SAND2022-13948C

High-Performance GMRES Multi-Precision Benchmark: Design, Performance, and Challenges

Ichitaro Yamazaki*, Christian Glusa*, Jennifer Loe*, Piotr Luszczek†,
Sivasankaran Rajamanickam*, and Jack Dongarra†
*Sandia National Laboratories, Albuquerque, New Mexico, USA
†University of Tennessee, Knoxville, Tennessee, USA

***Abstract*— We propose a new benchmark for high-performance (HP) computers. Similar to High Performance Conjugate Gradient (HPCG), the new benchmark is designed to rank computers based on how fast they can solve a sparse linear system of equations, exhibiting computational and communication requirements typical in many scientific applications. The main novelty of the new benchmark is that it is now based on Generalized Minimum Residual method (GMRES) (combined with Geometric Multi-Grid preconditioner and Gauss-Seidel smoother) and provides the flexibility to utilize lower precision arithmetic. This is motivated by new hardware architectures that deliver lower-precision arithmetic at higher performance. There are other machines that do not follow this trend. However, using a lower-precision arithmetic reduces the required amount of data transfer, which alone could improve solver performance. Considering these trends, an HP benchmark that allows the use of different precisions for solving important scientific problems will be valuable for many different disciplines, and we also hope to promote the design of future HP computers that can utilize mixed-precision arithmetic for achieving high application performance. We present our initial design of the new benchmark, its reference implementation, and the performance of the reference mixed (double and single) precision Geometric Multi-Grid solvers on current top-ranked architectures. We also discuss challenges of designing such a benchmark, along with our preliminary numerical results using 16-bit numerical values (half and bfloat precisions) for solving a sparse linear system of equations.**

I. Introduction

Table I lists current top-ranked HP computers, some emerging architectures, and the early-access platforms for the US Exascale Computing Project (ECP). These platforms represent current hardware trends: though typical scientific applications require double precision accuracy, some emerging hardware for HP scientific applications can deliver lower-precision arithmetic at higher performance. There are other machines that do not follow this trend and provide the same performance for double and single precision arithmetic. However, even in that case, using a lower-precision arithmetic reduces the required amount of data transfer. Since the performance of scientific applications on the HP computer is often limited by the communication bandwidth, if not by communication latency, reducing the required communication volume alone could reduce the simulation time. Moreover, many emerging architectures can perform the arithmetic operations in a precision lower than single (e.g. in half precision), which can deliver significantly higher performance and further reduce the communication volume. Considering these trends, there is a growing interest in utilizing lower-precision arithmetic capabilities for scientific applications.

In response to these growing interests, there are several efforts to investigate multi-precision algorithms on emerging computers [5]. This includes the xSDK multi-precision project funded by the ECP [1]. Since a significant portion of scientific simulation time may be spent solving sparse linear systems of equations, xSDK's effort includes development of multi-precision algorithms for solving such linear systems, numerical and performance studies of the resulting multi-precision solvers, and deployment of resulting mixed-precision software.

In this paper, we propose a new benchmark that can harness emerging hardware trends, as well as algorithmic and software efforts to utilize lower precision for solving important scientific or engineering problems. Specifically, the new benchmark ranks the computers based on how fast they can solve a sparse linear system of equations (exhibiting computational and communication requirements typical in many scientific applications), while providing the flexibility to utilize lower precision arithmetic. Considering the current hardware and software trends, an HP benchmark that allows the use of different precisions for solving important scientific problems will be valuable for many different disciplines.

II. Related Work

There are three popular benchmarks for HP computers, which are closely related to the one proposed in this paper. The first, and oldest, is the High Performance Linpack (HPL) benchmark [15] that measures the performance (floating point operations per second, or flop/s in short) of computers solving a dense linear system of equations in double precision. It is based on a dense LU factorization. With a proper implementation, HPL performance is dominated by the dense matrix-matrix multiply (**GEMM**). This kernel exhibits uniform memory access with enough data reuses and parallelism to fully utilize the compute power of current HP computers. As a result, HPL can obtain close to the peak computational performance of the target computer. It is used to rank HP computers for the Top500 list [4] and provides historical data since its release. On the other hand, the compute and communication patterns demonstrated by HPL may not be typical in many of the current scientific or engineering applications, especially

Sandia National Laboratories is a multimission laboratory managed and operated by National Technology & Engineering Solutions of Sandia, LLC, a wholly owned subsidiary of Honeywell International Inc., for the U.S. Department of Energy's National Nuclear Security Administration under contract DE-NA0003525.

	Benchmark ranking (Pflop/s) HPL, HPCG, HPL-AI rank (Pflop/s)	GPU	GPU Peak Performance (TFlop/s) FP64	FP32	FP16
Frontier (ORNL)	1 (1,102), N/A, 1 (6,861)	AMD MI250X	26.5	26.5	191
Fugaku (RIKEN)	2 (442.0), 1 (16.0), 2 (2,000)	Fujitsu A64 FX	3.4	6.7	13.5
Summit (ORNL)	4 (148.6), 2 (2.9), 3 (1,411)	NVIDIA V100	7.5	19.5	N/A
Perlmutter (NERSC)	7 (70.8), 4 (1.9), 5 (590)	NVIDIA A100	9.7	19.5	312
Spock (ORNL)	N/A	AMD MI100	11.5	23.1	184
Crusher (ORNL)	N/A	AMD MI250X	26.5	26.5	191

TABLE I: Current top-ranked HP computers (June 2022), and emerging architectures [4], [2].

those applications that aim to utilize the full HP computers on the Top500 list.

When simulating physical systems on a parallel computer, a significant portion of the simulation time may be spent solving large sparse linear systems of equations arising from the discretization of partial differential equations (PDEs). In many applications, an iterative method is used for solving these linear systems. This led to the development of High Performance Conjugate Gradient (HPCG) benchmark [14] that tests the HP computer's ability to solve a sparse symmetric positive definite (SPD) linear system. The benchmark uses an iterative Krylov solver, Conjugate Gradient (CG) [19], combined with Geometric Multi-Grid (GMG) preconditioning and a symmetric variant of a Gauss-Seidel (GS) smoother. This is a popular solver configuration used in many scientific applications, but more importantly, unlike compute-bound HPL, the resulting sparse solver exhibits computation and data access patterns that are more commonly found in scientific applications. In particular, HPCG exhibits a more irregular memory access pattern, less parallelism, and a lower ratio of computation to communication. As a result, HPCG performance is often limited by communication latency or bandwidth (see the HPL and HPCG performance gaps in Table I). Its benchmark performance is meant to represent typical application performance. It is designed to rank computers based on how fast they can solve an important problem that appears in the applications targeting top HP computers and to promote the design of future HP computers that achieve high application performance.

Recently, a newer benchmark called High Performance Linpack for Accelerator Introspection (HPL-AI) [2] was released. Like HPL, the benchmark measures performance of a dense linear system solve, but here it uses a mixed-precision algorithm. Specifically, the compute-intensive LU factorization, which is typically the most time-consuming part of the benchmark, is computed in a lower precision, while iterative refinement is performed to obtain double precision accuracy of the computed solution. Computers that deliver lower-precision arithmetic at higher performance often obtain much higher HPL-AI performance than HPL performance (see Table I).

