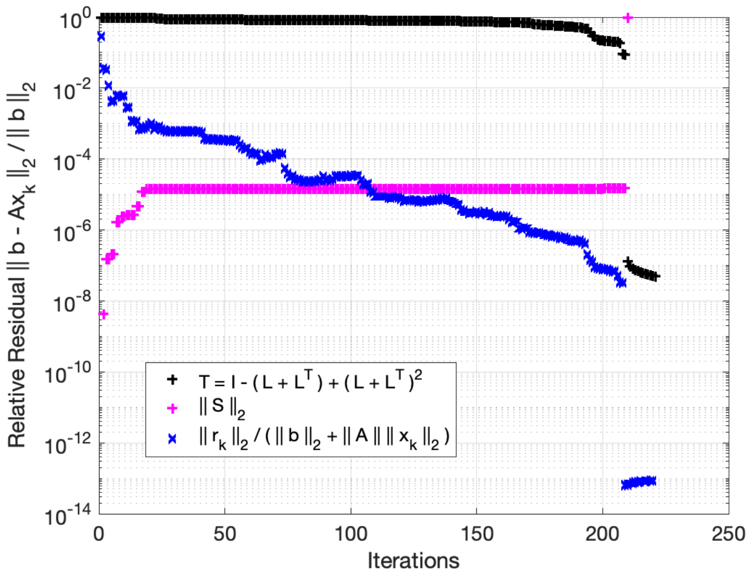


Figure 12: GMRES residuals and loss of orthogonality $\|S\|_2$ for impcol_e matrix

the overall runtime by cutting down the memory access volume. The most elegant way to realize a mixed precision preconditioner storing the preconditioner data in low precision but performing all computations in high precision is the memory accessor introduced in Section 8. In Figure 13 and Figure 14 we visualize the performance of mixed precision sparse approximate inverse preconditioners for general and symmetric positive definite problems, respectively [49].

6. Mixed Precision Strategies for Multigrid

Design and development of mixed-precision strategies for numerical solvers typically takes two main forms: a defect-correction based approach, and an implicit task-based approach. Defect correction techniques rely on identifying a defect in the numerical solution process and formulating a correction to the defect. They are easy to implement, less intrusive, and can be adapted to correct multiple defects. A classic example of this approach is the iterative refinement strategy. Implicit task-based techniques, on the other hand, rely on domain knowledge to make informed approximations and apply task-based mixed precision within solvers. This is a more intrusive, albeit general, approach that does not rely on correcting a defect. For both of these approaches, the guiding principle is to use low precision in a practical way so as to make efficient use of memory capacity, improve data motion, and optimize the number of flops to benefit faster computation and communication.

6.1 MIXED-PRECISION ALGEBRAIC MULTIGRID

Multigrid is a scalable linear solver algorithm widely used in many scientific applications for the solution of linear systems of equations. Its effectiveness comes from the complementary processes of relaxation - to handle high frequency error, and coarse grid correction - to handle low frequency error. Recent efforts in [50, 51] have presented some theoretical analysis on the convergence of mixed-precision strategies for multigrid.

In designing mixed-precision strategies for AMG, it makes sense to identify the key floating point operations in the AMG algorithm that either represent bottlenecks to performance, or are sensitive to approximation. These include smoothing operations (approximate solvers, relaxation and complex smoothers), grid transfer operations (matrix-vector and matrix-matrix multiplications), residual calculation and solution updates (vector additions). Furthermore, at lower depths of the AMG hierarchy, the algorithm is memory bound. Thus, to achieve an efficient mixed-precision algorithm, one needs to consider strategies for both

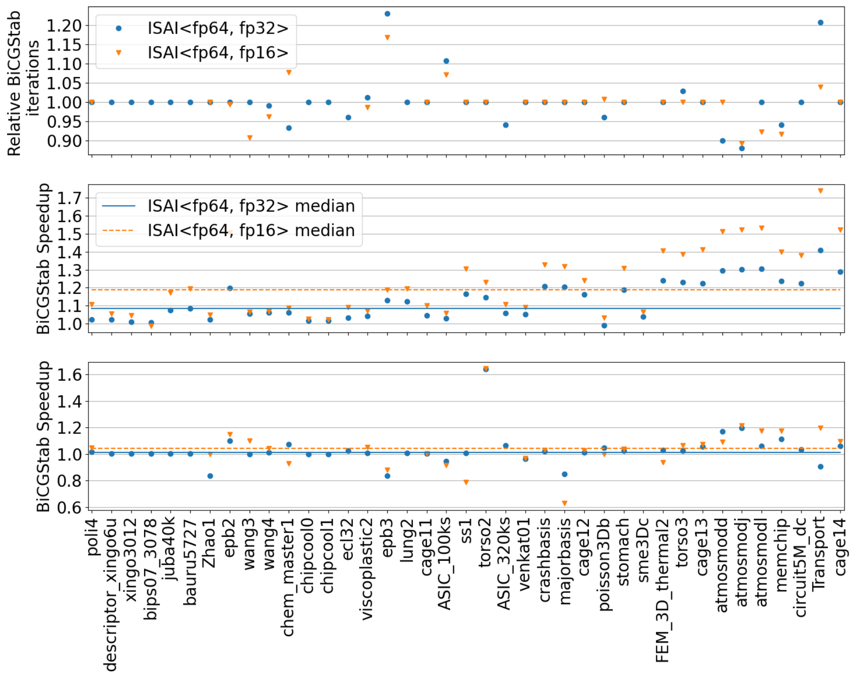


Figure 13: *Top*: Relative iteration counts for BiCGSTAB using lower precision preconditioner storage compared to double precision. Cases where half precision use results in lower iteration counts are related to rounding effects. *Center*: Speedup of a single mixed precision ISAI application vs. double precision. The speedup ratios ignore the quality degradation of the preconditioner when using low precision storage. *Bottom*: BiCGSTAB speedup when using an ISAI preconditioner stored in lower precision instead of a double precision ISAI. Missing dots indicate the loss of convergence when using the mixed precision ISAI variant. The horizontal lines display the median among all values without loss of convergence. BiCGSTAB runtimes do not include the preconditioner generation. *Note*: The matrices are sorted according to their nonzero count along the x-axis.

