



LAWRENCE
LIVERMORE
NATIONAL
LABORATORY

Towards Use of Mixed Precision in ECP Math Libraries

H. Anzt, E. Boman, M. Gates, S. Kruger, X. Li, J. Loe,
D. Osei-Kuffuor, S. Tomov, Y. Tsai, U. M. Yang

November 11, 2020

Disclaimer

This document was prepared as an account of work sponsored by an agency of the United States government. Neither the United States government nor Lawrence Livermore National Security, LLC, nor any of their employees makes any warranty, expressed or implied, or assumes any legal liability or responsibility for the accuracy, completeness, or usefulness of any information, apparatus, product, or process disclosed, or represents that its use would not infringe privately owned rights. Reference herein to any specific commercial product, process, or service by trade name, trademark, manufacturer, or otherwise does not necessarily constitute or imply its endorsement, recommendation, or favoring by the United States government or Lawrence Livermore National Security, LLC. The views and opinions of authors expressed herein do not necessarily state or reflect those of the United States government or Lawrence Livermore National Security, LLC, and shall not be used for advertising or product endorsement purposes.

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

Towards Use of Mixed Precision in ECP Math Libraries

ECP Multiprecision Team (Lead: Hartwig Anzt)

Hartwig Anzt, Erik G. Boman, Mark Gates, Scott Kruger, Sherry Li, Jennifer Loe, Daniel Osei-Kuffuor, Stan Tomov, Yaohung M. Tsai, Ulrike Meier Yang

January 21, 2021

TABLE OF CONTENTS

1	Introduction	2
2	Ginkgo	2
3	heFFTe	8
4	hypre	10
	4.1 Approach:	11
	4.2 Current status and future plans:	12
5	MAGMA	12
6	PETSc/TAO	13
7	SLATE	15
8	SuperLU	16
9	Trilinos and KokkosKernels	17
	9.1 Software design	18
	9.2 Iterative Refinement Experiments with GMRES	18
	9.3 Kernel Speedup for GMRES-IR	19
	9.4 Polynomial Preconditioning	21
	9.5 Future work	21

1. Introduction

The use of multiple types of precision in mathematical software has the potential to increase its performance on new heterogeneous architectures. The xSDK project focuses both on the investigation and development of multiprecision algorithms as well as their inclusion into xSDK member libraries. This report summarizes current efforts on including and/or using mixed precision capabilities in the math libraries Ginkgo, heFFTe, hypre, MAGMA, PETSc/TAO, SLATE, SuperLU, and Trilinos, including KokkosKernels. It contains both numerical results from libraries that already provide mixed precision capabilities, as well as descriptions of the strategies to incorporate multiprecision into established libraries.

2. Ginkgo

Memory Accessor separating the arithmetic format from the memory format. In the GINKGO library, we realized the memory accessor separating the memory format from the arithmetic format. This allows to compact data before invoking memory operations which, for memory bound algorithms, results in a performance increase. However, compacting data via lossy compression generally has significant effects on the numerical properties of an algorithm. Not every algorithm can accept casting data to lower precision for the memory operations. We can identify at least two scenarios where the use of a lower precision format in the memory operations can be acceptable: 1) linear operators that are by design only approximations, e.g., preconditioners; and 2) iterative methods that can recover from using lower precision in the memory operations by performing additional iterations. For those, a accessor applying lossy compression can render performance benefits if the runtime savings coming from the reduced data access volume surpass the overhead in terms of inferior numerical properties, e.g., slower convergence. Thus, it is reasonable that the GINKGO team spent significant efforts on the design and prototype realization of a cross-platform accessor. The design document is available in the ECP-iteral confluence: <https://confluence.exascaleproject.org/display/STMS05/xSDK+Project+Documents?preview=/29001460/101223279/main.pdf>. For now, the GINKGO team has used this prototype accessor in two example algorithms that can be seen as role models for the two algorithm classes accepting lossy compression: 1) an adaptive precision bloc-Jacobi preconditioner; and 2) a compressed basis (CB-) GMRES.

Compressed Basis GMRES.

It is important to note that Krylov subspace methods that operate on long recurrences, such as GMRES, optimize the solution approximation in the generated Krylov subspace independently of how that subspace is constructed. Assembling the Krylov subspace out of A -orthonormal vectors is expected to optimize the attainable accuracy for a certain subspace dimension. However, the optimization may find a solution approximation of comparable quality in a subspace generated out of nearly A -orthogonal vectors that correspond, for example, to perturbed A -orthonormal vectors. Furthermore, as long as the perturbations remain of moderate magnitude, a lower solution approximation quality can be compensated via extending the subspace dimension by adding additional nearly A -orthogonal search directions. This strategy only breaks down if the additional search directions become linear combinations of those already spanned by the current basis. However, provided the perturbations are moderate and the restart parameters are chosen much smaller than the problem dimension n , this scenario remains unlikely.

In summary, if the solution optimization process is realized in high precision, the perturbations of the Krylov search direction should not prevent the convergence of the iterative algorithm to a solution approximation with acceptable accuracy. This observation opens the door to a memory accessor that compresses the Krylov search direction in memory. Concretely, our expectations are that:

- E1) using lossy compression, realized by maintaining the Krylov search directions in low precision in memory, will only have a small impact on the convergence of the iterative method;
- E2) a potential convergence degradation can be compensated by adding extra search directions; and
- E3) provided the Krylov search directions do not become linearly dependent, the attainable accuracy will continue to depend on the condition number of the problem and the arithmetic format employed for the solution optimization step, but remain unaffected by the format used to store the Krylov search directions.

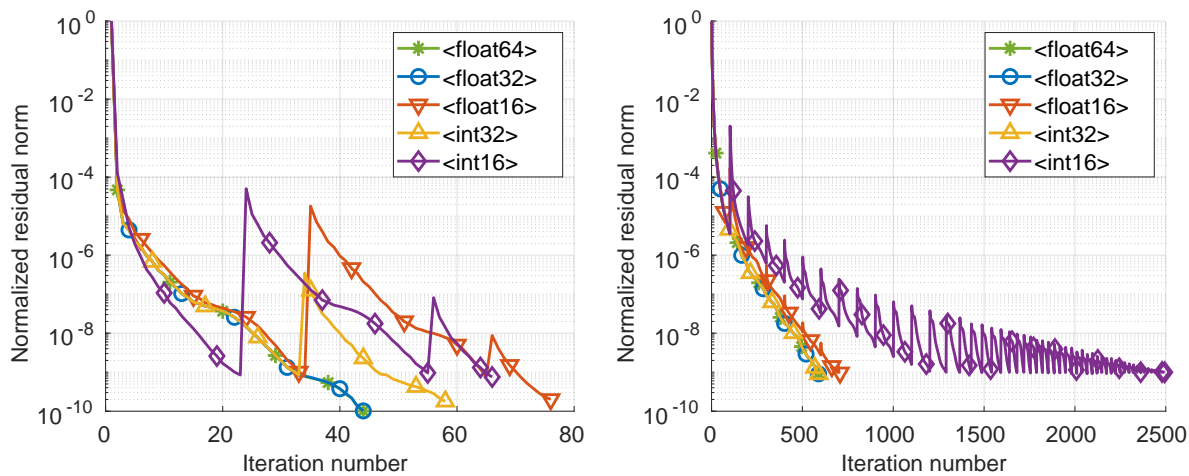


Figure 1: Convergence of the CB-GMRES variants for the `circuit5M.dc` and `Ser-ena` problems.

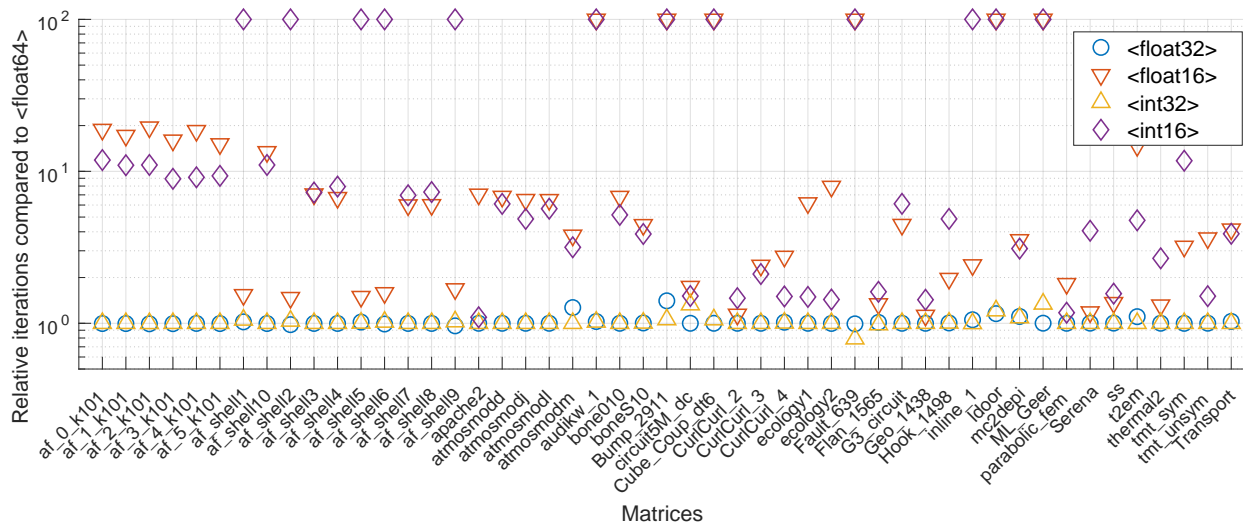


Figure 2: Iteration overhead of the CB-GMRES variants relative to the DP GMRES iteration count for a residual threshold.

Employing a memory accessor based on lossy compression for the Krylov search direction results in the CB-GMRES [1]. A high performance implementation of this algorithm is available in the GINKGO numerical linear algebra package [2]. In this realization, the user can choose among different formats for the storage of the orthonormal basis in memory: IEEE single precision and IEEE half precision, as well as 32-bit and 16-bit integer formats that represent the floating point values in the Krylov vectors as fractions relative to a reference value [1]. Independently of the format in memory for the Krylov search directions, all arithmetic operations (including the Krylov search direction computation) use IEEE double precision.