HPL-AI addresses the current hardware trends. However, it only tests the machine's ability to handle compute-intensive tasks. We are missing a benchmark to test the machine's ability to exploit mixed-precision arithmetic for solving a sparse linear system with computation and communication patterns common in scientific applications. Lack of such a benchmark may lead the hardware and scientific communities to neglect both recent hardware trends (wasting available higher compute power at lower precision) and the recent algorithmic and software efforts to utilize lower-precision algorithms.

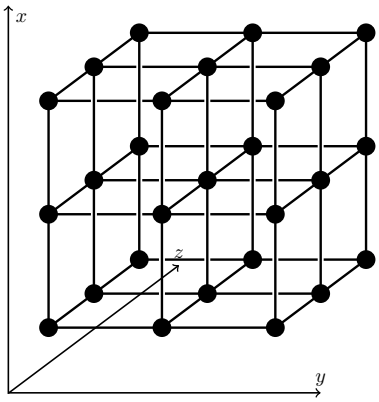
The new benchmark aims to fill this gap. Similar to HPL-AI, the new benchmark is based on mixed-precision iterative refinement. However, while HPL-AI uses direct LU factorization for solving a dense linear system, the new benchmark uses an iterative Krylov solver, the Generalized Minimum Residual method (GMRES) [28], to solve a non-symmetric sparse linear system. The benchmark allows the use of lower precisions for the GMRES iteration, but to obtain double-precision solution accuracy, the approximate solution is updated in double precision. This algorithm dates back to a paper published by Turner and Walker in 1992 [30], but in recent years it has gained more attention ([11], [12], [6], [7], [31]), especially with the recent hardware trends discussed in Section I. We alternatively considered allowing the use of lower-precision arithmetic in HPCG (e.g., a preconditioner or SpMV in lower precision), but GMRES with mixed-precision iterative refinement often gives more robust solution convergence, especially for ill-conditioned problems (making the validation easier). In addition, both solvers rely on similar computational kernels that are commonly found in scientific applications, but GMRES has more balanced use of sparse and dense operations. (The full orthogonalization process of GMRES uses more dense operations than the three-term recurrence of CG.)

The new benchmark is named High Performance GMRES with Multi Precision (HPGMP). We stress that it is designed to measure the computer's capability to utilize lower precisions for the computation and communication patterns common in scientific applications. Thus, we do *not* aim to develop a scalable or robust linear solver. In fact, for a weak-parallel scaling, similar to HPCG, the number of HPGMP iterations required for convergence increases with the number of processes.

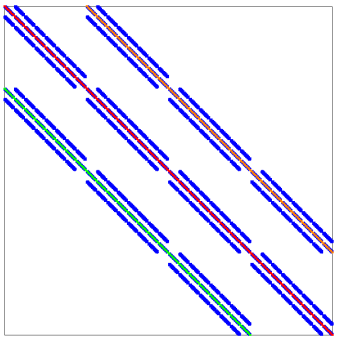
Table II summarizes these four benchmarks. Considering the current hardware, algorithmic, and software trends, an HP benchmark that allows the use of different precisions for solving a realistic scientific problem will be valuable to many different disciplines.

	Dense Problem (Compute Intensive)	Real-World Sparse Problem (Communication Intensive)
Uniform Precision	HPL	HPCG
Multiple Precisions	HPL-AI	HPGMP (new)

TABLE II: HP Benchmarks



(a) 27-point stencil.



(b) Local submatrix with $n_x = n_y = n_z = 8$, where diagonal red points, and off-diagonal blue, green, and orange points have numerical values of 26, -1, $-1-\beta$, and $-1+\beta$, respectively.

Fig. 1: 27-point stencil problem used for the benchmark.

III. Problem Description

As in HPCG, the benchmark problem is on a regular 3D grid with a 27-point stencil as shown in Figure 1. For HPGMP, however, we use a parameter β to allow a non-symmetric numerical value pattern (the diagonal and off-diagonal entries have the numerical values of 26 and -1, except for the off-diagonal entries, which correspond to the edges connecting to the point directory above and below, have the numerical values of $-1-\beta$ and $1+\beta$, respectively). The matrix corresponds to a finite difference discretization of an advection-diffusion problem. We have tested the benchmark with $\beta = 0.0$ and $\beta = 0.5$. In this paper, we show results with $\beta = 0.0$, which corresponds to the same SPD problem that HPCG solves. Compared with $\beta = 0.5$, non-preconditioned GMRES for $\beta = 0.0$ needed more iterations to converge and GMG resulted in a greater reduction in the iteration count.

For the benchmark run, the user specifies the dimension (n_x, n_y, n_z) of the submatrix local to each MPI process (e.g., largest size allowed by the available local memory). The global matrix is distributed on a 3D process grid (p_x, p_y, p_z), and hence, the global matrix dimensions are $(n_x p_x, n_y p_y, n_z p_z)$.

The right-hand-side vector is computed such that the entries of the exact solution vector are all one. The initial approximate solution vector is a vector of all zeros.

Input: input vector $y \in \mathbb{R}^{n\times 1}$ **Output:** output vector x
1: $x = 0$,
2: **if** not coarse grid **then**
3: Update x by applying Pre-smoothing to x
4: Compute residual vector $r = y - Ax$
5: Compute input vector for coarse grid by applying restriction operator R to r, $y_c = Rr$
6: Call GMG on a coarser grid, GMG(y_c, x_c)
7: Compute x by applying prolongation P to x_c, $x = Px_c$
8: Update x by applying Post-smoothing to x
9: **else**
10: Compute x by applying forward GS to r
11: **end if**

Fig. 2: GMG preconditioner.

IV. Algorithm Description

HPGMP is based on the Generalized Minimum Residual method (GMRES) [28], an iterative Krylov solver for solving a nonsymmetric linear system of equations $Ax = b$. At its jth iteration, GMRES generates an orthonormal basis vector of the Krylov subspace $\mathcal{K}_{j+1} = \text{span}(r_0, (AM)r_0, (AM)^2 r_0, \ldots, (AM)^j)$, where $r_0 = b - Ax_0$ is the initial residual vector given by the initial approximate solution x_0, and M it a preconditioner used to accelerate GMRES convergence. It then computes the approximate solution that minimizes the residual norm over the generated projection subspace.

The benchmark consists of the following main components:

- To accelerate GMRES convergence, it uses three levels of a Geometric Multigrid (GMG) preconditioner with one forward sweep of Gauss-Seidel (GS) for pre and postsmoothing on each level. The restriction operator halves the number of points in each direction of the 3D mesh such that the coarser grid has 8 times fewer points than the finer grid. One forward sweep of the GS iteration is used at the final coarse level. Figure 2 shows the GMG pseudocode. The HPCG benchmark uses the same GMG preconditioner but with a symmetric Gauss-Seidel smoother.
- After GMG, we apply a sparse-matrix multiply (SpMV) on the resulting vector. The generated Krylov basis vector is then orthogonalized against the previous basis vectors using the Classical Gram Schmidt process with reorthogonalization (CGS2).
- As the iteration proceeds, the cost of storing and computing the basis vectors becomes expensive. To reduce the cost, the iteration is restarted after a fixed number of basis vectors are computed. At the restart, the approximate solution is updated such that the residual norm is minimized on the Krylov subspace generated. The resulting residual vector is used as the starting vector for the next cycle.