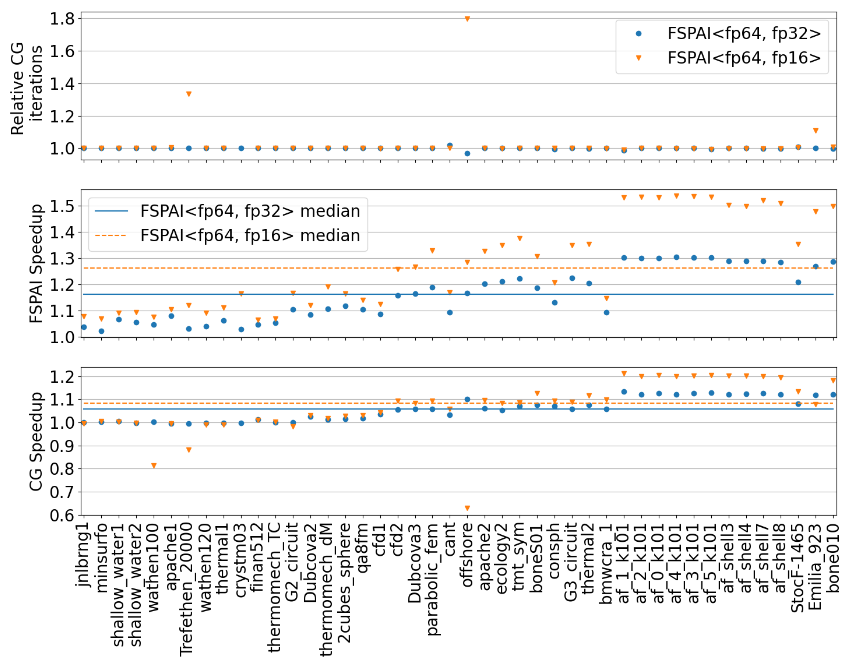
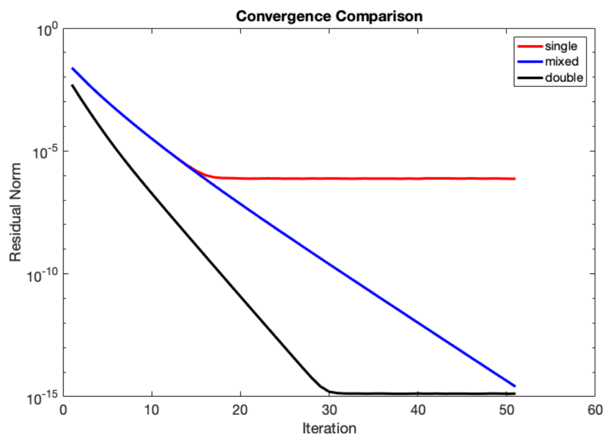


Figure 14: *Top*: Relative iteration counts for CG using lower precision preconditioner storage compared to double precision. *Center*: Speedup of a single mixed precision FSPAI application vs. double precision. The speedup ratios ignore the quality degradation of the preconditioner when using low precision storage. *Bottom*: CG speedup when using an FSPAI preconditioner stored in lower precision instead of a double precision FSPAI.



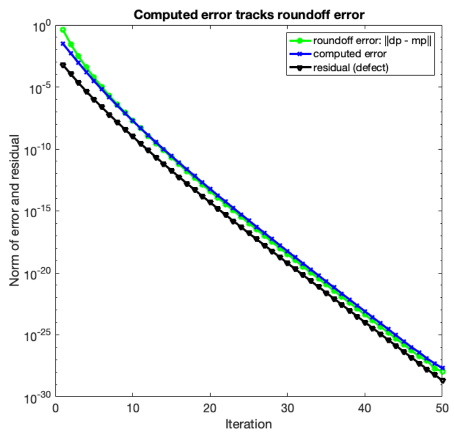


Figure 15: Left: Convergence behavior of double, single and mixed-precision AMG. **Right:** Comparison of roundoff error and computed error correction from defect correction algorithm. Also shows the residual (defect).

faster computation and communication. In the implicit task-based approach, one would also need to perform the appropriate error analysis to study the impact of approximating one or more of the key operations (in low precision, for example) on the final solution of the AMG algorithm. The defect-correction approach by design addresses most of these issues, and is easy to adapt into existing codes. Thus, we follow this approach in our work.

The standard V-cycle formulation of the AMG algorithm can be viewed as a defect correction strategy on a hierarchy of grids. Here, the defect is characterized by the residual computed at each level of the hierarchy, and the correction is computed on subsequent grid levels. In regards to mixed precision, various scenarios may be considered. One strategy is to consider performing the AMG V-cycle in low precision, but evaluating the residual and solution updates in high precision. While this appears to satisfy the classic iterative refinement strategy, in the context of AMG, this also requires that the inter-grid operators and in particular the coarse grid matrix, be available in high precision. In other words, from the point of view of implementation, this requires a setup of two AMG hierarchies (one for each precision), just to compute the matrix needed for the residual calculation. An alternative strategy is to consider performing the AMG V-cycle in high precision, but performing the smoother operations in low precision. We note that in this case, a low precision equivalent of the matrix is also required at each level of the hierarchy, in order to perform the smoother operation. However, unlike the former strategy, we need not compute or store low precision inter-grid operators nor compute a low precision version of the matrix. We only need to store a copy of the already computed high precision matrix. Yet another alternative to the aforementioned scenarios is to consider performing defect-correction only on the fine grid. This approach is easy to implement and is equivalent to performing iterative refinement on the original (fine grid) problem, where the correction step is computed by applying an AMG v-cycle. Figure 15 shows the convergence behavior of using this approach as a mixed-precision strategy for AMG, compared to a fixed double or single precision AMG solve. The results also depicts how the roundoff error is accurately tracked by the error correction to eventually achieve double precision accurate convergence for the mixed-precision strategy.

6.2 ENABLING MIXED-PRECISION CAPABILITIES IN HYPRE

Currently, the hypre library can be built in one of three precisions - single, double and longdouble. The choice of precision type is prescribed by the user at build time by setting the option *--enable-<precision-type>*. The goal for the mixed-precision integration is to extend the build system to allow a unity build of all three precisions supported by hypre, in order to facilitate mixed-precision solver development. Integrating multiprecision capabilities into an existing library like hypre can be quite challenging. A systematic approach to the integration is necessary to:

1. Limit user impact: provide new options to users while staying true to the existing interface
2. Limit developer impact: reduce burden on developers by automating code transformation to support multiprecision build
3. Obtain multiprecision code that is maintainable, portable, and performant

To achieve these objectives, we proposed terminology to classify functions in hypre into three categories:

- Multiprecision functions: These are functions that rely on precision for storage or computation
- Multiprecision Methods: These are multiprecision functions that have a hypre solver or matrix object in their argument. These objects will store precision information.
- Mixed-Precision Methods: These are functions that use a combination of multiprecision functions

Notice that by definition, multiprecision methods and mixed-precision methods are also multiprecision functions, since they rely on precision. However, one major difference between them is that the function precision for the methods is determined at run-time, while the multiprecision functions have precision prescribed at compile time. This also means that the multiprecision functions will require multiple builds for each precision type, while the mixed-precision and multiprecision methods need to be built only once.

Using the proposed terminology, we developed some basic rules and guidelines to help guide the multiprecision integration:

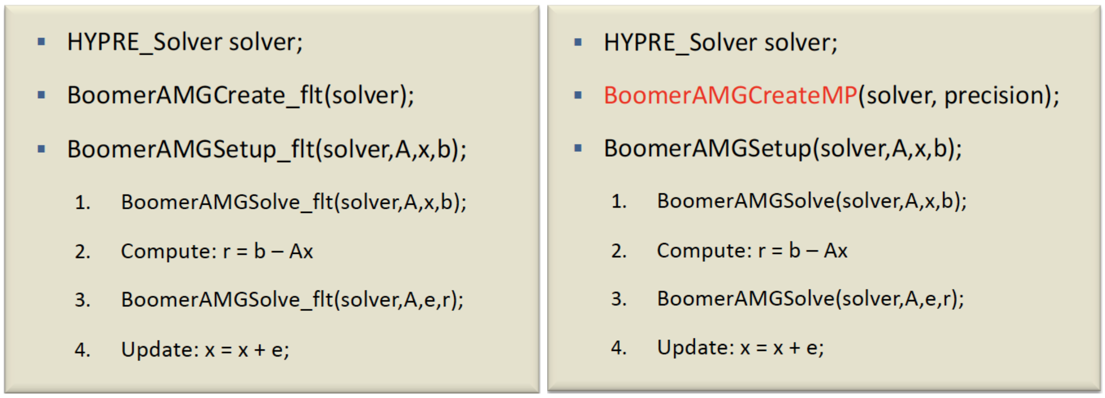
- A multiprecision function can only call another multiprecision function of the same type, or a non multiprecision function
- A function calling a multiprecision function must also be a multiprecision function or a mixed-precision method
- A function calling a mixed-precision method must also be a mixed-precision method

Implementing the integration within hypre requires that we first extend the build system to allow a multiprecision build of all the supported types. We provide support for the build option *--enable-mixed-precision*, which if specified by the user, activates various compile definitions and C-preprocessor (CPP) macros to help with code transformation. The code transformation is achieved by populating a list of multiprecision functions into a header file. These functions are wrapped in a CPP macro that modifies any occurence of the function names (in header or source files) at compile time. This results in distinct symbols of the object files for the built libraries in the respective precisions and avoids any name collisions in the library. For example, the function `hypre_foo(hypre_object obj)` would be transformed into `hypre_foo_<precision_type>(hypre_object obj)` for precision type single, double and longdouble respectively.

