


A survey of numerical linear algebra methods utilizing mixed-precision arithmetic

The International Journal of High Performance Computing Applications
2021, Vol. 35(4) 344–369
© The Author(s) 2021
Article reuse guidelines:
sagepub.com/journals-permissions
DOI: 10.1177/10943420211003313
journals.sagepub.com/home/hpc


Ahmad Abdelfattah¹, Hartwig Anzt^{1,2} , Erik G Boman³,
Erin Carson⁴, Terry Cojean², Jack Dongarra^{1,5,6}, Alyson Fox⁷,
Mark Gates¹, Nicholas J Higham⁶, Xiaoye S Li⁸, Jennifer Loe³ ,
Piotr Luszczek¹, Srikara Pranesh⁶, Siva Rajamanickam³,
Tobias Ribizel² , Barry F Smith⁹, Kasia Swirydowicz¹⁰,
Stephen Thomas¹⁰, Stanimire Tomov¹, Yaohung M Tsai¹
and Ulrike Meier Yang⁷

Abstract

The efficient utilization of mixed-precision numerical linear algebra algorithms can offer attractive acceleration to scientific computing applications. Especially with the hardware integration of low-precision special-function units designed for machine learning applications, the traditional numerical algorithms community urgently needs to reconsider the floating point formats used in the distinct operations to efficiently leverage the available compute power. In this work, we provide a comprehensive survey of mixed-precision numerical linear algebra routines, including the underlying concepts, theoretical background, and experimental results for both dense and sparse linear algebra problems.

Keywords

Mixed-precision arithmetic, numerical mathematics, linear algebra, high-performance computing, GPUs

1. Introduction

In computational numerics, the accuracy of a computed result and the time needed to complete the algorithm both heavily depend on the floating point format used for storage and arithmetic operations. While we acknowledge the existence of other formats, here we focus on fixed-size floating point formats composed of a sign bit, an exponent, and a significand. Roughly speaking, the length of the exponent determines the value range of a floating point format, and the length of the significand determines the relative accuracy of the format in that range. While a sufficient exponent range is necessary for the meaningful data representation, generally the accuracy of an algorithm output strongly correlates with the significand length.

The cost of communication and computation in a numerical application grows with the size of the floating point format. Communication here can mean access to main memory and data transfers within computing cores, in between distinct compute cores, or between distinct compute nodes. The cost of data transfers is composed of a constant access latency and the transfer time that reflects the ratio between message size and data transfer rate.

Consequently, when ignoring the latency, the communication cost linearly increases with the size (in bits) of the floating point format. This implies that the runtime impact of communicating values in different formats is hardware independent and only depends on the size of the floating point formats. Specifically, for the widely adopted IEEE754 formats for double precision (64 bit) and single precision (32 bit) (IEEE), the runtime difference for memory/communication operations is roughly $2\times$, independent

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

Corresponding author:

Hartwig Anzt, Karlsruher Institut für Technologie (KIT), Germany.
Email: hartwig.anzt@kit.edu; hanzt@icl.utk.edu

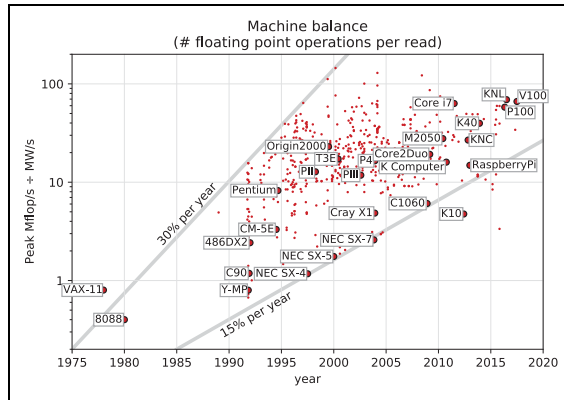


Figure 1. Evolution of the machine balance of processors over different hardware generations.

of the hardware platform. At the same time, the cost of performing arithmetic operations heavily depends on the hardware support for computations in a certain floating point format, and the acceleration/slowdown of computing in different formats can vary significantly between hardware architectures. For example, on Intel's 64-bit processors, the compute performance in single precision is twice the compute performance of double precision. On AMD's Radeon VII GPU, the ratio between the single-precision performance and the double-precision performance is about $4\times$.

Given the performance differences for computing and communicating in different precision formats, there is a long history of efforts that aim to improve the performance of numerical algorithms by carefully combining precision formats. The overall goal of these mixed-precision algorithms is to accelerate the applications with the use of lower precision formats while maintaining the high accuracy of the output.