Figure 3 shows the pseudocode of restarted GMRES, where the restart length of 30 is used for the benchmark results in this paper (i.e., $m = 30$).

Input: $A \in \mathbb{R}^{n\times n}$, $b \in \mathbb{R}^{n\times 1}$, initial guess $x_0 \in R^{n\times 1}$, relative residual tolerance $rTol$
Output: approximate solution x_m

1: $r = b - Ax$,
2: $\gamma = \|r\|_2$
3: **while** not converged **do**
4: $v_1 = r_0/\gamma$, and $h_{1,1} = 0$
5: **for** $j = 1 : m$ **do**
6: *// GMG preconditioner M, followed by SpMV*
7: $w_j = AMv_j$
8: *// CGS2 orthogonalization*
9: $w_j = w_j - V_j t_j$ with $t_j = V_j^T w_j$
10: $h_{1:j,j} = t_j$
11: $w_j = w_j - V_j t_j$ with $t_j = V_j^T w_j$
12: $h_{1:j,j} = h_{1:j,j} + t_j$
13: $h_{j+1,j} = \|w_j\|_2$
14: $v_{j+1} = w_j/h_{j+1,j}$
15: **end for**
16: $\hat{d} = \arg\min_{y\in\mathbb{R}^m} \|\gamma e_1 - H_{1:m+1,1:m}y\|_2$
17: $x = x + V_m\hat{d}$
18: $r = b - Ax$
19: $\gamma = \|r\|_2$
20: **end while**

Fig. 3: GMRES(m) with CGS2 orthogonalization and GMG preconditioner M. Steps in blue may be performed in lower precision; the remaining steps are in double precision.

The resulting solver exhibits many of the computational and communication patters that are typical in applications:

- Sparse matrix vector multiply (SpMV):
  - point-to-point communication among the 7 to 26 neighboring processes (`1, $n_x$, or $n_x^2$ elements to exchange for each SpMV`)
  - local sparse-matrix vector multiply on 27-point stencil (`Total of` $54nm$ `Flops / restart cycle`)
- Geometric Multigrid (GMG):
  - Restriction and prolongation of the vectors on the fine and coarse grids (`no MPI communication, local SpMV with a rectangular sparse matrix, restriction operator has one nonzero entry per row`).
  - Sparse matrix vector multiply on 27-point stencil of different size at each level (for computing residual vector). (`Total of` $(54 \times 73)/64nm$ `Flops / restart cycle`)
  - Local sparse triangular solves on 27-point stencil of different size at each level (for pre and post smoothing based on forward GS). (`Total of` $(54\times 73)/64nm$ `Flops / restart cycle`)
  - Sparse triangular solve as the coarse gird solver (`Total` $81/512nm$ `Flops / restart cycle`)
- Orthogonalization (CGS2):
  - BLAS-2 dot-product that requires local atomic operations and global reduction among all the processes. (`Total of` $2n(1 + m)m$ `Flops / restart cycle`)
  - BLAS-2 vector update that is embarrassingly parallel. (`Total of` $2n(1 + m)m$ `Flops / restart cycle`)

With $m = 30$, we perform about the same number of flops for GMG and CGS2.

The benchmark allows the use of lower precision arithmetic for the GMRES iterations (Lines 5 through 16 of Figure 3), while to obtain double-precision solution accuracy, the approximate solution is updated in double precision (Lines 17 through 19). To maintain stability, the starting vector for the inner-iteration is also scaled in double precision (Line 4). The algorithm originated in [30] but several variants of the solver have been proposed and analyzed (e.g., using LU factorization in a lower precision as a preconditioner [11], [12], using a lower precision for orthogonalization [18], using up to five different precisions with low-rank compression in lower precision to approximate an LU-based preconditioner [6], [7]). The potential of such multi-precision solvers to obtain higher performance than the uniform-precision solver has been demonstrated in recent papers [24], [25], [31].

In Sections VII and IX, we discuss the potential uses of lower precision in the new benchmark, along with other allowed optimizations.

V. Reference GMRES Implementation

The reference implementation of the HPGMP benchmark is available at [3]. It uses many components from the HPCG reference implementation (e.g. problem construction, GMG, SpMV, etc.). These components are used to generate the basis vectors in parallel, while the least-square problem is solved by each MPI process, redundantly (Line 16 of Figure 3). One difference is that HPGMP uses C++ templates to make it easier to use different scalar types. Figure 4 shows the solver interfaces for the reference implementation of GMRES.

The reference implementation also contains CUDA and HIP backends for running with NVIDIA and AMD GPUs, respectively. It uses GPUs for generating the basis vectors, while the least-square problem is solved on a CPU. It still relies on MPI for exchanging data between the MPI processes, but all local operations are performed solely relying on vendor libraries. Hence, the reference implementation does not contain any custom CUDA or HIP codes. Specifically, the CUDA backend relies on the CuBLAS, CuSparse, and CUDA libraries:

- CuBLAS is used to implement CGS2 orthogonalization.
- The GS smoother is implemented using CuSparse's SpMV and SpTRSV.
- The prolongation and restriction operators are based on CuSparse's SpMV.
- The CUDA library is used for memory management operations such as Malloc, Memcpy, and Memset.

Similarly, the HIP backend relies on rocBLAS, rocSparse, and HIP routines, with no custom HIP functions.

Because of the design choice to avoid custom CUDA or HIP code, we perform the halo-exchange among neighboring MPI processes for SpMV on the CPU. Hence, each MPI process first copies all the vector entries local to the process to the CPU. (Local elements are stored contiguously in memory before the interface elements, such that the local entries can be copied to CPU in a single CudaMemcpy call.) Then each

```
template<class SparseMatrix_type, class CGData_type, class Vector_type>
int GMRES(const SparseMatrix_type & A, CGData_type & data, const Vector_type & b, Vector_type & x,
          const int restart_length, const int max_iter, const typename SparseMatrix_type::scalar_type tolerance,
          int & niters, typename SparseMatrix_type::scalar_type & normr, typename SparseMatrix_type::scalar_type & normr0,
          double * times, bool doPreconditioning);
```

(a) Uniform-precision GMRES.

```
template<class SparseMatrix_type, class SparseMatrix_type2, class CGData_type, class CGData_type2, class Vector_type>
int GMRES_IR(const SparseMatrix_type & A, const SparseMatrix_type2 & A_lo,
             CGData_type & data, CGData_type2 & data_lo, const Vector_type & b_hi, Vector_type & x_hi,
             const int restart_length, const int max_iter, const typename SparseMatrix_type::scalar_type tolerance,
             int & niters, typename SparseMatrix_type::scalar_type & normr, typename SparseMatrix_type::scalar_type & normr0,
             double * times, bool doPreconditioning);
```

(b) Mixed-precision GMRES.