6.3 CURRENT STATUS AND FUTURE PLANS:

The proposed terminology, rules and guidelines, and automated code transformation approach all contribute to a systematic process for integrating multiprecision capabilities into hypre, with minimal impact on the developer while achieving the desired goals of this effort. The process has been successfully applied to a proxy library (AMG2017), which comprises a subset of hypre. This allows us to demonstrate how users can make use of multiprecision functions and methods available to them to develop mixed-precision algorithms. Figure 16 shows an illustration of how a user might implement a mixed-precision iterative refinement solver using either a multiprecision function or a multiprecision method approach. The latter approach benefits from new functionality that allows users to prescribe the precision type for the hypre objects they create, which is then used internally to call the appropriate multiprecision functions associated with that object.

The next stages of this effort involves applying this systematic process to the main hypre library and testing and evaluating various functionalities to identify any outstanding issues. In addition, new task-based mixed-precision solver strategies would be developed as new solver components within hypre, including the defect correction based mixed-precision AMG solver described earlier. Performance evaluations of these new capabilities will also be performed to show the benefits of the mixed-precision strategies.



```
▪ HYPRE_Solver solver;
▪ BoomerAMGCreate_flt(solver);
▪ BoomerAMGSetup_flt(solver,A,x,b);
    1. BoomerAMGSolve_flt(solver,A,x,b);
    2. Compute: r = b – Ax
    3. BoomerAMGSolve_flt(solver,A,e,r);
    4. Update: x = x + e;
```

```
▪ HYPRE_Solver solver;
▪ BoomerAMGCreateMP(solver, precision);
▪ BoomerAMGSetup(solver,A,x,b);
    1. BoomerAMGSolve(solver,A,x,b);
    2. Compute: r = b – Ax
    3. BoomerAMGSolve(solver,A,e,r);
    4. Update: x = x + e;
```

Figure 16: An illustration of a user-defined function to perform mixed-precision AMG using defect-correction. **Left:** Using a multiprecision function approach. **Right:** Using a multiprecision method approach. The solver object stores the precision, which is used in the setup and solve routines implicitly.

7. Mixed Precision FFT

The goal of the CLOVER FFT-ECP project is to provide highly optimized and scalable multidimensional Fast Fourier Transforms (FFTs) targeting exascale systems [52, 53]. However, the FFTs are well known to be memory-bound, especially on hybrid systems with GPUs, where the local computations can be significantly accelerated, e.g., more than 40× with V100 GPUs in *heFFTe* compared to multicore CPUs [54]. This behavior is observed in all state-of-the-art distributed FFT libraries [55]. Hence, inter-processes communication is the main bottleneck for parallel FFTs, and indeed, there are cases where it can take more than 97% of the runtime [56, 57]. The work presented in this section, aims to alleviate the communication bottleneck for parallel FFTs, and hence further accelerate its computation. For this, we developed new computational techniques and software that use mixed-precision methodologies.

We worked on three techniques, described next: The first uses data compression to reduce communication, the second uses typical FP32-FP32-FP16 data conversions also to reduce communications as well as intermediate storage obtained by truncating the mantissa of FP64 numbers; and finally, some new contributions to MPI including a proposal and implementations for mixed-precision MPI that has FP64 input and output, but internally sends less data by fusing the above data compression techniques with the MPI communications.

7.1 DATA COMPRESSION TO REDUCE COMMUNICATION

We propose to apply some compression techniques on the data before each communication, and decompression after the data is received. The idea is that if the GPUs are used less than 3% of time, we can use the GPUs for data compression/decompression, and speedup the overall execution by reducing the communication volume.

We consider two classes of compression techniques. The first class consists of the well-known casting operation which is very efficient due to the hardware support provided (discussed in the next subsection). The second class is composed of the techniques that rely on more sophisticated algorithms and so are not as efficient but can potentially offer more advantages in terms of data compression. For example, the ZFP library [58] provides lossless and lossy compression. Moreover, it offers the possibility to control for instance either the accuracy, or the compression rate, with CPU and/or GPU support. However, for some algorithms to be of interest, certain conditions must be satisfied. For example, in ZFP, the data must have some meaning/smoothness in order to make it possible to compress at a fixed compression rate and then decompress with a lower maximum error compared with casting operation [59, 60].

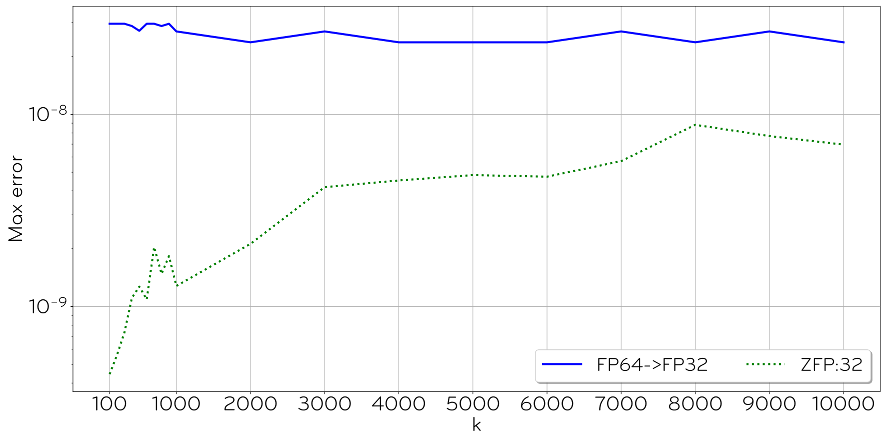


Figure 17: Comparison of the evolution of the accuracy between casting from FP64 to FP32 and ZFP with 32 bits when data to compress becomes less smooth.

To illustrate this, we consider a problem of size 100 000 given by $f(x,k) = sin(2k\Pi \times x/100000)$, where k allows us to control the smoothness of the data. We compare the casting operation with ZFP for a compression rate of two, which corresponds to casting from FP64 to FP32. The results are given in Figure 17. We note that the maximum error for the casting operation is around $1e-7$, while for ZFP starts below $1e-9$ when $k = 100$. Thus, if the data have certain smoothness, ZFP is able to compress and decompress data better than with the casting operation. However, when k increases, the function f loses this property and so the maximum error grows and becomes closer to the one obtained using casting operation. Ultimately, ZFP applied on random data, gives a maximum error of the same order as the casting operation.

Next, we focus on the casting operations since they are easier to analyze, and the compression on GPUs gives very good performance (approximately 700GB/s on NVIDIA V100 GPU).

7.2 APPROXIMATE FFTS WITH SPEED-TO-ACCURACY TRADE-OFFS

We consider two casting operations: FP64 to FP32, and FP64 to FP16, which give us a compression rate of two and four, respectively. Applying it in the context of FFT, a compression rate of two for example gives a speedup very close to two.

