A Survey of Numerical Methods Utilizing Mixed Precision Arithmetic

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

Ahmad Abdelfattah[1], Hartwig Anzt[1,2], Erik G. Boman[3], Erin Carson[4], Terry Cojean[2], Jack Dongarra[1,5,6], Mark Gates[1], Thomas Grützmacher[2], Nicholas J. Higham[6], Sherry Li[8], Neil Lindquist[1], Yang Liu[8], Jennifer Loe[3], Piotr Luszczek[1], Pratik Nayak[2], Sri Pranesh[6], Siva Rajamanickam[3], Tobias Ribizel[2], Barry Smith[9], Kasia Swirydowicz[10], Stephen Thomas[10], Stanimire Tomov[1], Yaohung M. Tsai[1], Ichi Yamazaki[3], Urike Meier Yang[7]

[1]University of Tennessee, Knoxville, USA
[2]Karlsruhe Institute of Technology, Karlsruhe, Germany
[3]Sandia National Lab, Albuquerque, USA
[4]Charles University, Prague, Czech Republic
[5]Oak Ridge National Lab, Oak Ridge, USA
[6]University of Manchester, Manchester, UK
[7]Lawrence Livermore National Lab, USA
[8]Lawrence Berkeley National Lab, Berkeley, USA
[9]Argonne National Lab, Argonne, USA
[10] National Renewable Energy Lab, Boulder, USA

Technical report from the Exascale Computing Project.

1 Introduction

Within the past years, hardware vendors have started designing low precision special function units in response to the demand of the Machine Learning community and their demand for high compute power in low precision formats. Also the server-line products are increasingly featuring low-precision special function units, such as the NVIDIA tensor cores in ORNL's Summit supercomputer providing more than an order of magnitude higher performance than what is available in IEEE double precision. At the same time, the gap between the compute power on the one hand and the memory bandwidth on the other hand keeps increasing, making data access and communication prohibitively expensive compared to arithmetic operations. Having the choice between ignoring the hardware trends and continuing the traditional path, and adjusting the software stack to the changing hardware designs, the US Exascale Computing Project decided for the aggressive step of building a multiprecision focus effort to take on the challenge of designing and engineering novel algorithms exploiting the compute power available in low precision and adjusting the communication format to the application specific needs. To start the multiprecision focus effort, we survey the numerical linear algebra community and summarize all existing multiprecision knowledge, expertise, and software capabilities in this landscape analysis report. We also include current efforts and preliminary results that may not yet be considered "'mature technology," but have the potential to grow into production quality within the multiprecision focus effort. As we expect the reader to be familiar with the basics of numerical linear algebra, we refrain from providing a detailed background on the algorithms themselves but focus on how mixed- and multiprecision technology can help improving the performance of these methods and present highlights of application significantly outperforming the traditional fixed precision methods.

Contents

2 Dense Linear Algebra

2.1 Low Precision BLAS

The revolution of machine learning applications and artificial intelligence (AI) spiked an interest in developing high-performance half-precision arithmetic (16-bit floating-point format), since most AI applications do not necessarily require the accuracy of single or double precision [1]. Half precision also enables machine learning applications to run faster, not only because of the faster arithmetic, but also because of the reduction in memory storage and traffic by a factor of 2× against single precision, and by a factor of 4× against double precision.

In terms of vendor support, NVIDIA, Google, and AMD manufacture hardware that is capable of performing floating point arithmetic using 16-bit formats. Google's Tensor Processing Units (TPUs) are customized chips that are mainly designed for machine learning workloads using the `bfloat16` format. AMD also provides half-precision capabilities, and their software stack shows support for both the `bfloat16` format and the IEEE format [2]. The theoretical performance of half-precision on AMD GPUs follows the natural 2×/4× speedups against single/double precisions, respectively. As an example, the Mi50 GPU has a theoretical FP16 performance of 26.5 Tflop/s, against a 13.3 Tflop/s for FP32 and 6.6 Tflop/s for FP64. But perhaps the most widely accessible hardware with half-precision capability are NVIDIA GPUs, which have introduced half-precision arithmetic since the Pascal architecture. Throughout this section, we will focus on NVIDIA GPUs and its math libraries to highlight half-precision developemnts for numerical kernels.

NVIDIA GPUs implement the "binary16" format which is defined by the IEEE-754 standard [2]. While the Pascal GPU architecture introduced hardware support for FP16 arithmetic, the Volta architecture, which powers the Summit supercomputer,[1] comes with hardware acceleration units (called Tensor Cores) for matrix multiplication in FP16. These Tensor Cores are theoretically 12× faster than the theoretical FP16 peak performance of the preceding architecture. Applications taking advantage of the Tensor Cores can run up to 4× faster than using the regular FP16 arithmetic on the same GPU. The Tensor Cores are also able to perform a mixed-precision multiplication, with a low precision input (e.g. half-precision) and a higher precision output (typically single-precision). The Tensor Core units are discussed in more details in Section 2.1.1.

In terms of half-precision Basic Linear Algebra Subroutines (BLAS), most of the available routines consider only dense matrix multiplications (GEMMs). From the perspective of machine learning applications, most of the performance critical components in training/inference can be reformulated to take advantage of the GEMM kernel. As for dense linear algebra, many high level algorithms are built to extract their high performance from GEMM calls. Therefore, accelerating such performance-critical steps through FP16 GEMM (HGEMM) would propagate the performance advantage to the entire algorithm, while keeping other numerical stages in their original precision(s). An example of this practice is the mixed precision dense LU factorization [3], which is used to accelerate the solution of $Ax = b$ in double precision, see Section 2.2.

2.1.1 Hardware Acceleration of Half Precision

The CUDA Toolkit is one of the first programming models to provide half-precision (i.e., FP16) arithmetic. Early support was added in late 2015 for selected embedded GPU models that are based on the Maxwell architecture. The FP16 arithmetic has become mainstream in CUDA-enabled GPUs since the Pascal architecture. In general, half precision has a dynamic range that is significantly smaller than single or double precision.

The Volta and Turing architectures introduce hardware acceleration for matrix multiplication in FP16. The hardware acceleration units are called Tensor Cores. They can deliver a theoretical peak performance that is up to 8× faster than the peak FP32 performance. As an example, each Volta V100 GPU has 640 Tensor Cores, evenly distributed across 80 multiprocessors. Each Tensor Core possesses a mixed-precision $4 \times 4 \times 4$ matrix processing array which performs the operation $D = A \times B + C$, where A, B, C and D are 4×4 matrices. The inputs A and B must be represented in FP16 format, while C and D can be represented in FP16 or in FP32 formats. It is also possible that C and D point to the same matrix.

[1] https://www.olcf.ornl.gov/summit/

NVIDIA's vendor library cuBLAS provides various optimized routines, mostly GEMMs, that can take advantage of the Tensor Core acceleration if configured accordingly. As an example, the routine `cublasHgemm` implements the GEMM operation for real FP16 arithmetic.

Apart from the vendor library, taking advantage of the Tensor Cores in a custom kernel is possible also through the use of low-level APIs that are provided by the programming model. As shown in Figure 1, Tensor Cores deal with input and output data through opaque data structures called *fragments*. Each fragment is used to store one matrix. Fragments can be loaded from shared memory or from global memory using the `load_matrix_sync()` API. A similar API is available for storing the contents of an output fragment into the shared/global memory of the GPU. The `mma_sync()` API is used to perform the multiplication. The user is responsible for declaring the fragments as required, and calling the APIs in the correct sequence.

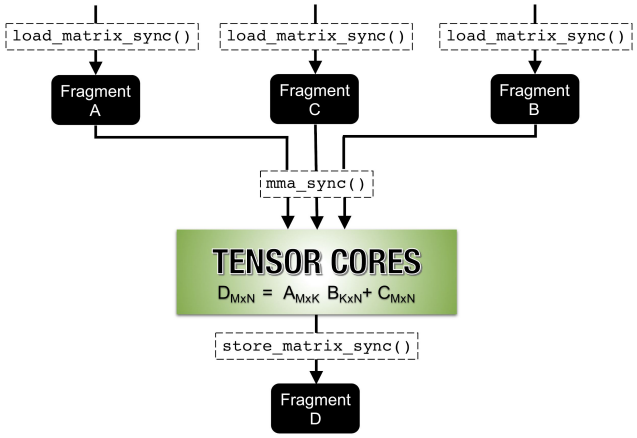

Figure 1: Programmability of the Tensor Core units

The programming model imposes some restrictions to the programming of the Tensor Cores. First, the GEMM dimensions (M, N, K), which also control the size of the fragments, are limited to three discrete combinations, namely (16, 16, 16), (32, 8, 16), and (8, 32, 16). Second, the operations of load, store, and multiply must be performed by one full warp (32 threads). Finally, the load/store APIs require that the leading dimension of the corresponding matrix be multiple of 16-bytes. As an example, a standard GEMM operation of size (16, 16, 16) requires three `load_matrix_sync()` calls (for A, B, and C), one `mma_sync()` call, and then a final `store_matrix_sync()` call to write the result. The latest CUDA version to date (10.1) provides direct access to the Tensor Cores through an instruction called `mma.sync`. The instruction allows one warp to perform four independent GEMM operations of size (8, 8, 4). However, using the explicit instruction may lead to long-term compatibility issues for open-source libraries as new architectures are released.

2.1.2 Half-precision GEMM (HGEMM)

The cuBLAS library provides several routines that take advantage of the reduced FP16 precision. Figure 2 shows the performance of three different HGEMM kernels. An HGEMM kernel with half-precision output can achieve up to 30 Tflop/s of performance if the tensor cores are turned off. While this is around 2× the single-precision performance, a significantly higher performance can be achieved if the tensor cores are turned on. As the figure shows, the tensor cores are capable of delivering an asymptotic 100 Tflop/s, which is 5× the asymptotic performance of a non-accelerated HGEMM.

However, perhaps the most interesting performance graph of Figure 2 is the HGEMM with FP32 output. The reason is that its performance is close to the accelerated HGEMM kernel, but with much more precision on the output. This is of particular importance for mixed-precision algo-

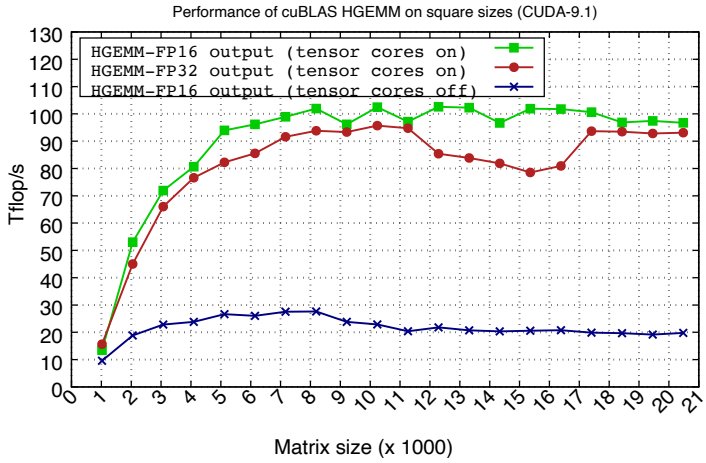

Figure 2: Performance of different HGEMM kernel from the cuBLAS library on square sizes. Results are shown on a Tesla V100 GPU using CUDA-9.1.

rithms [3, 4]. To put this fact in more perspective, Figure 3 shows the forward error between the three different HGEMM kernels, with respect to the single-precision GEMM kernel from the Intel MKL library. The forward error is computed as $\frac{\|R_{cuBLAS} - R_{MKL}\|_F}{\sqrt{k+2}|\alpha|\|A\|_F\|B\|_F + 2|\beta|\|C\|_F}$, where k is equal to the matrix size. The first surprising observation is that an HGEMM operation with FP16 output is more accurate if the tensor cores are turned on, which means that the utilization of the tensor core units achieves both better performance and higher accuracy. The second observation is that performing HGEMM with FP32 output achieves at least two more digits of accuracy when compared with the other two HGEMM variants. Given that HGEMM with FP32 output is mostly within 90% of the peak tensor core throughput, it is clearly the best option for mixed-precision algorithms that target achieving higher accuracy while taking advantage of the half-precision.

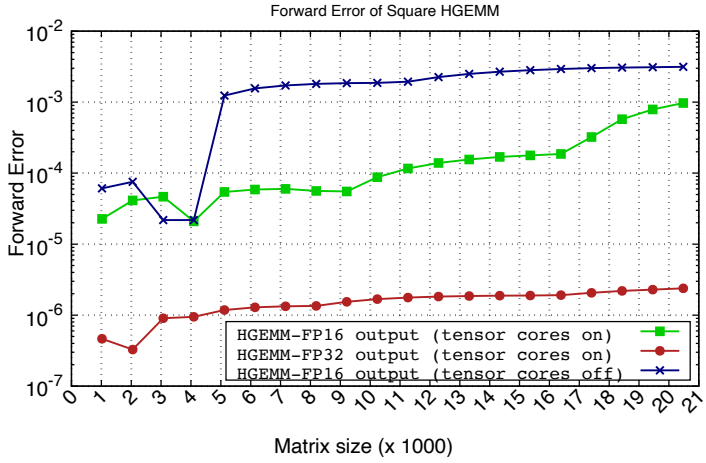

Figure 3: Forward error of HGEMM with respect to MKL SGEMM ($C = \alpha AB + \beta C$). Results are shown for square sizes using cuBLAS 9.1 and MKL 2018.1. The forward error is computed as $\frac{\|R_{cuBLAS} - R_{MKL}\|_F}{\sqrt{k+2}|\alpha|\|A\|_F\|B\|_F + 2|\beta|\|C\|_F}$, where k is equal to the matrix size.

2.1.3 Batch HGEMM

Apart from the vendor-supplied BLAS, few efforts have focused on building open-source BLAS routines that utilize the tensor cores of NVIDIA GPUs. An example of such efforts is in the MAGMA library [5], which has a batch HGEMM kernel that makes use of the tensor cores [6]. The kernel

builds an abstraction layer over the tensor cores to overcome their size restrictions, so that arbitrary blocking sizes can be used by the kernel. The batch HGEMM kernel in MAGMA outperforms cuBLAS for relatively small sizes, as shown in Figure 4. The same work also shows that extremely small matrix (e.g. whose sizes ≤ 10) may not necessarily benefit from tensor core acceleration.

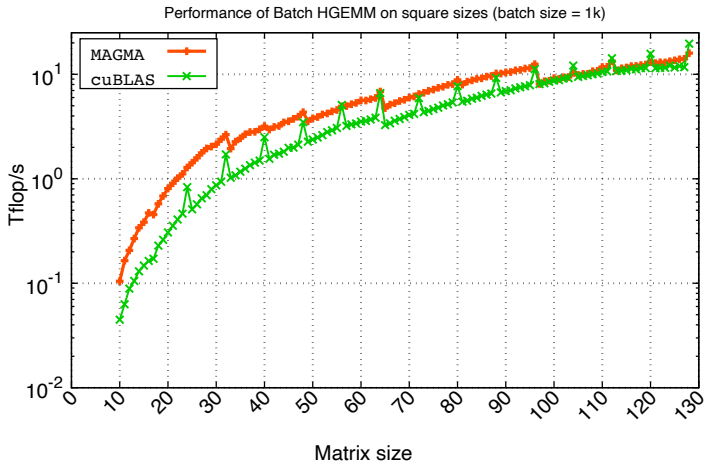

Figure 4: Performance of the batch HGEMM kernel on square sizes. Results are shown on a Tesla V100 GPU using CUDA-9.1.

2.2 Classical Iterative Refinement

On modern architectures, the performance of 32-bit operations is often at least twice as fast as the performance of 64-bit operations. There are two reasons for this. Firstly, 32-bit floating point arithmetic is usually twice as fast as 64-bit floating point arithmetic on most modern processors. Secondly the number of bytes moved through the memory system is halved.

Mixed precision algorithms stem from the observation that, in many cases, a single precision solution of a problem can be refined to the point where double precision accuracy is achieved. The refinement can be accomplished, for instance, by means of the Newtons algorithm (see Equation (1)) which computes the zero of a function $f(x)$ according to the iterative formula

$$x_{n+1} = x_n - \frac{f(x_n)}{f'(x_n)} \tag{1}$$

In general, we would compute a starting point and $f(x)$ in single precision arithmetic and the refinement process will be computed in double precision arithmetic.

If the refinement process is cheaper than the initial computation of the solution, then double precision accuracy can be achieved nearly at the same speed as the single precision accuracy.

A common approach to the solution of linear systems, either dense or sparse, is to perform the LU factorization of the coefficient matrix using Gaussian elimination. First, the coefficient matrix A is factored into the product of a lower triangular matrix L and an upper triangular matrix U. Partial row pivoting is in general used to improve numerical stability resulting in a factorization $PA = LU$, where P is a permutation matrix. The solution for the system is achieved by first solving $Ly = Pb$ (*forward substitution*) and then solving $Ux = y$ (*backward substitution*). Due to round-off errors, the computed solution x carries a numerical error magnified by the condition number of the coefficient matrix A.