Fig. 4: Solver interfaces for reference implementation.

process packs the local elements needed by each neighboring process and sends them to that process. Similarly, the received interface elements are copied to the appropriate nonlocal portion of the vector on the CPU and then copied back to the GPU. In addition, since neither NVIDIA nor AMD provides mixed-precision kernels (including copy), our mixed-precision GMRES copies the vector $\hat{d}$ to CPU for type-casting, and then copies back to GPU (Line 17 in Figure 3).

VI. Benchmark Description

Our benchmark consists of two steps (Figure 5). The first step is designed to verify that the optimized version of the solver, which is provided by the participant and potentially uses mixed-precision, can solve the linear system to double-precision accuracy. This verification is done by running the optimized solver on a fixed-size problem to a specified accuracy, using a fixed number of MPI processes. If the solver converges, the benchmark records the number of iterations needed for the optimized code to reach the specified solution accuracy. Otherwise, it is marked as a failure. The reference implementation of the solver is also run with the same parameters, and the number of iterations required for convergence is recorded. For our experiments, we set $n_x = n_y = n_z = 80$ and run to a tolerance of $\|b - Ax\|_2 < 10^{-9}\|b\|_2$. We used four MPI processes for the verification step in order to start the benchmark from a single node in our experiments (nodes of Spock and Crusher have only four GPUs). However, for the finalized benchmark, we may require the user to use a larger number of MPI processes for this step.

The second step benchmarks the optimized version of the solver. The benchmark runs the optimized solver multiple times for a fixed number of iterations (300 iterations $\times$ 10 runs) and for a minimum run time (30 minutes). Then it records the required number of flops and the total solve time. These values are used to compute the "raw" Gflop/s. The final Gflop/s value used for ranking (Eqn. 1) is penalized by the ratio of the reference to optimized iteration counts from Step 1. (The penalty reflects the fact that, while each mixed-precision iteration may be faster, the solver may perform more iterations than a uniform double precision solver to converge for a real-life problem.) The reasoning behind this two-step process is further explained in Section VIII-A.

Step 1: Verification

1) Run both optimized and reference solvers using a fixed problem size on a fixed number of MPI processes to a fixed tolerance.
   a) Record the required numbers of iterations for both optimized and reference solvers, i_o and i_r
   b) Compute penality factor $i_p = \min(1.0, i_r/i_o)$

Step 2: Benchmark

1) Run optimized solver for a fixed number of iterations using user-specified problem size and number of MPI processes
2) Repeat until reaching the minimum number of solves or the minimum time in minutes, required by the benchmark
   a) Record time and flop count
   b) Compute "raw" Gflop/s for optimized run
   c) Compute benchmark Gflop/s, $i_p \times$ "raw" Gflop/s.

Fig. 5: Benchmark steps.

name	value
Solver parameters	
restart cycle, m	30
GMG levels	3
GS sweeps	1
Step 1 (Validation)	
problem size (n_x, n_y, n_z)	(80,80,80)
convergence tol	10^{-9}
# of MPI procs	4
Step 2 (Benchmark)	
# of iterations	300
# of minimum solves	10
minimum time	30 minutes (disabled)

TABLE III: Parameters.

$$\textbf{Final Gflops/s} = \min\left(1.0, \frac{\text{reference iteration count}}{\text{optimized iteration count}}\right) \times \text{"raw" Gflop/s} \quad (1)$$

Required parameter values and those used in our experiments are summarized in Table III. Some of the parameter

values are chosen for convenience (e.g., "# of MPI procs" in the Verification step, and "# of minimum solves" and "minimum time" in Benchmark step). We also did not include the setup time needed for CUDA and HIP backends when computing the Gflop/s; this value will be included in the real benchmark runs but will be amortized over the multiple solves.

Our reference implementation is coupled with the benchmark suite and is available at [3].

VII. Allowed Optimizations

Benchmark optimizations are allowed as follows:

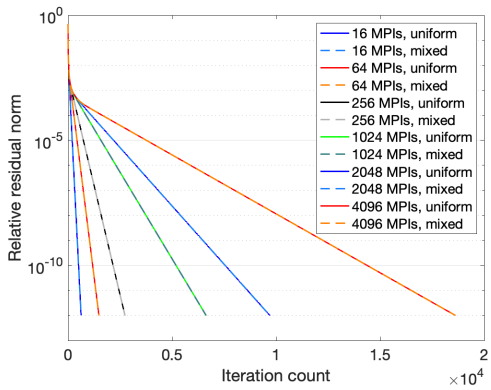
- Hardware-specific optimizations are allowed (e.g. specialized data structures or communication schemes).
- The matrix may be permuted/reordered for the GS smoother to exploit more parallelism. If the matrix permutation leads to an increase in the iteration count for the Verification phase of the benchmark, the benchmark performance will be penalized in the Benchmark phase.
- Any precision may be used for the inner GMRES iteration. Again, if the iteration count increases using lower precision, then the benchmark performance will be penalized in the second phase.
- Matrix scaling is not allowed. It may be possible to scale the vectors and/or matrices so that their numerical values fit within the numerical range of a lower precision [21]. However, matrix scaling may also be used as a preconditioner to improve convergence and, thus, is prohibited.
- Algorithmic changes are not allowed. These include s-step/communication-avoiding [22], pipelined [17], or randomized [8] variants of GMRES. Also prohibited are single-reduce or other variants of CGS orthogonalization [10] and iterative variants of Gauss Seidel [29], [9].
- It is prohibited to use knowledge of the structure or spectral properties of the problem (i.e. the matrix should be treated as a general matrix and both the numerical values and indices of the non-zero entries should be explicitly loaded for SpMV).

VIII. Numerical and Performance Results with Reference Implementation

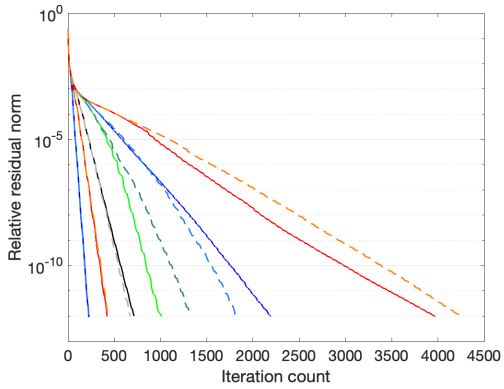
The best-known variation of the mixed-precision GMRES algorithm uses single (fp32) precision for the inner GMRES iterations (blue lines in Figure 3). We now show the parallel weak-scaling results of this mixed-precision variation of the reference benchmark implementation.

A. Numerical Convergence Studies

The following solver convergence results was performed on the (CPU-only) Skybridge cluster at Sandia National Labs. Each node has two 8-core 2.6 GHz Intel Sandy Bridge CPUs. Recall that, per the linear problem setup, the size of the 3D grid local to each MPI process is fixed. We set $n_x = n_y = n_z = 40$ and test both the uniform (all fp64) and mixed (fp32-fp64) precision solvers using up to 4096 MPI processes. Figure 6 shows the convergence of the benchmark solvers. The bottom plot in the figure uses the required GMG preconditioner. For



(a) Without preconditioner.