Besides casting, we can also compress data by trimming the mantissa. This can give us finer control over the accuracy vs. the more coarse casting operation. Trimming the mantissa also corresponds to sending MPI data using a stride, so it can be done efficiently, thus the reduction in communication results in proportional reduction of the overall execution time.

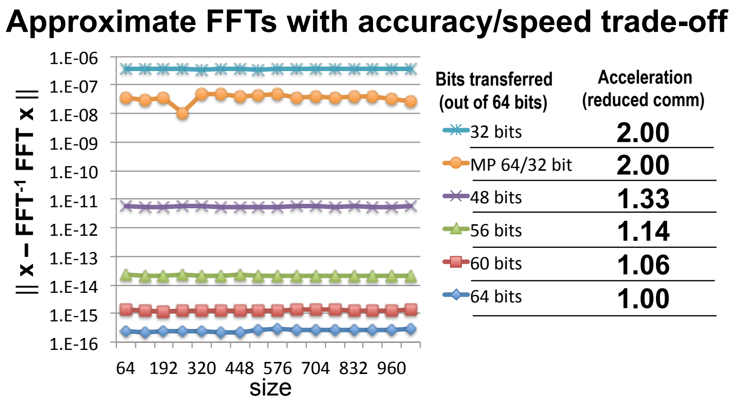


Figure 18: Evolution of the accuracy of the FFT algorithm with respect to the number of bits in the mantissa.

Figure 18 shows the impact of reducing the number of bits as well as the theoretical acceleration obtained by reducing the volume of communication. The accuracy is given by the norm of the difference between the input problem and the inverse of the FFT, i.e., $\|x - FFT^{-1}(FFT(x))\|$. We first note that the accuracy for 64 bits is around the double-precision machine ($\approx 1e-16$), and, for 32 bits, around the single-precision machine ($\approx 1e-8$), as expected. We observe that the more the mantissa is trimmed, the closer the accuracy is to the 32 bits accuracy.

Now, if we do the computation in double-precision but the communication in single-precision, referred to as MP 64/32 in the figure, the accuracy is better than doing everything with 32 bits. This means that, compared with 64 bits, the overall execution can be accelerated twice, while at the same time having a better accuracy than with the 32 bits.

7.3 TOWARDS MIXED-PRECISION MPI

We have collaborated with MPI developers, including George Bosilca and collaborators, towards new MPI developments for FFT [56] as well as a proposal and implementation for mixed-precision MPI that has FP64 input and output, but internally sends less data by fusing the above data compression techniques with the MPI communications. We implemented our approach on top of MPI, mimicking the MPI_Alltoallv API so that the integration in applications, and *heFFTe* in particular, is straightforward.

We re-implemented the all-to-all algorithm using the classical ring algorithm while fusing the compression into the MPI algorithm. In substance, each process p involved in the ring sends at step i data to process $p+i$. However, each send is replaced by *csend* which is a two-stage call: first, the compression of the data, and second, the send of the compressed data where the volume is reduced according the compression rate. Similarly, each receive is replaced by a *crecv*, which is also a two-stage operation where the received data need to be decompressed.

Additionally, for performance purposes, we replace the classical two-sided communication scheme by the One-Sided Communication scheme (OSC). This new approach allows us to remove unnecessary operations, (mainly synchronization), and to speedup the overall communication time.

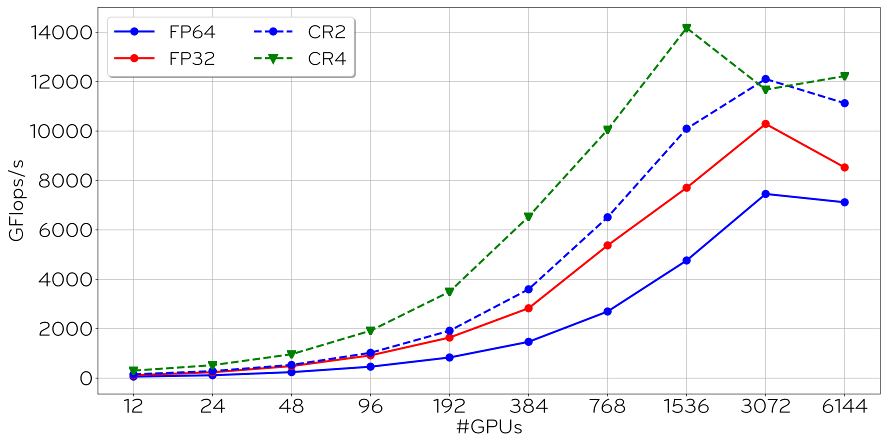


Figure 19: Evolution of the performance of *heFFTe* code when the number of GPUs increases, with a fixed problem size of $1,024^3$.

We now compare the performance of *heFFTe*, where the data is compressed during the communication, with the original *heFFTe* code on Summit supercomputer. We perform a strong scaling on a problem size of $1,024^3$ starting from two nodes (six GPUs per node) up to $1,024$ nodes. We consider as references the FP64 and FP32 *heFFTe* codes, both doing the computation and the communication using the same data-precision. We introduce *CR2* and *CR4*, which correspond to a compression rate of two and four, respectively. It means that the data is cast using FP64 to FP32 and FP64 to FP16, respectively.

In Figure 19, we display preliminary results on the evolution of the performance (GFlops/s) of *heFFTe* when the number of GPUs increases. The solid lines correspond to the references while the dashed lines represent the performance with compression. As expected, since FP32 involves twice fewer bits, the volume of communication is divided by two and so the performances are around 2× better.

The CR2 curve shows a greater speedup than the FP32, with the same volume of communication. This shows that our implementation does not suffer from the overhead of compressing the data. Moreover, the use of the One-Sided Communication improves the overall performance, reaching up to 2.5× speedup compared with FP64. With a compression rate of four (CR4), we also exceed the 4× speedup up to 384 GPUs. Then, the scalability starts to decrease. For all curves, we observe a performance drop at 6,144 GPUs. This is due to the fact that the volume of each message becomes too small and the latency becomes dominant. We even see it earlier for CR4 since the volume of communication is divided by four compared with FP64.

#GPU	FP64	FP32	FP64 → FP32
12	6.00e-15	4.96e-06	1.94e-07
24	6.17e-15	4.91e-06	2.20e-07
48	5.92e-15	4.49e-06	3.01e-07
96	6.00e-15	3.47e-06	3.90e-07
192	5.11e-15	3.54e-06	3.99e-07
384	5.25e-15	4.44e-06	5.09e-07
768	5.29e-15	3.13e-06	5.44e-07
1536	5.38e-15	3.06e-06	5.57e-07
3072	5.61e-15	4.37e-06	6.03e-07
6144	4.84e-15	3.25e-06	5.72e-07

Table 1: Comparison of the FFT accuracy when using compression in the communication with both references FP64 and FP32.

Finally, Table 1 shows the comparison between the reference and the casting operation from FP64 to FP32 (CR2). We observe that the mixed-precision gives one order of magnitude better accuracy compared with a unique working-precision of FP32. Furthermore, our approach allows us to consider lower precision without having the computational kernel usually needed with a unique working-precision.