To validate E1) and E2), in Figure 1 we visualize the convergence of CB-GMRES for two example problems; and in Figure 2 we report the iteration overhead of CB-GMRES relative to the standard GMRES using IEEE double precision for all memory operations. We acknowledge that the convergence overhead can become relevant. However, for the problems/format combinations where the search directions remain linearly independent, the convergence degradation can be compensated with some extra iterations that generate additional Krylov search directions.

To answer the question of whether CB-GMRES can outperform the standard GMRES algorithm in practice, we implemented and ran both algorithms on heavily-optimized GPU backends [3]. The GINKGO implementations are algorithmically identical and use the same kernels for CB-GMRES and GMRES. Therefore, the only runtime differences come from the CB-GMRES algorithm employing the memory accessor for in-register compression/decompression of the Krylov search directions. In Figure 3 we display the speedup achieved by the CB-GMRES over the standard GMRES implementation on an NVIDIA V100 GPU (top) and the newer NVIDIA A100 GPU (bottom), respectively. We note that the use of IEEE single precision as target memory storage format improves the time-to-solution performance for all test cases. While the actual speedup values vary between $1.1\times$ and $1.75\times$, the speedup averages $1.4\times$ for both GPU generations. The performance benefits obtained using the customized 32-bit integer format are comparable. The 16-bit memory formats (both IEEE and integer) introduce perturbations that cause significant convergence delays, (see Figure 2) which cannot be compensated by the faster access to the Krylov search directions.

Adaptive Precision Block-Jacobi. When using a preconditioning scheme inside an iterative solver, an important consideration is whether the preconditioner is a constant operator or not. The reason is that, for example, Krylov iterative solvers working with short recurrences (e.g., CG, BiCGSTAB, CGS, etc.) expect the preconditioner to be a constant operator so that the orthogonality of Krylov search directions translates to the orthogonality of the preconditioned Krylov search directions. Obviously, a non-constant preconditioner breaks this assumption, and an additional, potentially expensive, orthogonalization step is needed to preserve the convergence of the iterative method. Thus, the idea of leveraging a preconditioner operating with a low precision format, inside an iterative method that performs the arithmetic on a high precision format, has to consider these effects carefully. The situation is different if the preconditioner decouples the memory format from the arithmetic format, and employs a lower precision format only for the memory operations but preserves the high precision format in all arithmetic operations. Then, even though of potentially lower quality, the preconditioner remains a constant operator. At the same time, many preconditioners, including the block-Jacobi preconditioner, are memory-bound on virtually all modern hardware architectures. Thus, the runtime cost of applying a preconditioner closely relates to the memory access volume, not the arithmetic operations. Taking both aspects into account, a preconditioner separating the memory format from the arithmetic format, and employing a lower precision format only for the memory operations, is numerically more attractive, and provides the same runtime benefits than a preconditioner using lower precision for both arithmetic and memory operations. This makes the memory accessor an attractive building block for high performance preconditioning. However, unlike its usage inside an iterative solver, the accessor has to account for the numerical properties of the preconditioner and potentially adjust the memory precision to the preconditioner quality. For the block-Jacobi preconditioner, the memory accessor has to: 1) preserve the regularity of the preconditioner (i.e., converting the inverted diagonal blocks does not turn a diagonal block singular); 2) keep the numeric values within the range of the format; and 3) preserve the quality of the preconditioner (i.e., the iteration count of the iterative solver should not be impacted by the preconditioner being stored in a lower precision format). A block-Jacobi preconditioner may choose a global memory precision format such that these requirements are fulfilled, and then store all inverted diagonal blocks in this precision format using the memory accessor. A more aggressive strategy determines the lowest precision format fulfilling the requirements on the local basis, for each block individually. While the latter approach naturally increases the complexity, it generally allows

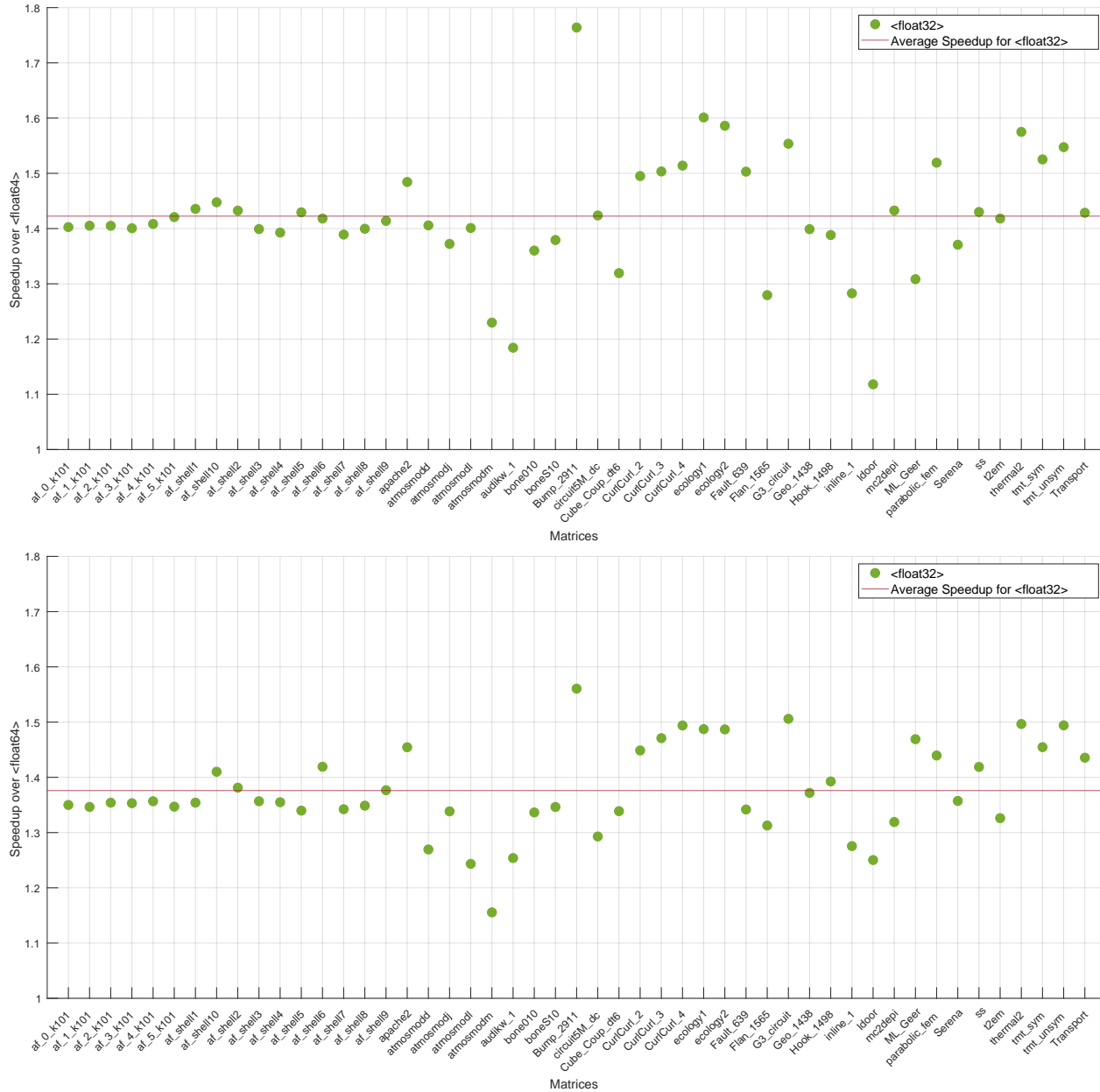


Figure 3: CB-GMRES speedup over standard GMRES on the NVIDIA V100 GPU (top) and the NVIDIA A100 GPU (bottom), respectively.

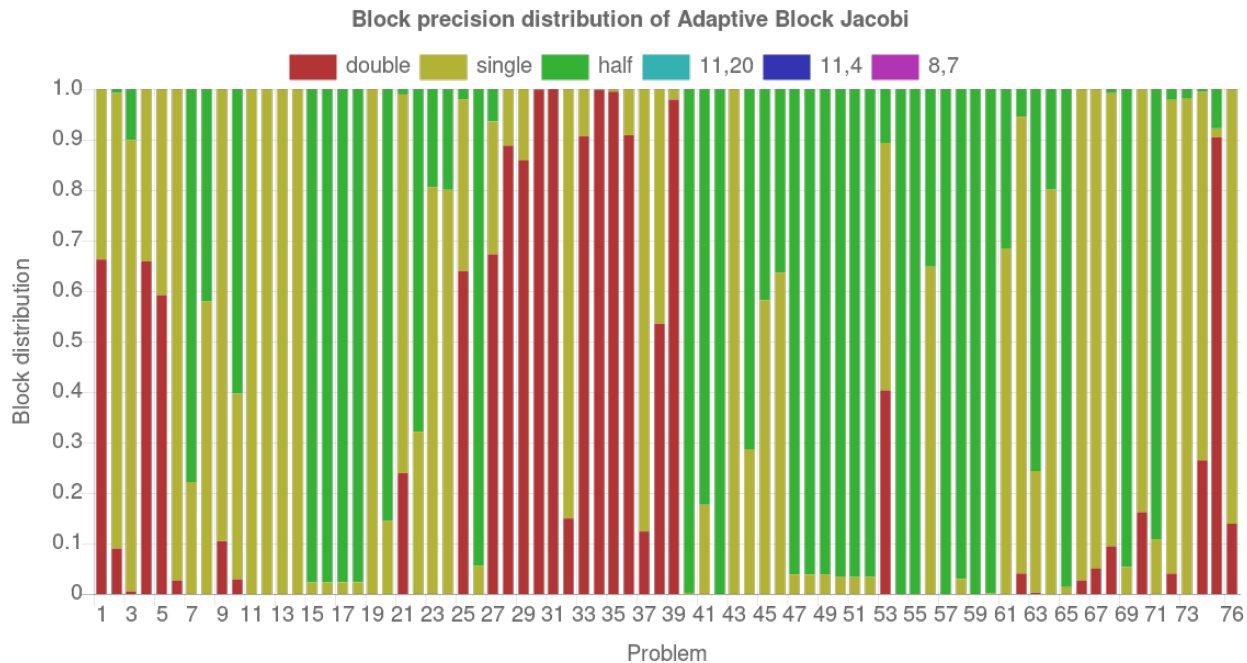


Figure 4: Distribution of the admissible memory formats for a block-Jacobi preconditioner employing a maximum block size of 24 and preserving 2 digits of preconditioner accuracy. The notation for the non-standard formats specify the length (in bits) of the exponent and the significand.

for a much more aggressive reduction of the memory access volume, therewith enabling larger runtime reductions. Controlling the memory format on a local basis requires the algorithm to determine the optimal memory precision for each block individually by carefully increasing the length of the significand and the exponent. An analysis identifying suitable precision formats, for a block-Jacobi preconditioner with a block size upper bound of 24 that preserves 2 digits of preconditioner accuracy, reveals that, for some problems, an aggressive memory format reduction results in a large discrepancy in terms of memory formats, see Figure 4.