(b) With GMG preconditioner.

Fig. 6: Comparison of convergence histories using uniform and mixed (double and single) precision ($n_x = 40$) on SkyBridge Supercomputer at Sandia National Labs.

comparison, the top plot shows convergence for the solver with no preconditioning. We make the following two points:

1) Even with the GMG preconditioner, the solver is not weak-scalable; the iteration count increases significantly with increases in the number of MPI processes. This poses a challenge in validating and benchmarking the optimized code since it indicates that enforcing double-precision accuracy of the solution on a large number of MPI ranks requires too many iterations to be practical. This demonstrates the reason that the Benchmark step (Step 2) runs the solver to a fixed number of iterations rather than to a particular tolerance.

2) The figure demonstrates another difference in iteration counts between the solvers: Without preconditioning, the problem needs many more iterations to converge, but the convergence curves of the uniform and mixed precision solvers match, and both solvers use (essentially) the same number of iterations. This behavior has been observed in [18], [24], [25], [31]. On the other hand, with GMG preconditioning (applied lower precision), the mixed-precision solver could require a different number of iterations to converge than the uniform-precision preconditioned solver, especially on a large number of MPI processes. (The figure shows that the mixed-

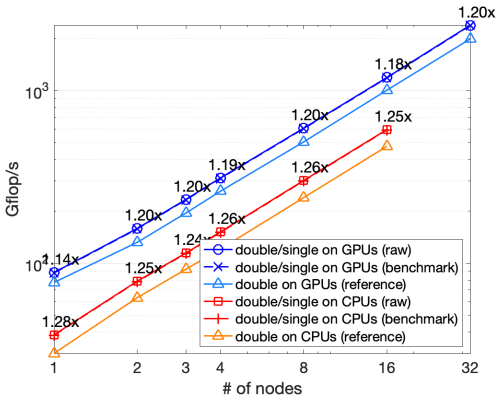
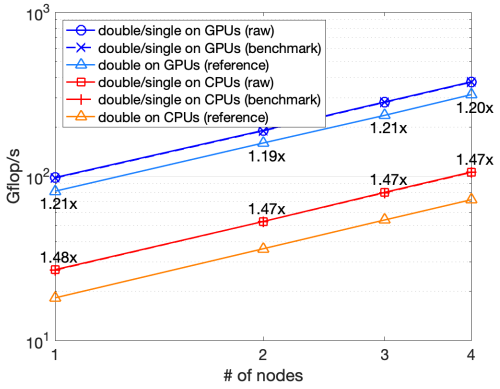
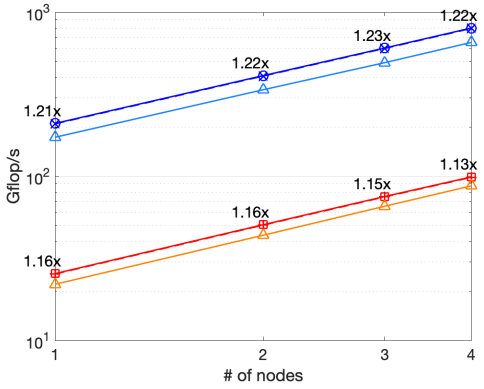


Fig. 7: Benchmark performance results on Summit (6 NVIDIA V100 GPUs or 42 IBM Power9 CPU cores per node) using CUDA 10.1.168. Speedups are mixed-precision over uniform-precision.



(a) Spock (4 AMD MI100 GPUs or 64 AMD EPYC 7662 CPU cores per node) using ROCm 4.5.0.



(b) Crusher (8 AMD MI250X GPUs or 63 AMD EPYC 7AA53 CPU cores per node) using ROCm 4.5.0.

Fig. 8: Benchmark performance results on AMD GPUs. Speedups are mixed-precision over uniform-precision.

precision solver could require either more or fewer iterations to converge. Use of mixed-precision arithmetic in multi-grid has been analyzed in [26].) Since the Verification is run on a smaller number of processes, it may not capture the increase in the iteration count using low precision on the larger number of processes used for the Benchmark step. Moreover, if the difference in the required numbers of uniform and mixed precision iterations increases with a larger number of MPI processes, then we may not want to capture the difference and penalize the participant, who is running the benchmark on a larger number of processes, with a larger factor.

We stress that the new benchmark is designed to measure the computer's capability to perform computation and communication typical for HPC applications, while providing the flexibility to use various precisions; it is not meant to be a scalable solver for the selected linear system (defined in Section III). For this purpose, we feel that the Verification step (Step 1) provides sufficient information about the optimized solver's performance in achieving the required solution accuracy: It captures and penalizes the benchmark results if the iteration count increases with use of lower precisions in the validation phase. This reflects the reality that the mixed-precision solver may need to perform more floating-point operations than a uniform precision solver for a real-life problem.

B. Parallel Weak-Scaling Performance Results

Next we benchmark the solver performance on top-ranked HP computer architectures at Oak Ridge Leadership Computing Facility: Summit supercomputer with NVIDIA GPUs, and Spock and Crusher with AMD GPUs. (See Table I for the GPU peak performance.) We enabled OpenMP for both CPU and GPU builds, and ran the benchmark with the default number of threads. The codes were compiled using the optimization flag `-O3`.

When solver performance is bound by the bandwidth needed to move matrix values, indices, and vector values, the maximum speedup which can be obtained using single precision arithmetic is about $1.6\times$ (maximum speedup of about $2.5\times$ for local SpMVs was reported in [25] due to a reduction in local cache misses on the GPU). Figures 7 and 8 show performance of the mixed-precision reference implementation compared to the uniform-precision implementation as computed by the Benchmark step (see Section VI). These results demonstrate that it is possible to obtain speedups of up to $1.2\times$ with our reference implementation, even without any optimizations. We note that even though the AMD MI250X GPU provides the same peak performance for double and single precision computations, mixed-precision GMRES still obtains speedups (because the amount of data to be moved was reduced by half).

In the figures, "raw" and "benchmark" Gflop/s are essentially the same because with the parameters used for the experiments (in Table III), the uniform and mixed precision GMRES converged very similarly for our validation phase (using just four processes).