In conclusion, the techniques presented show that reducing the communication volume (through different types of compression) results in corresponding reductions of the execution times. Since the FFT is an orthogonal transformation, the loss of accuracy in the input translates in approximately the same loss of accuracy in the output, except that we gain roughly an extra digit of accuracy due to computations being performed in FP64 arithmetic. Thus, the new techniques will be especially appealing for applications that use approximate FFTs, FFTs as preconditioners, and in general applications that know their inputs to FFT are not of full FP64 accuracy (i.e., about 16 decimal digits of accuracy). Among the largest applications that rely on FFTs, we have those from molecular dynamics simulations, such as LAMMPS [61], part of the EXAALT ECP project; and those from cosmological simulations, such as HACC [62]. For these applications, kernels such as long-range Coulombic solvers and N-body interactions require efficient and scalable FFT computations. For several type of simulations from these fields, the tolerance on the FFT result can be big enough to allow a mixed-precision computation, which can yield considerable speedups and is part of our future work plans.

8. Memory Accessor

The memory accessor decouples the floating point format used in arithemtic operations from the floating point format that is used in memory access and communication. On a high level, the memory accessor is a compression mechanism that compresses data before doing memory or communication operations, and decompresses data before doing arithmetic operations. While in theory any compression scheme, both lossless and lossy compression are options that can be used in the memory accessor scheme, the realization of the accessor has requirements that are hard to meet with complex compression algorithms: 1) operate on small data chunks; 2) support and provide good performance for random data access; 3) incur no overhead to memory-bound operations. Among many compression strategies, a simple one fulfilling these requirements is the lossy compression based on converting to a lower precision floating point format. We

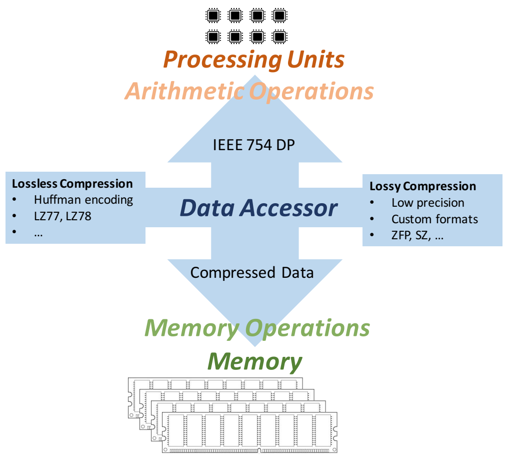


Figure 20: Accessor separating the memory format from the arithmetic format and realizing on-the-fly data conversion in each memory access.

realized the memory accessor with using a lower precision format for memory operations in the Ginkgo software ecosystem with support for OpenMP multicore, NVIDIA GPUs, and AMD GPUs. The memory accessor doing all arithmetic operations in double precision but using a more compact floating point format for memory operations can be used in two ways: One option is to replace memory-bound low precision numerical routines with accessor-based routines that achieve the same performance but reduce rounding errors as all arithmetic operations and accumulations are done in a higher precision format. The second option is to replace memory-bound high precision numerical routines with accessor-based routines that sacrifice some accuracy but speed up the application.

For the first use case, we employed the memory accessor in BLAS 2 operations on NVIDIA GPUs. In Figure 21 we visualize the performance (left) and accuracy (right) of accessor-BLAS routines that use single precision for all memory operations and double precision for all arithmetic operations. The visualizations reveal that the accessor-BLAS routines match the single precision BLAS routines in performance, often even outperform the cuBLAS routines. This implies that the accuracy gain compared to single precision BLAS (right-hand side in Figure 21) indeed comes for free.

The second use case for the memory accessor is more involved as it requires algorithm technology able to cope with increased rounding effects either by compensating them or accepting them as design choice, e.g. in the form of preconditioners. In Ginkgo, we use the memory accessor to store Krylov basis vectors in Compressed Basis Krylov solvers (see Section 4.2) or to reduce the memory footprint of preconditioners (see Section 5).

9. Software featuring mixed- and multiprecision functionality

9.1 GINKGO

Ginkgo is a modern linear algebra library engineered towards performance portability, and productivity. To achieve these goals, the library design is guided by combining ecosystem extensibility with heavy, architecture-specific kernel optimization using the platform-native languages CUDA (NVIDIA GPUs), HIP (AMD GPUs), DPC++ (Intel GPUs) and OpenMP (Intel/AMD/ARM multicore). Ginkgo also aims for user-friendliness in the context of designing and using mixed- and multiprecision algorithms. In fact, Ginkgo employs three mechanisms to enable productivity and performance in this aspect: 1) Ginkgo features the memory accessor that encapsulates on-the-fly compression for decoupling the memory precision from the arithmetic precision, see Section 8 This allows to accelerate the performance of memory-bound algorithms that can compensate or tolerate some information loss in the memory operations. 2) Ginkgo allows to mix & match precisions in the sense of combining linear operators and vectors stored in different precision formats without explicit conversion. 3) Ginkgo features production-ready mixed- and multiprecision algorithms like mixed precision SAI preconditioning Section 5 or CB-GMRES Section 4.2. 1) Ginkgo uses a static template