The GINKGO team deployed a production-ready high performance implementation of an adaptive precision block-Jacobi preconditioner in the GINKGO numerical linear algebra package [3]. It employs a memory accessor to decouple the memory format from the arithmetic format, and adjusts the memory format to the numerical requirements for each block individually, as described previously. To assess the benefits of using the adaptive precision block-Jacobi preconditioner inside an iterative solver, in Figure 5 we visualize the effect on a Conjugate Gradient (CG [5]) solver. The iteration and runtime speed-ups (or slow-downs) are both relative to a block-Jacobi CG that is algorithmically identical and employs the same kernels and block size settings, but differs in the fact that it stores the block-Jacobi preconditioner in the high precision format (IEEE 754 double precision). Hence, like in the CB-GMRES case, any speed-ups are directly related from the memory accessor reducing the volume of memory accesses. For the experiments, we choose the highly-optimized CUDA backend of the implementation. In Figure 5, we consider a block-Jacobi preserving 2 digits of preconditioner accuracy. While preserving less preconditioner accuracy allows for more aggressive precision format reduction, and therewith larger potential speedups, it may also incur significant iteration overhead for cases where a full-precision preconditioner would offer considerably higher approximation quality. For the conservative preservation of 2 digits of preconditioner accuracy, the iteration overheads are very moderate and the runtime benefits more consistent across the distinct problems, averaging to a 20% speedup over the standard block-Jacobi preconditioner on the V100 architecture, and 5% speedup on the A100 architecture. At this point, we note that the adaptive-precision preconditioner in GINKGO is heavily optimized for the V100 architecture, and that the same version was employed on the A100 GPU. Higher speedups may thus be possible by specific tuning it for the A100 architecture.

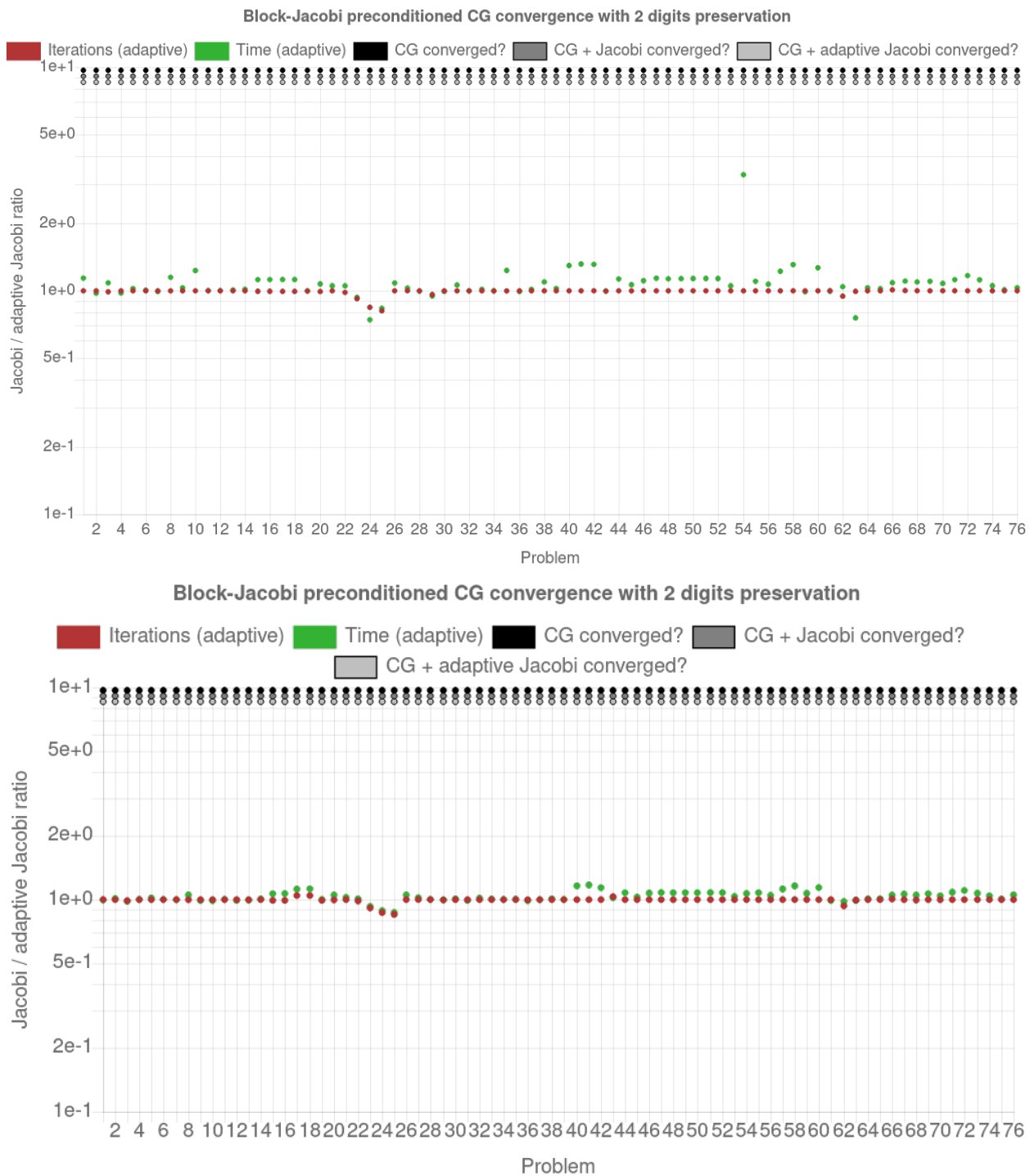


Figure 5: Runtime speedup and iteration speedup of the admissible memory formats for a block-Jacobi preconditioner employing a maximum block size of 24 and preserving 2 digits of preconditioner accuracy over a standard block-Jacobi preconditioner [4]. The performance results reflect the CUDA backend of the GINKGO library running on an NVIDIA V100 GPU (top) and the NVIDIA A100 GPU (bottom). The test matrices are taken from the Suite Sparse Matrix Collection, the problem characteristics can be found in the Appendix.

3. heFFTe

Fast Fourier transforms (FFTs) are used in applications ranging from molecular dynamics and spectrum estimation to machine learning, fast convolution and correlation, signal modulation, wireless multimedia applications, and others. The goal of the *Highly Efficient FFTs for Exascale* (**heFFTe**) library [6] is to provide sustainable high-performance multidimensional FFTs for Exascale platforms. Currently, heFFTe v2.0 achieves very good scalability on large-scale GPU-accelerated systems with a performance that is close to 90% of the roofline peak [7]. However, FFTs are memory bound, and therefore, to accelerate them, it is crucial to avoid and optimize the FFTs' communications. As many applications do not need full working precision accuracy and would benefit by a possibility for trade-off of accuracy for speed, we developed mixed-precision algorithms and high-performance implementations in the heFFTe library.

Figure 6 shows the profiles of 1024^3 FFTs running heFFTe in double precision on four Summit nodes using the CPUs (left) vs. the GPUs (right). This is a typical profile that illustrates the memory-bound nature of the FFT computation. Indeed, note that the MPI communications take 97% of the time for this particular case. Thus, any acceleration of the communications will bring proportional acceleration of the overall algorithm. We exploited several ideas to accomplish this. First, we investigated opportunities to directly accelerate MPI communications. Second, we looked into compression techniques, both lossless and lossy. Third, since GPUs are mostly idle, we investigated the use of higher accuracy in the computations than in the MPI communications. A combination of the three ideas was also developed.

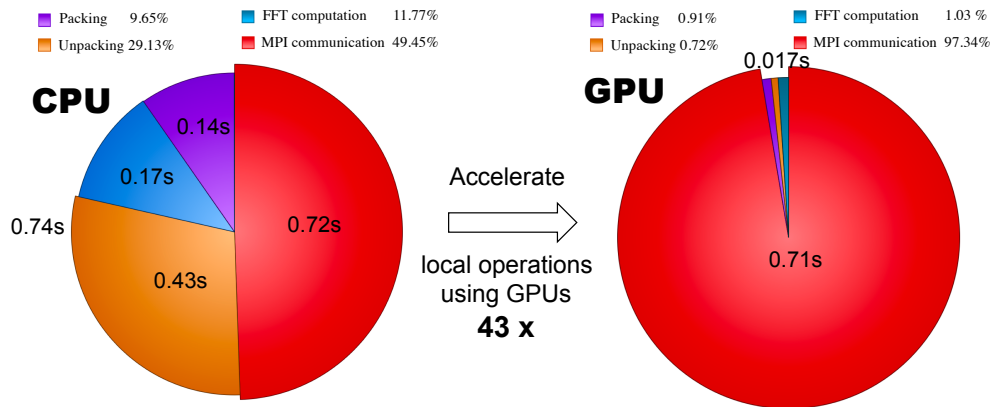


Figure 6: Profile of a 3D FFT of size 1024^3 on 4 Summit nodes – using CPUs with 128 MPI processes, 32 MPIs per node, 16 MPIs per socket (Left) vs. using GPUs with 24 MPI processes, 6 MPIs per node, 3 MPI per socket, 1 GPU per MPI (Right).