Table IV shows the breakdown of solve times and the performance obtained by individual algorithm components on NVIDIA V100 GPUs (all of these data are part of the Benchmark output). We ran GMRES to a relative residual norm tolerance of 10^{-12}. The performance of the reference

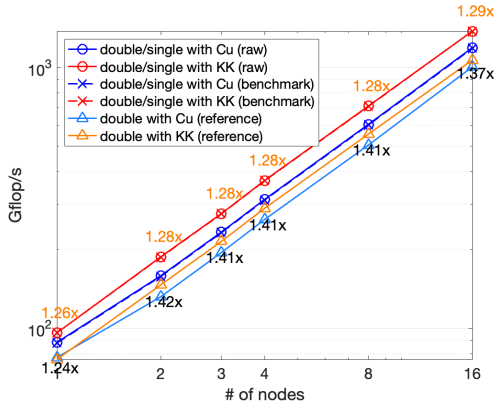
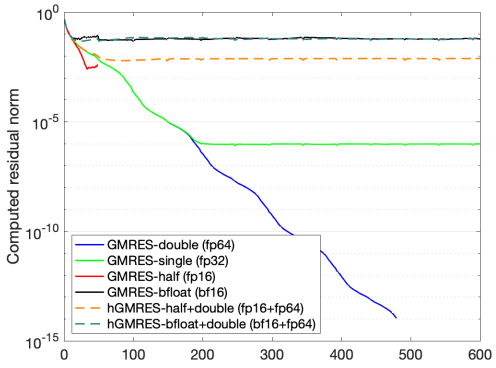
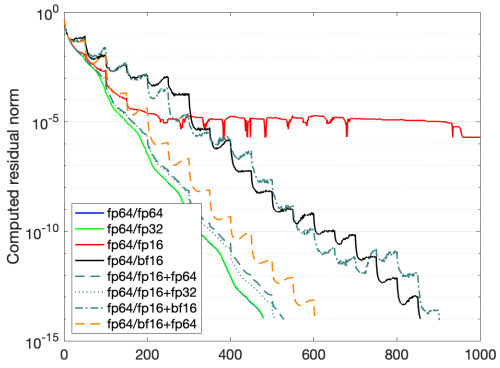


Fig. 9: Comparison of Benchmark performance on Summit (6 NVIDIA V100 GPUs) using CUDA 10.1.168. and Kokkos-Kernels 3.6.1. Speedups indicate mixed-precision Kokkos-Kernels over uniform-precision Kokkos-Kernels (orange) and over uniform-precision CuSparse (black).



(a) Convergence history of uniform-precision GMRES. ('hGMRES' computes and stores dot products in higher precision.)



(b) Convergence history of mixed-precision GMRES (with iterative refinement).

Fig. 10: Experimental results using 16-bit half precision and bfloat precision on an NVIDIA A100 GPU. The problem is a standard 3D Laplacian with no preconditioning.

implementation is dominated by SpTRSV within GMG. Compared to other kernels, SpTRSV has less parallelism to be exploited on a GPU, and its performance may be more limited by latency. As a result, GMG obtained the lowest performance and the smallest speedup among the main components.

To see if an optimized implementation can obtain higher speedup, Figure 9 shows benchmark results on the Summit supercomputer, where we replaced the reference GS smoother implementation (based on CuSparse) with the implementation in Kokkos-Kernels [13], [27]. We see higher speedups can be obtained using more optimized codes.

IX. Discussions

Before we conclude, we discuss some potential extensions and challenges of the benchmark.

A. Potential of 16-bit Precisions

Though our reference implementation is templated with only two precisions (one for the solution vector and the other for the "inner" GMRES iteration), our goal is to allow arbitrary precision for any subset of the inner GMRES iteration operations, while capturing any increase in the iteration count with the validation phase. Here we examine the potential of using 16-bit precisions, either half (fp16) or bfloat (bf16) on a single NVIDIA A100 GPU (NERSC Perlmutter supercomputer). Table V shows the machine epsilons and representative numerical ranges for precisions used in the experiments.

Figure 10 shows initial numerical results using GMRES without preconditioning to solve a standard Laplacian on a 3D 7-point stencil ($n_x = 50$). Our experiments use Kokkos and Kokkos-Kernels [16], [27] for the underlying linear algebra. Though we only show performance on an NVIDIA GPU, Kokkos enables thread-parallel performance that is portable to different manycore architectures using a single code base.

Figure 10a shows the convergence history of uniform-precision GMRES in different precisions. For orthogonalization, we used Kokkos-Kernels GEMV, but we implemented our own mixed-precision dot product for computing vector norms. Both single and bfloat precisions provide a wide enough range of representative numerical values for stable GMRES convergence (even though GMRES stagnates at a higher residual norm than in double precision, due to the larger machine epsilon). On the other hand, half precision's numerical range is much smaller, and GMRES encounters numerical breakdown after a few iterations. In our experiments, numerical issues typically arose when computing dot products in GMRES.

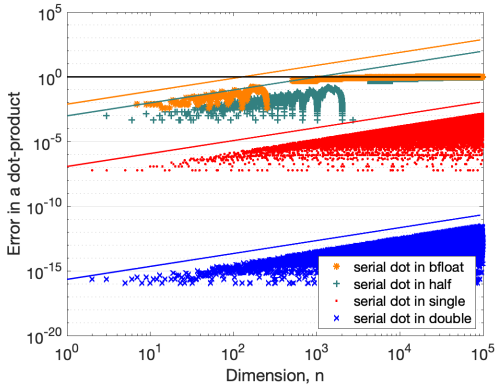
For more insight, Figure 11 shows the numerical errors from computing a dot-product of a unit-norm vector in different precisions. We test two implementations: i) a serial code where each vector element is multiplied and accumulated in sequence and ii) a hierarchical parallel code. Numerical errors of the serial implementation follow the expected upper bound of $\mathcal{O}(n\epsilon)$. The parallel implementation has much smaller numerical errors, with magnitude almost independent of the vector length. This is partially expected as the error depends on the block size used for the parallel implementation rather than on the full vector length [20, Section 3.1]. This suggests that while parallel dot-products using half or bfloat precision within

	Time (s)						Gflop/s			
	Setup	Opt	GMG	SpMV	Ortho	Total	GMG	SpMV	Orth	Total
uniform			51.5	3.8	2.5	60.2	298.6	1195.7	4130.3	504.7
mixed	3.9	5.0	44.5	2.4	1.8	50.1	345.6	1867.1	5729.5	605.6
speedup			1.16	1.56	1.39	1.20	1.15	1.56	1.39	1.20

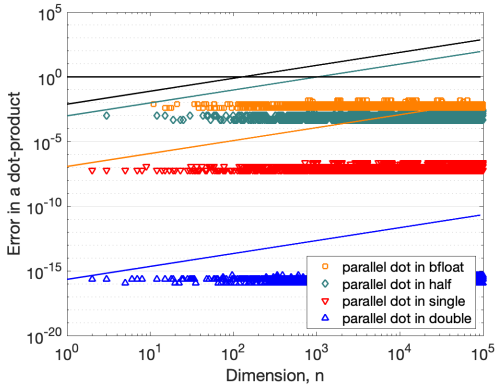
TABLE IV: Breakdown of iteration time with 48 GPUs on 8 Summit nodes.

	fp64	fp32	fp16	bf16
maximum	$1.80 \cdot 10^{308}$	$3.40 \cdot 10^{38}$	$6.55 \cdot 10^{4}$	$3.39 \cdot 10^{38}$
minimum	$2.22 \cdot 10^{-308}$	$1.17 \cdot 10^{-38}$	$5.96 \cdot 10^{-8}$	$1.18 \cdot 10^{-38}$
epsilon	$2.22 \cdot 10^{-16}$	$1.19 \cdot 10^{-7}$	$9.77 \cdot 10^{-4}$	$7.81 \cdot 10^{-3}$

TABLE V: Numerical precisions used in the experiments.