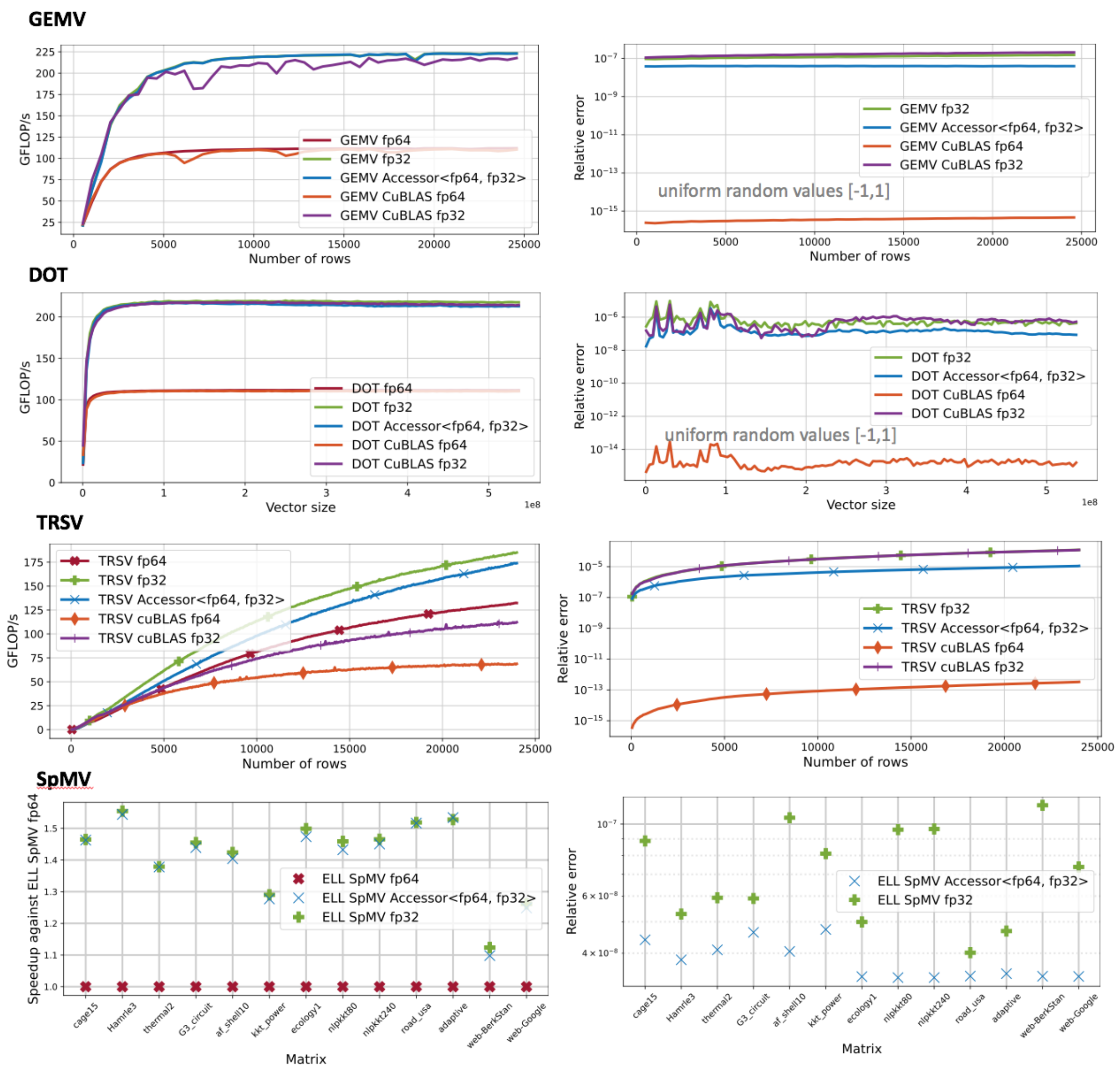


Figure 21: Performance (left) and accuracy (right) of accessor-BLAS routines in comparison to double precision and single precision BLAS. The values are uniform random distributed values in $[-1, 1]$. All experiments are run on an NVIDIA V100 GPU.

parameter for the value type and a template parameter for the integer type to allow for compilation in different precision formats. Standard value type formats supported are IEEE double precision, IEEE single precision, double complex precision, and single complex precision.

9.2 KOKKOS CORE, KOKKOS KERNELS, AND TRILINOS ADDITIONS

Trilinos, Kokkos Core and Kokkos Kernels have incorporated several mixed- and multiprecision capabilities recently. First, we have added half precision capabilities in Kokkos Core. This allows ECP applications and software technology projects to use a portable arithmetic traits implementation for half precision and use Kokkos programming model paradigms (e.g. reduction) with half precision types. We also support using a higher precision for certain operations (e.g. atomics) for performance reasons. Second, we have used these Kokkos functionality within Kokkos Kernels to support half precision sparse/dense linear algebra. In addition, Trilinos now has capabilities for single precision MueLu (multigrid) and IfPack2 (factorization-based) preconditioners within its linear solvers. The GMRES-IR solver has a Trilinos implementation and a Kokkos-only implementation for single node evaluation. We are also introducing mixed- and multiprecision linear algebra kernels in Kokkos Kernels for exploiting tensor cores better. This work is in progress.

9.3 MAGMA

MAGMA provides dense (BLAS and LAPACK), sparse, and batched linear algebra routines for single nodes with GPUs. Multiple precisions are supported and a number of mixed-precision iterative refinement solvers based on LU, Cholesky, and QR factorizations. MAGMA v2.6.1 was released on July 13, 2021, adding HIP support for AMD GPUs (former hipMAGMA). Support for AMD GPUs is now full in the four standard precisions (single and double real and complex), as well as the mixed-precision iterative refinement solvers.

9.4 HEFFTE

The heFFTe library provides multidimensional FFTs in multiple precisions. For this period new optimizations for AMD and Nvidia GPUs were added, as well as support for Intel GPUs. The current release is heFFTe v2.1 from April 2021.

9.5 PLASMA

PLASMA [63] (Parallel Linear Algebra for Scalable Multicores and Accelerators) is a numerical software library with dense linear system solvers, least squares problems, eigenvalue and singular value solvers. PLASMA heavily relies on modern OpenMP functionality such as data-dependent tasking and target-offload pragma directives. The former allow dynamic generation and traversal of DAG of tasks based on their data dependences thus allowing explicit control of data sharing and load balancing. The latter allows data placement and transfer with execution on the hardware accelerators.

9.6 PETSC

The Portable Extensible Toolkit for Scientific computation (PETSc) library delivers scalable solvers for nonlinear time-dependent differential and algebraic equations and for numerical optimization. It is written to enable users writing applications to separate themselves from the performance issues associated with high-performance computing, including accelerator support. This separation will allow PETSc users from C/C++, Fortran, or Python to employ their preferred GPU programming model, such as Kokkos, RAJA, SYCL, HIP, CUDA, or OpenCL [64, 65, 66, 67, 68, 69], on upcoming exascale systems. In all cases, users will be able to rely on PETSc's large assortment of composable, hierarchical, and nested solvers, as well as advanced time-stepping and adjoint capabilities and numerical optimization methods running on the GPU. A more detailed discussion of the PETSc approach may be found in Ref. [70].

PETSc's goal for multi-precision is to maintain the runtime flexibility of our users while enabling the performance gains enabled by reduced precision on the accelerators. The approach to mixed precision in PETSc focuses on developing an abstraction layer to vendor algebra libraries (cuBLAS/cuSPARSE, rocBLAS/rocSPARSE, ...), and other libraries that support mixed precision such as Ginkgo [71]. The multi-precision work is closely tied to the work on incorporating backends in PETSc discussed above. Our current work focuses on creating the correct abstractions to expose these cutting-edge capabilities while having a code base that is robust and maintainable. Looking forward, enhancing multi-precision capabilities are planned in PETSc 4 (a planned refactor of PETSc that takes advantage of the lessons learned in PETSc over the past decade, and new hardware and software capabilities), while maintaining an easy upgrading path for our large user base.

9.7 HYPRE

As previously noted in section 6.2, the hypre linear solver library currently supports independent builds in single, double, and longdouble precisions. Efforts are currently underway to make the library support a unity build that will enable one to create and access single, double, and longdouble variants of the same (or different) hypre solver, and use them interoperably. We have demonstrated this capability on a proxy that includes key components of the hypre solver library, and successfully developed new mixed-precision solvers from this build. Current efforts are focused on enabling this capability on the entire hypre library in a systematic way that is portable and easy to maintain.

Acknowledgments

This work was supported by the US Exascale Computing Project (17-SC-20-SC), a collaborative effort of the U.S. Department of Energy Office of Science and the National Nuclear Security Administration. This work was performed under the auspices of the U.S. Department of Energy by Lawrence Livermore National Laboratory under Contract DE-AC52-07NA27344. LLNL-TR-825909.

References

[1] Ahmad Abdelfattah, Hartwig Anzt, Erik G. Boman, and Erin Carson et al. A Survey of Numerical Methods Utilizing Mixed Precision Arithmetic. *International J. of High-Performance Computing Applications*, 2021. to appear.

[2] Mark Gates, Jakub Kurzak, Ali Charara, Asim YarKhan, and Jack Dongarra. Slate: Design of a modern distributed and accelerated linear algebra library. In *International Conference for High Performance Computing, Networking, Storage and Analysis (SC19)*, Denver, CO, 2019-11 2019. ACM, ACM.

[3] Kadir Akbudak, Paul Bagwell, Sebastien Cayrols, Mark Gates, Dalal Sukkari, Asim YarKhan, and Jack Dongarra. Slate performance improvements: Qr and eigenvalues. Technical Report 17, ICL-UT-21-02, 2021-04 2021.

[4] Azzam Haidar, Stanimire Tomov, Jack Dongarra, and Nicholas J. Higham. Harnessing GPU Tensor Cores for Fast FP16 Arithmetic to Speed Up Mixed-precision Iterative Refinement Solvers. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage, and Analysis*, SC '18, pages 47:1–47:11, Piscataway, NJ, USA, 2018. IEEE Press.

[5] Azzam Haidar, Harun Bayraktar, Stanimire Tomov, Jack Dongarra, and Nicholas J. Higham. Mixed-precision iterative refinement using tensor cores on gpus to accelerate solution of linear systems. Technical report, 2020-11 2020.