For this period, we redesigned the all-to-all routines in heFFTe by using advanced features available in MPI, such as One-Sided Communication, as well as integrating compression techniques during communication that reduce the volume of data exchange. Since the overall performance depends on the capability of compressing the data faster than the network speed, we must compress on the GPUs. Our experiments focus on a compression rate of two and four, which corresponds to a casting operation from fp64 to fp32, and fp64 to fp16. Experimental results show that pipelining the compression with communication allows us to reach almost the maximum theoretical speedup, that is 3.x out of 4 when compressing four times. Moreover, since the used compression techniques involve a loss of accuracy, we studied its impact in the heFFTe library for large scale multidimensional FFTs. We compared our approach with a full reduced precision execution, where both the data and the computation are in reduced precision. We showed that when the volume of communication is compressed twice, i.e., from double-complex to single-complex, the performance of heFFTe are similar to the single-complex representation without compression while the accuracy is one order of magnitude better.

Figure 7 illustrates the acceleration that is currently achieved by using one-sided MPI communications to speedup the Alltoallv routines used in heFFTe. Note that performance may depend on the number of processes used per node, which is a tunable parameter in our implementation. In this case the heFFTe

acceleration is up to 31%.

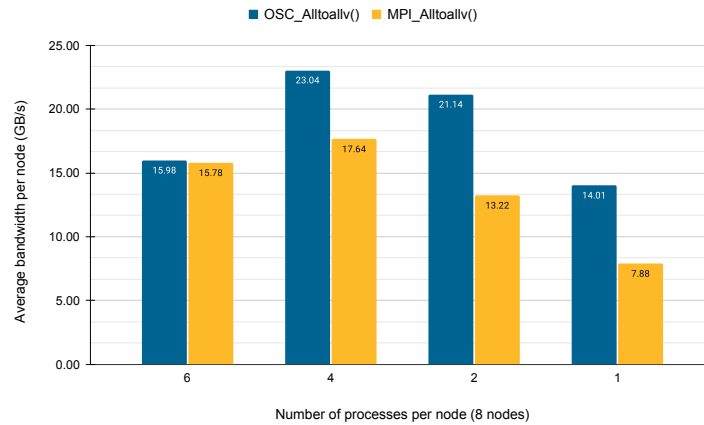


Figure 7: Average node bandwidth with different number of MPI processes per node using MPI Alltoallv vs. our one-sided Alltoallv implementation (OSC.Alltoallv). Results are using 8 nodes of Summit.

Figure 8 shows the strong scalability and Gflops rates of the mixed-precision FFTs in heFFTe for a problem of size 1024^3 on Summit. Note that single precision (fp32) FFT (denoted Ref(float)) is about 2x faster than the double precision (fp64) FFT (denoted Ref(double)). Direct use of half precision (fp16) typically fails due to overflows. To enable the use of fp16 we developed mixed-precision fp16-fp32 FFTs (CR-4). Note that CR-4 is about 4x faster than the fp64 FFT. CR-2 is a mixed-precision fp32-fp64 FFT. Note that this is twice faster than the fp64 FFT and also a little faster even than the fp32 FFT, due to the one-sided MPI acceleration.

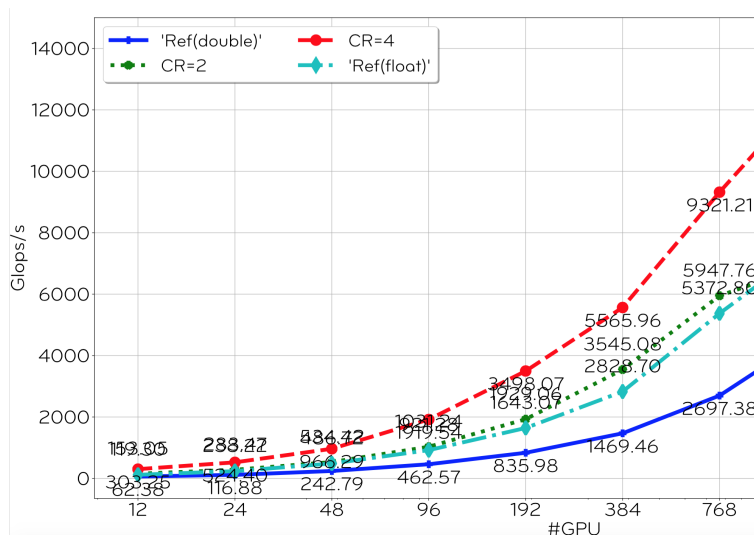


Figure 8: Strong scalability and Gflops rates of the mixed-precision FFTs in heFFTe for a problem of size 1024^3 on Summit.

The developments presented enable the use of multi-precision FFTs. The performance increase of using lower precision is proportional to the difference in storage compared to the higher precision FFTs. This is important because it enables the development of mixed-precision FFT-based solvers. Indeed, similar to sparse Krylov solvers, where mixed-precision can be applied due to a possible acceleration of their main building block – SpMV using lower precision storage, PDE-based linear solvers and eigensolvers based on

FFTs can also be accelerated.

Furthermore, we developed mixed-precision FFTs that increase the accuracy of their fixed lower-precision counterpart by doing the computations in higher precision, while retaining the speed of the low-precision FFT (see Figure 8). Table 1 shows the accuracy of the different FFTs, and quantifies the difference. For example, note that the mixed-precision fp32-fp64 FFT is one-order of magnitude more accurate than its fp32 counterpart. The algorithm runs the computations in fp64, but the MPI communications are redesigned to cast the data to fp32 and benefit from reducing the communications. The receiving end is casting the data back to fp64 before resuming computations. The mixed-precision MPI is designed to work with different compression techniques, both lossy (as in the case of casting) and lossless.

#GPU	fp64	fp32	fp32-fp64
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 evolution of the maximum error in heFFTe for FFTs of size 1024^3 when casting from fp64 to fp32 with the classical FFT using either fp64 or fp32, when the number of GPUs increases.

Related to the accuracy of the algorithms, recall that as a “rule of thumb”, if the condition number of a problem is 10^k , then you may lose up to k digits of accuracy on top of what would be lost due to round-off errors during the computation. Since FFT is an orthogonal transformation, its condition number is one, so one can expect not to lose accuracy due to the algorithm. In other words, the FFT error due to compression (forward error) is at most the compression error (backward error). Thus, certain number of correct digits in the input are expected to get translated into the same number of correct digits in the output, which we use to provide stable FFTs for very low-precision arithmetic, including fp16. Further, since FFT can be viewed as a matrix-vector product, the expectation is that for size N , the round-off error in the dot-product accumulations of the matrix-vector product will grow with N , but Higham provides computational example endorsing the “rule of thumb” that error tends to grow with \sqrt{N} . Indeed, this is observed in the experiments in Table 1, where the fixed-precision FFTs error grows with \sqrt{N} , while the mixed-precision is not affected by similar growth due to the use of higher precision arithmetic. This can be further explored in future work for the development of other mixed-precision FFTs and FFT-based applications incorporating speed-to-accuracy trade-offs.

4. hypre

Task-based mixed precision solver strategies may be realized in existing solver libraries in a number of ways: (1) directly adding the new solver capability as a new component of the library (eg. Mixed-precision Krylov or strategies based on defect correction, such as mixed-precision iterative refinement), or (2) modify the library’s build system to implicitly realize mixed-precision capabilities by mixing solvers of different precisions (eg. double precision Krylov solver with a single precision preconditioner). The former approach allows more fine-grained mixed-precision strategies to be integrated into the library. Note that this approach can also be incorporated into the latter approach for completeness. The integration of mixed-precision solver strategies within hypre follows the latter approach, with the overarching goal of minimizing the impact on current user experience with hypre.

```

...
HYPRE_Solver solver , precondition;
HYPRE_IJMatrix A, B;
HYPRE_IJVector b, x;
/* solver precision */
HYPRE_SolverPrecision solver_precision;
HYPRE_SolverPrecision precon_precision;
...
/* Create the matrix. Note the use of precision here. */
solver_precision = HYPRE_REAL_DOUBLE;
precon_precision = HYPRE_REAL_SINGLE;
HYPRE_IJMatrixCreate_MP(MPLCOMMLWORLD, ilower, iupper, ilower, iupper,
                        &A, solver_precision);
HYPRE_IJMatrixCreate_MP(MPLCOMMLWORLD, ilower, iupper, ilower, iupper,
                        &B, precon_precision);
...
/* Create the rhs and solution. Here, we only account for the solver
   precision. Since the preconditioner solve is done internally, we can pass the
   appropriate vector types there. */
HYPRE_IJVectorCreate(MPLCOMMLWORLD, ilower, iupper, &b, solver_precision);
HYPRE_IJVectorCreate(MPLCOMMLWORLD, ilower, iupper, &x, solver_precision);
...
/* Create solver */
HYPRE_ParCSRPCGCreate(MPLCOMMLWORLD, &solver, solver_precision);
...
/* Create AMG preconditioner */
HYPRE_BoomerAMGCreate(&precond, precon_precision);
...

```

Figure 9: Code snippets from a driver that demonstrates how to incorporate mixed precision into hypre from the user’s point of view.

4.1 APPROACH:

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, but setting the option `--enable-<precision-type>`. The initial goal for the mixed-precision integration is to extend the build system to allow a unity build of all three precision supported by hypre. A user would then obtain this build by setting the build option `--enable-mixed-precision`. In addition to changes to the build system, we also need to modify existing code to support the multiple precision build. We use a macro to transform existing function names to change the resulting symbols of the object files for the built libraries in the respective precisions. This is necessary to avoid any name collisions in the object files. 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. In addition, corresponding header files have to be modified to reflect these changes. Finally, new wrapper functions are developed to call the corresponding function in the appropriate precision. That is, function `hypre_foo(hypre_object obj)` would now serve as a wrapper function that would call the appropriate `hypre_foo_<precision_type>(hypre_object obj)`, given the `precision.type` of the associated hypre object. This is achieved via a switch statement over the different precision types. Additional functionality are also necessary to allow users to pass precision information to the hypre objects they create and wish to use. Making these changes to an existing library can be a tedious task and it is convenient to utilize scripts whenever possible to modify existing code and generate new code.