In order to improve the computed solution, we can apply an iterative process which produces a correction to the computed solution at each iteration, which then yields the method that is commonly known as the *iterative refinement* (IR) algorithm. As Demmel points out [7], the nonlinearity of the round-off errors makes the iterative refinement process equivalent to the Newtons method applied to the function $f(x) = b - Ax$. Provided that the system is not too ill-conditioned, the algorithm

produces a solution correct to the working precision. Iterative refinement in double/double precision is a fairly well understood concept and was analyzed by Wilkinson [8], Moler [9] and Stewart [10].

Iterative refinement is a long-standing method that was programmed by Wilkinson in the 1940s for the ACE digital computer. The idea is to improve the computed solution of a linear system by iteratively solving a correction equation and adding the correction to the current solution; for a comprehensive treatment, Higham [11, Chap. 12].

The algorithm can be modified to use a mixed precision approach. The three tasks original solve/-factorization, residual computation, and correction equation solvecan be done in the same precision (fixed precision) or in different precisions (mixed precision). The original usage was mixed precision, with the residual computed at twice the working precision.

For the current work, the factorization $PA = LU$ and the solution of the triangular systems $Ly = Pb$ and $Ux = y$ are computed using single precision arithmetic. The residual calculation and the update of the solution are computed using double precision arithmetic and the original double precision coefficients (see Algorithm 1). The most computationally expensive operation, the factorization of the coefficient matrix A, is performed using single precision arithmetic and takes advantage of its higher speed. The only operations that must be executed in double precision are the residual calculation and the update of the solution (they are denoted with an ϵ_d in Algorithm 1).

Algorithm 1 Mixed precision, Iterative Refinement for Direct Solvers

1: $LU \leftarrow PA$ ▷ (ϵ_s)
2: Solve $Ly = Pb$ ▷ (ϵ_s)
3: Solve $Ux_0 = y$ ▷ (ϵ_s)
4: **for** $k = 1, 2, \ldots$ **do**
5: $r_k \leftarrow b - Ax_{k-1}$ ▷ (ϵ_d)
6: Solve $Ly = Pr_k$ ▷ (ϵ_s)
7: Solve $Uz_k = y$ ▷ (ϵ_s)
8: $x_k \leftarrow x_{k-1} + z_k$ ▷ (ϵ_d)
9: Check convergence
10: **end for**

We observe that the only operation with computational complexity of $O(n^3)$ is handled in single precision, while all operations performed in double precision are of at most $O(n^2)$ complexity. The coefficient matrix A is converted to single precision for the LU factorization and the resulting factors are stored in single precision while the initial coefficient matrix A needs to be kept in memory. Therefore, one drawback of the following approach is that the it uses 50% more memory than the standard double precision algorithm.

The method in Algorithm 1 can offer significant improvements for the solution of a sparse linear system in many cases if:

1. single precision computation is significantly faster than double precision computation;
2. the iterative refinement procedure converges in a small number of steps;
3. the cost of each iteration is small compared to the cost of the system factorization. If the cost of each iteration is too high, then a low number of iterations will result in a performance loss with respect to the full double precision solver. In the sparse case, for a fixed matrix size, both the cost of the system factorization and the cost of the iterative refinement step may substantially vary depending on the number of nonzeroes and the matrix sparsity structure; this will be addressed in Section 4.2. In the dense case, results are more predictable.

Note that the choice of the stopping criterion in the iterative refinement process is critical. Formulas for the errors computed at each step of Algorithm 1 can be obtained for instance in [12, 13].

Recently, Carson and Higham [14] analyzed a three-precision iterative refinement scheme (factorization precision, working precision, residual precision) and concluded that if the condition number of A is not too large, namely $\kappa_\infty(A) = \|A\|_\infty \left\|A^{-1}\right\|_\infty < 10^4$, then using FP16 for the $O(n^3)$ portion (the LU factorization) and (FP32, FP64) or (FP64, FP128) as the (working, residual) preci-

sion for the $O(n^2)$ portion (refinement loop), one can expect to achieve forward error and backward error on the order of 10^{-8} and 10^{-16} respectively. We note that, if $\hat{x}$ is an approximate solution of $Ax = b$ the forward error is defined by $\frac{\|\hat{x}-x\|_\infty}{\|x\|_\infty}$ and the backward error is defined by $\min\{\epsilon : (A + \Delta A)\hat{x} = b, \|\Delta A\| \leq \epsilon \|A\|\}$ and can be evaluated as $\frac{\|r\|_2}{\|A\|_2\|\hat{x}\|_2}$, where $r = b - A\hat{x}$.

2.3 GMRES-IR

Carson and Higham [15] proposed the use of Generalized Minimum Residual (GMRES) [16] preconditioned by the FP16 LU factors as the solver in correction equation and showed that in this case the constraint on the condition number can be relaxed to $\kappa_\infty(A) < 10^8$ when the (working, residual) precision is (FP32, FP64) and to 10^{12} when the (working, residual) precision is (FP64, FP128). We refer to [17, Table 2.2] for limiting condition number, and forward, backward errors. Analysis covering this GMRES-based approach when two precisions are used with the residual precision equal to the working precision is given in [17].

The idea is that the GMRES solver will provide a better and more stable approximate solution to $Az_k = r_k$ than the basic triangular solve, which is directly affected by the quality of the low-precision LU factors. Using GMRES, we can still guarantee that the solution of the correction equation $Az_k = r_k$ has some correct digits and a residual at the level of the convergence tolerance requested by the algorithm. The convergence tolerance of the refinement process is chosen to be of the order of the unit roundoff of the low-precision arithmetic used during the factorization (e.g., we use 10^{-4} or 10^{-8} when the LU factorization is in FP16 or FP32, respectively). This variant is called GMRES-based iterative refinement (GMRES-IR) by Carson and Higham, and it is described in Algorithm 2. Note that U^{-1} and L^{-1} are never explicitly formed; instead matrixvector products $U^{-1}L^{-1}Ay$ needed by GMRES are computed by multiplication by A followed by two triangular solves. Since this paper focuses on the practical usage and possible performance gains rather than error analysis, we point the reader to [11, 15, 17] for detailed error analysis of the IR and GMRES-IR techniques.

Algorithm 2 Iterative refinement using GMRES (GMRES-IR)

1: Convert A to A_f from precision u_w to u_f
2: Perform LU factorization of A_f in precision u_f
3: Find the initial solution x_0 using the computed LU factorization of A_f in precision u_f then cast x_0 to precision u_w
4: // Refinement loop, outer loop
5: **repeat**
6: Compute residual $r_k = b - Ax_k$ in precision u_r and cast it to u_w ▷ Residual
7: Solve $U^{-1}L^{-1}Az_k = U^{-1}L^{-1}r_k$ by GMRES in precision u_w ▷ Correction
8: Correct the current solution $x_{k+1} = x_k + z_k$ in precision u_w ▷ Update
9: **until** x_k is accurate enough

The design and implementation of numerical algorithms that efficiently exploit current highly parallel computer architectures is a challenge, especially if close to peak performance is to be achieved. Since in the GMRES-IR approach $\mathcal{O}(n^3)$ operations are in lower precision, this idea allows a new class of iterative refinement solvers and a number of computational techniques that allow us to solve fundamental $Ax = b$ problems close to peak FP64 performance. The developments open up directions for future work, including further optimizations, development of a full set of mixed-precision factorizations, linear system solvers as well as eigensolvers and singular value decomposition (SVD).

2.3.1 Scaling

It is clear that the use of low precision floating-point arithmetic in iterative refinement can lead to significant speedups. However, fp16 has a small dynamic range, and therefore encountering overflow, underflow, and subnormal numbers is very likely. To address these issues now we discuss scaling algorithms, which are presented in [18], [19] and [20]. We refer interested readers to these references for more details.

We consider a two-sided diagonal scaling prior to converting to fp16: A is replaced by RAS, where

$$R = \mathrm{diag}(r_i), \quad S = \mathrm{diag}(s_i), \qquad r_i, s_i > 0, \quad i = 1\colon n.$$

Such scaling algorithms have been developed in the context of linear systems and linear programming problems. Despite the large literature on scaling such problems, no clear conclusions are available on when or how one should scale; see [21] for a recent experimental study. In contrast to previous studies, where the aim of scaling has been to reduce a condition number or to speed up the convergence of an iterative method applied to the scaled matrix, we scale in order to help squeeze a single or double precision matrix into half precision, with a particular application to using the resulting half precision LU factors for iterative refinement.

Our usage of two-sided diagonal scaling is given in Algorithm 3.

Algorithm 3 (Two-sided diagonal scaling then round). This algorithm rounds $A \in \mathbb{R}^{n\times n}$ to the fp16 matrix $A^{(h)}$, scaling all elements to avoid overflow. $\theta \in (0, 1]$ is a parameter.

1: Apply any two-sided diagonal scaling algorithm to A, to obtain diagonal matrices R, S.
2: Let β be the maximum magnitude of any entry of RAS.
3: $\mu = \theta x_{\max}/\beta$
4: $A^{(h)} = fl_h(\mu(RAS))$

We consider two different algorithms for determining R and S; both algorithms are carried out at the working precision. One option is row and column equilibration, which ensures that every row and column has maximum element in modulus equal to 1—that is, each row and column is equilibrated. The LAPACK routines `xyyEQU` carry out this form of scaling [22]. A symmetry-preserving two-sided scaling is proposed by Knight, Ruiz, and Uçar [23]. The algorithm is iterative and scales simultaneously on both sides rather than sequentially on one side then the other.

2.4 Mixed-precision Factorizations

Haidar at al. [3] proposed IR methods using mixed-precision factorizations. While classical IR and extensions like the GMRES-IR use fixed-precision factorizations (e.g., in precision u_f as illustrated in Algorithm 2), mixed-precision factorizations apply higher precision (e.g., u_w) at critical parts of the algorithm to get extra-precise factorizations while retaining the performance of the low-precision counterpart. The developments were applied to GPU Tensor Cores and illustrate that FP16 can be used to get FP64 accuracy for problems with $\kappa_\infty(A)$ of up to 10^5, compared to a more typical requirement of $\kappa_\infty(A) < 10^4$. The work illustrates that mixed-precision techniques can be of great interest for linear solvers in many engineering areas. The results show that on single NVIDIA V100 GPU the new solvers can be up to four times faster than an optimized double precision solver [3],[4], [24].

The mixed-precision factorizations were motivated by the need to get extra precision when working with very low precisions, like the FP16. Also, this allows to easily overcome implementation issues and other limitations of using FP16 arithmetic, and thus harness the power of specialized hardware, like the GPU Tensor Cores, for a larger range of scientific computing applications.

A building-block for the mixed-precision factorizations is mixed-precision BLAS. Having mixed-precision BLAS can enable the ease of developing many mixed-precision LAPACK algorithms. Currently, cuBLAS provides mixed FP32-FP16 precision HGEMM that uses the GPU's Tensor Cores FP16 acceleration. In this GEMM, the input `A` and `B` matrices can be FP32, internally get casted to FP16, used to compute a GEMM on Tensor Cores in full (FP32) accuracy, and the result stored back on the GPU memory in FP32. There are two main benefits of having such mixed-precision BLAS routines. First, note that this mixed-precision HGEMM is almost as fast as the non-mixed FP16 precision only HGEMM (see Figure 2), and second, the use of mixed-precision gains about one more decimal digit of accuracy (see Figure 3).

Besides the two main benefits outlined above, the availability of mixed-precision GEMM also enables us to easily develop other mixed-precision algorithms, e.g., LAPACK, and in particular, the various mixed-precision factorizations that we recently added in MAGMA [3]. Figure 5 shows the performance of the mixed-precision LU (marked as "FP16-TC hgetrf LU"). Note that this factor-

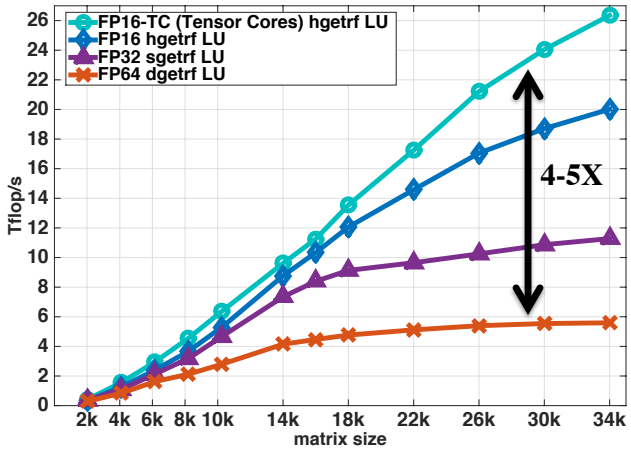

Figure 5: Mixed-precision LU (hgetrf) in MAGMA and its speedup vs. FP64 LU.

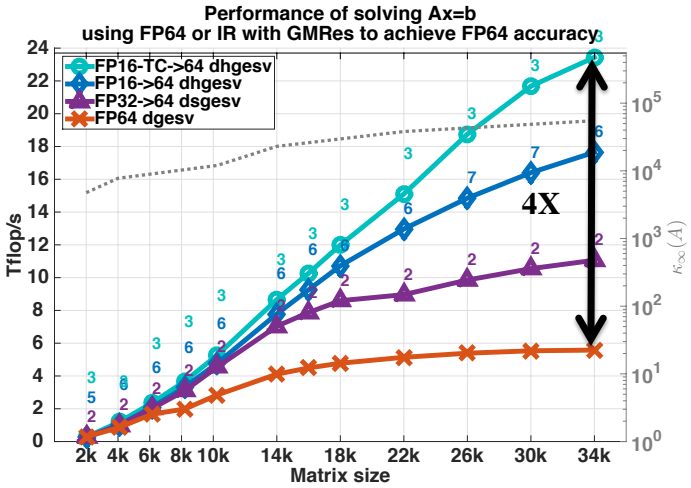

Figure 6: Mixed-precision iterative refinement in MAGMA and acceleration vs. FP64 solvers. Note $\approx$ 2% overhead per iteration, and more than 2× less overhead in terms of iterations for mixed-precision LU vs. regular FP16 LU (the 3 vs. 7 iterations until FP64 convergence).

ization is about 4 – 5× faster than dgetrf. Its data storage is in FP32 and the implementation is the same as sgetrf, except that it uses the mixed-precision HGEMMs for the trailing matrix updates.

Figure 6 shows the mixed-precision iterative refinement in MAGMA [3]. The 4× overall acceleration is due to a number of optimizations. First, note that the 3 iterations to get to FP64 accuracy led to loss of about 2 Tflop/s compared to the hgetrf performance (24 Tflop/s vs. 26 Tflop/s), i.e., the overhead of one iteration can be deduced as being about 2%. Loosing 75%, e.g., through up to 40 iterations, would lead to no acceleration compared to FP64 solver. This overhead per iteration is very low, which is due to fusing all data conversions with computational kernels. Without fusion, the overhead would have been easily about 3× higher. Second, note that the iterative refinement using the mixed-precision factorization has more than 2× smaller overhead in terms of iterations to solution (the 3 vs. 7 iterations until FP64 convergence). This is due to the extra digit of accuracy that the mixed-precision HGEMM has over the FP16 HGEMM, which also translates to a more accurate mixed-precision LU.

2.5 Cholesky Factorization

In the previous section we considered scaling of a general and symmetric matrix, we now assume that we are given a symmetric positive definite matrix $A \in \mathbb{R}^{n\times n}$ in finite precision arithmetic of precision u and wish to compute a Cholesky factorization in finite precision arithmetic with precision $u_h > u$. The most practically important cases are where (u_h, u) = (half, single), (half, double), or (single, double). Naively, we might first form $A^{(h)} = fl_h(A)$, where fl_h denotes the operation of rounding to precision u_h, and then compute the Cholesky factorization of $A^{(h)}$ in precision u_h. For half precision this procedure can fail for two reasons. First, if fp16 is used then the limited range might cause overflow during the rounding. Second, for both bfloat16 and fp16, $A^{(h)}$ might not be (sufficiently) positive definite, because a matrix whose smallest eigenvalue is safely bounded away from zero with respect to single precision or double precision may become numerically indefinite under the perturbation induced by rounding to half precision. The second issue can also be encountered when a double precision matrix is rounded to single precision. To overcome these problems we will use scaling and shifting.

2.5.1 Scaling

The first step is to scale the matrix A to $H = D^{-1}AD^{-1}$, $D = \mathrm{diag}(a_{ii}^{1/2})$, and D will be kept at precision u. Because Cholesky factorization is essentially numerically invariant under two-sided diagonal scaling (as can be seen from the recurrence relations for the Cholesky factors), the sole reason for scaling is to reduce the dynamic range in order to avoid overflow and reduce the chance of underflow, for fp16. For bfloat16 or single precision it will not usually be necessary to scale, and we can work with A throughout. For the rest of the presentation we will always scale, for simplicity of notation. Two-sided diagonal scaling before rounding to low precision was used in [18] in the context of LU factorization. The scaling used there equilibrates rows and columns; our scaling with D is the natural analogue of that for symmetric positive definite matrices. For Cholesky factorization we need an extra ingredient to ensure a successful factorization, which we consider next.