[6] Cade Brown, Ahmad Abdelfattah, Stanimire Tomov, and Jack Dongarra. Design, optimization, and benchmarking of dense linear algebra algorithms on amd gpus. In *2020 IEEE High Performance Extreme Computing Conference (HPEC)*, pages 1–7, 2020.

[7] Ahmad Abdelfattah, Azzam Haidar, Stanimire Tomov, and Jack J. Dongarra. Analysis and Design Techniques towards High-Performance and Energy-Efficient Dense Linear Solvers on GPUs. *IEEE Transactions on Parallel and Distributed Systems*, 29(12):2700–2712, 2018.

[8] Jack J Dongarra. Improving the accuracy of computed matrix eigenvalues. Technical report, Argonne National Lab., IL (USA); Argonne National Lab.(ANL), Argonne, IL, 1980.

[9] Jack J Dongarra. Algorithm 589: Sicedr: A fortran subroutine for improving the accuracy of computed matrix eigenvalues. *ACM Transactions on Mathematical Software (TOMS)*, 8(4):371–375, 1982.

[10] Jack J Dongarra, Cleve B Moler, and James Hardy Wilkinson. Improving the accuracy of computed eigenvalues and eigenvectors. *SIAM Journal on Numerical Analysis*, 20(1):23–45, 1983.

[11] J. Demmel, Y. Hida, W. Kahan, X.S. Li, S. Mukherjee, and E.J. Riedy. Error bounds from extra-precise iterative refinement. *ACM Trans. Math. Softw.*, 32(2):325–351, June 2006.

[12] J. Demmel, Y. Hida, E.J. Riedy, and X.S. Li. Extra-precise iterative refinement for overdetermined least squares problems. *ACM Transactions on Mathematical Software (TOMS)*, 35(4):28, 2009.

[13] Jennifer A. Loe, Christian Glusa, Ichitaro Yamazaki, Erik Boman, and Sivasankaran Rajamanickam. Experimental evaluation of multiprecision strategies for GMRES on GPUs. *2021 IEEE International Parallel and Distributed Processing Symposium Workshops (IPDPSW)*, pages 469–478, 2021.

[14] Erin Carson and Nicholas J. Higham. A New Analysis of Iterative Refinement and Its Application to Accurate Solution of Ill-Conditioned Sparse Linear Systems. *SIAM Journal on Scientific Computing*, 39(6):A2834–A2856, 2017.

[15] Erin Carson and Nicholas J. Higham. Accelerating the Solution of Linear Systems by Iterative Refinement in Three Precisions. *SIAM Journal on Scientific Computing*, 40(2):A817–A847, 2018.

[16] Neil Lindquist, Piotr Luszczek, and Jack Dongarra. Improving the Performance of the GMRES Method using Mixed-Precision Techniques. In *Smoky Mountains Conference Proceedings*, 2020.

[17] Eric Bavier, Mark Hoemmen, Sivasankaran Rajamanickam, and Heidi Thornquist. Amesos2 and Belos: Direct and iterative solvers for large sparse linear systems. *Scientific Programming*, 20(3):241–255, 2012.

[18] Michael A. Heroux, Roscoe A. Bartlett, Vicki E. Howle, Robert J. Hoekstra, Jonathan J. Hu, Tamara G. Kolda, Richard B. Lehoucq, Kevin R. Long, Roger P. Pawlowski, Eric T. Phipps, Andrew G. Salinger, Heidi K. Thornquist, Ray S. Tuminaro, James M. Willenbring, Alan Williams, and Kendall S. Stanley. An overview of the Trilinos project. *ACM Trans. Math. Softw.*, 31(3):397–423, September 2005.

[19] H. Carter Edwards, Christian R. Trott, and Daniel Sunderland. Kokkos: Enabling manycore performance portability through polymorphic memory access patterns. *Journal of Parallel and Distributed Computing*, 74(12):3202–3216, 2014. Domain-Specific Languages and High-Level Frameworks for High-Performance Computing.

[20] Jennifer A. Loe, Heidi K. Thornquist, and Erik G. Boman. Polynomial Preconditioned GMRES in Trilinos: Practical Considerations for High-Performance Computing. In *Proceedings of the 2020 SIAM Conference on Parallel Processing for Scientific Computing*, pages 35–45, 2020.

[21] José I Aliaga, Hartwig Anzt, Thomas Grützmacher, Enrique S Quintana-Ortí, and Andrés E Tomás. Compressed basis gmres on high performance gpus. *arXiv preprint arXiv:2009.12101*, 2020.

[22] Mark Hoemmen. *Communication-avoiding Krylov subspace methods*. PhD thesis, 2010.

[23] Erin Claire Carson. *Communication-avoiding Krylov subspace methods in theory and practice*. PhD thesis, University of California, Berkeley, 2015.

[24] Erin Carson and Tomáš Gergelits. Mixed precision s-step Lanczos and conjugate gradient algorithms. *arXiv preprint arXiv:2103.09210*, 2021.

[25] C. C. Paige. Accuracy and effectiveness of the Lanczos algorithm for the symmetric eigenproblem. *Lin. Alg. Appl.*, 34:235–258, 1980.

[26] Timothy A Davis and Yifan Hu. The University of Florida sparse matrix collection. *ACM Transactions on Mathematical Software (TOMS)*, 38(1):1–25, 2011.

[27] Serge Gratton, Ehouarn Simon, David Titley-Peloquin, and Philippe Toint. Exploiting variable precision in GMRES. *SIAM J. Sci. Comput. (to appear)*, 2020.

[28] Åke Björck. Solving linear least squares problems by Gram-Schmidt orthogonalization. *BIT Numerical Mathematics*, 7(1):1–21, 1967.

[29] Christopher C Paige. Accuracy and effectiveness of the Lanczos algorithm for the symmetric eigenproblem. *Lin. Alg. Appl.*, 34:235–258, 1980.

[30] Christopher C Paige, Miroslav Rozložník, and Zdeněk Strakoš. Modified gram-schmidt MGS, least squares, and backward stability of MGS-GMRES. *SIAM J. Matrix Anal. Appl.*, 28(1):264–284, 2006.

[31] Christopher C Paige. The effects of loss of orthogonality on large scale numerical computations. In *International Conference on Computational Science and Its Applications*, pages 429–439. Springer, 2018.

[32] Luc Giraud, Serge Gratton, and Julien Langou. A rank-k update procedure for reorthogonalizing the orthogonal factor from modified Gram–Schmidt. *SIAM J. Matrix Anal. Appl.*, 25(4):1163–1177, 2004.

[33] Christopher C Paige and Zdeněk Strakoš. Residual and backward error bounds in minimum residual Krylov subspace methods. *SIAM J. Sci. Comput.*, 23(6):1898–1923, 2002.

[34] Jasper van den Eshof and Gerard LG Sleijpen. Inexact Krylov subspace methods for linear systems. *SIAM J. Matrix Anal. Appl.*, 26(1):125–153, 2004.

[35] Valeria Simoncini and Daniel B Szyld. Theory of inexact Krylov subspace methods and applications to scientific computing. *SIAM J. Sci. Comput.*, 25(2):454–477, 2003.

[36] K. Świrydowicz, J. Langou, S. Ananthan, U. Yang, and S. J. Thomas. Low synchronization Gram-Schmidt and GMRES algorithms. *Numer. Lin. Alg. Appl.*, 2020.

[37] Nicholas J Higham. The accuracy of solutions to triangular systems. *SIAM J. Numer. Anal.*, 26(5):1252–1265, 1989.