It is clear from the above approach that at the end of the build, one would obtain a library that is 3 times the size of the current hypre build. However, one could imagine that not all code units need to be

transformed for a build in multiple precision, since some functionalities may be completely independent of the precision. Thus, one could consider a more refined approach that identifies only appropriate code units that need to be built in multiple precisions to support the rest of the library. That is, a subset of all the code units that would achieve the required mixed-precision result. To do this, we would need to first identify code units that utilize floating point input and arithmetic as candidates for transformation to support multiple precisions. Leveraging the modular design of hypre’s data structures and components, we could obtain a hierarchy of dependencies for each component (eg. the different solvers in hypre) and identify code units necessary to support the mixed precision build of the component. Putting these together, we can obtain a subset of the entire hypre code base that needs to be transformed to support a mixed precision build for hypre. While this refined approach could lead to a smaller build of hypre with mixed-precision capabilities, it would require some additional restructuring of existing code in order to separate transformable code units from those that need not be transformed. The hypre team is evaluating both of the above approaches to identify which option is most appropriate from the point of view of code maintenance and portability.

As previously noted, this effort is being conducted with significant consideration for our users and current developers. In the current design, the user interface will remain largely unchanged. New functions for creating hypre objects would need to be introduced for users to prescribe the appropriate precision types for the objects they create. These functions will complement existing “create” functions leaving the original function intact (eg. `HYPRE_OBJCreate(hypre_object obj)` and a corresponding `HYPRE_OBJCreate(hypre_object obj, HYPRE_PRECISION precision_type)`), see Figure 9 for an example, how mixed precision functions would be called by a user. From the developer’s point of view, new changes to the code structure of hypre as a result of these new capabilities would be documented, and subsequent code development will follow the new structure.

4.2 CURRENT STATUS AND FUTURE PLANS:

To get a good understanding of how users may utilize this new capability in hypre and how the code modifications would impact existing users, we have designed a driver that demonstrates how the solvers in hypre may be accessed and used with the mixed-precision build. This driver helped advice how users could communicate precision information through the hypre interface to the appropriate objects when they are created. It also helped confirm that subsequent references to precision would be done outside the user’s view, leaving the remaining user interface intact.

Using the scripts developed to transform code units to support building hypre in multiple precision, we have successfully demonstrated building the Krylov solvers in hypre in multiple precisions. Thus, one can create and access single, double, and longdouble variants of the same or different Krylov solver, and use them interoperably. The next steps include applying this strategy to hypre’s matrix and vector data structures to allow users to assemble linear systems of different precision types for use in solvers and preconditioners of the corresponding precision type. This will enable a comprehensive test of the mixed-precision build (eg. double precision Krylov solver with and single precision preconditioner) and help identify any outstanding issues. In addition, new task-based mixed-precision solver strategies would be developed as new solver components within hypre. In particular, we will consider a mixed-precision multigrid solver and a solver based on defect correction.

5. MAGMA

MAGMA is a numerical linear algebra library targeting heterogeneous architectures. We already have a mixed precision linear solver which can utilize the high FP16 performance from the tensor core from NVIDIA Volta architecture[8]. Now we are implementing a **mixed precision iterative refinement algorithm for symmetric and Hermitian eigenvalue problems**. It is based on the SICE algorithm from Dongarra et al.[9]. The original SICE algorithm is designed for unsymmetric cases and solve a shifted and rank one updated system $(A - \lambda I + uv^T)c = \lambda x - Ax$ at each iteration. The right hand side is the residual vector and the rank one updated is constructed in a way that the solution vector c will contain the correction to both the eigenvalue λ and corresponding eigenvector x . In the original paper the system is solved with two series of Givens rotations on top of Schur decomposition, which is hard to be parallelized to fit modern architectures. Our proposed algorithm targets symmetric and Hermitian systems and performs the tridiagonalization in low

precision which has $O(n^3)$ cost. The tridiagonal eigensolver is still in high precision in order to capture the eigenvalues which gap is smaller than ϵ_{low} but not duplicated. Then in the iterative refinement step, the shifted rank one updated tridiagonal system is solved with Sherman-Morrison formula:

$$(A + uv^T)^{-1} = A^{-1} - \frac{A^{-1}uv^T A^{-1}}{1 + v^T A^{-1}u}$$

Furthermore, we are blocking the algorithm so it could refine multiple eigenpairs simultaneously instead of only one at a time. In the implementation, we are optimizing the performance by rearranging the tasks to reduce idle time on both CPU and GPU. Comparing fixed precision eigensolvers which will only apply the back transformation to the requested eigenvectors, we need the transformation matrix Q from the tridiagonalization $A = QTQ^T$ to be explicitly formed for later refinement. But we could start it earlier here as soon as the tridiagonalization is completed because it does require the eigenvectors to be ready.

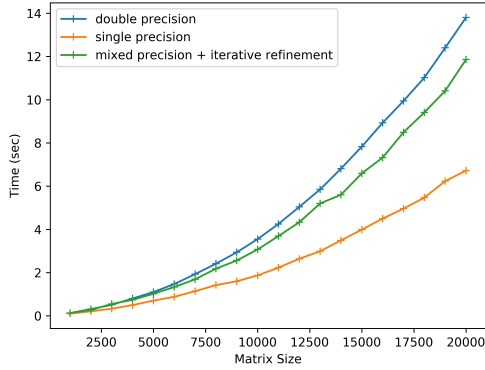
For the performance results, we are using the same Intel Xeon E5-2650 v3 CPU but paired with two different GPUs: NVIDIA Volta V100 and NVIDIA Pascal GTX1060. The Volta V100 is a workstation GPU with peak performance 14 Tflop/s in single precision and 7 Tflop/s in double precision (1:2). However, the GTX1060 is a gaming GPU without native double precision hardware unit support. It's peak performance in single precision is 4.375 Tflop/s but only 136.7 Gflop/s in double precision (1:32). In figure 10 we are showing the performance comparison between fixed precision and mixed precision algorithms for real symmetric and complex Hermitian matrices. Both algorithms are using the latest two stages tridiagonalization[10] which first reduce the matrix from symmetric to band then band to tridiagonal. The mixed precision algorithm is performing the reduction in single precision and at the end refining the solution eigenvalues and eigenvector toward double precision accuracy. For all the subfigures, the x-axis is the matrix size and the y-axis is time to solution, the lower the better. And the largest 32 eigenvalues are requested. On V100, there is 1.16x and 1.45x speedup do use mixed precision comparing to double precision. The single precision is shown as reference and its accuracy is much lower than the other two. The complex problem is showing higher speedup because of the higher arithmetic intensity. For the GTX1060, the speedup becomes 2x 3.6x for real and complex. The double precision performance on GTX1060 is crippled and the mixed precision algorithm can benefit from moving most of the heavy operations to single precision. Overall, there are noticeable performance improvements using mixed precision algorithm while only a small part (256) of the eigenvalues and eigenvectors are requested.

The results we have shown are well-condition cases but the clustering of eigenvalues will slow down the convergence as the conditioning of eigenvectors is related to the gap between their corresponding eigenvalues. It will require some shifting and scaling for each of the clusters and it is part of our future work. The paper with more details has been submitted to IEEE International Parallel & Distributed Processing Symposium (IPDPS) 2021 and under review. The implementation will be in the eigref branch in MAGMA repository and be merged into default in later release after the documentations are prepared.

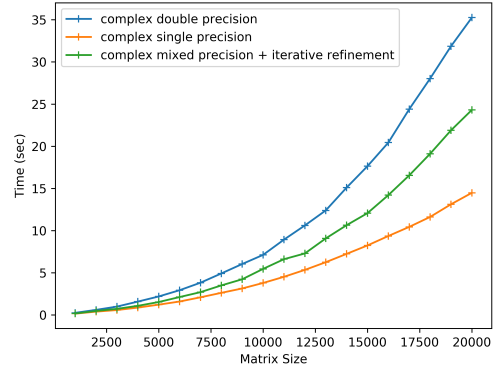
6. PETSc/TAO

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, as seen in Figure ???. 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 [11, 12, 13, 14, 15, 16], on upcoming exascale systems. In all cases, users will be able to rely on PETSc's large assortment of composable, hierarchical, and nested solvers [17], as well as advanced time-stepping and adjoint capabilities and numerical optimization methods running on the GPU.

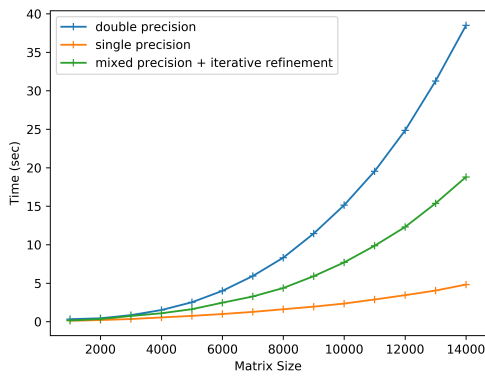
An application for solving time-dependent partial differential equations, for example, may compute the Jacobian using Kokkos and then call PETSc's time-stepping routines and algebraic solvers that use CUDA, cuBLAS, and cuSPARSE; see Figure 11. Applications will be able to mix and match programming models, allowing, for example, some application code in Kokkos and some in CUDA. The flexible PETSc back-end support is accomplished by **sharing data** between the application and PETSc programming models but not



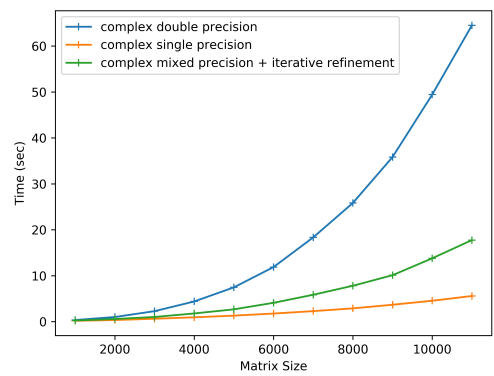
(a) Real symmetric matrices performance on NVIDIA Volta V100.