2.5.2 Shifting

We now convert H to the lower precision u_h, incorporating a shift to ensure that the lower precision matrix is sufficiently positive definite for Cholesky factorization to succeed, as discussed in [19, Sec. 2]. We will shift H by $c_n u_h I$, where c_n is a positive integer constant, to obtain $G = H + c_n u_h I$. Since the diagonal of H is I, this shift incurs no rounding error and it produces the same result whether we shift in precision u then round or round then shift in precision u_h.

For fp16, in view of the narrow range we will also multiply the shifted matrix by a scalar to bring it close to the overflow level $x_{\max}$, in order to minimize the chance of underflow and of subnormal numbers being produced. So our final precision-u_h matrix is constructed as

$$\begin{aligned} G &= H + c_n u_h I, \\ \beta &= 1 + c_n u_h, \quad \mu = \theta x_{\max}/\beta, \\ A^{(h)} &= fl_h(\mu G), \end{aligned} \tag{2}$$

where $\theta \in (0,1)$ is a parameter. Note that $\beta = \max_{ij} |g_{ij}|$, so the largest absolute value of any element of $A^{(h)}$ is $\theta x_{\max}$. Note also that since the growth factor for Cholesky factorization is 1 (see, e.g., [11, Prob. 10.4]), there is no danger of overflow during Cholesky factorization of $A^{(h)}$.

We refer to [19, Sec. 3.3] for an analysis regarding the choice of c_n. However since the estimates are pessimistic, we take the pragmatic approach of taking c_n to be a small constant c. If the Cholesky factorization fails we will increase c and try again. We will determine experimentally how large c should be for a range of problems of interest. Based on this we give the low precision Cholesky factorization algorithm in Algorithm 4.

2.6 Iterative Refinement for Least Squares Problems

We consider the linear least squares problem $\min_x \|Ax - b\|_2$, where $A \in \mathbb{R}^{m\times n}$ with $m \geq n$ has full rank. The ideas of mixed-precision iterative refinement and GMRES-IR can be adapted to the least squares case. Least squares problems may be ill conditioned in practice, and so rounding errors may

Algorithm 4 (Cholesky factorization in precision u_h). Given a symmetric positive definite $A \in \mathbb{R}^{n\times n}$ in precision u this algorithm computes an approximate Cholesky factorization $R^T R \approx \mu D^{-1} A D^{-1}$ at precision u_h, where $D = \mathrm{diag}(a_{ii}^{1/2})$. The scalar $\theta \in (0,1]$ and the positive integer c are parameters.

1: $D = \mathrm{diag}(a_{ii}^{1/2})$, $H = D^{-1}AD^{-1}$ % Set $h_{ii} \equiv 1$ instead of computing it.
2: $G = H + cu_h I$
3: $\beta = 1 + cu_h$
4: $\mu = \theta x_{\max}/\beta$
5: $A^{(h)} = fl_h(\mu G)$
6: Attempt Cholesky factorization $A^{(h)} = R^T R$ in precision u_h.
7: **if** Cholesky factorization failed **then**
8: $c \leftarrow 2c$, goto line 2
9: **end if**

result in an insufficiently accurate solution. In this case, iterative refinement may be used to improve accuracy, and it also improves stability.

2.6.1 Cholesky-Based Approach

The normal equations method solves

$$A^T Ax = A^T b$$

using the Cholesky factorization of $A^T A$ (see Section 2.5). In general, this method is deprecated by numerical analysts because it has a backward error bound of order $\kappa_2(A)u$ [11, sect. 20.4] and the Cholesky factorization can break down for $\kappa_2(A) > u^{-1/2}$, but it is used by statisticians with some justification [25]. Here, we assume that A is (sufficiently) well conditioned. We propose the GMRES-IR-based least squares solver given in Algorithm 5.

Algorithm 5 (Cholesky-based GMRES-IR for the least squares problem) Let a full rank $A \in \mathbb{R}^{m\times n}$, where $m \geq n$, and $b \in \mathbb{R}^m$ be given in precision u. This algorithm solves the least squares problem $\min_x \|b - Ax\|_2$ using Cholesky-based GMRES-IR. The scalar $\theta \in (0,1]$ and the positive integer c are parameters.

1: Compute $B = AS$, where $S = \mathrm{diag}(1/\|a_j\|_2)$, with a_j the jth column of A.
2: $\mu = \theta x_{\max}$
3: $B^{(h)} = fl_h(\mu^{1/2} B)$
4: Compute $C = B^{(h)T} B^{(h)}$ in precision u_h.
5: Compute the Cholesky factorization $C + cu_h \mathrm{diag}(c_{ii}) = R^T R$ in precision u_h.
6: **if** Cholesky factorization failed **then**
7: $c \leftarrow 2c$, goto line 5
8: **end if**
9: Form $b^{(h)} = fl_h(SA^T b)$.
10: Solve $R^T R y_0 = b^{(h)}$ in precision u_h and form $x_0 = \mu S y_0$ at precision u.
11: **for** $i = 0\colon i_{\max} - 1$ **do**
12: Compute $r_i = A^T(b - Ax_i)$ at precision u_r and round r_i to precision u.
13: Solve $MA^T Ad_i = Mr_i$ by GMRES at precision u, where $M = \mu S R^{-1} R^{-T} S$ and matrix–vector products with $A^T A$ are computed at precision u_r, and store d_i at precision u.
14: $x_{i+1} = x_i + d_i$ at precision u.
15: **if** converged **then**
16: return x_{i+1}, **quit**
17: **end if**
18: **end for**

We make some comments on the algorithm. Line 1 produces a matrix B with columns of unit 2-norm. The computation $C = B^{(h)T} B^{(h)}$ on line 4 produces a symmetric positive definite matrix with constant diagonal elements $\mu = \theta x_{\max}$, so overflow cannot occur for $\theta < 1$. The shift on line 5 is analogous to that in Algorithm 4, but here the matrix C is already well scaled and in precision u_h so there is no need to scale C to have unit diagonal.

There are two reasons why explicitly forming $C = B^{(h)T}B^{(h)}$ in Algorithm 5 is reasonable from the numerical stability point of view. First, C is used to form a preconditioner, so the usual problems with forming a cross product matrix (loss of significance and condition squaring) are less of a concern. Second, if we are working in fp16 on an NVIDIA V100 we can exploit the tensor cores when forming C to accumulate block fused multiply-add operations in single precision; this leads to a more accurate C, as shown by the error analysis of Blanchard et al. [26].

For the computed $\hat{R}$ we have

$$\hat{R}^T\hat{R} \approx B^{(h)T}B^{(h)} \approx \mu S A^T A S,$$

or

$$(A^TA)^{-1} \approx \mu S \hat{R}^{-1}\hat{R}^{-T} S,$$

so we are preconditioning with an approximation to the inverse of A^TA. We apply the preconditioned operator MA^TA to vectors at precision u_r. Computing $y = A^TAx$ costs $4mn$ flops and $SR^{-1}R^{-T}y$ costs another $2n^2 + n$ flops, making $4mn + 2n^2 + n$ flops in total. For $m \gg n$ and large n, computing $y = A^TAx$ costs a factor $n/4$ fewer flops than the mn^2 flops needed to form A^TA, while for $m \approx n$ the difference is a factor $n/6$. For large n, even allowing for the fact that the flops we are comparing are at different precisions, as long as GMRES converges quickly the cost of the refinement stage should be negligible compared with the cost of forming A^TA and computing the Cholesky factorization.

Related to this work is the Cholesky–QR algorithm for computing a QR factorization $A = QR$. It forms the cross-product matrix A^TA, computes the Cholesky factorization $A^TA = R^TR$, then obtains the orthogonal factor Q as $Q = AR^{-1}$, and this process can be iterated for better numerical stability; see, for example, [27], [28], [29], [30]. Our work differs in that we do not compute Q, we carry out the Cholesky factorization in lower precision than the working precision, and we solve a least squares problem using preconditioned iterative refinement.

2.6.2 Augmented Matrix Approach

Another approach to mixed precision least squares iterative refinement was presented by Carson, Higham, and Pranesh in [20]. This approach is based on the method of using the QR factorization

$$A = Q\begin{bmatrix} R \\ 0 \end{bmatrix},$$

where $Q = [Q_1, Q_2] \in \mathbb{R}^{m\times m}$ is an orthogonal matrix with $Q_1 \in \mathbb{R}^{m\times n}$ and $Q_2 \in \mathbb{R}^{m\times(m-n)}$, and $R \in \mathbb{R}^{n\times n}$ is upper triangular. The unique least squares solution is $x = R^{-1}Q_1^Tb$ with residual $\|b - Ax\|_2 = \|Q_2^Tb\|_2$.

An iterative refinement approach that works even when $Ax = b$ is inconsistent was suggested by Björck [31]. Refinement is performed on the augmented system

$$\begin{bmatrix} I & A \\ A^T & 0 \end{bmatrix}\begin{bmatrix} r \\ x \end{bmatrix} = \begin{bmatrix} b \\ 0 \end{bmatrix}, \tag{3}$$

which is equivalent to the normal equations. In this way, the solution x_i and residual r_i for the least squares problem are simultaneously refined. Björck [31] shows that the linear system can be solved by reusing the QR factors of A.

Existing analyses of the convergence and accuracy of this approach in finite precision assume that at most two precisions are used; the working precision u is used to compute the QR factorization, solve the augmented system, and compute the update. A second precision $u_r \le u$ is used to compute the residuals. Typically $u_r = u^2$, in which case it can be shown that as long as the condition number of the augmented system matrix is smaller than u^{-1}, the refinement process will converge with a limiting forward error on the order of u; see [32] and [11, sect. 20.5] and the references therein.

The work [20] shows that the three-precision iterative refinement approach of Carson and Higham [14] can be applied in this case; the theorems developed in [14] regarding the forward error and normwise and componentwise backward error for iterative refinement of linear systems are applicable. The only thing that must change is the analysis of the method for solving the correction equation since we now work with a QR factorization of A, which can be used in various ways.

The work in [20] also extends the GMRES-based refinement scheme of [15] to the least squares case and shows that one can construct a left preconditioner using the existing QR factors of A such that GMRES provably converges to a backward stable solution of the preconditioned augmented system. Further, it is shown that an existing preconditioner developed for saddle point systems can also work well in the GMRES-based approach in practice, even though the error analysis is not applicable. We refer the reader to [20] for further details.

2.7 Quantized Integer LU Factorization

Quantization is a technique widely being used in deep learning inference [33, 34]. While the model is usually still trained in single precision, quantization compress the data and use lower precision to carry out the computation in inference stage which is applying the trained model to new data for real application. For an int8 quantized model, the data is converted into 8-bit integers. The computation and communication are reduced 4 times comparing to 32-bit single precision while the accuracy lost is acceptable (usually $< 1\%$ for predictive models). Integer arithmetic is available on most hardware architectures. Field Programmable Gate Array (FPGA)s are usually more capable in integer operations and might not have floating-point number arithmetic units. New Application-Specific Integrated Circuit (ASIC)s for deep learning inference are also moving toward using mostly integer arithmetic for quantized neural networks. This motivated to investigate the use of integer arithmetic for the Gaussian elimination (LU factorization) with partial pivoting.

2.7.1 Quantized Integer LU Algorithm

Storage format	i in 32-bit integer
Represented real number	$R(i) = i/2^{32} \times 2^0$
Conversion from `double` precision number α	$i \leftarrow \texttt{int32}(\alpha \times 2^{32})$
Conversion to `double` precision number α	$\alpha \leftarrow \texttt{double}(i)/2^{32}$
Addition	$R(i) + R(j) = i/2^{32} + j/2^{32} = (i+j)/2^{32} = R(i+j)$
Multiplication	$R(i) \times R(j) = i/2^{32} \times j/2^{32} = (i \times j)/2^{64}$ $= (i \times j/2^{32})/2^{32} = R(i \times j/2^{32})$

Table 1: Proposed Fixed-point Number Representation

The basic idea is to scale down numbers to fit into a fixed-point number representation: $i/2^{32} \times 2^0$ where i is in 32 bits integer. The exponent will not change under addition or multiplication so can be ignored. The addition under is form is simply integer addition. Multiplication becomes: $i/2^{32} \times j/2^{32} = i \times j/2^{64} = (i \times j/2^{32})/2^{32}$. To compute $i \times j/2^{32}$ can be done with 32 bits integer multiply and return the high 32 bits in the 64 bits result. Note that this operation can be done in one instruction on modern CPU instruction set architectures (ISAs) including `x64` and `ARM`. Table 1 summarizes the proposed fixed-point number representation.

Algorithm 6 shows for LU factorization with partial pivoting based on integer arithmetic. The computation inside the loop is mainly 32-bit integer arithmetic. Line 9 requires 64-bit integer division but only once per column. The scale in line 10 will remain in `int32` range because the pivot has larger magnitude then other elements in the column. The update in line 11 is 32-bit integer multiply but we only need the high 32 bits in 64 bits results.

The input integer r determines the number of bits $(32 - r)$ we are actually using while converting A into integer. Because the matrix would grow during the factorization and we do not have any dynamic scaling during the factorization, it might hit the integer range and overflow at some point. To avoid it, we first scale the matrix into $[-2^{-r}, 2^{-r}]$. The higher r is, the more room we will have from the integer range. But less accurate the input matrix would be after converted into `int32`.

2.7.2 Quantized Integer LU Numerical Results

Figure 7 shows the normalized backward error $\|Ax - b\|_\infty / \|A\|_\infty \|x\|_\infty$ vs. input matrix size. The algorithm is implemented in `MATLAB R2018b`. Each element of the matrix is generated from uniform random distribution: `uniform`(-1,1). Each point is the the geometric average over 30 random matrices and error bars indicate the 15% and 85% percentiles. The result from single and double

Algorithm 6 LU factorization with partial pivoting based on integer arithmetic.

Input: n by n matrix A in double precision.
Integer r for the range while normalizing A.
Declare identity matrix P as permutation matrix.
$m \leftarrow \max(A) \times 2^r$; $A \leftarrow A/m$ ▷ Normalize A into $[-2^{-r}, 2^{-r}]$
$A_{int} \leftarrow \text{int32}(A \times 2^{32})$ ▷ Convert A into proposed fixed-point representation.
for $i = 1 \ldots n$ **do** ▷ Main loop over columns
    pivot $\leftarrow (\arg\max |A_{int}[i{:}n, i]|) + i - 1$ ▷ Find the pivot index.
    swap$(A_{int}[i,:], A_{int}[\text{pivot},:])$ ▷ Swap rows.
    swap$(P[i,:], P[\text{pivot},:])$
    $\alpha \leftarrow \text{int64}(2^{32})/A[i,i]$ ▷ Find the scale with integer division.
    $A_{int}[i{:}n, i] \leftarrow \alpha A_{int}[i{:}n, i]$ ▷ Scale the column.
    $A_{int}[i+1{:}n, i+1{:}n] \leftarrow A_{int}[i+1{:}n, i+1{:}n] - A_{int}[i+1{:}n, i] \times A_{int}[i, i+1{:}n]/2^{32}$
    ▷ Integer rank-1 update with a division using integer shift.
end for
$L \leftarrow$ lower triangular part of double$(A)/2^{32}$ with unit diagonal.
$U \leftarrow$ upper triangular part of double$(A)/2^{32}$ including diagonal.
Return: P, L, U as the result of factorization such that $P(A/m) = LU$

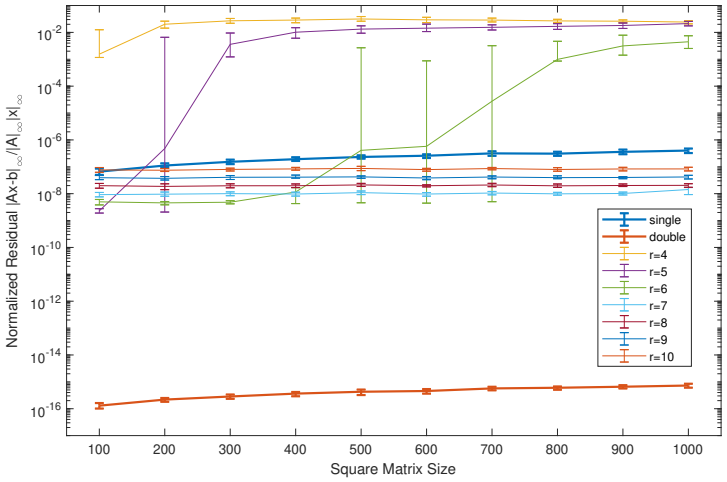

Figure 7: Normalized backward error vs. input matrix size for different number of usable bits r. Result in single and double precision are also shown.

precision LU factorization is also reported in bold lines as reference. For the results which are $> 10^{-6}$, overflow occurred and the algorithm failed. Otherwise there's no numerical error during the factorization. The error is only introduced in the conversions between floating-point and fixed-point format, not during integer factorization. The backward error grows with r since the input is truncated more at the conversion. But still when $r = 10$ it is still using $32 - 10 = 22$ bits and the result is comparable with single precision which is using 23 mantissa bits.

2.7.3 Future Work on Quantized Integer LU

We would like to show case that it is possible to have low precision factorization using integer arithmetic. For the next step we will conduct more detailed error analysis. Extend to shorter integers such as `int16` and `int8`. Also to tackle the overflow problem, we would like to consider dynamically scale the columns during factorization to keep the numbers in range. And the same as per channel quantization in deep learning, assign a different range to each column might also be a feasible approach.