But while the idea of mixed-precision algorithms has been around for several decades, recent hardware trends have motivated increased research and development activities. Within the past few years, hardware vendors have started designing special-purpose units for low-precision arithmetic in response to the machine learning community's demand for high compute power in low-precision formats. Also, the server-level products are increasingly featuring low-precision special function units (e.g., NVIDIA Tensor Cores in V100 GPUs) providing about $16\times$ higher performance than what can be achieved in IEEE double precision. Exploiting this compute power efficiently could offer up to an order of magnitude of speedup to compute-bound algorithms. 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 progressively more expensive compared with arithmetic operations (Figure 1). Given the over provisioning of modern hardware for arithmetic operations, it may be a rational decision for memory-bound

algorithms to compress all data in cache before communicating with remote processors or main memory

In this paper, we present mixed-precision linear algebra algorithms and the attainable performance advantages for dense linear algebra (Section 3) and for sparse linear algebra (Section 4). We conclude in Section 5 with an outlook on current algorithm development and perspectives for mixed-precision technology on future architectures. We note that this survey is focusing on numerical linear algebra operating on explicitly-available linear operators, matrix-free methods remain outside the scope of this work.

2. Precision formats, hardware realization, and notation

Before presenting mixed-precision algorithms, we want to establish some notation we use throughout the rest of the paper, and provide some background on precision formats and their realization in hardware. We exclusively focus on floating point formats that are composed of a sign bit, a sequence of exponent bits, and a sequence of significand bits. The distinct precision formats then differ in the composition in terms of how many bits are used for the exponent and how many bits are used for the significand. Generally, we use the term *high precision* for precision formats that provide high accuracy at the cost of a larger memory volume (in terms of bits) and *low precision* to refer to precision formats that compose of fewer bits (smaller memory volume) and provide low(er) accuracy. Unless explicitly stated, we think of IEEE double precision when using the term *high precision* and IEEE754 single precision when using the term *low precision* (IEEE, 2019). These formats are of particular interest as they are natively supported by a broad range of hardware architectures. However, in particular with the rise of machine learning, a number of architectures now also provide native support for floating point formats, that are even more compact than IEEE754 single precision. In particular IEEE754 half precision and BFloat16 are formats that experience increased interest by the community. For all floating point formats, their bitwise configuration determines the characteristics. Roughly speaking, the length of the exponent of a precision format determines the range of a format, the length of the significand determines the precision of a format. Relevant indicators in that context are the largest and smallest representable numbers in a format, and the unit roundoff of a format u . In Table 1 we list some of the most relevant formats used in modern scientific high-performance computing along with their key characteristics. Traditionally, hardware and software are strictly coupling the precision format used for arithmetic operations and for memory operations. However, given that most architectures are nowadays overprovisioned for arithmetic operations, there exist trends to break up this strict coupling. On the hardware side, a recent example of an architecture breaking up the coupling between memory format and arithmetic format are the NVIDIA Tensor Cores integrated into NVIDIA's

Table 1. Parameters for various floating point arithmetics. “Range” denotes the order of magnitude of the smallest subnormal ($x_{\min,s}$) and largest and smallest positive normalized floating point numbers. BFloat16 does not support subnormal numbers.

Arithmetic	Size (bits)	Range			Unit roundoff u
		$x_{\min,s}$	x_{\min}	x_{\max}	
BFloat16	16	—	1.2×10^{-38}	3.4×10^{38}	3.9×10^{-3}
IEEE FP16	16	6.0×10^{-8}	6.1×10^{-5}	6.6×10^4	4.9×10^{-4}
IEEE FP32	32	1.4×10^{-45}	1.2×10^{-38}	3.4×10^{38}	6.0×10^{-8}
IEEE FP64	64	4.9×10^{-324}	2.2×10^{-308}	1.8×10^{308}	1.1×10^{-16}
IEEE FP128	128	6.5×10^{-4966}	3.4×10^{-4932}	1.2×10^{4932}	9.6×10^{-35}

Volta GPU architecture v10 (2017). These special function units designed to perform high-performance matrix-matrix multiplication take half precision (FP16) input data, but compute in FP32 (v10, 2017). On the software side, the concept of a memory accessor separating the memory precision from the arithmetic precision pursues the same goal: computing in higher precision while handling the data in lower precision in memory (Anzt et al., 2019b). In 4.3 we will detail the software-based approach in more detail. For the V100 GPU experiments in Section 3, we claim that the algorithms operating on the Volta Tensor Cores use half precision, acknowledging that internally, the arithmetic operations are using higher precision after converting the half-precision input data.