(a) Serial implementation.



(b) Parallel implementation.

Fig. 11: Numerical errors from dot-products of a unit-norm vector with numerical value of each entry set to be $1/n^{1/2}$. The solid lines show $n\epsilon$.

GMRES have larger round-off errors than those with higher precisions, their round-off errors are not amplified by the vector length. We note that the Kokkos-Kernels implementation of dot-product (used in the GMRES runs for Figure 10) had similar numerical errors to our own parallel implementation, even though for half or bfloat precision, Kokkos-Kernels accumulates the dot-product in single precision before type-casting to the output scalar type.

When computing the dot-product of the basis vectors in GMRES, their resulting numerical values are upper-bounded by $\|AM^{-1}\|^2\|v_j\|^2$, where the norm of the matrix A is within the numerical range of the precision (e.g., $\|A\|_1 = 56$), and v_j is normalized to have unit-norm. However, the numerical overflow of dot-products is still possible, especially in half precision (e.g., when the residual norm becomes small, the normalization of the residual vector at restart could cause significant round-off errors, and the norm of the resulting starting vector v_j may significantly deviate from one). Numerical issues may also arise when the dot-product is rounded to zero, causing "lucky" breakdown [28] before the solution converges. This numerical issue still arose even when using the Kokkos-Kernels implementation of dot-product in half precision, which casts the numerical values to single precision for computing the dot-product. Since the final value is cast back down to half precision, the solver still fails if the computed single-precision result is outside the half precision numerical range limit.

We found that this numerical issue may be avoided if we use a higher precision for computing dot-products *and* for storing the final computed value (including the residual norm at restart). In Figure 10a, our implementation of this "hybrid" GMRES (hGMRES in the figure) uses half or bfloat for the GMRES iteration but uses double precision for computing and storing the dot-products. As shown in the figure, storing dot product results in double precision helps the half (fp16) precision solver avoid breakdown.

Finally, Figure 10b shows the convergence history of several mixed-precision GMRES variations. Here, dotted lines are with hGMRES that computes and stores dot-products in precision higher than that GMRES iteration uses (specified after the '+' sign in the legend). Though it may require more iterations, mixed-precision GMRES with bfloat or half precision can converge to double precision accuracy when the dot-product is implemented carefully using a higher precision. In particular, hGMRES using a combination of half and single precisions for the "inner" iteration converged similarly to uniform double precision GMRES; the combination of half and bfloat resulted in a slower convergence rate (CUDA does not currently support intrinsic casting between half and bfloat, so we casted half to float before casting to bfloat).

These numerical results demonstrate the potential of using various precisions for the benchmark implementation in order to improve the computational performance or reduce the communication volume.

B. Smoother Options

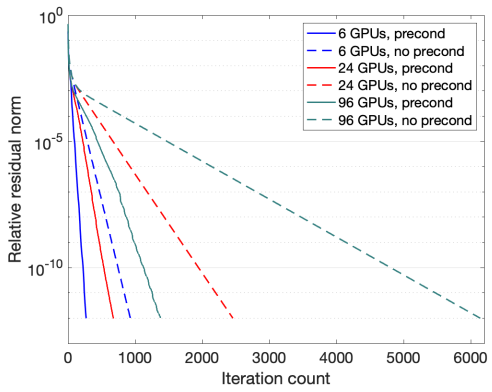
While we want HPGMP to provide flexibility to utilize various precisions, the benchmark needs to validate that the mixed-precision solver still converges to double-precision accuracy. If lower-precision arithmetic leads to an increase in the iteration

count, then solver performance should be penalized "appropriately" to account for the additional operations required to solve a real-life problem to convergence.

Figure 12 shows an extreme case, where we compare the iteration count and time-to-solution of our reference implementation on NVIDIA V100 GPUs, with and without GMG preconditioning. Without preconditioning, GMRES requires more iterations to converge but has a shorter time-to-solution. Hence, if GMG performance can be improved by using a lower precision, then participants could use the lowest possible precision for preconditioning and obtain the highest possible performance, despite more iterations. The Validation phase must capture this increase in the iteration count. However, we note that our reference implementation relies on vendor-optimized but generic libraries, which may not be ideal for this particular case. With an optimized version of GS smoothing, preconditioned GMRES may be faster than non-preconditioned GMRES. This is one of our current investigation focuses (see Section X).

As we have seen, the performance and speedup available from using lower precision arithmetic are primarily limited by the GS smoother with SpTRSV. There have been extensive studies to improve the parallel performance of GS and SpTRSV on a GPU, but they may still not provide enough parallelism to utilize the emerging manycore architectures, and its performance may be limited by latency. While GS with SpTRSV is a well-known and robust smoother option, there are trends that favor other options with more parallelism (e.g. a polynomial smoother with SpMV, though this option is typically only available for symmetric matrices). We are considering whether the benchmark should capture these trends.

The speedup gained from using lower precision provides good improvements in run-time for scientific applications. However, the HPGMP speedup is much smaller than the speedups obtained by the compute-bound HPL-AI benchmark, due to the nature of sparse computation and communication patterns, which are typical in applications. (The maximum speedup for HPGMP is about $1.6\times$ using single precision, compared to $5 \sim 10\times$ obtained for HPL-AI). A potential concern is that participants will not be motivated to optimize and run the benchmark for such a relatively small speedup.



(a) Convergence plot.

	No precond			GMG precond		
#GPUs	#iters	time-to-sol	time/iter	#iters	time-to-sol	time/iter
6	929	2.00	0.002	271	4.53	0.016
24	2456	6.35	0.002	676	11.94	0.017
96	6141	14.97	0.002	1376	25.91	0.019

(b) Iteration time in seconds.

Fig. 12: Comparison of uniform-precision GMRES in double precision with or without preconditioner on Summit.

X. Conclusions and Next Steps

In this paper, we proposed a new benchmark called High-Performance GMRES with Multi Precision (HPGMP). The new benchmark ranks high-performance (HP) computers based on how well they perform computational and communication requirements typical in scientific applications while allowing use of lower precision arithmetic. The paper presented our initial design with the reference implementation and initial performance studies. Considering current hardware and software trends, an HP benchmark that allows use of different precisions for solving important scientific problems will be valuable for many disciplines.

We are making further performance and numerical investigations, and these results will be available soon, including:

- Using an "optimized" GS implementation, combined with matrix reordering to exploit more parallelism. This could significantly impact the performance of the benchmark. We have shown results using the default GS implementation in Kokkos-Kernels, but we are looking to see whether further improvements are possible [13], [23].
- Using half or bfloat precision for the GMG preconditioner. Since vendor libraries (like CuBLAS/CuSparse and rocBLAS/rocSparse) do not provide full coverage of half and bfloat precisions for the computational kernels our benchmark requires, this will require custom CUDA or HIP codes, or relying on libraries like Kokkos-Kernels.
- Running benchmark results on other top-ranked machines at larger scales, and on emerging new architectures.