2.8 Symmetric Eigenvalue Problems

In [35] an algorithm is described for determining rigorous error bounds for a simple eigenvalue and its associated eigenvector. The algorithm has the pleasing feature of providing an improved eigenpair as a by-product. This approach assumes that an eigenpair is given. No assumptions are made about how that eigenpair was found, whether through some knowledge of the physical problem, an initial eigenvalue decomposition in a lower precision or a clever guess.

We are interested in improving the accuracy of an eigenvalue- eigenvector pair. Consider the eigenvalue problem $Ax = \lambda x$, where λ and x have been found by some means. Because they were arrived at by some calculation on a computer with finite precision or by some insight into the problem, they are, in general, not the true eigenvalue and eigenvector, but an approximation. We know, however, that there exist μ and $\tilde{y}$ such that $A(x + \tilde{y} = \lambda + \mu) = (\lambda + \mu)(x + \tilde{y})$ is the exact solution to the eigenvalue problem, where μ and $\tilde{y}$ are the corrections to the computed λ and x.

We will normalize x such that $\|x\|_\infty = 1$ and say $x_s = 1$, where the s^{th} component of x is the largest. This can be done because we have one degree of freedom in our choice of the components for x. We will assume that the s^{th} component of x is exact and no correction is needed. This determines the value of $\tilde{y}_s$, which is the correction to x_s. Because x_s is exact, the value of $\tilde{y}_s$ is zero. This also determines the degree of freedom in the corrected vector, $x + \tilde{y}$, through the relationship between x and $\tilde{y}$, namely $(x + \tilde{y}_s) = 1$.

We can rewrite equation $A(x+\tilde{y} = \lambda+\mu) = (\lambda+\mu)(x+\tilde{y})$ as $(A-\lambda I)\tilde{y}-\mu x = \lambda x-Ax+\mu\tilde{y}$. Note that $\lambda x - Ax$ is the residual for the computed eigenvalue and eigenvector. If we look more closely at the product $(A - \lambda I)\tilde{y}$, we discover that because $\tilde{y}_s = 0$, the s^{th} column of $(A - \lambda I)$ does not participate in the product with $\tilde{y}$. In the formulation of $(A - \lambda I)\tilde{y} - \mu x$, we can replace the s component of $\tilde{y}$, which is zero, by the value μ and the s^{th} column of $(A - \lambda I)$ by $-x$ to arrive at $(A - \lambda I)\tilde{y} - \mu x$.

We will define y by $y \equiv \tilde{y} + \mu e_s$, where e_s is the s^{th} column of the identity matrix. So the s^{th} component of the newly defined y has the value μ; i.e., $y_s = \mu$. We will also define the matrix B as the matrix $(A-\lambda I)$ with the s^{th} column replaced by $-x$. Thus we can rewrite $(A-\lambda I)\tilde{y}-\mu x = \lambda x-Ax+\mu\tilde{y}$ as $By = r + y_s\tilde{y}$, where $r = \lambda x - Ax$.

Because the $n + 1$ element of the solution vector is known, we will solve with the truncated form of B, truncated so the n+l row and n+l column are no longer present. This truncation can be done because we know the solution vector has a zero in the $(n + 1)^{th}$ position. The above equation is a nonlinear equation defining the correction y. This system can be solved by the following iterative method for solving,

$By^{(p+1)} = r + y_s^{(p)}\tilde{y}^{(p)}$, where $\tilde{y}^p = y_s^{(p)} - y_s^{(p)}e_s$. This is the approach used in [36].

Algorithm 7 Iterative refinement for symmetric eigenvalue problem.

Input: $A = A^T \in \mathbb{R}^{n\times n}$, $\widehat{X} \in \mathbb{R}^{n\times\ell}$, $1 \le \ell \le n$
Output: $X' \in \mathbb{R}^{n\times\ell}$, $\widetilde{D} = \mathrm{diag}\left(\widetilde{\lambda}_i\right) \in \mathbb{R}^{\ell\times\ell}$, $\widetilde{E} \in \mathbb{R}^{\ell\times\ell}$, $\omega \in \mathbb{R}$
function $\left[X', \widetilde{D}, \widetilde{E}, \omega\right] \leftarrow$ REFSYEV(A, $\widehat{X}$)
    $R \leftarrow \mathbb{I}_n - \widehat{X}^T\widehat{X}$
    $S \leftarrow \widehat{X}^T A\widehat{X}$
    $\widetilde{\lambda}_i \leftarrow s_{ii}/(1 - r_{ii})$ for $i = 1, \ldots, \ell$ ▷ Compute approximate eigenvalues.
    $\widetilde{D} \leftarrow \mathrm{diag}\left(\widetilde{\lambda}_i\right)$
    $\omega \leftarrow 2\left(\left\|S - \widetilde{D}\right\|_2 + \|A\|_2\|R\|_2\right)$
    $e_{ij} \leftarrow \begin{cases} \frac{s_{ij}+\widetilde{\lambda}_j r_{ij}}{\widetilde{\lambda}_j - \widetilde{\lambda}_i} & \text{if } \left|\widetilde{\lambda}_i - \widetilde{\lambda}_j\right| > \omega \\ r_{ij}/2 & \text{otherwise} \end{cases}$ for $1 \le i, j \le \ell$ ▷ Compute the entries of the refinement matrix $\widetilde{E}$.
    $X' \leftarrow \widehat{X} + \widehat{X}\widetilde{E}$ ▷ Update $\widehat{X}$ by $\widehat{X}(\mathbb{I}_n + \widetilde{E})$
end function

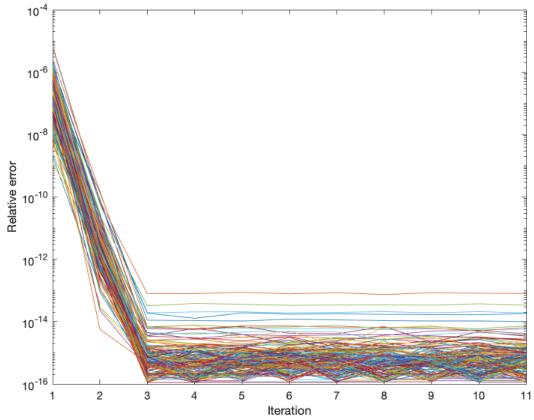

Figure 8: Convergence of eigenvalue refinement from single to double precision for $n = 200$.

Algorithm 7 shows another approach using an iterative refinement procedure for solving a symmetric eigenvalue problem [37]. This methods succeeds also for clustered eigenvalues [38]. Line 4, 5, and 10 represent the compute intensive parts of the algorithm, which amounts to 4 calls to matrix-matrix multiply function `xGEMM`. Line 8 is the matrix norm. The original analysis uses 2-norm but it is suggested to approximate it using the Frobenius norm because it is much easier to compute in practice. Line 9 is an element-wise operation to construct the refinement matrix E. Line 10 is the update of eigenvectors by applying the refinement matrix E. High-precision arithmetic is required for all computations except line 8 for matrix norm. Although the algorithm can work for only a subset of eigenvectors, it is only refining in the corresponding subspace. Hence the refinement process could be limited. In other words, the desired accuracy might be unattainable if only a part of the spectrum is refined in higher precision.

Figure 8 shows the convergence behavior of algorithm 7 on a real symmetric matrix of size $n = 200$ when refining the entire eigen-specturm. Each line represents the convergence of one eigenvalue, and the normalized residual $||Ax - \lambda x||/||A||||x||$ is plotted against subsequent iteration numbers.

Iterative refinement based on linear solve is also possible [36]. Algorithm 8 is the procedure called SICE which, in each iteration, solves a linear system resulting from a rank-1 update in order to refine a single eigen-pair. The rank-1 update is introduced while replacing one column in $A - \lambda I$ to remove one degree of freedom on eigenvector correction and, at the same time, compute a correction for the corresponding eigenvalue. The original formulation [36] solves the system with two series of Givens rotations to make it upper triangular. This process is hard to parallelize on modern architectures. Also, some form of orthogonalization should be considered while using the algorithm to refine more than one eigenvalue.

In many applications, we are satisfied with a subset of the eigenvalue eigenvector pairs. In this case, it can be much more efficient to use an algorithm such as the Multiple Relatively Robust Representations (MRRR) [39] to compute the eigenpairs once the matrix has been reduced to a tri-diagonal form. Though this method by itself is less accurate than its counterparts (Divide and Conquer and QR), [40] show that using a mixed precision approach can be beneficial to improve the accuracy of the solve and the overall time to solution. The mixed precision approach here also shows promise in improving the orthogonality over its single precision and other solver counterparts.

3 Data and communication compression for multiprecision algorithms

A fundamental requirement for accelerating scientific computations through multiprecision use is the ability to efficiently convert data between the different floating point formats employed and to minimize the communications associated with these data movements. Techniques and implementations to accomplish this efficiently have been developed usually ad-hoc, e.g., implemented and tuned for particular algorithms and implementations that use mixed-precision. Thus, although there are some solutions that address particular challenges, there are no standards, often there are no user-level interfaces to lower-level building blocks, and therefore not extracted as independent, supported

Algorithm 8 SICE algorithm for iteratively refining computed eigenvalue.

function $[x, \lambda] \leftarrow \mathsf{SICE}(A, x_0, \lambda_0)$
  $[Q, T] \leftarrow$schur(A) ▷ Schur decomposition to find $A = QTQ^T$ where T is quasi upper triangular.
  $[m, s] \leftarrow max(x_0)$ ▷ Find maximum value and index in the eigenvector.
  $x_0 \leftarrow x_0/m$ ▷ Normalize
  for $i = 1, 2, \ldots$ **do**
    $c \leftarrow -x_{i-1} - (A - \lambda_{i-1}I)[:, s]$ ▷ Column s of $A - \lambda_{i-1}I$
    $d \leftarrow Q^T c$
    $f \leftarrow e_s^T Q$ ▷ Row s of Q
    Solve the rank-1 updated system $Q(T - \lambda_{i-1}I + df^T)Q^T y_i = Ax_{i-1} - \lambda_{i-1}x_{i-1}$
    $\lambda_i \leftarrow \lambda_{i-1} + y_i[s]$ ▷ Eigenvalue correction.
    $x_i \leftarrow x_{i-1} + y_i$ ▷ Eigenvector correction.
    $x_i[s] \leftarrow x_{i-1}[s]$ ▷ Restore x[s].
    if $2 \times y_i[s] > y_{i-1}[s]$ **then**
      Break from for loop.
    end if
  end for
  $x \leftarrow x_i$
  $\lambda \leftarrow \lambda_i$
end function

libraries or components infrastructure that other developers can use. To address this, we have been investigating a number of building blocks that can be extracted and included in numerical libraries for the development of mixed-precision algorithms. The components that are of interest for data and communication compression are discussed in the subsequent subsections.

3.1 Data conversions

Many mixed-precision algorithms need to convert data between different standard IEEE formats. For example, LAPACK supports this type of data conversion as needed for its mixed-precision iterative refinement solvers. Support is provided through auxiliary routines for either general or triangular matrices, following standard LAPACK naming conventions and matrix representations. For example, general matrices can be converted from FP64 to FP32 as follows:

```
zlag2c(M, N, zA, LDA, cA, LDCA, INFO)
dlag2s(M, N, dA, LDA, sA, LDSA, INFO).
```

The first example is for casting double complex to single complex matrix, and the second for double to single real matrix. The other way around (from single to double) is also provided through the `clag2z` and `slag2d` routines.

The interfaces for converting triangular matrices are:

```
zlat2c(UPLO, N, zA, LDA, cA, LDCA, INFO)
dlat2s(UPLO, N, dA, LDA, sA, LDSA, INFO)
```

and the ones for going from single to double are `clat2z` and `slat2z`, respectively.

These routines, following LAPACK's interfaces, are also provided in MAGMA for GPUs. MAGMA also adds conversion from single to half precision (FP32 to FP16) for general matrices:

```
slag2h(M, N, sA, LDA, hA, LDHA, INFO, QUEUE)
```

and the corresponding `hlag2s`. These routines are well optimized for NVIDIA GPUs, and also supported for AMD GPUs (through hipMAGMA). MAGMA also provides the batched equivalent for batches of conversions.

A more specialized for mixed-precision calculations library may have to support a more complete set of data conversion routines, e.g., for arrays, strided arrays, tensors, etc., and more combinations

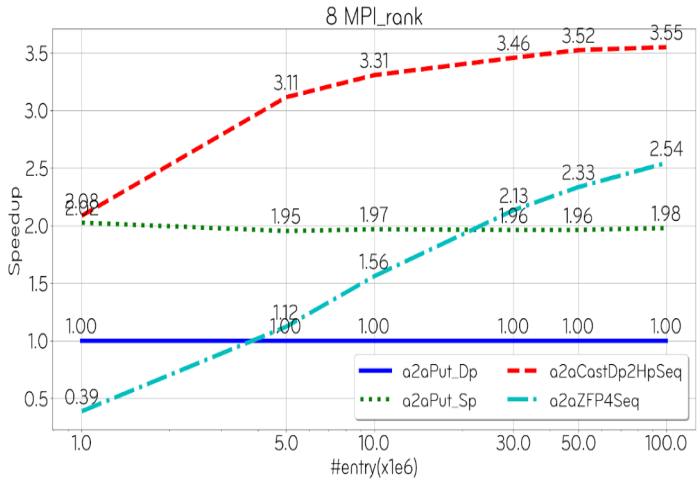

Figure 9: Speedup of All2All by 4× compression using cast (red) vs. 2× compression using cast (dotted green) vs. 4× compression using ZFP on Nvidia V100 GPUs.

of formats, including user/application defined. For example, some mixed-precision FFT algorithms (see Section 3.3) use dynamic "splitting" of a high precision array (e.g., FP32) into two lower-precision (e.g., FP16) arrays. See also Section 7 for further discussion and extensions on formats.

3.2 Data compression

Data compression for reducing communications is another component needed for the development of mixed-precision algorithms. The simplest form that we consider is the casting, as discussed in Section 3.1. This is an example of a lossy compression. Casting from FP64 to FP32 for example, leads directly to a loss of about 8 decimal digits of accuracy, but reduces the data size by a factor of two. Casting has been used to accelerate the FP64 solvers in MAGMA up to 4× using the mixed-precision iterative refinement techniques [24, 4, 3] and we use it as benchmark to evaluate the potential of using other compression algorithms.

We evaluated for example the possibility to use ZFP compression. ZFP provides lossy compression algorithms, where the compression mode can be specified by the user as either fixed rate, fixed accuracy, or fixed precision [41]. Analysis for the round-off error introduced by ZFP in compressing floating-point data is presented in [42]. The values in this experiment are taken random numbers and the compression specified is 4×. Note that compared to casting, the compression rate is as casting to FP16, but the accuracy is comparable to casting to FP32. These results make it feasible to use tools like ZFP to accelerate memory-bound codes, e.g., like FFT (see Section 3.3), up to 4× while loosing about 8 decimal digits of accuracy.

3.2.1 Mixed-precision MPI

Of interest is MPI extension that fuses subsequent data conversions with the MPI communications. The conversion must be user specified and includes casting or other data compression or conversion mechanisms, where a single MPI call will convert the input data as specified, send the converted data, and the corresponding MPI call will receive and convert the result again, as specified by the user. Our MPI collaborators have developed preliminary mixed-precision MPI for All2All and P2P communication using casting. The results show that asymptotically, for large enough data, the MPI communications can be accelerated proportional to the data compression, i.e., the conversion is negligible. The implementations are for CPUs, as well as GPUs using GPU-direct communications.

Our preliminary results can also use ZFP to compress the data. Figure 9 illustrates an acceleration result for All2All in FP64 (marked as base, i.e., the acceleration of 1 line).

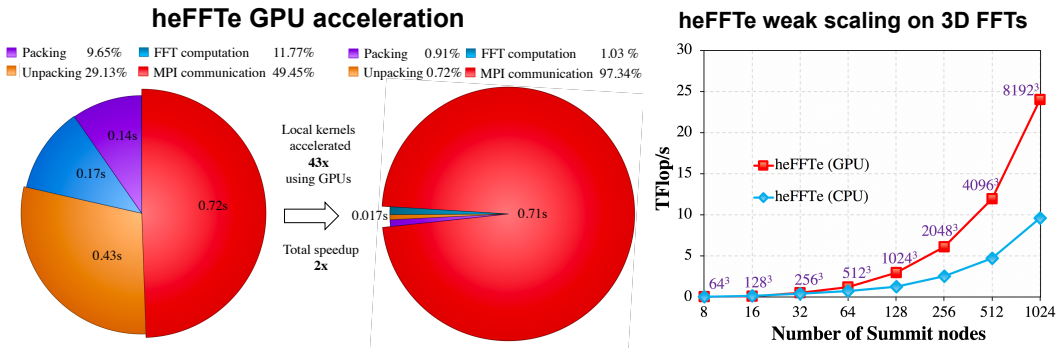

Figure 10: **Left**: heFFTe acceleration using GPUs vs. CPUs of 1024^3 FFT on 4 Summit nodes. Note than nodal computations are accelerated 43× using GPUs. **Right**: heFFTe weak scalability with sizes indicated on the graph on up to 1024 nodes (6 V100 GPUs; double complex arithmetic; starting and ending with bricks; performance assumes $5N^3log_2N^3$ flops).