3. Dense linear algebra

Carefully designed mixed-precision dense linear algebra algorithms can leverage the potential performance advantages of low-precision arithmetic. With this in mind, we start the section in Section 3.1 by presenting basic linear algebra subroutines specifically designed to exploit the compute power of NVIDIA’s Tensor Cores, which provide high compute power in low precision. Building on low-precision Basic Linear Algebra Subprograms (BLAS) and guided by the concept of Newton’s method (Section 3.2), it is possible to derive high-performance linear solvers running in low precision that, embedded in an iterative refinement (Section 3.3), succeed in generating high-accuracy solutions while conducting most of the work in low-precision arithmetic. The standard approach is based on factorizing a matrix in low precision and using an iterative refinement scheme in high precision to recover a high-accuracy solution (see Section 3.3). However, for numerical reasons, it can be advantageous to use the factorization computed in low precision as a preconditioner for a Generalized Minimum Residual (GMRES) iterative solver embedded in an iterative refinement loop (see Section 3.4). Using sophisticated scaling and shifting techniques, symmetry and positive definiteness of a system matrix can be exploited in a Generalized Minimum Residual-based Iterative Refinement (GMRES-IR) variant using a low-precision Cholesky factorization as a preconditioner (see Section 3.5). In Section 3.6, we present some performance

results demonstrating the potential of these techniques on modern GPU architectures. The scope of mixed-precision iterative refinement is not limited to linear systems and extends to least-square problems (Section 3.7) and eigenvalue solvers (Section 3.8).

3.1. Low-precision BLAS

The revolution of machine learning applications and artificial intelligence (AI) spiked an interest in developing high-performance 16-bit, half-precision floating point arithmetic (FP16), because most AI applications do not necessarily require the accuracy of 32-bit, full-precision floating point arithmetic (FP32) or 64-bit, double-precision floating point arithmetic (FP64) (Gupta et al., 2015). FP16 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 $2\times$ against FP32, and by a factor of $4\times$ against FP64.

In terms of vendor support, NVIDIA, Google, and AMD manufacture hardware that is capable of performing FP16 arithmetic. Google’s Tensor Core Processing Units (TPUs) are customized chips that are mainly designed for machine learning workloads using the 16-bit brain floating point (BFloat16) format. AMD also provides half-precision capabilities, and their software stack shows support for both the BFloat16 format and the IEEE FP16 format, (IEEE). The theoretical performance of half precision on AMD GPUs follows the expected $2\times$ speedup against FP32 and $4\times$ speedup against FP64. As an example, the Mi50 GPU has a theoretical FP16 performance of 26.5 TFLOP/s vs. 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’s GPUs, which first supported half-precision arithmetic in the Maxwell GPU architecture. Throughout this section, we will focus on NVIDIA’s GPUs and math libraries to highlight half-precision developments for numerical kernels.

While NVIDIA’s Maxwell GPU architecture introduced hardware support for IEEE 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 $12\times$ faster than the theoretical

FP16 peak performance of the preceding architecture (Pascal architecture). Applications taking advantage of the Tensor Cores can run up to $4\times$ 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 detail in Section 3.1.1.

In terms of half-precision BLAS, most of the available routines provide 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 (Haidar et al., 2018b), which is used to accelerate the solution of $Ax = b$ in double precision, see Section 3.3.

3.1.1. Hardware acceleration of half precision. The CUDA Toolkit is one of the first programming models to provide half-precision (i.e., FP16) arithmetic. Support was added in late 2015 for selected embedded GPU models based on the Maxwell architecture, and FP16 arithmetic has become mainstream in CUDA-enabled GPUs since the Pascal architecture. FP16 has a dynamic range that is significantly smaller than single or double precision (see Table 1).

The Volta and Turing architectures introduced hardware acceleration for matrix multiplication in FP16 using the aforementioned Tensor Cores. Using Tensor Cores for FP16, these GPUs can deliver a theoretical peak performance that is up to $8\times$ 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 that 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.

NVIDIA's cuBLAS library 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 also possible through low-level APIs that are provided by the programming model. As shown in Figure 2, Tensor Cores deal with input and output data through opaque data structures called *fragments*. Each fragment is used to store one matrix.

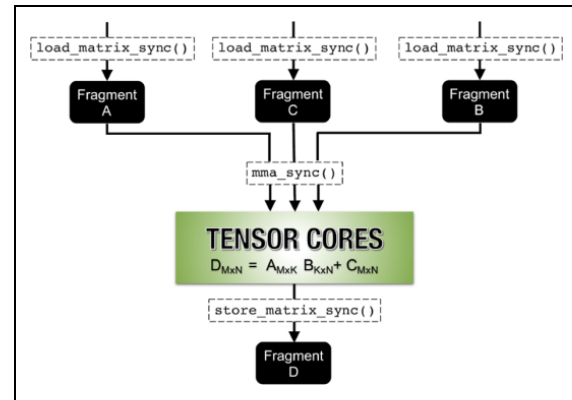


Figure 2. Programmability of the Tensor Core units.