(b) Complex Hermitian matrices performance on NVIDIA Volta V100.



(c) Real symmetric matrices performance on NVIDIA Pascal GTX1060.



(d) Complex Hermitian matrices performance on NVIDIA Pascal GTX1060.

Figure 10: MAGMA performance comparison between fixed precision and mixed precision algorithms for real symmetric and complex Hermitian matrices on NVIDIA V100 and GTX1060 GPUs.

sharing the programming models' internal data structures. Because the data is shared, there are no copies between the programming models and no loss of efficiency.

Performance portability in PETSc

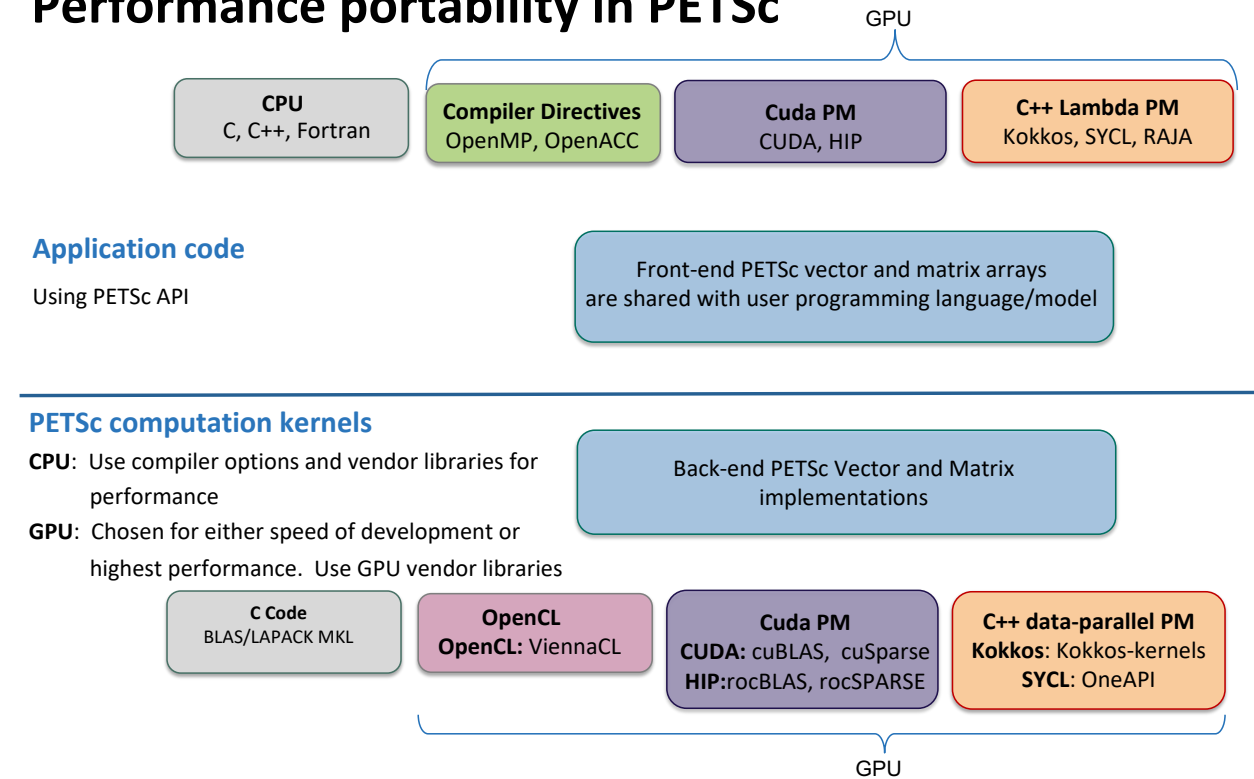


Figure 11: PETSc application developers will be able to use a variety of programming models for GPUs independently of PETSc's internal programming model.

PETSc develops its own backends in addition to using portability middle-ware such as Kokkos. Such a design provides rapid development of new solvers while still enabling a low-dependency, highly-performant library for our users. This approach has also enabled us to better understand the fundamental challenges in developing for the variety of computing hardware as the ancillary challenges, such as code complexity, are being developed. A more detailed discussion of our approach may be found in Ref. [18].

The approach to mixed precision in PETSc focuses on developing an abstraction layer to vendor algebra libraries (cuBLAS/cuSPARSE, rocBLAS/rocSPARSE, ...), which will enable PETSc solver methods to become mixed-precision by changing precision between the CPU and GPU. This is a complement to work on incorporating backends in PETSc discussed above. 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 maintaing an easy upgrading path for our large user base.

7. SLATE

SLATE has an initial implementation of mixed-precision iterative refinement, for both LU (*gesvMixed*) and Cholesky (*posvMixed*), sketched in algorithm 1. The idea behind the algorithm is to do the $O(n^3)$ work of factoring the matrix and the $O(n^2k)$ work of triangular solves in lower precision (e.g., single), and then use iterative refinement to improve the result to high precision (e.g., double) accuracy, for a linear system of dimension n with k right-hand sides (RHS). Only the $O(n^2k)$ work is done in high precision, to compute residuals and update the solution vector. The code is templated on both the low and high precisions, and works for both real and complex types.

Algorithm 1 Mixed precision Cholesky factorization in low precision (lo) and high precision (hi). The mixed precision LU factorization is similar, just replacing the LL^H factorization with $P^H LU$ factorization.

$A_{lo} = A_{hi}$	▷ cast from high to low precision
$A_{lo} = L_{lo}L_{lo}^H$	▷ factor, in low precision
Solve $L_{lo}L_{lo}^H x_{lo} = b_{lo}$	▷ initial solve, in low precision
$x_{hi} = x_{lo}$	▷ cast from low to high precision
while not converged do	
$r_{hi} = A_{hi}x_{hi}$	▷ compute residual, in high precision
$r_{lo} = r_{hi}$	▷ cast from high to low precision
Solve $L_{lo}L_{lo}^H d_{lo} = r_{lo}$	▷ compute correction, in low precision
$d_{hi} = d_{lo}$	▷ cast from low to high precision
$x_{hi} = x_{hi} - d_{hi}$	▷ apply correction, in high precision
end while	

This implements Newton’s Method to solve $f(x) = Ax - b = 0$, with the iteration

$$x^{(k+1)} = x^{(k)} - A^{-1}r = x^{(k)} - A^{-1}(Ax - b),$$

where $A^{-1}r$ is done by solving $Ad = r$ using the LU or Cholesky factorization in low precision (single). Thus, given a sufficiently close initial solve, convergence is quadratic. Only the highlighted portions are done in high precision (double).

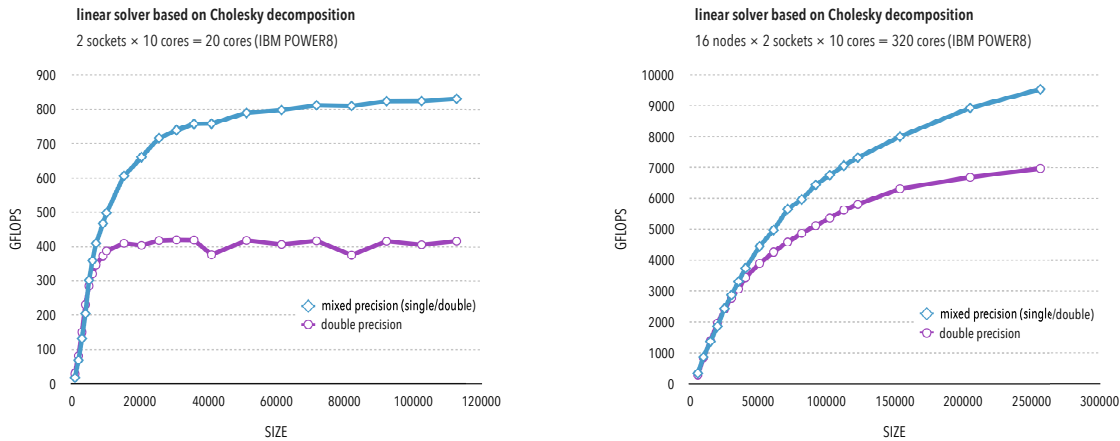


Figure 12: Mixed precision Cholesky results on 1 node (left) and 16 nodes (right).

On a single node, mixed-precision Cholesky exhibits the expected 2× performance improvement over double-precision Cholesky, shown in fig. 12. However, on 16 nodes, it achieves a lower 1.35× speedup. We are currently investigating this less-than-expected performance, which appears to come from SLATE’s parallel BLAS routines not being optimized for this case. The BLAS routines (`trsm` triangular solves, `gemm` general matrix multiply, and `hemm` Hermitian matrix multiply) are optimized for the case where the output matrix is large (large n and k). However, in these solvers, there are only a small number of RHS, so k is small. If there are a large number of RHS, the $O(n^2k)$ iterative refinement time becomes large, making mixed-precision refinement unattractive.

8. SuperLU

SuperLU is a general purpose library for the direct solution of large, sparse, nonsymmetric systems of linear equations. The library is written in C and is callable from either C or Fortran program. It uses MPI, OpenMP and CUDA to support various forms of parallelism.

For the mixed precision work, we currently focus on the distributed memory library SuperLU-DIST. In the past, this library supports only two data types: double-precision real and double-precision complex. In both versions, only working precision iterative refinement subroutines are provided for improved backward stability, even though the two versions can be compiled and used together in a single executable. In the last few months, we did the following to transition the code to use mixed precision arithmetics: 1) We use a macro code generator to generate the single precision code, both LU factorization and triangular solve; 2) For the single precision code, we implement a mixed precision iterative refinement routine. At present, only the residual computation uses double precision. In the future, we will add double precision variables for the updated solution vector as well.