Note that here we use ZFP and manage to accelerate the MPI communication more than 2× while loosing only about 8 decimal digits of accuracy, i.e., we achieve an acceleration that outperforms the corresponding version that uses cast to FP32 (while having the same accuracy). The data sending itself is accelerated close to 4× as it is compressed 4×, but the overall acceleration drops when adding the cost for the data compression and decompression. This means that acceleration theoretically still can go up to about 4× in implementations where the GPU work on compression and decompression is pipelined and overlapped with the communication.

3.3 Approximate Fast Fourier Transforms

One application of the mixed-precision technologies described in this Section is the acceleration of multidimentional FFTs through mixed-precision algorithms. We found that more than dozen of ECP applications use FFTs in their codes, e.g., including LAMMPS, HACC, ExaAM, and applications from the Copa co-design center [43]. ECP applications that require FFT-based solvers suffer from the lack of fast and scalable 3D FFT routines for distributed-heterogeneous parallel systems as the ones projected for the upcoming exascale computing systems, and some of the applications have indicated interest in exploring the use of approximate FFTs that trade some loss of accuracy for increase in performance.

To address the above needs for high-performance scalable FFTs on large-scale GPU systems, we developed and released the *Highly Efficient FFTs for Exascale* (**heFFTe**) library [44, 45, 46]. heFFTe v0.2 features very good weak, as well as strong, scalability and performance that is close to 90% of the roofline peak (see Figure 10). However, after accelerating the local/nodal computations of about 43× using GPUs (vs CPUs), the main bottleneck becomes the MPI communications. Currently, on typical FFT problems the GPUs can be used only about 1% of the time, i.e., the GPUs are free to be used for other computations 99% of the time, while the rest 99% of the time is spent in MPI communications. Thus, any acceleration in the MPI communications would translate into the same acceleration of the overall FFT computation.

3.3.1 Approximate FFTs with accuracy-for-speed tradeoff

One idea to accelerate FFTs using mixed-precision is through a tradeoff of accuracy for speed. Here we reduce the communication volume by compressing the data, e.g., by casting or compression as outlined in Section 3.2.

Multidimensional FFTs use All2All-type of MPI communications, where first data is packed (locally on each GPU, using and benefiting from GPUs' high bandwidth), next is the MPI communication, and finally data is unpacked locally on each GPU [43, 44]. As packing and unpacking are memory bound and involve going through the data once, the addition of casting or other type of compression can be fused with the packing/unpacking and thus significantly remove its overhead. This idea is explored through the use of mixed-precision MPI that fuses all these operations (as in Section 3.2.1). Quantification of the speedups obtained vs. the reduction in accuracy is as illustrated in Figure 9.

Current work is on overlapping the use of the GPUs for compression/decompression with the MPI communications through pipelining. This is possible as FFTs are memory bound and GPUs can be free to do other computations up to 99% of the time, e.g., as benchmarked for the Summit hardware configuration.

3.3.2 Accuracy control

Accuracy requirements are mainly application dependent and can be controlled through specifying a desired compression rate. The accuracy of the resulting mixed-precision FFTs will be higher than the corresponding FFT using "low" precision for both storage and computation, while retaining the same performance. Moreover, going to the extreme, it is possible to tune the mixed-precision FFTs to be of the same accuracy, e.g., as FP64 FFTs. For example, this can be done through evaluating the loss of decimal digits in local computations due to round-off and use an appropriate data compression rate, so that only the valid digits get to be communicated.

3.3.3 Dynamic splitting

Another idea in accelerating multidimensional FFTs is that instead of compression, the high-precision FFT data can be dynamically split into two scaled data sets of lower-precision data, apply the FFT transformations on the two sets in parallel and combine the result at the end. This was explored in the context of FP32 data that gets split into two FT16 sets in order to apply fast GPU Tensor Cores computations on the FP16 data [47, 48], e.g., by going to a small radix, e.g., 4, where the FFT matrix can be represented exactly (even in FP16) and benefit from Tensor Cores accelerated HGEMMs. An extension of this idea can take into account that the FFT matrix is never assembled, except for a small radix matrix in order to apply it on the data as GEMM. Thus, without much computational overhead (because of the memory-bound nature of FFT) one can use higher precision Fourier matrix and computations (than the precision that stores the data) to accelerate the entire FFT computation.

4 Multiprecision Sparse Factorizations

Direct methods for sparse systems can also benefit from using lower precision formats. The idea is to perform the expensive calculations in lower precision, taking advantage of the faster speed often provided by hardware. Then, some cheaper "fixup" algorithm is employed to recover the accuracy at a higher precision. Sparse factorizations, such as sparse LU and QR factorizations, are most often used to construct sparse direct solvers. Two related but orthogonal research directions can be taken here. The first is about the factorizations themselves, and the second is in the context of direct solvers.

4.1 Multiprecision sparse LU and QR

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

The benefit of using lower precision is two fold: for step 2), we can use very fast lower precision vendor-provided GEMM functions, e.g. those utilizing NVIDIA's Tensor Cores. For the gather/scatter in steps 1) and 3), the amount of data movement would be reduced.

For the dense case the main benefit comes from accelerated GEMM speed. But in the sparse case, GEMM is only one part of the three steps above. Furthermore, the dimensions of the GEMM kernel calls is generally smaller and of non-uniform size throughout factorization. Therefore, the speed gain from GEMM alone is limited. We will need to design new schemes to enhance overlap of GEMM computation with gather/scatter operations.

4.2 Multiprecision sparse direct solvers

For the dense case, in Section 2.2 we revisited the mixed precision iterative refinement (IR) algorithms with adaptive precision adjustment depending on convergence history. The algorithms can deliver high accuracy to the solution even when the expensive LU and QR factorizations are done in lower precision. We recall the IR algorithm using three precisions in Algorithm 9 [49, 50]. This algorithm is already available in **x**GERFSX functions in LAPACK.

The following three precisions are used:

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

Algorithm 9 Three-precisions Iterative Refinement for Direct Solvers

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

With the above setup and adaptive adjustment of ε_x and ε_r, the algorithm converges with small normwise error and error bound if the normwise condition number of A does not exceed $1/(\gamma(n)\varepsilon_w)$. Similarly, the algorithm converges with small componentwise error and error bound if the componentwise condition number of A does not exceed $1/(\gamma(n)\varepsilon_w)$. Moreover, this IR procedure can return to the user the reliable error bounds both normwise and componentwise. The error analysis in [49] should all carry through to the sparse cases.

The following are example configurations of the precisions:

- $\varepsilon_w = 2^{-53}$ (IEEE-754 double precision), $\varepsilon_x = 2^{-53}$, $\varepsilon_r = 2^{-106}$ (double-double)
- $\varepsilon_w = 2^{-16}$ (B-float), $\varepsilon_x = 2^{-24}$, $\varepsilon_r = 2^{-53}$

Our plan is first to extend the above algorithm to the sparse direct solvers SuperLU and STRUMPACK. While doing so, we will address the following open questions:

- When ε_w is bfloat16, the error analysis and error bounds may need be revisited.
- The relative cost of sparse LU/QR (lines 1 and 5) and sparse matvec (line 4) is different from the dense counter part. For a typical 3D PDE discretized problem, the respective costs are $O(n^2)$ and $O(n^{4/3})$. Thus, the ratio between "expensive" and "cheap" is smaller than the dense case. We need to be more mindful with the higher precision calculations.

5 Multiprecision efforts in Krylov solver technology

The scope of our review includes both Lanczos-based (short-term recurrence) and Arnoldi-based (long-term recurrence) methods and the associated methods for solving linear systems of equations $Ax = b$. In the context of long-term recurrence methods, we consider the Arnoldi-QR algorithm with the modified Gram-Schmidt implementation of the Generalized Minimum Residual (GMRES) Krylov subspace method for iteratively solving linear systems of equations. The emphasis here is to

examine the approaches employed to date that incorporate mixed-precision floating point arithmetic to speed-up computations, and yet retain some or all of the numerical properties of the original algorithms in full double precision arithmetic (i.e. representation error and loss of orthogonality).

5.1 Lanczos-CG

5.1.1 Theoretical Results

We first summarize very briefly the most well-known results on the finite precision behavior of Lanczos and CG methods, and discuss how such results could potentially be extended to the mixed precision case and existing progress in this area. We note that there is a huge literature on the finite precision behavior of Lanczos-based methods which we cannot hope to fully describe here. For a more thorough account and historical references, we point the reader to the manuscript of Meurant and Strakoš [51].

Fundamental relations dealing with the loss of orthogonality and other important quantities in finite precision Lanczos have been derived by Chris Paige [52]. These results were subsequently used by Anne Greenbaum to prove backward stability-like results for the CG method [53]; namely, Greenbaum showed that CG in finite precision can be seen as exact CG run on a larger linear system, in which the coefficient matrix has eigenvalues in tight clusters around the eigenvalues of the original matrix (where the diameter of these clusters depends on properties of the matrix and the machine precision). Greenbaum also proved fundamental results on the maximum attainable accuracy in finite precision in what she calls "recursively computed residual methods", which includes CG, BICG, BICGSTAB, and other Lanczos-based methods [54]. The results of Paige and Greenbaum have also been extended to s-step Lanczos/CG variants in [55], where it is shown that s-step Lanczos in finite precision behaves like classical Lanczos run in a lower "effective" precision (where this "effective" precision depends on the conditioning of the polynomials used to generate the s-step bases). We believe that these existing results can be extended to the mixed precision case; in Paige's analysis [52], he first defines an ε_0 quantity that is used for errors in inner products and an ε_1 quantity that comes from errors in matrix-vector products, but then these quantities are combined in later theorems in order to simplify the analysis. It is possible to expand upon his analysis and keep these two quantities separate; such results could also then be interpreted in the framework of Greenbaum [53].

Existing results in the area of mixed precision Lanczos-based methods are contained within the work on "inexact Krylov subspace methods", which also applies to Arnoldi-based methods; see, e.g., the manuscripts of Simoncini and Szyld [56], and van den Eshof and Sleijpen [57]. Within such frameworks, it is assumed that the matrix-vector products are computed with some bounded perturbation (which can change in each iteration) and all other computation is exact. These methods were motivated by improving performance in applications where the matrix-vector products dominate the cost of the computation, e.g., when the matrix is dense or the application of A involves solving a linear system. Many theoretical results on "inexact Krylov subspace methods", mostly focused on the maximum attainable accuracy, have been proved in the literature. A surprising result is that the inexactness in the matrix-vector products can be permitted to grow in norm as the iterations progress at a rate proportional to the inverse of the residual norm without affecting the maximum attainable accuracy. However, a crucial practical question is whether inexactness will affect the convergence behavior *before* the attainable accuracy is reached; this is entirely possible in the case of short-term recurrence methods such as CG and has not been well-studied theoretically.

5.1.2 Practical Applications

We briefly mention works which make use of mixed precision Krylov subspace methods in practical applications, focusing on performance rather than on theoretical results.

One instance of this is in the work of Clark et al. [58], which uses mixed precision CG and BICGSTAB methods implementing the "reliable update" strategy of Sleijpen and van der Vorst [59] within a Lattice QCD application run on GPUs. The idea behind the "reliable update" strategy is that the true residual is computed and used to replace the recursively updated residual in select iterations, thus improving the attainable accuracy; this is done in conjunction with batched updates to the solution vector. By using higher (double) precision only in the true residual computations and group updates (and single or half precision for the rest of the computation), the authors claim they are able to achieve full double precision accuracy. This deserves further theoretical study, which we

believe can be achieved by extending the results in [59] and the related work of van der Vorst and Ye [60] to the mixed precision setting.

5.2 Arnoldi-QR MGS-GMRES

For MGS-GMRES the mixed precision work by Gratton et. al. [61] is the most recent and appropriate - and in particular the loss-of-orthogonality relations due to Björck [62] and Paige [52], later refined by Paige, Rozložnik and Strakoš [63], are employed in order to provide tolerances for mixed single–double computations. MGS-GMRES convergence stalls (the norm-wise relative backward error approaches ε) when linear independence of the Krylov vectors is lost, and this is signaled by Paige's S matrix norm $\|S\|_2 = 1$. The S matrix [64] is derived from the lower triangular T matrix appearing in the rounding error analyses by Giraud et. al. [65].

To summarize, the Gratton et. al. [61] paper postulates starting from the Arnoldi-QR algorithm using the modified Gram-Schmidt algorithm and employing exact arithmetic in the MGS-GMRES iterative solver. The Arnoldi-QR algorithm applied to a non-symmetric matrix A produces the matrix factorization, with loss of orthogonality F_k

$$A V_k = V_{k+1} H_k, \quad V_{k+1}^T V_{k+1} = I + F_k \tag{4}$$

They next introduce inexact (e.g. single precision) inner products - this directly relates to the loss-of-orthogonality relations for the $A = QR$ factorization produced by MGS. The resulting loss of orthogonality, as measured by $\|I - Q^T Q\|_2$, grows as $\mathcal{O}(\varepsilon)\kappa(A)$ as was derived by Björck [62] and $\mathcal{O}(\varepsilon)\kappa([\, r_0, AV_k \,])$ for Arnoldi-QR - which is described by Paige, Rozložník and Strakoš [66, 63] and related work. The inexact inner products are given by

$$h_{ij} = v_i^T w_j + \eta_{ij} \tag{5}$$

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

$$[\; r_0, \quad AV_k \;] = V_{k+1} [\; \beta\, e_1, \quad H_k \;], \tag{6}$$

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

$$F_k = \bar{U}_k + \bar{U}_k^T, \quad U_k = \begin{bmatrix} v_1^T v_2 & \cdots & v_1^T v_{k+1} \\ & \ddots & \\ & & v_k^T v_{k+1} \end{bmatrix} \tag{7}$$

Define the matrices,

$$N_k = \begin{bmatrix} \eta_{11} & \cdots & \eta_{1k} \\ & \ddots & \\ & & \eta_{kk} \end{bmatrix}, \quad R_k = \begin{bmatrix} h_{21} & \cdots & h_{2k} \\ & \ddots & \\ & & h_{k+1,k} \end{bmatrix} \tag{8}$$

The loss of orthogonality relation derived by Björck [62], for the $A = QR$ factorization via the modified Gram-Schmidt algorithm can be applied to the Arnoldi-QR algorithm to obtain

$$N_k = -[\; 0, \quad U_k \;]\, H_k = -U_k R_k \tag{9}$$

The complete loss of orthogonality (linear independence) of the Krylov vectors in MGS-GMRES signals the minimum error is achieved and GMRES then stalls or really can go no further than when the norm-wise relative backward error reaches $\mathcal{O}(\varepsilon)$. Gratton et al. [61] show how to maintain sufficient orthogonality in order to achieve a desired relative residual error level - by switching the inner products from double to single at certain tolerance levels and combine this with inexact matrix-vector products as in van den Eshof and Sleijpen [57] and Simoncini and Szyld [56].

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

5.3 Alterative Approaches

Although somewhat outside the scope of this review, we can demonstrate that it is possible to modify the Gratton et al. [61] analysis based on the inverse compact WY form of the MGS algorithm, introduced by Świrydowicz et al. [67]. Rather than treat all of the inner products in the MGS-GMRES algorithm equally, consider the strictly upper triangular matrix $U = L^T$ from the loss of orthogonality relations. We introduce single precision $L_{:,j-1} = Q_{j-1}^T q_{j-1}$ and double precision triangular solve $r = (I + L_{j-1})^{-1} Q_{j-1}^T a$ to update R - as this would directly employ the forward error analysis of Higham [68]. The former affects the loss of orthogonality, whereas the latter affects the representation error for QR - but then also for Arnoldi-QR. This could allow more (or most) of the inner products to be computed in single precision.

Evidence for maintaining orthogonality is provided in Figure 11, with $\|I - Q^T Q\|$ plotted for $A = QR$ using the inner products in standard MGS (blue) in double precision versus the inverse compact WY MGS (red) with $Q_{j-1}^T q_{j-1}$ in single precision (simulated in MATLAB) - and we observe at least the same or slightly higher error levels. The x-axis is log condition number for randomly generated matrices. The lower triangular solve is computed in double precision.

Barlow [69] contains similar if not the same algorithm formulations in block form. His work is related to Björck's 1994 paper [70, Section 7] which derives the triangular matrix T using a recursive form for MGS, and which is referred to as a "compact WY" representation in the literature. While Björck used a lower triangular matrix for the compact WY form of MGS, Malard and Paige [71] derived the upper triangular form, also employed by Barlow, which reverses the order of elementary projectors. The latter is unstable in that a backward recurrence leads to $\mathcal{O}(\varepsilon)\kappa^2(A)$ loss of orthogonality. An interesting observation from Julien Langou is that the upper triangular form is less stable than the lower triangular (even though the backward-forward algorithm results in re-orthogonalization; see the algorithm in Leon, Björck, Gander [72]).