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.

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 a 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.

3.1.2. Half-precision GEMM (HGEMM). The cuBLAS library provides several routines that take advantage of the reduced FP16 precision. Figure 3 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 $2\times$ the single-precision performance, 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 $5\times$ the asymptotic performance of a non-accelerated HGEMM.

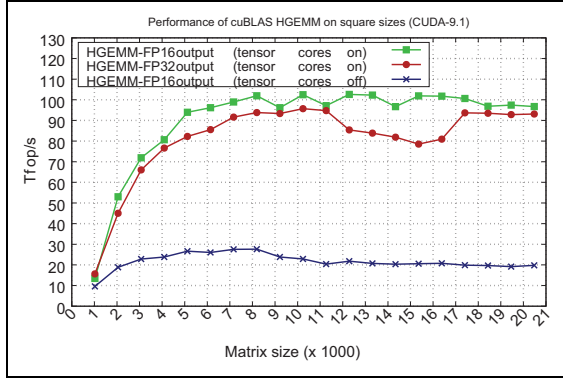


Figure 3. Performance of different HGEMM kernel from the cuBLAS library on square sizes. Results are shown on a Tesla V100 GPU using CUDA-9.1.

However, perhaps the most interesting performance graph of Figure 3 is the HGEMM with FP32 output, where we can see that the performance is close to the accelerated HGEMM kernel but with much more precision on the output. This is of particular importance for mixed-precision algorithms (Haidar et al., 2018b, 2017). To put this in perspective, Figure 4 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:

$$\frac{\|R_{cuBLAS} - R_{MKL}\|_F}{\sqrt{k + 2\|\alpha\|_F\|B\|_F + 2\|\beta\|_F\|C\|_F}},$$

where k is equal to the matrix size, and α and β are the two scalars in the standard GEMM operation ($C = \alpha AB + \beta C$). The first surprising observation is that an HGEMM operation with FP16 output is more accurate if the Tensor Cores are turned on (as the accumulation in the Tensor cores happens in FP32), 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.

3.1.3. Batch HGEMM. Apart from the vendor-supplied BLAS, a few efforts have focused on building open-source BLAS routines that utilize NVIDIA Tensor Cores. An example of such efforts is in the MAGMA library (Agullo et al., 2009) which has a batch HGEMM kernel that makes use of the Tensor Cores (Abdelfattah et al., 2019). 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

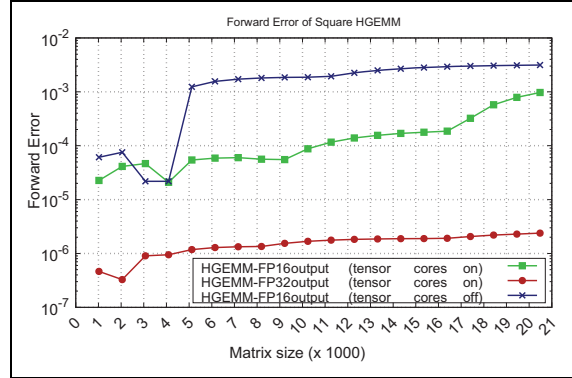


Figure 4. Forward error of HGEMM with respect to MKL SGEMM ($C = \alpha AB + \beta C$). Results are shown for square sizes using cuBLAS 9.1 and MKL 2018.1.

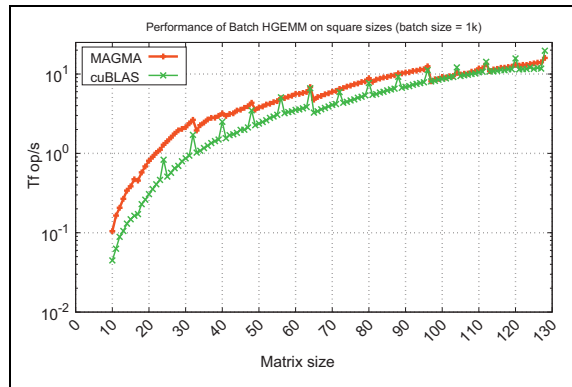


Figure 5. Performance of the batch HGEMM kernel on square sizes. Results are shown on a Tesla V100 GPU using CUDA-9.1.

relatively small sizes, as shown in Figure 5. The same work also shows that extremely small matrices (e.g., with sizes ≤ 10) do not necessarily benefit from Tensor Core acceleration.