The direct solution process is dominated by the sparse LU factorization, which in turn is dominated by the Schur complement update (SCU) in each block elimination step. Fig 13 depicts the algorithm corresponding to the local portion of SCU on each MPI process. Unlike the dense LU factorization, where dense matmul is the sole operation in SCU, the sparse SCU requires gather and scatter operations before and after the matmul operation. Gather operation is relatively cheap, which is mainly memory copy. But scatter operation is more costly, because it involves memory access with indirect addressing.

- 1: Loop through N steps of panel factorization:
- 2: **for** $k = 1, \dots, N$ **do**
- 3: Gather sparse $A(:,k)$ and $A(k,:)$ into dense work[] arrays
- 4: Call dense matmul (GEMM) on work[] arrays
- 5: Scatter work[] arrays into trailing sparse data structures
- 6: **end for**

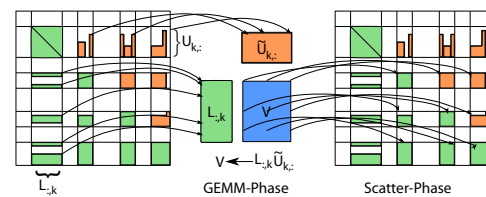


Figure 13: Local Schur complement update in SuperLU on each MPI process.

In the current CUDA implementation, we can offload GEMM operations to GPU and call GEMM in the cuBLAS library. The panel factorization and gather/scatter operations remain on CPU.

We ran both single precision and double precision codes on multiple nodes of Summit, using 10 nodes and all 6 GPUs per node. Five matrices of medium to large size are selected from SuiteSparse for the experiments; they are: audikw_1.rb, Ga19As19H42.rb, Geo_1438.rb, Serena.rb, and nlpkkt80.rb. Fig. 14 shows the results of using 60 CPU cores (60 MPI tasks) together with 60 GPUs, on 10 nodes. The left plot shows runtime breakdown in the double precision LU. The Panel factorization and the Gather operation are relatively fast. The Scatter operation can take nontrivial time, up to 50% total time for Ga19As19H42. GEMM time does not dominate, at most 33% total time for Geo_1438. In fact, we tried to use Tensor Cores version of cublasSgemm routine, and observe only a couple of percents performance benefit, but the solution accuracy has suffered. The “Other” part is mostly due to MPI communication and other data movements. The right plot compares the runtime between single and double precision LU. The total time improvements are 28-53%. When examining each phase, we observe the largest speedup in GEMM, up to 83%. The second largest speedup is in Panel factorization, up to 54%. The Scatter operations show at most 20% speedup.

We will pursue several tasks in the near future: 1) experiment with larger matrices and larger node counts; 2) implement a more accurate version of iterative refinement, and a mixed precision GMRES to improve accuracy; 3) develop a new CUDA kernels that perform gather/scatter on GPU.

9. Trilinos and KokkosKernels

Trilinos [19] is a large toolkit for scientific computing, consisting of 60+ packages. The most important package for parallel matrix and vector objects is Tpetra. Tpetra relies on KokkosKernels for parallel linear algebra on the node. The Kokkos library [11] provides an abstraction layer for performance portable programming models for multiple platforms, including all commonly used CPU and GPU architectures. The main package for iterative solvers is Belos, while multiple packages provide preconditioners, such as Ifpack2, MueLu, and ShyLu.

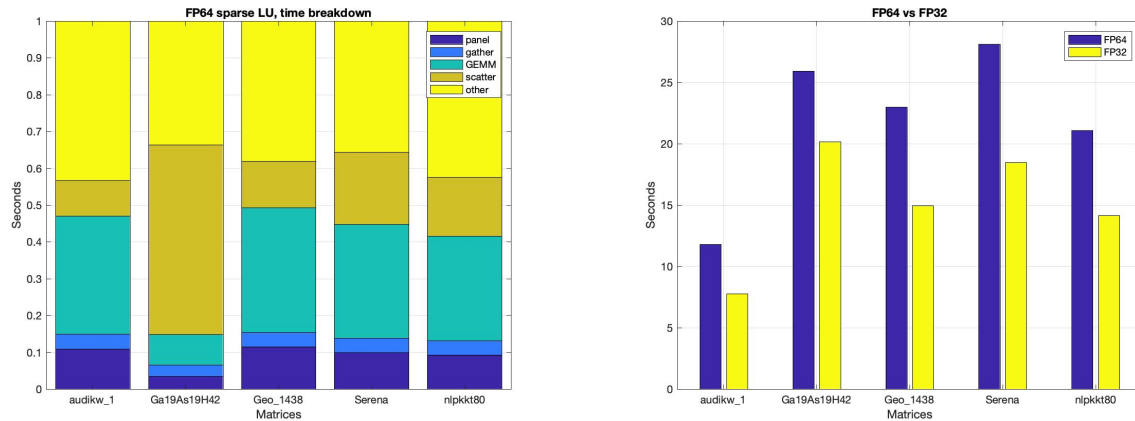


Figure 14: On Summit with 10 nodes, each using 6 MPIs and 6 GPUs. Runtime breakdown in the double precision LU (Left). Comparison of single precision vs. double precision LU (Right).

9.1 SOFTWARE DESIGN

All the packages are templated on a *scalar type*. Currently, only double and float are supported and tested in Trilinos. Kokkos and KokkosKernels are in the process of adding support for half precision. To allow mixed precision solvers, one could potentially add another template type (so two scalar types). However, this would increase the complexity of the code and make things more difficult for users, so we have decided against that approach. Instead, the templated scalar type shall correspond to the precision of the input/output. Algorithms (solvers, preconditioners) may use another scalar type internally, but this is not exposed to the users. This makes the user interface cleaner.

Belos does not contain its own linear algebra implementation but instead relies on abstracted linear algebra interfaces through the Belos MultiVector Traits. We created a Kokkos-based adapter for Belos so that all Krylov basis vectors are stored in Kokkos Views and operated on via the MultiVector interface. Belos' templates assume that all operations are carried out in the same scalar type; there are no built-in capabilities to mix and match precisions within a solver. We can perform operations outside of a solver using a different precision (as we do with computing residuals in our GMRES-IR implementation), but any kernels which use multiple precisions internally must be managed entirely in the linear algebra adaptor.

9.2 ITERATIVE REFINEMENT EXPERIMENTS WITH GMRES

We focus on iterative solvers (Krylov methods) as they are the most commonly used linear solvers within DOE/ECP. The most popular solvers are CG (for SPD systems) and GMRES (for nonsymmetric systems). We focus on GMRES as it is the most general method. GMRES with Iterative Refinement (GMRES-IR) [20, 21] is a well-known algorithm for using a Krylov solver with multiple precisions. However, it has not been well-studied on modern high-performance architectures, and is not standard in linear solver software implementations.

Algorithm 2 Iterative Refinement with GMRES Error Correction

- 1: $r_0 = b - Ax_0$ [double]
 - 2: **for** $i = 1, 2, \dots$ until convergence: **do**
 - 3: Use GMRES(m) to solve $Au_i = r_i$ 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**
-

We use GMRES(50) and perform iterative refinement at each restart. That is, we perform 50 iterations

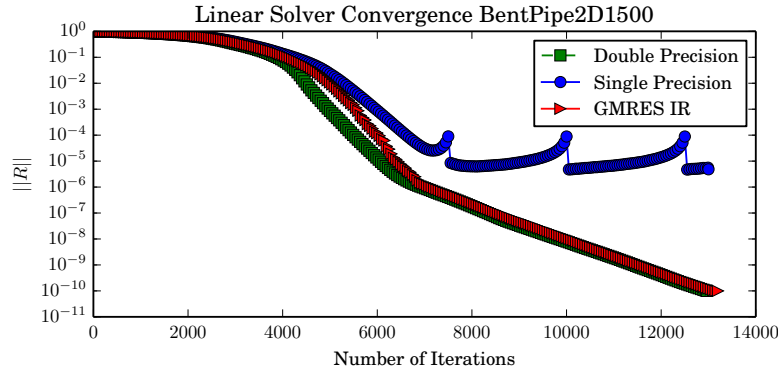


Figure 15: Relative residual norm convergence for the matrix BentPipe2D1500. Single precision GMRES(50) is represented by blue circles, double precision by green squares, and mixed precision GMRES(50)-IR by red triangles.

of GMRES in float precision, compute the new residuals in double precision, and then restart the GMRES iteration using the truncated residual in float precision. (Algorithm 2.) For all problems, we use a right-hand side vector of all ones. Each GMRES iteration uses 2 passes of classical Gram-Schmidt (CGS2) orthogonalization. This allows orthogonalization operations to be grouped into one GPU kernel launch with four calls to GEMV. Solvers are run to a relative residual tolerance of 10^{-10} . Each matrix is stored in Compressed Sparse Row (CSR) format. Experiments were run on a Power 9 CPU using a single V100 GPU. All operations on the CPU are run in serial.

9.3 KERNEL SPEEDUP FOR GMRES-IR

We use the matrix BentPipe2D1500. This is the 2D Bent Pipe convection-diffusion problem from the Galeri package with $n_x = 1500$. So we have $n = 2,250,000$ and $nnz = 11,244,000$. This matrix is strongly convection-dominated, so GMRES without preconditioning needs many iterations to converge to a tolerance of $1e-10$. Convergence plots are in Figure 15. The single precision only solver reaches a maximum convergence of $4.7e-6$, and the double precision solver needs 12,967 iterations to converge. GMRES-IR needs 263 cycles of 50 iterations, so 13,150 iterations, and its convergence curve closely follows that of the double precision solver. This phenomenon has been observed by both [22] and [23].

Figure 16 shows the solve times of the GMRES double and IR solvers. Table 2 gives specific timings and speedups of different kernels. Using GMRES-IR gives about 32% speedup over GMRES double. The solve times do not include time required to copy the matrix A from double precision to single precision. The two GEMV kernels give 28 to 57% speedup, but the SpMV gives a spectacular 148% speedup. The bar segment labeled “other” indicates time solving the least squares problems and performing non-GPU operations. For GMRES-IR, it also includes the computation of the new residual in double precision.