Barlow [69] employs the Householder compact WY representation of reflectors and also refers to the work of Chiara Puglisi [73] – discussed in Joffrain et al. [74] – and this is referred to as the "inverse compact WY" representation of Householder; this originally comes from Walker's work on Householder GMRES [75]. Barlow then extends this approach to the block compact WY form of MGS; see also the technical report by Sun [76]. The contribution by Świrydowicz et al. [67] was to note that there exists an inverse compact WY representation for MGS - having the projector

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

and to "lag" the norm $\|q_{j-1}\|_2$ so that these can be computed in one global reduction. Barlow [69] makes this connection for blocks (and in effect this is given in his equation (3.10)) and references Puglisi [73].

Björck and Paige [77] made the link between Householder and MGS based on the observation made by Sheffield. Paige defines this to be augmentation and Gratton et al. [61] also references this work. Paige has also recently extended these augmentation ideas to Lanczos. The T matrix appears in Paige's work with Wülling [78] and then later in [64] to derive the loss of orthogonality matrix $S = (I + L_{j-1}^T)^{-1} L_{j-1}^T$. This also appears in the work of Giraud, Gratton and Langou [65]; Langou also worked with Barlow and Smoktunowicz [79] on the Pythagorean trick to reduce cancellation error in the computation of vector norms and a Cholesky-like form of classical Gram-Schmidt (CGS).

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

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

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

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

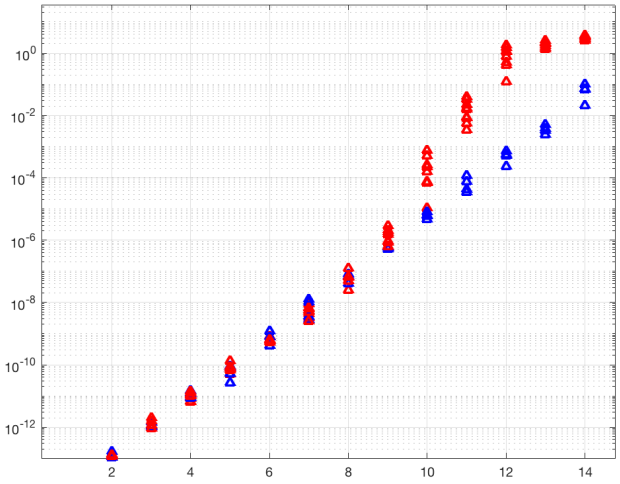

Figure 11: Loss of Orthogonality for Mixed Single-Double MGS Algorithm

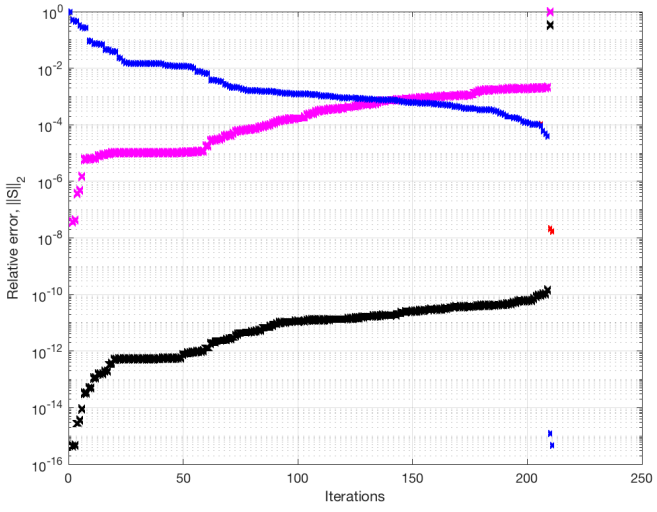

Figure 12: GMRES residuals and loss of orthogonality $\|S\|_2$ for impcol_e matrix

6 Multiprecision Preconditioners

In the iterative solution process of large sparse systems, preconditioners are an important building block facilitating satisfactory convergence. The concept of preconditioning is to turn an ill-conditioned liner system $Ax = b$ into a (left-) preconditioned system $MAx = Mb$ ($AMy = b$, $x = My$ for right-preconditioning), which allows for faster convergence of the iterative solver. The convergence characteristics typically depend on the conditioning of the target system. For an ill-conditioned A, the preconditioner is also required to be ill-conditioned. Otherwise, the preconditioner can not be expected to improve the conditioning of the problem or the convergence of the iterative solver. In that respect, the preconditioner basically tries to approximate the inverse of the system matrix. Obviously, if the preconditioner is the exact inverse, the solution is readily available. However, computing the exact inverse is prohibitively expensive, and in most cases, the preconditioner is just a rough approximation of the system matrix inverse. As a consequence, it is natural to question the need for using high precision for a preconditioner that is inherently carrying only limited accuracy. Indeed, choosing a lower precision format for the preconditioner is a valid strategy as long as the accuracy loss induced by using a lower precision format neither impacts the preconditioner accuracy nor its regularity. For example, Trilinos allows the use of low precision preconditioners inside high precision iterative solvers, see Section 9, and the hypre team works on multigrid methods running the first cycles in lower precision. However, the use of lower precision in the preconditioner application results in different rounding effects than when using high precision. Specifically, the rounding effects make the preconditioner non-constant as the rounding effects are not only larger than in high precision, but also depend on the input data [80]. As a result, low precision preconditioners can only be used to accelerate an iterative method that can handle non-constant preconditioners, i.e., can converge even if the preconditioner changes in-between iterations. For the Krylov subspace solvers generating search directions orthogonal to the previous search direction, a changing preconditioner requires an additional orthogonalization of the preconditioned search direction against the previous preconditioned search direction. The flexible Krylov solvers (e.g. FGMRES, FCG) contain this additional orthogonalization and are therefore slightly more expensive. At the same time, they do allow for using low precision preconditioners, which can compensate for the additinal cost.

An alternative workaround is to decouple the memory precision from the arithmetic precision, see Section 7, and only store the preconditioner in low precision but apply it in high precision [80]. Running all arithmetic in high precision keeps the preconditioner constant, and removes the need for the additional orthogonalization of the preconditioned search direction. On the other hand, decoupling memory precision from arithmetic precision requires to convert in-between the formats on-the-fly when reading data from main memory. Fortunately, most iterative solvers and preconditioners are memory bound, and the conversion can be hidden behind the memory transfers. A production-ready implementation of an adaptive precision block-Jacobi preconditioner decoupling memory precision from arithmetic precision is available in the Ginkgo library, see Section 9.

7 Multiprecision efforts decoupling the arithmetic format from the memory format

Across the complete hardware technology foodchain, we are witnessing a widening gap between the compute power in terms of float point operations per second on the one side and the communication power in terms of memory bandwidth. In modern processor technology, retrieving values from main memory takes several orders of magnitude longer than performing arithmetic operations, and communicating between distinct nodes of a cluster is again orders of magnitude slower than main memory access. In consequence more and more algorithms hit the memory wall – and already today, virtually all applications inside the ECP ecosystem are memory bound on modern hardware architectures. With no disruptive hardware changes on the horizon, we are facing a situation where all applications suffer from the slow communication to main memory or in-between nodes.

A promising – and maybe the only promising – strategy to overcome this problem is to utilize the bandwidth capacity more carefully, reduce the communication volume and the number of communication points, and whenever possible, trade communication against computations. Specifically, the idea is to radically decouple the memory precision from the arithmetic precision, employ high

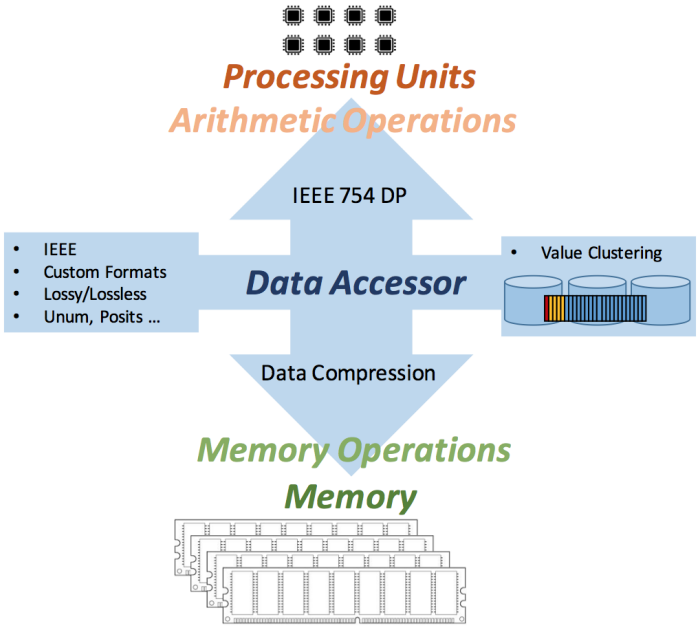

Figure 13: Accessor separating the memory format from the arithmetic format and realizing on-the-fly data conversion in each memory access.

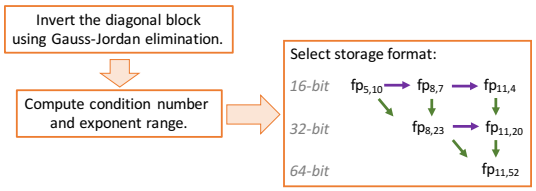

Figure 14: Storage format optimization for block-Jacobi: Starting from the most compact storage (left top), the format is extended in exponent bits to fit the data range (rightwards) and to preserve regularity (downwards) until both is satisfied.

precision only in the computations, and lower the precision as much as possible when accessing data in main memory or communicating with remote processors [81]. An important aspect in this context is the design of a "*memory accessor*" that converts data on the fly between the IEEE high precision arithmetic format and the memory/communication format, see Figure 13. Also, the memory/communication format does not necessarily have to be part of the IEEE standard, but can also be an arbitrary composition of sign, exponent, and significand bits [82], or even nonstandard formats like L. Gustafson's Unum format [83]. Obviously, there is a close link to the idea to complement the format separation with compression techniques, like proposed in Section 3.

A proof-of-concept for the idea of decoupling arithmetic precision from memory precision is the adaptive precision block-Jacobi preconditioner [80] available in the Ginkgo sparse linear algebra library. The idea here is to compute a block-Jacobi preconditioner in high precision, but then store the distinct inverted diagonal blocks in the lowest floating point precision format that avoids overflow and still preserves the regularity of the preconditioner, see Figure fig. 14.

This storage format is chosen for each diagonal block individually, respectively reflecting the characteristics. Figure 15 (top) visualizes the distribution of formats when storing the inverted diagonal blocks of size 24 for symmetric positive definite matrices of the Suite Sparse Matrix Collection. Obviously, converting to a lower format generally reduces the accuracy of the linear operator, but as block-Jacobi preconditioners ignore all off-(block)diagonal entries, they are typically only a rough approximation of the matrix inverse and therewith by design only have very limited accuracy. Experimental results reveal that the use of a lower precision format for storing the inverted diagonal

Figure 15: Top: Distribution of floating point formats among the distinct blocks when preserving 1 digit accuracy of each inverted diagonal block. Each column represents one symmetric positive definite matrices of the Suite Sparse Matrix Collection. Bottom: Impact on the top-level CG solver solving the system-induced linear problem. For most systems, the convergence rate is unaffected by the use of a lower storage precision format, all preconditioner applications are faster, resulting in an average 20% runtime reduction.

blocks has in most cases only negligible effects on the preconditioner effectiveness and the outer solver convergence. At the same time, storing the inverted diagonal blocks in lower precision reduces the memory access volume in every preconditioner application, therewith accelerating the bandwidth bound iterative solution process, see Figure 15. For the adaptive precision block-Jacobi preconditioner, is important that the accessor converts the inverted diagonal blocks back to the IEEE standard precision not only for performance reasons – leveraging the highly-optimized IEEE floating point arithmetic of the processors – but also for numeric reasons: Using working precision in the arithmetic operations of the precnditioner application preserves the preconditioner as a constant operator, applying a preconditioner in lower precision would result in a non-constant preconditioner and require the use of a (more expensive) flexible iterative solver [80].

7.1 Using different precision formats in Multigrid methods

Multigrid methods are highly effective iterative methods. There are basically two types of multigrid methods: geometric multigrid methods (GMG) and algebraic multigrid methods (AMG). GMG requires actual grids on each level to generate its components, whereas AMG can be considered more like a black box method, in that it can be given a matrix and right hand side and will generate the components for each level automatically using sensible heuristics. These methods are an interesting target for multiprecision treatment due to their different components which affect the overall algorithm in different ways. GMG and AMG components combine smoothers, coarser grid, restriction and prolongation operators on each level. In addition, it is of interest to investigate changes in precision on different levels. Finally, GMG and AMG can be used as preconditioners to other solvers, i.e. there is potential to use lower precision across the whole preconditioner. Historically, most work focused on the use of a lower precision GMG or AMG method as a preconditioner to a double precision solver. Additionally, there are attempts to apply ZFP [41] within MG or establish an error analysis framework for AMG.

Ljungkvist and Kronbichler[84, 85] successfully use mixed precision to solve the Laplace problem for different orders with a matrix-free geometric multigrid approach. Their solver infrastructure allows for using mixed-precision arithmetic that performs the multigrid V-cycle in single precision with an outer correction in double precision, increasing throughput by up to 83 percent.

Similarly, Glimberg et al [86] use a single precision multigrid to precondition a double precision defect correction scheme and solve the Laplace problem within a nonlinear water wave application on a GPU architecture. They achieve a speedup of up to 1.6 of the mixed precision version over the double precision version, a speedup of 1.9 for a purely single precision version.

Yamagishi and Matsumura [87] also apply a single precision multigrid to a double precision conjugate gradient solver to the Poisson/Helmholtz problem within their non-hydrostatic ocean model. They report a speedup up to 2 for a single precision Matvec over a double precision one and improved overall times using this approach, however they compare the full application run only to their CPU version.

There are various publications that pursue the same strategy of using a single precision AMG preconditioner to a double precision solver.

Emans and van de Meer [88]perform a careful analysis of the individual kernels of preconditioned Krylov solvers on multi-core CPUs, including sparse matrix-vector multiplications (SpMV) which make up a large portion of AMG. They also consider the effect of communication, where lower precision leads to smaller size messages, but latencies are still an issue, particularly on the coarsest levels of AMG. They find that the use of mixed precision for the preconditioner barely affects convergence and therefore speedups for the kernels, which were between 1.1 and 1.5, can potentially carry over to the whole solver and lead to improvements of runtimes within CFD applications.

Sumiyoshi et al [89] investigate AMG performance on a heterogeneous computer architecture with both CPUs and GPUs for isotropic and anisotropic Poisson problems. They consider smoothed aggregation AMG as a stand-alone solver. They carefully analyze different portions of the algorithm on five different architectures, including one multi-core CPU cluster. They report speedups between 1.2 and 1.6 on the GPU-CPU architectures for the mixed-precision implementation over the double precision version. These speedups are related to SpMV performance (between 1.6 and 1.8) on these architectures. However, the mixed-precision version was slightly slower on the CPU-only architecture, which achieved barely any improvement for the SpMV operations.

Richter et al [90] examine the performance of a single precision AMG solver (ML and PETSc) applied to a double precision PCG solver. They apply the method for an electrostatic simulation of the high voltage isolator on a GPU/CPU computer architecture. Their mixed precision version takes about 84 percent of the time of the double precision version.

An approach described in a presentation by Kate Clark [91] takes the use of mixed precision even further to involve half precision. Clark and collaborators achieved good results using a double precision defect correction approach with a single precision Krylov solver and a half precision AMG preconditioner.

Another interesting related study by Fox and Kolasinski [92] examines the use of ZFP, a lossy compression algorithm, within multigrid. Due to the local structure of ZFP, ZFP can easily be integrated into numerical simulations without changing the underlying algorithms. However, since ZFP is a lossy algorithm, it will introduce some error, thus, it is important to understand if the error caused by ZFP overwhelms other traditional sources of error, such as discretization error. The study shows that for MG on a Poisson problem, applying ZFP to the approximation vector, can significantly decrease memory use and thus is expected to decrease runtimes, while the generated errors stay below discretization error. Since a hardware version of ZFP is not available yet, no actual runs were possible, however the results show good potential to use GMG and/or AMG as a preconditioner.

Currently, Tamstorf et al [93] appear to be the only ones who have investigated the theory of multi-precision multigrid methods. Their original intent was to improve the appearance of the movement of cloth within Disney movies, which requires higher than FP64 accuracy. However, their theory applies equally to decreased precision. They have created a theoretical framework with rigorous proofs for a mixed-precision version of multigrid for solving the algebraic equations that arise from discretizing linear elliptic partial differential equations (PDEs). The arising matrices being sparse and symmetric positive definite enable the use of the so-called energy or A norm to establish convergence and error estimates. Bounds on the convergence behavior of multigrid are developed and analyzed as a function of the matrix condition number. Both theoretical and numerical results confirm that convergence to the level of discretization accuracy can be achieved with mixed-precision versions of V-cycles and full multigrid. This framework is inspired by the results of Carson and Higham [15] but ultimately provides tighter bounds for many PDEs. Tamstorf et al [94] further extend their theoretical framework to include the quantization error. They use the bounds to guide the choice of precision level in their progressive-precision multigrid scheme by balancing quantization, algebraic and descretization errors. They show that while iterative refinement is susceptible to quantization errors during the residual and update computation, the V-cycle used to compute the correction in each iteration is much more resilient, and continues to work if the system matrices in the hierarchy become indefinite due to quantization.