3.1.4. Error bounds. What can we say about rounding error bounds for low-precision BLAS? Hardware that employs low-precision matrix multiplication with accumulation at high precision, such as the the NVIDIA tensor cores, requires a careful analysis that takes account of the internal precisions and the matrix size. A general such analysis, which quantifies the gain from the use of higher precision accumulation, is given in Blanchard et al. (2020). A second question concerns the interaction of precision and dimension: an error bound with a constant nu (say) provides no information if $nu > 1$, such as when $n = 10^4$ and u is the unit roundoff for half precision (see Table 1). However, standard rounding error bounds are based on worst-case analyses. By making probabilistic assumptions about the rounding errors one can obtain bounds in which the problem size is replaced by its square root (Higham and Mary 2019), and even smaller constants can be obtained for data

with zero or small mean (Higham and Mary 2020). These analyses provide a theoretical explanation for the practical findings that acceptable accuracy can be obtained for large n and low precisions.

3.2. Newton's method

Newton's method is widely used for solving systems of nonlinear equations $F(x) = 0$, where $F : \mathbb{R}^n \rightarrow \mathbb{R}^n$. It takes the form:

$$x_{k+1} = x_k - F'(x_k)^{-1}F(x_k), \quad (3.1)$$

where F' denotes the Jacobian of F , which is assumed to be nonsingular. In practice, of course, the explicit inverse is not computed, but rather a system of equations is solved with $F'(x_k)$ as the coefficient matrix. Newton's method lends itself to an obvious mixed-precision implementation, whereby early iterations are carried out at low precision, and the precision is increased as the iteration converges. Mixed precision can also be used within a Newton step: $F(x_k)$ can be evaluated at a precision higher than the working precision to inject more information into the iteration. A detailed analysis of Newton's method in mixed-precision floating point arithmetic is given by Tisseur (2001). Our interest here is in using Newton's method as a tool for refining approximate solutions computed at low precision to the linear equations problem and the eigenvalue problem, as we will discuss in later sections.

3.3. Iterative refinement

A common approach to the solution of linear systems, either dense or sparse, is to perform an LU factorization of the coefficient matrix. 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, which results 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). Because of roundoff errors, the computed solution x carries a numerical error magnified by the condition number of the coefficient matrix A .

Iterative refinement attempts to improve \hat{x} by forming the residual $r = b - A\hat{x}$, solving $Ad = r$ using the LU factors, and then updating $\hat{x} \leftarrow \hat{x} + z$. This process is repeated as necessary. Iterative refinement is effectively Newton's method (3.1) applied to $F(x) = b - Ax$. The method goes back to the beginning of the digital computer era and has been analyzed by Wilkinson (1963), Moler (1967), Stewart (1973), and Higham (1997, 2002).

Iterative refinement inherently presents opportunities for a mixed-precision approach, and there is a wealth of existing work in this area. The three tasks—original solve/factorization, residual computation, and correction

equation solve—can 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.

We define various precisions. Assume that the data A , b , and the solution \hat{x} are stored in working precision u . We let u_ℓ represent the precision at which the LU factorization of A is computed and u_r represent the precision at which the residuals are computed. We also define u_s to be the precision at which the correction equation is effectively solved; in practice this will either be u or u_ℓ . A generic mixed-precision iterative refinement algorithm is given in Algorithm 3.1.

Algorithm 3.1. Mixed-precision iterative refinement.

```

1: Compute the LU factorization with partial pivoting
    $PA = LU$   $\triangleright (u_\ell)$ 
2: Solve for initial solution  $x_0$   $\triangleright (u_\ell)$ 
3: for  $k = 1, 2, \dots$  do
4:    $r_k \leftarrow b - Ax_{k-1}$   $\triangleright (u_r)$ 
5:   Solve  $Ad_k = r_k$   $\triangleright (u_s)$ 
6:    $x_k \leftarrow x_{k-1} + d_k$   $\triangleright (u)$ 
7:   Check convergence
8: end for

```

In Table 2 we give citations to work which analyzes iterative refinement in mixed precision. We list whether the analysis applies to LU or an arbitrary solver (“arb”) as well as whether the analyses of backward and/or forward error are normwise (“N”) or componentwise (“C”). We note that, if \hat{x} is an approximate solution of $Ax = b$, the normwise forward error is defined by

$$\|\hat{x} - x\|/\|x\|,$$

and the normwise backward error is defined by

$$\min\{\varepsilon : (A + \Delta A)\hat{x} = b, \|\Delta A\| \leq \varepsilon\|A\|\}$$

which can be evaluated as

$$\|r\|/(\|A\|\|\hat{x}\|),$$

where $r = b - A\hat{x}$.

Table 2. Summary of existing analyses of mixed-precision iterative refinement.