We aim to develop HPGMP into a solid benchmark suite that promotes the design of future HP computers that achieve high application performance utilizing lower precisions.

Acknowledgment

This work was supported by the 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. Sandia National Laboratories is a multimission laboratory managed and operated by National Technology and Engineering Solutions of Sandia, LLC, a wholly owned subsidiary of Honeywell International, Inc., for the U.S. Department of Energy's National Nuclear Security Administration under contract DE-NA-0003525. This paper describes objective technical results and analysis. Any subjective views or opinions that might be expressed in the paper do not necessarily represent the views of the U.S. Department of Energy or the United States Government.

REFERENCES

[1] Extreme-scale scientific Software Development Kit for the Exascale Computing Project (xSDK4ECP). https://www.exascaleproject.org/research-project/xsdk4ecp.

[2] The High Performance LINPACK for Accelerator Introspection (HPL-AI) benchmark. https://icl.utk.edu/hpl-ai.

[3] Hpgmp webpage. https://github.com/iyamazaki/hpcg/tree/hpgmp-cuda.

[4] Top500. https://www.top500.org.

[5] Ahmad Abdelfattah, Hartwig Anzt, Erik G. Boman, Erin Carson, Terry Cojean, Jack Dongarra, Mark Gates, Thomas Grützmacher, Nicholas J. Higham, Sherry Li, Neil Lindquist, Yang Liu, Jennifer Loe, Piotr Luszczek, Pratik Nayak, Sri Pranesh, Siva Rajamanickam, Tobias Ribizel, Barry Smith, Kasia Swirydowicz, Stephen Thomas, Stanimire Tomov, Yaohung M. Tsai, Ichitaro Yamazaki, and Urike Meier Yang. A survey of numerical methods utilizing mixed precision arithmetic, 2020.

[6] Patrick Amestoy, Alfredo Buttari, Nicholas Higham, Jean-yves L'Excellent, Théo Mary, and Bastien Vieuble. Five-precision GMRES-based iterative refinement. 2021.

[7] Patrick Amestoy, Alfredo Buttari, Nicholas Higham, Jean-yves L'Excellent, Théo Mary, and Bastien Vieuble. Combining sparse approximate factorizations with mixed precision iterative refinement. Technical report, The University of Manchester, 2022.

[8] Oleg Balabanov and Laura Grigori. Randomized gram–schmidt process with application to gmres. *SIAM Journal on Scientific Computing*, 44(3):A1450–A1474, 2022.

[9] Luc Berger-Vergiat, Brian Kelley, Sivasankaran Rajamanickam, Jonathan Hu, Katarzyna Swirydowicz, Paul Mullowney, Stephen Thomas, and Ichitaro Yamazaki. Two-stage Gauss-Seidel preconditioners and smoothers for Krylov solvers on a GPU cluster, 2021.

[10] Daniel Bielich, Julien Langou, Stephen Thomas, Kasia Świrydowicz, Ichitaro Yamazaki, and Erik G. Boman. Low-synch Gram–Schmidt with delayed reorthogonalization for Krylov solvers. *Parallel Computing*, 112:102940, 2022.

[11] 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.

[12] 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.

[13] M. Deveci, E. G. Boman, K. D. Devine, and S. Rajamanickam. Parallel graph coloring for manycore architectures. In *IEEE Int. Par. Dist. Proc. Symp. (IPDPS)*, pages 892–901, 2016.

[14] Jack Dongarra, Michael A Heroux, and Piotr Luszczek. High-performance conjugate-gradient benchmark: A new metric for ranking high-performance computing systems. *The International Journal of High Performance Computing Applications*, 30:3–10, 2015.

[15] Jack Dongarra, Piotr Luszczek, and Antoine Petitet. The linpack benchmark: Past, present, and future. *Concurrency and Computation: Practice and Experience*, 15:803–820, 2003.

[16] 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.

[17] P. Ghysels, T. J. Ashby, K. Meerbergen, and W. Vanroose. Hiding global communication latency in the GMRES algorithm on massively parallel machines. *SIAM Journal on Scientific Computing*, 35(1):C48–C71, 2013.

[18] S. Gratton, E. Simon, D. Titley-Péloquin, and P. Toint. Exploiting variable precision in GMRES. *ArXiv*, abs/1907.10550, 2019.

[19] M. Hestenes and E. Stiefel. Methods of conjugate gradients for solving linear systems. *J. Nat. Bur. Standards*, 49:409–436, 1952.

[20] Nicholas J. Higham. *Accuracy and Stability of Numerical Algorithms*. Society for Industrial and Applied Mathematics, Philadelphia, PA, USA, 2nd edition, 2002.

[21] Nicholas J. Higham, Srikara Pranesh, and Mawussi Zounon. Squeezing a matrix into half precision, with an application to solving linear systems. *SIAM Journal on Scientific Computing*, 41(4):A2536–A2551, 2019.

[22] M. Hoemmen. *Communication-avoiding Krylov subspace methods*. PhD thesis, EECS Department, University of California, Berkeley, 2010.

[23] B. Kelley and S. Rajamanickam. Parallel, portable algorithms for distance-2 maximal independent set and graph coarsening. In *2022 IEEE International Parallel and Distributed Processing Symposium (IPDPS)*, pages 280–290, 2022.

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

[25] J. A. Loe, C. A. Glusa, I. Yamazaki, E. G. Boman, and S. Rajamanickam. Experimental evaluation of multiprecision strategies for gmres on gpus. In *2021 IEEE International Parallel and Distributed Processing Symposium Workshops (IPDPSW)*, pages 469–478, 2021.

[26] Stephen F. McCormick, Joseph Benzaken, and Rasmus Tamstorf. Algebraic error analysis for mixed-precision multigrid solvers. *SIAM Journal on Scientific Computing*, 43(5):S392–S419, 2021.

[27] Sivasankaran Rajamanickam, Seher Acer, Luc Berger-Vergiat, Vinh Dang, Nathan Ellingwood, Evan Harvey, Brian Kelley, Christian R Trott, Jeremiah Wilke, and Ichitaro Yamazaki. Kokkos Kernels: Performance portable sparse/dense linear algebra and graph kernels. *arXiv preprint arXiv:2103.11991*, 2021.

[28] Y. Saad and M. Schultz. GMRES: A generalized minimal residual algorithm for solving nonsymmetric linear systems. *SIAM Journal on Scientific and Statistical Computing*, 7:856–869, 1986.

[29] Daniel B. Szyld and Mark T. Jones. Two-stage and multisplitting methods for the parallel solution of linear systems. *SIAM Journal on Matrix Analysis and Applications*, 13(2):671–679, 1992.

[30] Kathryn Turner and Homer F. Walker. Efficient high accuracy solutions with GMRES(m). *SIAM J. Sci. Stat. Comput.*, 13(3):815–825, 1992.

[31] Yingqi Zhao, Takeshi Fukaya, Linjie Zhang, and Takeshi Iwashita. Numerical investigation into the mixed precision gmres(¡i¿m¡/i¿) method using fp64 and fp32. *Journal of Information Processing*, 30:525–537, 2022.