8 Low precision and multiprecision technology for Machine Learning

Modern high-performance computing (HPC) hardware continues to experience an ongoing shift towards supporting a variety reduced-precision formats for representing floating-point numbers in order to offer a much increased performance rate. However, portability is often of little concern as the hardware tends to serve only a specific set of workloads that are of special interest to the particular vendor. The examples include Intel's Cascade Lake Vector Neural Network Instructions (VNNI) and the recently announced Xe platform for graphics cards, AMD's Radeon Instinct cards (MI5, MI8, MI25, MI55, MI60) and NVIDIA's compute cards from the Pascal, Volta, and Turing series. Finally, ARM included 16-bit floating point (FP16) in its NEON vector unit specification VFP 8.2-A. These accelerators follow two types of specifications for 16-bit floating-point format: IEEE-compliant FLOAT16 and extended-range BFLOAT16.

At the same time, a new breed of accelerators take the use of reduced precision to a new level as they target new machine learning workloads. This new hardware includes Cloud AI 100 by Qualcomm, Dot-Product Engine by HPE, Eyeriss by MIT [95], TPU by Google [96], MAERI by Georgia Institute of Technology [97] Deep Learning Boost by Intel, CS-1 by Cerebras, and Zion by Facebook.

In general, the machine learning community has been more aggressive in evaluating multiple precision to the extent that even a 1-bit Stochastic Gradient Descent has been considered [98]. The

typical use case in machine learning is to use the training with 32-bit arithmetic and use different precision for the inference task. The quantization for the inference is supported in popular frameworks like TensorFlow [99] and pyTorch [100]. Quantization is the approach to store the tensors and compute on them using bitwidths lower than floating point bitwidths. Even in machine learning frameworks, the support for quantizations is limited to just the key functionality needed for a convolutional neural networks or recurrent neural networks with some limited hardware support. For example, pyTorch and TensorFlow supports 8-bit quantization for activation and weights. This allows using 8-bits for inference where the additional 2-4x performance is necessary. On the training front, it has been shown that 16-bit training is sufficient for certain tasks [1, 101]. The recent Gordon Bell winner demonstrated that lower-precison training can be used for scientific machine learning tasks as well [102].

The analogous effort to the work in deep learning to the examples of our interest in scientific computing involves training the network in lower precision and performing inference in a higher one [103, 104]. The compute imbalance between training and inference is even higher than that of factorization and the subsequent iterative refinement. Another difference is that in the context of neural network training, lowering the precision may be incorporated into the model as a regularizer.

9 Multiprecision capabilities of xSDK Math Libraries and Interoperability

9.1 Ginkgo

Ginkgo is a modern sparse linear algebra library able to run on multi- and manycore architectures [105]. The library design is guided by combining ecosystem extensibility with heavy, architecture-specific kernel optimization using the platform-native languages CUDA (NVIDIA GPUs), HIP (AMD GPUs), or OpenMP (Intel/AMD/ARM multicore). The software development cycle ensures production-quality code by featuring unit testing, automated configuration and installation, Doxygen code documentation, as well as a Continuous Integration (CI) and Continuous Benchmarking framework.

Ginkgo uses a static template parameter for the value type and a template parameter for the integer type to allow for compilation in different precision formats. Standard value type formats supported are IEEE double precision, IEEE single precision, double complex precision, and single complex precision. Theoretically, Ginkgo can also be compiled for any other (arbitrary) precision format, but the support on both the hardware and the software side is very limited outside the IEEE standard.

Aside from being compilable for different precision formats, Ginkgo features the adaptive precision block-Jacobi preconditioner, decoupling the memory precision from the arithmetic precision, and optimizing the storage format for the inverted diagonal block individually. Even though heavily leveraging advanced multiprecision technology, the numerical considerations of the adaptive precision block-Jacobi preconditioner are fully automated and hidden from the user who can employ the functionality as black-box algorithm without numerical degradation. Building upon the knowledge gained in the adaptive precision block-Jacobi, Ginkgo is currently employing the accessor concept to consequently separate the memory precision from the arithmetic precision, see section 7.

A orthogonal multiprecision technology that is under consideration for integration into Ginkgo is the multiprecision SpMV based on value clustering.

9.2 heFFTe

The Highly-Efficient FFTs for Exascale (heFFTe) library provides fast and robust multi-dimensional FFT routines for Exascale platforms. heFFTe leverages established but *ad hoc* software tools that have traditionally been part of application codes, but not extracted as independent, supported libraries. These multidimensional FFTs rely on third-party 1D FFTs, either from FFTW or from vendor libraries.

FFTs are used in many domain applications–including molecular dynamics, spectrum estimation, fast convolution and correlation, signal modulation, and wireless multimedia applications. For example, distributed 3-D FFT is one of the most important kernels used in molecular dynamics computations, and its performance can affect an application's scalability on larger machines. Similarly, the performance of the first principle calculations depends strongly on the performance of the FFT

solver. Specifically, for DOE, we found that more than a dozen ECP applications use FFT in their codes. However, the current state-of-the-art FFT libraries are not scalable on large heterogeneous machines with many nodes, or even on one node with multiple high-performance GPUs (e.g., several NVIDIA V100 GPUs). To address these needs, the heFFTe v0.2 library release demonstrates very good weak and strong scalability, and a very high performance that is close to 90% of the roof-line theoretical peak performance. This is achieved through (1) efficient use of GPUs' high bandwidth, (2) algorithms to reduce global communications, when possible, and (3) employment of GPUDirect technologies as well as MPI optimizations. heFFTe provides multi-precision capabilities with support for both single and double precision arithmetic. heFFTe is a C++ library, and the arithmetic used is templated, so that other precisions can be easily added. Current work is on adding mixed-precision capabilities using mixed-preision MPI and compression, as described in Section 3.2.

9.3 hypre

hypre is a software library of high-performance preconditioners and solvers for the solution of large, sparse linear systems of equations on massively parallel computers. The hypre library was created with the primary goal of providing users with advanced parallel preconditioners. The library features parallel multigrid solvers for both structured and unstructured grid problems. For ease of use, these solvers are accessed from the application code via hypre's conceptual linear system interfaces, which allow a variety of natural problem descriptions and include a structured, a semi-structured and a linear-algebraic interface. The (semi-)structured interfaces are an alternative to the standard matrix-based interface, give users a more natural means for describing linear systems and provide access to structured multigrid solvers, which can take advantage of the additional information.

9.4 Kokkos Kernels

The Kokkos Kernels project primarily focuses on performance-portable kernels for sparse/dense linear algebra and graph kernels. Kokkos Kernels relies on Kokkos programming model for portability. The focus of sparse linear algebra kernels has been to support the requirements of frameworks such as Trilinos and computational science applications. The sparse linear algebra data structure used is a compressed row storage. Kokkos Kernels provides kernels for sparse matrix-vector multiplication, sparse matrix-matrix multiplication, ILU(k) factorization, Gauss-Seidel preconditioner, triangular solves when the triangular factors arise from direct solvers or incomplete factorizations. All these kernels are templated on the matrix and the vector type allowing multiple precision support from the initial software design. Kokkos Kernels also supports dense linear algebra kernels for team-level BLAS and LAPACK functionality. This allows computational science applications to use BLAS and LAPACK operations in the "inner-loop" when programming for accelerators. The BLAS and LAPACK functionality is also templated on the scalar type allowing multiprecision use. Kokkos Kernels also support graph kernels such as distance-1 coloring, distance-2 coloring and triangle counting kernels.

9.5 MAGMA

MAGMA provides LAPACK and a large number of highly optimized dense and sparse linear algebra (LA) routines for heterogeneous architectures. Besides LAPACK, other dense LA routines in MAGMA include BLAS, Batched BLAS and LAPACK, and mixed-precision factorizations and solvers. A MAGMA Sparse component provides support for sparse iterative solvers and preconditioners, a number of sparse matrix formats and conversion routines, SpMV/MM and auxiliary kernels.

MAGMA addresses the complex challenges of heterogeneous compute environments by providing hybridized software that combines the strengths of different algorithms for different hardware components. MAGMA's LA algorithms target hybrid manycore systems featuring GPUs specifically and thus enable applications to fully exploit the power offered by each of the hardware components. MAGMA provides solvers for linear systems, least squares, eigenvalue problems, and singular value problems. Designed to be similar to LAPACK in functionality, data storage, and interface, the MAGMA library allows scientists to seamlessly port any linear algebra reliant software components to heterogeneous architectures. MAGMA allows applications to fully exploit the power of current heterogeneous systems of multi/many-core CPUs and multi-GPUs to deliver the fastest possible time to accurate solution within given energy constraints.

MAGMA provides mixed-precision solvers using LU, Cholesky, or QR factorizations. In terms of low precision developments, the latest MAGMA release to-date (v 2.5.3) provides an optimized batch HGEMM kernel that outperforms the vendor BLAS for relatively small sizes. It also provides a mixed-precision linear solver for $Ax = b$ in double-precision, while taking advantage of half-precision during the LU factorization. The mixed-precision solver is up to 4× faster than a direct FP64 solver, and converges to double precision accuracy if the condition number of the matrix $\kappa_\infty(A)$ is up to 10^5.

9.6 PETSc

PETSc is a suite of data structures and routines for the scalable solution of scientific applications modeled by partial differential equations; TAO is a scalable library for numerical optimization. PETSc/TAO can be easily used in application codes written in C, C++, Fortran, and Python.

PETSc is written in pure C89 (recently extended to support portions of C999 that are supported by the more recent Microsoft C compilers). The emphasis has always been on ultimate portability to the Fortran and C standards. At the same time we have always insured PETSc compilers completely with C++ as well and that the C compiled version can be used from C++ compiled code. The largest hassle in this regard has always been the differences between complex number handling in C and C++ requiring extensive code to handle the differences. PETSc can be built only for a single scalar type and precision at a time, for example real numbers and quad precision. Since C does not offer templates, managing multiple integrated precision's is difficult. For CPUs, PETSc supports half-precision (ARM only), single, double, quad (GNU compilers only). The above applies for CPU based systems. For GPU's, where large improvements in time to solution are possible with less precision, PETSc will use its GPU interfaces to allow computing with a selected precision at runtime on the GPUs. If the numerical values are in, say, double on the CPU they would be converted to, for example, single when transferred to the GPU for the computation. Of course, the more desirable case where the data remains on the GPU will require less conversion, except when particularly desired, for example, ill-conditioning requires a portion of the computation to be done with more precision.

9.7 PLASMA

PLASMA (Parallel Linear Algebra Software for Modern Architectures) [5] is a software package based on modern OpenMP for solving problems in dense linear algebra. PLASMA provides implementations of state-of-the-art algorithms using modern task scheduling techniques. PLASMA provides routines for solving linear systems, least squares problems, eigenvalue problems, and singular value problems. PLASMA is based on OpenMP and its data-dependence tracking and task scheduling. PLASMA library allows scientists to easily port their existing software components from LAPACK to PLASMA to take advantage of the new multicore architectures. PLASMA provides LAPACK-style interface for maximum portability and compatibility. An interface with more efficient data storage is also provided to achieve performance as close as possible to the computational peak performance of the machine.

9.8 SLATE

SLATE is a distributed, GPU accelerated library for dense linear algebra, intended as a replacement for ScaLAPACK. To this end, SLATE provides parallel Basic Linear Algebra Subprograms (BLAS), norms, linear systems solvers, least square solvers, singular value and eigenvalue solvers. It is written using modern C++, with ScaLAPACK and LAPACK compatible wrappers.

SLATE provides mixed-precision solvers using both LU and Cholesky factorization. The factorization is done in a lower precision, then iterative refinement is applied to improve the accuracy to a higher precision. The code is templated on the two precisions; currently single/double and single-complex/double-complex are supported. Future plans include using half precision and a more robust GMRES refinement mechanism.

9.9 STRUMPACK

STRUMPACK is a distributed, GPU accelerated library for dense and sparse linear algebra using rank-structured matrix approximations, including hierarchically semiseparable (HSS), hierarchically off-diagonal low rank (HODLR), butterfly, and a non-hierarchical format called block low rank (BLR). The baseline sparse STRUMPACK is a multifrontal sparse LU direct solver. The frontal matrices in the sparse factors can be approximated with the above rank-structured formats, serving as effective sparse preconditioners with nearly optimal complexity in flops and memory. Sparse STRUMPACK relies on ButteryPACK for the HODLR and butterfly formats, and provides C++ interfaces to the ButteryPACK Fortran library.

STRUMPACK is written using modern C++, with templated datatypes to support various precisions, including real and complex, single and double precisions. It can also support half-precision. Currently, iterative refinement and GMRES are performed in the same working precision as factorization.

9.10 SuperLU

SuperLU is a distributed, GPU accelerated sparse direct solver for general sparse linear systems, using supernodal techniques in LU factorization and triangular solves. It uses MPI+OpenMP+CUDA to support various forms of parallelism. Routines are also provided to equilibrate the system, estimate the condition number, calculate the relative backward error, and estimate error bounds for the refined solutions.

SuperLU is written in C and is callable from either C or Fortran program. The code base uses macros to template the datatypes, so it can support the mixture of various precisions, including real and complex, single, double and half precisions. Currently, iterative refinement is performed in the same working precision as factorization. Work is in progress to provide lower precision factorization coupled with higher precision iterative refinement.

9.11 Trilinos

The Trilinos Project is a premier software framework in scientific computing for the solution of large-scale, complex multiphysics engineering and scientific problems. Trilinos is object-oriented and organized into about 60 different packages, each with a specific focus. These packages include linear and nonlinear solvers, preconditioners (including algebraic multigrid), graph partitioners, eigensolvers, and optimization algorithms, among other things. Users are required to install only the subset of packages related to the problems they are trying to solve. Trilinos supports MPI+X, where X can be CUDA, OpenMP, etc. (anything Kokkos supports).

In Trilinos, the scalar type is a template parameter, typically set to IEEE double precision while also IEEE single precision is fully supported. Users can employ preconditioners that are compiled in single precision inside a double precision outer solver - however have to account for the numerical effects, i.e., may need a flexible Krylov solver (FCG / FGMRES). A brief discussion of using mixed precision in the Belos package was given in [106]. Other scalar types than single and double may also be used, however, this is not common and not supported in the explicit template instantiation (ETI) build system.

10 IEEE Formats and Format Conversion

10.1 Emulator

The half-precision (fp16) floating-point format, defined in the 2008 revision of the IEEE standard for floating-point arithmetic, and a more recently proposed half-precision format bfloat16, are increasingly available in GPUs and other accelerators. While the support for low precision arithmetic is mainly motivated by machine learning applications, as discussed in earlier sections, general purpose numerical algorithms can benefit from it too. Since the appropriate hardware is not always available, and one may wish to experiment with new arithmetics not yet implemented in hardware, software simulations of low precision arithmetic are needed. In [107], Higham and Pranesh discuss a strategy to simulate low precision arithmetic using arithmetic of higher precision, and correctness

of such simulations is explained via rounding error analysis. A MATLAB function function chop[2] is provided, that can be used to efficiently simulate fp16, bfloat16, and other low precision arithmetics, with or without the representation of subnormal numbers and with the options of round to nearest, directed rounding, stochastic rounding, and random bit flips in the significand. Interested readers are referred to [107] for further details.

10.2 Rounding Error Analysis

Traditional rounding error analysis in numerical linear algebra leads to backward error bounds involving the constant $\gamma_n = nu/(1-nu)$, for a problem size n and unit roundoff u. In light of large-scale and possibly low-precision computations, such bounds can struggle to provide any useful information. In [108], Higham and Mary develop a new probabilistic rounding error analysis that can be applied to a wide range of algorithms. By using a concentration inequality and making probabilistic assumptions about the rounding errors, they show that in several core linear algebra computations γ_n can be replaced by a relaxed constant $\widetilde{\gamma}_n$ proportional to $\sqrt{n\log n}u$ with a probability bounded below by a quantity independent of n. The new constant $\widetilde{\gamma}_n$ grows much more slowly with n than γ_n. We refer to [108], [109] for further details.

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.

Disclaimer

[2] https://github.com/higham/chop

References

[1] Suyog Gupta, Ankur Agrawal, Kailash Gopalakrishnan, and Pritish Narayanan. Deep Learning with Limited Numerical Precision. In *Proceedings of the 32Nd International Conference on International Conference on Machine Learning - Volume 37*, ICML'15, pages 1737–1746. JMLR.org, 2015.

[2] IEEE Standard for Floating-Point Arithmetic. *IEEE Std 754-2008*, pages 1–70, Aug 2008.

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

[4] Azzam Haidar, Panruo Wu, Stanimire Tomov, and Jack Dongarra. Investigating half precision arithmetic to accelerate dense linear system solvers. In *Proceedings of the 8th Workshop on Latest Advances in Scalable Algorithms for Large-Scale Systems*, pages 1–8, 2017.