	Solver	F.E.	B.E.	Prec.
Moler (1967)	LU	N	—	$u \geq u_r$
Stewart (1973)	LU	N	—	$u \geq u_r$
Skeel (1980)	LU	C	C	$u \geq u_r$
Higham (1997)	arb	C	C	$u \geq u_r$
Tisseur (2001)	arb	N	N	$u \geq u_r$
Langou et al. (2006)	LU	N	N	$u_\ell \geq u = u_r$
Carson and Higham (2017)	arb	C	—	$u \geq u_r$
Carson and Higham (2018)	arb	C	C, N	$u_\ell \geq u_s \geq u \geq u_r$

The work in Langou et al. (2006) is notable as it suggests that with double (FP64) as the working precision, the factorization $PA = LU$ and the solution of the triangular systems $Ly = Pb$ and $Ux = y$ can be carried out in single precision (FP32). As a result, the only operation with computational complexity of $O(n^3)$ is handled in the lower precision, while all operations performed in working precision (triangular solves) and residual precision (residual and solution updates) are of at most $O(n^2)$ complexity. The coefficient matrix A is converted to factorization precision for the LU factorization, and the resulting factors are stored in factorization precision, while the initial coefficient matrix A needs to be kept in memory. Therefore, using FP32 as factorization precision for a FP64 problem, this approach increases the memory requirement by 50% compared to the standard algorithm.

Building on the work of Langou et al. (2006), Carson and Higham (2018) recently analyzed the general three-precision iterative refinement scheme presented in Algorithm 3.1. The analysis generalizes previously studied cases (e.g., the two-precision analyses of Higham, 1997; Langou et al., 2006), and also covers the case where three different precisions are used. It also allows for an arbitrary solver to be used in line 5 of Algorithm 3.1. For LU-based iterative refinement (i.e., when the solve in line 5 is done via triangular solve with the computed LU factors), it is shown that with the factorization done at half the working precision and the residual computed at double the working precision, forward and backward errors on the order of the working precision are still attainable, as long as the condition number of A is below some threshold. For example, as long as $\kappa_\infty(A) = \|A\|_\infty \|A^{-1}\|_\infty < 10^4$, then using FP16 for the $O(n^3)$ portion (the LU factorization) and (FP32, FP64) or (FP64, 128-bit, quadruple-precision floating point arithmetic (FP128)) as the (working, residual) precision 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. A full summary of the attainable componentwise forward errors and the normwise and componentwise backward errors for LU-based iterative refinement with various precision combinations can be found in Carson and Higham (2018, table 7.1).

When mixed precision is used, the method in Algorithm 3.1 can offer significant improvements for the solution of linear systems in many cases: if the low-precision computation is significantly faster than the high-precision computation, if the iterative refinement procedure converges in a small number of steps, and if the cost of each iteration is small compared with the cost of the system factorization. If the cost of each iteration is too high, then a high 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 vary substantially depending on the number of nonzeros and the matrix sparsity structure; this will be addressed in Section

4.1. In the dense case, results are more predictable. Note that the choice of the stopping criterion in the iterative refinement process is critical. For an analysis of convergence criteria see Demmel et al. (2006).

3.4. GMRES-IR

As mentioned, the analysis in Carson and Higham (2018) allows for a general solver for the correction equation within iterative refinement. Carson and Higham (2017) first proposed the use of the GMRES method (Saad and Schultz, 1986) preconditioned by the FP16 LU factors as the solver in the correction equation. This variant is called GMRES-IR by Carson and Higham. Algorithmically, it follows the same structure as Algorithm 3.1, except line 5 is performed via preconditioned GMRES. A summary of results for various precision combinations can be found in Carson and Higham (2018, table 8.1). It is shown that in this case, the constraint on the condition number can be relaxed compared to the LU-based refinement scheme. For example, with factorization precision FP16, working precision FP32, and residual precision FP64, we can expect forward and backward errors on the order of FP32 as long as $\kappa_\infty(A) < 10^8$. We refer to Higham (2019, table 3.1) for limiting forward and backward errors for this GMRES-based approach when two precisions are used, with the residual precision equal to the working precision.

The idea behind GMRES-IR is that even though the low-precision LU factors may be of low quality, they can still be effective preconditioners in using the GMRES method to solve the correction equation $Ad_k = r_k$, resulting in an effective solve precision $u_s = u$. The condition number of the resulting preconditioned system is reduced enough to guarantee backward stability of the approximate solution computed by GMRES even for matrices that are nearly numerically singular with respect to the working precision. In contrast, using a basic triangular solve with the low-precision LU factors to solve $Ad_k = r_k$, for which $u_s = u_\ell$, provides no degree of relative accuracy once $\kappa_\infty(A)$ exceeds u_ℓ^{-1} . Using preconditioned GMRES, we can still guarantee that the solution of the correction equation has some correct digits and a residual at the level of the convergence tolerance requested by the algorithm despite the apparent low quality of the computed preconditioners.