Table 2: Speedup of different kernels for the matrix BentPipe2D1500.

	Double Belos	IR Belos	Speedup
Gemv (Trans)	20.20	15.78	1.28
Norm	1.72	1.49	1.15
Gemv (no Trans)	19.01	12.10	1.57
Total Orthogonalization	41.85	30.30	1.38
A*x	7.33	2.95	2.48
Total time	50.26	38.03	1.32

In Figure 17, we graph kernel speedups for the previous problem and two additional matrices: a 3D Laplacian, and the matrix UniFlow2D2500. The latter is another convection-diffusion problem from the Galeri package with $n = 6.25$ million. It is interesting to note that the kernel speedups are relatively

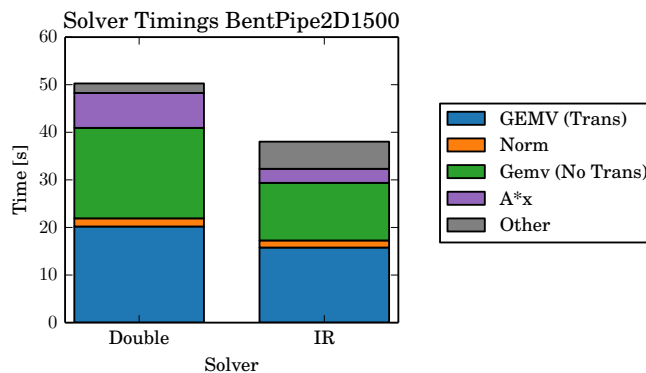


Figure 16: Solve times for GMRES(50) double (left) and IR (right) for the matrix BentPipe2D1500. Each bar represents total solve time, split up to give a breakdown of time spent in different kernels. The gap between the two bars represents timing for small dense (non-GPU) operations and, for GMRES-IR, computing residuals in double precision.

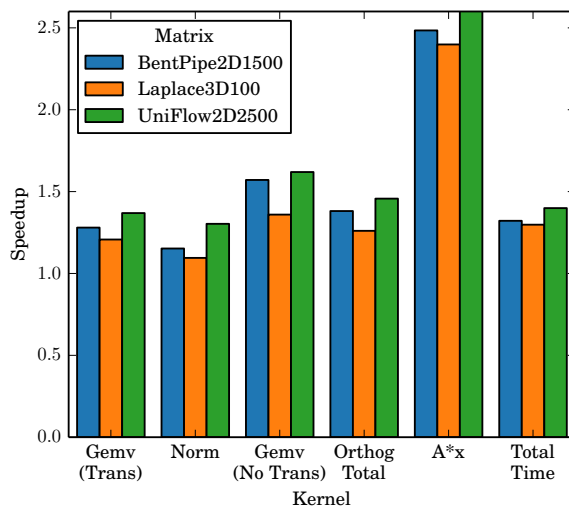


Figure 17: Speedups for different kernels going from GMRES double to GMRES-IR over three different PDEs.

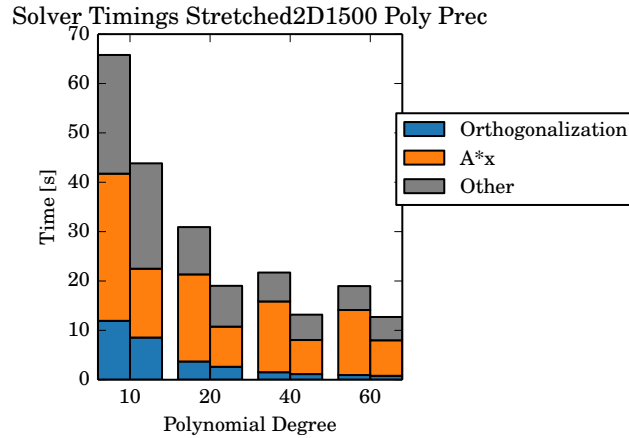


Figure 18: Solve times for polynomial preconditioned GMRES using polynomial degrees 10, 20, 40, and 60. For each polynomial degree, the bar on the left shows solve time for double precision GMRES, and the bar on the right gives timings for GMRES-IR.

consistent across the three problems. In particular, the SpMV kernel improves by 2.4 to 2.6 times. This requires further investigation, but it may be due to better caching with the low-precision matrix. The total solve times to convergence improve by 30 to 40%.

9.4 POLYNOMIAL PRECONDITIONING

We use a polynomial preconditioner based upon the GMRES polynomial (see details in [24]). For the GMRES-IR solver, the polynomial is both computed and applied entirely in single precision. The matrix is a 2D Laplacian over a stretched grid from the Galeri package, with 1500 grid points in each direction. Thus, we have $n = 2.25$ million as with the BentPipe2D1500 matrix, but here the bandwidth of the matrix is larger. Due to the difficulty of this problem, GMRES(50) cannot converge without preconditioning.

We apply polynomial preconditioners of degrees 10, 20, 40, and 60. With degree 10 polynomial preconditioning, GMRES (double) needs 3735 iterations to converge, and GMRES-IR needs 3750 iterations. By degree 60 this is reduced to 297 iterations (double) and 350 iterations (IR). Figure 18 shows the solve times for both double precision GMRES (left bars) and GMRES-IR (right bars). The “other” portion of each bar indicates time spent in small dense matrix operations, vector additions for the polynomial, and recomputing the residuals in double precision for GMRES-IR. For all polynomial degrees, GMRES-IR gives about 33% speedup in total solve time. Furthermore, unlike in the previous examples where solve time was dominated by orthogonalization, with polynomial preconditioning the cost begins to shift toward the sparse matrix-vector product. With high polynomial degrees of 40 and 60, the SpMV time dominates.

9.5 FUTURE WORK

In the future we plan to study other preconditioners such as block Jacobi. We also want to study the implications of choice of restart size on the convergence of GMRES-IR and compare to other recent research in this area. Additional work will aim to pinpoint an explanation for the greater than 2x speedup for the SpMV kernel when going from double to single precision. Kokkos and KokkosKernels are implementing support for half precision computations. We will incorporate these into the preconditioners and/or the Kokkos-Belos adapter to test half precision within linear solvers. Future performance tests will study other Krylov solvers such as CG and BiCGStab.

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.

References

- [1] Jose I. Aliaga, Hartwig Anzt, Thomas Grützmacher, Enrique S. Quintana-Ortí, and Andrés E. Tomás. Compressed basis GMRES on high performance GPUs. *SIAM J. on Scientific Computing*, 2020. Under review.
- [2] Hartwig Anzt, Terry Cojean, Goran Flegar, Pratik Nayak, and Enrique S. Quintana-Ortí. Ginkgo — a numerical linear operator library. <https://github.com/ginkgo-project/ginkgo/wiki>.
- [3] Hartwig Anzt, Terry Cojean, Goran Flegar, Fritz Goebel, Thomas Gruetzmacher, Pratik Nayak, Tobias Ribizel, Yu-Hsiang Tsai, and Enrique S Quintana-Orti. Ginkgo: A modern linear operator algebra framework for high performance computing. *arXiv preprint arXiv:2006.16852*, 2020.
- [4] Goran Flegar, Hartwig Anzt, Terry Cojean, and Enrique S Quintana-Ortí. Customized-precision block-jacobi preconditioning for krylov iterative solvers on data-parallel manycore processors. *ACM TOMS*, submitted.
- [5] Y. Saad. *Iterative Methods for Sparse Linear Systems*. SIAM, 2nd edition, 2003.
- [6] Stanimire Tomov, Alan Ayala, Azzam Haidar, and Jack Dongarra. Fft-ecp api and high-performance library prototype for 2-d and 3-d ffts on large-scale heterogeneous systems with gpus. ECP WBS 2.3.3.13 Milestone Report FFT-ECP STML13-27, 2020-01 2020. revision 01-2020.
- [7] Alan Ayala, Stanimire Tomov, Azzam Haidar, and Jack Dongarra. hefft: Highly efficient fft for exascale. In *International Conference on Computational Science (ICCS 2020)*, Amsterdam, Netherlands, 2020-06 2020.
- [8] 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 *SC18: International Conference for High Performance Computing, Networking, Storage and Analysis*, pages 603–613. IEEE, 2018.
- [9] 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.
- [10] Azzam Haidar, Hatem Ltaief, and Jack Dongarra. Parallel reduction to condensed forms for symmetric eigenvalue problems using aggregated fine-grained and memory-aware kernels. In *Proceedings of 2011 International Conference for High Performance Computing, Networking, Storage and Analysis*, pages 1–11, 2011.
- [11] 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.
- [12] 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.
- [13] Khronos SYCL Working Group. SYCL specification: Generic heterogeneous computing for modern C++, 2020. <https://www.khronos.org/-registry/SYCL/specs/sycl-2020-provisional.pdf>.
- [14] NVIDIA Corporation. *NVIDIA CUDA Toolkit*, 9.0 edition, 2018.
- [15] AMD. HIP programming guide, 2020. https://rocmdocs.amd.com/en/latest/Programming_Guides/HIP-GUIDE.html.
- [16] 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.

- [17] Jed Brown, Matthew G. Knepley, David A. May, Lois C. McInnes, and Barry F. Smith. Composable linear solvers for multiphysics. In *Proceedings of the 11th International Symposium on Parallel and Distributed Computing (ISPDC 2012)*, pages 55–62. IEEE Computer Society, 2012.
- [18] 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, 2020.
- [19] 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.
- [20] 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 J. Scientific Computing*, 39(6):A2834–A2856, 2017.
- [21] Erin Carson and Nicholas J. Higham. Accelerating the solution of linear systems by iterative refinement in three precisions. *SIAM J. Scientific Computing*, 40(2):A817–A847, 2018.
- [22] Neil Lindquist, Piotr Luszczek, and Jack Dongarra. Improving the performance of the GMRES method using mixed-precision techniques. In *Smoky Mountains Conference Proceedings*, 2020.
- [23] S. Gratton, E. Simon, D. Titley-Péloquin, and P. Toint. Exploiting variable precision in gmres. *ArXiv*, abs/1907.10550, 2019.
- [24] 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.