[5] Emmanuel Agullo, Jim Demmel, Jack Dongarra, Bilel Hadri, Jakub Kurzak, Julien Langou, Hatem Ltaief, Piotr Luszczek, and Stanimire Tomov. Numerical linear algebra on emerging architectures: The PLASMA and MAGMA projects. *Journal of Physics: Conference Series*, 180:012037, July 2009.

[6] Ahmad Abdelfattah, Stanimire Tomov, and Jack J. Dongarra. Fast Batched Matrix Multiplication for Small Sizes Using Half-Precision Arithmetic on GPUs. In *2019 IEEE International Parallel and Distributed Processing Symposium, IPDPS 2019, Rio de Janeiro, Brazil, May 20-24, 2019*, pages 111–122. IEEE, 2019.

[7] James W Demmel. Applied numerical linear algebra. 1997. *SIAM, Philadelphia*.

[8] James Hardy Wilkinson. *Rounding errors in algebraic processes*. Courier Corporation, 1994.

[9] Cleve B Moler. Iterative refinement in floating point. *Journal of the ACM (JACM)*, 14(2):316–321, 1967.

[10] Gilbert W Stewart. *Introduction to matrix computations*. Elsevier, 1973.

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

[12] James Demmel, Yozo Hida, William Kahan, Xiaoye S Li, Sonil Mukherjee, and E Jason Riedy. Error bounds from extra-precise iterative refinement. *ACM Transactions on Mathematical Software (TOMS)*, 32(2):325–351, 2006.

[13] Werner Oettli and William Prager. Compatibility of approximate solution of linear equations with given error bounds for coefficients and right-hand sides. *Numerische Mathematik*, 6(1):405–409, 1964.

[14] 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.

[15] 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.

[16] Youcef Saad and Martin H Schultz. Gmres: A generalized minimal residual algorithm for solving nonsymmetric linear systems. *SIAM Journal on scientific and statistical computing*, 7(3):856–869, 1986.

[17] Nicholas J. Higham. Error analysis for standard and GMRES-based iterative refinement in two and three-precisions. MIMS EPrint 2019.19, Manchester Institute for Mathematical Sciences, The University of Manchester, November 2019.

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

[19] Nicholas J. Higham and Srikara Pranesh. Exploiting lower precision arithmetic in solving symmetric positive definite linear systems and least squares problems. MIMS EPrint 2019.20, Manchester Institute for Mathematical Sciences, The University of Manchester, November 2019.

[20] Erin Carson, Nicholas J. Higham, and Srikara Pranesh. Three-precision GMRES-based iterative refinement for least squares problems. MIMS EPrint 2020.5, Manchester Institute for Mathematical Sciences, The University of Manchester, February 2020.

[21] Joseph M. Elble and Nikolaos V. Sahinidis. Scaling linear optimization problems prior to application of the simplex method. *Comput. Optim. Appl.*, 52(2):345–371, 2012.

[22] E. Anderson, Z. Bai, C. H. Bischof, S. Blackford, J. W. Demmel, J. J. Dongarra, J. J. Du Croz, A. Greenbaum, S. J. Hammarling, A. McKenney, and D. C. Sorensen. *LAPACK Users' Guide*. Society for Industrial and Applied Mathematics, Philadelphia, PA, USA, third edition, 1999.

[23] Philip A. Knight, Daniel Ruiz, and Bora Uçar. A symmetry preserving algorithm for matrix scaling. *SIAM J. Matrix Anal. Appl.*, 35(3):931–955, 2014.

[24] Azzam Haidar, Ahmad Abdelfattah, Mawussi Zounon, Panruo Wu, Srikara Pranesh, Stanimire Tomov, and Jack Dongarra. the design of fast and energy-efficient linear solvers: On the potential of half-precision arithmetic and iterative refinement techniques. In Yong Shi, Haohuan Fu, Yingjie Tian, Valeria V. Krzhizhanovskaya, Michael Harold Lees, Jack Dongarra, and Peter M. A. Sloot, editors, *Computational Science—ICCS 2018*, pages 586–600. Springer International Publishing, Cham, 2018.

[25] Nicholas J. Higham and G. W. Stewart. Numerical linear algebra in statistical computing. In A. Iserles and M. J. D. Powell, editors, *The State of the Art in Numerical Analysis*, pages 41–57. Oxford University Press, 1987.

[26] Pierre Blanchard, Nicholas J. Higham, Florent Lopez, Theo Mary, and Srikara Pranesh. Mixed precision block fused multiply-add: Error analysis and application to GPU tensor cores. MIMS EPrint 2019.18, Manchester Institute for Mathematical Sciences, The University of Manchester, September 2019.

[27] Takeshi Fukaya, Ramaseshan Kannan, Yuji Nakatsukasa, Yusaku Yamamoto, and Yuka Yanagisawa. Shifted cholesky qr for computing the qr factorization of ill-conditioned matrices. *SIAM J. Scientific Computing*, 42:A477–A503, 2020.

[28] Yusaku Yamamoto, Yuji Nakatsukasa, Yuka Yanagisawa, and Takeshi Fukaya. Roundoff error analysis of the CholeskyQR2 algorithm. *Electron. Trans. Numer. Anal.*, 44:306–326, 2015.

[29] Ichitaro Yamazaki, Stanimire Tomov, and Jack Dongarra. Mixed-precision Cholesky QR factorization and its case studies on multicore CPU with multiple GPUs. *SIAM J. Sci. Comput.*, 37(1):C307–C330, 2015.

[30] Ichitaro Yamazaki, Stanimire Tomov, and Jack Dongarra. Stability and performance of various singular value QR implementations on multicore CPU with a GPU. *ACM Trans. Math. Software*, 43(2):10:1–10:18, September 2016.

[31] Åke Björck. Iterative refinement of linear least squares solutions I. *BIT Numerical Mathematics*, 7(4):257–278, 1967.

[32] Åke Björck. Iterative refinement and reliable computing. In M. G. Cox and S. J. Hammarling, editors, *Reliable Numerical Computation*, pages 249–266. Oxford University Press, 1990.

[33] Pytorch 1.4.0 documentation: Quantization.

[34] Tensorflow lite 8-bit quantization specification.

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

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

[37] T. Ogita and K. Aishima. Iterative refinement for symmetric eigenvalue decomposition. *Japan Journal of Industrial and Applied Mathematics*, 35(3):1007–1035, 2018.

[38] T. Ogita and K. Aishima. Iterative refinement for symmetric eigenvalue decomposition II: clustered eigenvalues. *Japan J. Indust. Appl. Math.*, 36:435–459, 2019. https://doi.org/10.1007/s13160-019-00348-4.

[39] Dhillon, Inderjit S. and Parlett, Beresford N. and Vömel, Christof. The Design and Implementation of the MRRR Algorithm. *ACM Trans. Math. Softw.*, 32(4):0098–3500, 2006. https://doi.org/10.1145/1186785.1186788.

[40] Petschow, M. and Quintana-Ort, E. S. and Bientinesi, P. Improved Accuracy and Parallelism for MRRR-Based Eigensolvers—A Mixed Precision Approach. *SIAM Journal on Scientific Computing*, 36(2):C240–C263, 2014. https://doi.org/10.1137/130911561.

[41] P. Lindstrom. Zfp version 0.5.3, April.

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

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

[44] 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.

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

[46] Hejer Shaiek, Stanimire Tomov, Alan Ayala, Azzam Haidar, and Jack Dongarra. GPUDirect MPI Communications and Optimizations to Accelerate FFTs on Exascale Systems. *EuroMPI'19 Posters, Zurich, Switzerland*, (icl-ut-19-06), 2019-09 2019.

[47] A. Sorna, X. Cheng, E. D'Azevedo, K. Won, and S. Tomov. Optimizing the Fast Fourier Transform Using Mixed Precision on Tensor Core Hardware. In *2018 IEEE 25th International Conference on High Performance Computing Workshops (HiPCW)*, pages 3–7, 2018.

[48] Xaiohe Cheng, Anumeena Soma, Eduardo D'Azevedo, Kwai Wong, and Stanimire Tomov. Accelerating 2D FFT: Exploit GPU Tensor Cores through Mixed-Precision. 2018-11 2018.

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

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

[51] Gérard Meurant and Zdeněk Strakoš. The Lanczos and conjugate gradient algorithms in finite precision arithmetic. *Acta Numerica*, 15:471–542, 2006.

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

[53] Anne Greenbaum. Behavior of slightly perturbed Lanczos and conjugate-gradient recurrences. *Lin. Alg. Appl.*, 113:7–63, 1989.

[54] Anne Greenbaum. Estimating the attainable accuracy of recursively computed residual methods. *SIAM J. Matrix Anal. Appl.*, 18(3):535–551, 1997.

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

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

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

[58] Michael A Clark, Ronald Babich, Kipton Barros, Richard C Brower, and Claudio Rebbi. Solving lattice QCD systems of equations using mixed precision solvers on GPUs. *Computer Physics Communications*, 181(9):1517–1528, 2010.

[59] Gerard LG Sleijpen and Henk A van der Vorst. Reliable updated residuals in hybrid Bi-CG methods. *Computing*, 56(2):141–163, 1996.

[60] Henk A Van Der Vorst and Qiang Ye. Residual replacement strategies for Krylov subspace iterative methods for the convergence of true residuals. *SIAM J. Sci. Comput.*, 22(3):835–852, 2000.

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

[80] Hartwig Anzt, Jack Dongarra, Goran Flegar, Nicholas J Higham, and Enrique S Quintana-Ortí. Adaptive precision in block-jacobi preconditioning for iterative sparse linear system solvers. *Concurrency and Computation: Practice and Experience*, 31(6):e4460, 2019.

[81] Hartwig Anzt, Goran Flegar, Thomas Grützmacher, and Enrique S Quintana-Ortí. Toward a modular precision ecosystem for high-performance computing. *The International Journal of High Performance Computing Applications*, 33(6):1069–1078, 2019.

[82] Thomas Grützmacher, Terry Cojean, Goran Flegar, Fritz Göbel, and Hartwig Anzt. A customized precision format based on mantissa segmentation for accelerating sparse linear algebra. *Concurrency and Computation: Practice and Experience*, page e5418, 2019.

[83] J.L. Gustafson. *The End of Error: Unum Computing*. Chapman & Hall/CRC Computational Science. Taylor & Francis, 2015.

[84] Karl Ljungkvist and Martin Kronbichler. Multigrid for matrix-free finite element computations on graphics processors. *Technical report / Department of Information Technology, Uppsala University*, 2017.

[85] Karl Ljungkvist and Martin Kronbichler. Multigrid for matrix-free high-order finite element computations on graphics processors. *ACM Transactions on Parallel Processing*, 2019.

[86] Stefan Lemvig Glimberg, Allan Peter Engsig-Karup, and Morten G Madsen. A fast gpu-accelerated mixed-precision strategy for fully nonlinearwater wave computations. In *Proceedings of ENUMATH 2011*, 2011.

[87] Takateru Yamagishi and Yoshimasa Matsumura. Gpu acceleration of a non-hydrostatic ocean model with a multigrid poisson/helmholtz solver. *Procedia Computer Science*, 80:16581669, 2016.

[88] Maximilian Emans and Albert van der Meer. Mixed-precision amg as linear equation solver for definite systems. In *Proceedings of International Conference on Computational Science, ICCS 2010*, volume 1, page 175183, 2012.

[89] Yuki Sumiyoshi, Akihiro Fujii, Akira Nukada, and Teruo Tanaka. Mixed-precision amg method for many core accelerators. In *EUROMPI/ASIA 14: Proceedings of the 21st European MPI Users' Group Meeting*, page 127132, 2014.

[90] Christian Richter, Sebastian Schops, and Markus Clemens. Gpu-accelerated mixed precision algebraic multigrid preconditioners for discrete elliptic field problems. *IEEE Transactions on Magnetics*, 50(2), 2014.

[91] Kate Clark. Effective use of mixed precision for hpc. Smoky Mountain Conference 2019.

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

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

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

[95] Yu-Hsin Chen, Joel S. Emer, and Vivienne Sze. Eyeriss v2: A flexible and high-performance accelerator for emerging deep neural networks. *CoRR*, abs/1807.07928, 2018.

[96] Norman P. Jouppi, Cliff Young, Nishant Patil, David A. Patterson, Gaurav Agrawal, Raminder Bajwa, Sarah Bates, Suresh Bhatia, Nan Boden, Al Borchers, Rick Boyle, Pierre-luc Cantin, Clifford Chao, Chris Clark, Jeremy Coriell, Mike Daley, Matt Dau, Jeffrey Dean, Ben Gelb, Tara Vazir Ghaemmaghami, Rajendra Gottipati, William Gulland, Robert Hagmann, Richard C. Ho, Doug Hogberg, John Hu, Robert Hundt, Dan Hurt, Julian Ibarz, Aaron Jaffey, Alek Jaworski, Alexander Kaplan, Harshit Khaitan, Andy Koch, Naveen Kumar, Steve Lacy, James Laudon, James Law, Diemthu Le, Chris Leary, Zhuyuan Liu, Kyle Lucke, Alan Lundin, Gordon MacKean, Adriana Maggiore, Maire Mahony, Kieran Miller, Rahul Nagarajan, Ravi Narayanaswami, Ray Ni, Kathy Nix, Thomas Norrie, Mark Omernick, Narayana Penukonda, Andy Phelps, Jonathan Ross, Amir Salek, Emad Samadiani, Chris Severn, Gregory Sizikov, Matthew Snelham, Jed Souter, Dan Steinberg, Andy Swing, Mercedes Tan, Gregory Thorson, Bo Tian, Horia Toma, Erick Tuttle, Vijay Vasudevan, Richard Walter, Walter Wang, Eric Wilcox, and Doe Hyun Yoon. In-datacenter performance analysis of a tensor processing unit. *CoRR*, abs/1704.04760, 2017.

[97] Hyoukjun Kwon, Ananda Samajdar, and Tushar Krishna. Maeri: Enabling flexible dataflow mapping over dnn accelerators via reconfigurable interconnects. *ACM SIGPLAN Notices*, 53(2):461–475, 2018.

[98] Frank Seide, Hao Fu, Jasha Droppo, Gang Li, and Dong Yu. 1-bit stochastic gradient descent and its application to data-parallel distributed training of speech dnns. In *Fifteenth Annual Conference of the International Speech Communication Association*, 2014.

[99] Martín Abadi, Paul Barham, Jianmin Chen, Zhifeng Chen, Andy Davis, Jeffrey Dean, Matthieu Devin, Sanjay Ghemawat, Geoffrey Irving, Michael Isard, et al. Tensorflow: A system for large-scale machine learning. In *12th {USENIX} Symposium on Operating Systems Design and Implementation ({OSDI} 16)*, pages 265–283, 2016.

[100] Adam Paszke, Sam Gross, Soumith Chintala, Gregory Chanan, Edward Yang, Zachary DeVito, Zeming Lin, Alban Desmaison, Luca Antiga, and Adam Lerer. Automatic differentiation in pytorch. 2017.

[101] Matthieu Courbariaux, Yoshua Bengio, and Jean-Pierre David. Training deep neural networks with low precision multiplications. *arXiv preprint arXiv:1412.7024*, 2014.

[102] Thorsten Kurth, Sean Treichler, Joshua Romero, Mayur Mudigonda, Nathan Luehr, Everett Phillips, Ankur Mahesh, Michael Matheson, Jack Deslippe, Massimiliano Fatica, et al. Exascale deep learning for climate analytics. In *SC18: International Conference for High Performance Computing, Networking, Storage and Analysis*, pages 649–660. IEEE, 2018.

[103] Suyog Gupta, Ankur Agrawal, Kailash Gopalakrishnan, and Pritish Narayanan. Deep learning with limited numerical precision. *CoRR*, abs/1502.02551, 2015. accessed: 2018-08-01.

[104] Suyog Gupta, Ankur Agrawal, Kailash Gopalakrishnan, and Pritish Narayanan. Deep learning with limited numerical precision. In Francis Bach and David Blei, editors, *Proceedings of the 32nd International Conference on Machine Learning*, volume 37 of *Proceedings of Machine Learning Research*, pages 1737–1746, Lille, France, 2015. PMLR. accessed: 2018-08-01.

[105] Hartwig Anzt, Erik Boman, Rob Falgout, Pieter Ghysels, Michael Heroux, Xiaoye Li, Lois Curfman McInnes, Richard Tran Mills, Sivasankaran Rajamanickam, Karl Rupp, et al. Preparing sparse solvers for exascale computing. *Philosophical Transactions of the Royal Society A*, 378(2166):20190053, 2020.

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

[107] Nicholas J. Higham and Srikara Pranesh. Simulating low precision floating-point arithmetic. *SIAM Journal on Scientific Computing*, 41(5):C585–C602, 2019.

[108] Nicholas J. Higham and Theo Mary. A new approach to probabilistic rounding error analysis. *SIAM Journal on Scientific Computing*, 41(5):A2815–A2835, 2019.

[109] Nicholas J. Higham and Theo Mary. Sharper probabilistic backward error analysis for basic linear algebra kernels with random data. MIMS EPrint 2020.4, Manchester Institute for Mathematical Sciences, The University of Manchester, January 2020.