Since this paper focuses on the practical usage and possible performance gains rather than error analysis, we point the reader to Higham (2002), Carson and Higham (2017, 2018), and Higham (2019) for detailed error analysis of both standard iterative refinement and GMRES-IR. Of course, in order to be beneficial, it is necessary that the total number of GMRES iterations and the total number of refinement steps remains small. As shown in Carson and Higham (2017, 2018), this is indeed the case for many problems.

We note that the HPL-AI mixed-precision benchmark,² which is designed to take into account the availability of hardware accelerators for low-precision computation, is based on GMRES-IR.

3.4.1. 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.³

We consider a two-sided diagonal scaling prior to converting to FP16: A is replaced by RAS , where:

$$T = \text{diag}(t_i), \quad S = \text{diag}(s_i), \quad t_i, s_i > 0, \quad i = 1 : n.$$

Such scaling algorithms have been developed in the context of linear systems and linear programming problems. In contrast to previous studies (see Elble and Sahinidis, 2012), 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-precision or double-precision matrix into half precision, with a particular aim to use the resulting half-precision LU factors for iterative refinement.

Higham et al. (2019) propose the use of two-sided diagonal scaling given in Algorithm 3.2. Recall that x_{\max} denotes the largest finite floating point number (see Table 1).

Algorithm 3.2. (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: $A^{(h)} = fl_{\ell}(\mu(RAS))$
-

For FP16, in light of the narrow range, we will also multiply the shifted matrix by a scalar to bring it close to the overflow level x_{\max} and to minimize the chance of underflow and of subnormal numbers being produced.

Higham et al. (2019) recommend two different algorithms for determining R and S ; both algorithms are carried out at the working precision. The first option is row and column equilibration, which ensures that every row and column has the 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 (Anderson et al., 1999). The second option, for symmetric matrices, is a symmetry-preserving two-sided scaling proposed by Knight et al. (2014). The algorithm is iterative and scales simultaneously on both sides rather than sequentially on one side and then the other.

For more details on scaling see Higham et al. (2019), Higham and Pranesh (2021), and Carson et al. (2020).

3.5. Low-precision cholesky factorization

In the previous section, we considered general matrices. We now assume that we are given a symmetric positive definite matrix $A \in \mathbb{R}^{n \times n}$ in precision u and wish to compute a Cholesky factorization at precision $u_{\ell} > u$ for use in iterative refinement. The most practically important cases are where $(u_{\ell}, u) = (\text{half}, \text{single}), (\text{half}, \text{double}),$ or $(\text{single}, \text{double})$. The obvious approach is to form $A^{(\ell)} = fl_{\ell}(A)$, where fl_{ℓ} denotes the operation of rounding to precision u_{ℓ} , and then compute the Cholesky factorization of $A^{(\ell)}$ in precision u_{ℓ} . However, this approach 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, $A^{(\ell)}$ can fail to be (sufficiently) positive definite, because a matrix where the smallest eigenvalue is safely bounded away from zero with respect to single precision or double precision can become numerically indefinite under rounding to half precision. The second issue can also arise when a double-precision matrix is rounded to single precision. To overcome these problems, Higham and Pranesh (2019) propose scaling and shifting.

3.5.1. Scaling. The first step is to scale the matrix A to the unit diagonal matrix $H = D^{-1}AD^{-1}$, $D = \text{diag}(a_{ii}^{1/2})$, and D will be kept at precision u . Cholesky factorization is essentially numerically invariant under two-sided diagonal scaling, so 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 FP32, it is not usually necessary to scale, and we can work with A throughout.

3.5.2. Shifting. We now convert H to the lower precision u_{ℓ} , incorporating a shift to ensure that the lower precision matrix is sufficiently positive definite for Cholesky factorization to succeed. We shift H by $c_n u_{\ell} I$, where c_n is a positive integer constant, to obtain $G = H + c_n u_{\ell} I$. 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 and then round or round and then shift in precision u_{ℓ} .

Our final precision- u_{ℓ} matrix is constructed as:

$$G = H + c_n u_{\ell} I, \quad \beta = 1 + c_n u_{\ell}, \quad \mu = \theta x_{\max} / \beta, \quad (3.2)$$

$$A^{(h)} = fl_{\ell}(\mu G),$$

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 θx_{\max} . Note also that since the growth factor for Cholesky factorization is 1 (see Higham, 2002, Problem 10.4), there is no danger of overflow during Cholesky factorization of $A^{(h)}$.