[38] Jesse L Barlow. Block modified Gram–Schmidt algorithms and their analysis. *SIAM J. Matrix Anal. Appl.*, 40(4):1257–1290, 2019.

[39] Åke Björck. Numerics of Gram-Schmidt orthogonalization. *Lin. Alg. Appl.*, 197:297–316, 1994.

[40] J. Malard and C.C. Paige. Efficiency and scalability of two parallel QR factorization algorithms. In *Proceedings of IEEE Scalable High Performance Computing Conference*, pages 615–622. IEEE, 1994.

[41] Steven J Leon, Åke Björck, and Walter Gander. Gram-Schmidt orthogonalization: 100 years and more. *Numer. Lin. Alg. Appl.*, 20(3):492–532, 2013.

[42] Chiara Puglisi. Modification of the Householder method based on the compact WY representation. *SIAM J. Sci. Stat. Comput.*, 13(3):723–726, 1992.

[43] Thierry Joffrain, Tze Meng Low, Enrique S Quintana-Ortí, Robert van de Geijn, and Field G Van Zee. Accumulating Householder transformations, revisited. *ACM Transactions on Mathematical Software (TOMS)*, 32(2):169–179, 2006.

[44] Homer F Walker. Implementation of the GMRES method using Householder transformations. *SIAM J. Sci. Stat. Comput.*, 9(1):152–163, 1988.

[45] Xiaobai Sun. Aggregations of elementary transformations. Technical Report DUKE-TR-1996-03, Duke University, Durham, NC, 1996.

[46] Åke Björck and Christopher C Paige. Loss and recapture of orthogonality in the modified Gram-Schmidt algorithm. *SIAM J. Matrix Anal. Appl.*, 13(1):176–190, 1992.

[47] Christopher C Paige and Wolfgang Wülling. Properties of a unitary matrix obtained from a sequence of normalized vectors. *SIAM J. Matrix. Anal. Appl.*, 35(2):526–545, 2014.

[48] Alicja Smoktunowicz, Jesse L Barlow, and Julien Langou. A note on the error analysis of classical Gram–Schmidt. *Numerische Mathematik*, 105(2):299–313, 2006.

[49] Fritz Goebel, Thomas Grützmacher, and Hartwig Anzt. Mixed precision incomplete sparse approximate inverses. In *European Conference on Parallel Processing*, page accepted. Springer, Cham, 2021.

[50] Rasmus Tamstorf, Joseph Benzaken, and Stephen McCormick. Algebraic error analysis for mixed precision multigrid solvers. *SIAM Journal on Scientific Computing*, 2020. submitted.

[51] Rasmus Tamstorf, Joseph Benzaken, and Stephen McCormick. Discretization-error-accurate mixed precision multigrid solvers. *SIAM Journal on Scientific Computing*, 2020. submitted.

[52] Michael A. Heroux, Lois Curfman McInnes, Rajeev Thakur, Jeffrey S. Vetter, Xiaoye Sherry Li, James Aherns, Todd Munson, and Kathryn Mohror. Ecp software technology capability assessment report. 11 2020.

[53] Stanimire Tomov, Azzam Haidar, Alan Ayala, Daniel Schultz, and Jack Dongarra. Design and Implementation for FFT-ECP on Distributed Accelerated Systems. ECP WBS 2.3.3.09 Milestone Report ICL-UT-19-05, 2019-04 2019.

[54] Alan Ayala, S. Tomov, A. Haidar, and Jack Dongarra. heFFTe: Highly Efficient FFT for Exascale. In *ICCS 2020. Lecture Notes in Computer Science*, 2020.

[55] Interim report on benchmarking fft libraries on high performance systems. ICL Tech Report ICL-UT-21-03, 2021-07 2021.

[56] Alan Ayala, Stanimire Tomov, Xi Luo, Hejer Shaiek, Azzam Haidar, George Bosilca, and Jack Dongarra. Impacts of Multi-GPU MPI Collective Communications on Large FFT Computation. In *Workshop on Exascale MPI (ExaMPI) at SC19*, Denver, CO, 2019-11 2019.

[57] Alan Ayala, Stanimire Tomov, Miroslav Stoyanov, and Jack Dongarra. Scalability issues in FFT computation. In Victor Malyshkin, editor, *Parallel Computing Technologies*, pages 279–287, Cham, 2021. Springer International Publishing.

[58] P. Lindstrom. ZFP version 0.5.3, April.

[59] James Diffenderfer, Alyson Fox, Jeffrey Hittinger, Geoffrey Sanders, and Peter Lindstrom. Error analysis of zfp compression for floating-point data, 2018.

[60] Alyson Fox and Avary Kolasinski. Error analysis of inline ZFP compression for multigrid methods. 2019 Copper Mountain Conference for Multigrid Methods.

[61] Large-scale atomic/molecular massively parallel simulator, 2018. Available at https://lammps.sandia.gov/.

[62] JD Emberson, N. Frontiere, S. Habib, K. Heitmann, A. Pope, and E. Rangel. Arrival of First Summit Nodes: HACC Testing on Phase I System. Technical Report MS ECP-ADSE01-40/ExaSky, Exascale Computing Project (ECP), 2018.

[63] Emmanuel Agullo, Jack Dongarra, Bilel Hadri, Jakub Kurzak, Julie Langou, Julien Langou, Hatem Ltaief, Piotr Luszczek, and Asim YarKhan. Plasma users guide. Technical report, Technical report, ICL, UTK, 2009.

[64] H. Carter Edwards, Christian R. Trott, and Daniel Sunderland. Kokkos: Enabling manycore performance portability through polymorphic memory access patterns. *Journal of Parallel and Distributed Computing*, 74(12):3202–3216, 2014. Domain-Specific Languages and High-Level Frameworks for High-Performance Computing.

[65] David A Beckingsale, Jason Burmark, Rich Hornung, Holger Jones, William Killian, Adam J Kunen, Olga Pearce, Peter Robinson, Brian S Ryujin, and Thomas RW Scogland. RAJA: Portable performance for large-scale scientific applications. In *2019 IEEE/ACM International Workshop on Performance, Portability and Productivity in HPC (P3HPC)*, pages 71–81. IEEE, 2019.

[66] Khronos SYCL Working Group. SYCL specification: Generic heterogeneous computing for modern C++, 2020. https://www.khronos.org/-registry/SYCL/specs/sycl-2020-provisional.pdf.

[67] NVIDA. CUDA C++ programming guide, 2020. https://docs.nvidia.com/cuda/-pdf/CUDA_C_Programming_Guide.pdf.

[68] AMD. HIP Programming Guide v4.3. 2021.

[69] John E Stone, David Gohara, and Guochun Shi. OpenCL: A parallel programming standard for heterogeneous computing systems. *Computing in science & engineering*, 12(3):66–73, 2010.

[70] Richard Tran Mills, Mark F. Adams, Satish Balay, Jed Brown, Alp Dener, Matthew Knepley, Scott E. Kruger, Hannah Morgan, Todd Munson, Karl Rupp, Barry F. Smith, Stefano Zampini, Hong Zhang, and Junchao Zhang. Toward performance-portable PETSc for GPU-based exascale systems. *Parallel Computing*, 2021 (submitted).

[71] Hartwig Anzt, Terry Cojean, Yen-Chen Chen, Goran Flegar, Fritz Göbel, Thomas Grützmacher, Pratik Nayak, Tobias Ribizel, and Yu-Hsiang Tsai. Ginkgo: A high performance numerical linear algebra library. *Journal of Open Source Software*, 5(52):2260, 2020.