Higham and Pranesh (2021, Section 3.3) provide analysis suggesting the choice of c_n . A pragmatic approach is to take c_n to be a small constant, and if the Cholesky

factorization fails, increase c and try again. Based on this, we present the low-precision Cholesky factorization algorithm in Algorithm 3.3.

Algorithm 3.3. (Low-precision Cholesky factorization.) 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_ℓ , where $D = \text{diag}(a_{ii}^{1/2})$. The scalar $\theta \in (0, 1]$ and the positive integer c are parameters.

-
- 1: $D = \text{diag}(a_{ii}^{1/2})$, $H = D^{-1} A D^{-1}$
 - 2: $G = H + cu_\ell I$
 - 3: $\beta = 1 + cu_\ell$
 - 4: $\mu = \theta x_{\max} / \beta$
 - 5: $A^{(h)} = f_{l_\ell}(\mu G)$
 - 6: Attempt Cholesky factorization $A^{(h)} = R^T R$ in precision u_ℓ .
 - 7: **if** Cholesky factorization failed **then**
 - 8: $c \leftarrow 2c$, goto line 2
 - 9: **end if**
-

3.6. Mixed-precision factorizations

Haidar et al. (2018b) proposed iterative refinement methods using mixed-precision factorizations. While classical iterative refinement and extensions like the GMRES-IR use fixed-precision factorizations (e.g., in precision u_ℓ as illustrated in Algorithm 3.1), mixed-precision factorizations apply higher precision (e.g., u_w) at critical parts of the algorithm to obtain more accurate factorizations while retaining the performance of the low-precision counterpart.

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 one to easily overcome implementation issues and other limitations of using FP16 arithmetic and harness the power of specialized hardware (e.g., Tensor Cores) for a larger range of scientific computing applications.

The developments were applied to GPU Tensor Cores and illustrate that FP16 can be used to obtain 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 $4\times$ faster than an optimized double-precision solver (Haidar et al., 2017, 2018a, 2018b, 2020).

A building block for the mixed-precision factorizations is mixed-precision BLAS. Having mixed-precision BLAS can ease the development of many mixed-precision LAPACK algorithms. Currently, cuBLAS provides a mixed FP32-FP16 precision HGEMM that uses the GPU's Tensor Cores for FP16 acceleration. In this GEMM, the input matrices A and B can be FP32, be internally cast to FP16, used to compute a GEMM on Tensor Cores in full (FP32) accuracy, and then the result is stored back on the GPU memory in FP32. There are two main benefits to

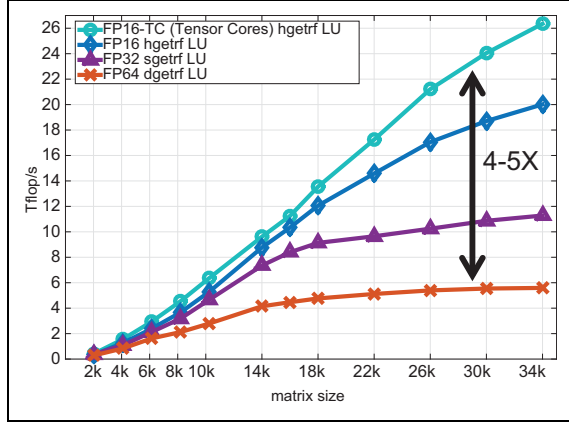


Figure 6. Mixed-precision LU (hgetrf) in MAGMA and its speedup vs. FP64 LU.

having such mixed-precision BLAS routines. First, note that this mixed-precision HGEMM is almost as fast as the non-mixed FP16 HGEMM (Figure 3). Second, the use of mixed-precision gains about one more decimal place of accuracy (Figure 4).

Aside from the two main benefits outlined above, the availability of mixed-precision GEMMs also enables us to easily develop other mixed-precision algorithms (e.g., LAPACK), including the various mixed-precision factorizations that we recently added in MAGMA (Haidar et al., 2018b). Figure 6 shows the performance of the mixed-precision LU (marked as “FP16-TC hgetrf LU”). Note that this factorization is about $4\times$ – $5\times$ 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 7 shows the mixed-precision iterative refinement in MAGMA (Haidar et al. 2018b), which uses a backward error criterion for convergence. The $4\times$ overall acceleration is due to a number of optimizations. First, note that the 3 iterations to get to FP64 accuracy led to a loss of about 2 TFLOP/s compared with the hgetrf performance (24 TFLOP/s vs. 26 TFLOP/s) (i.e., the overhead of one iteration can be deduced as being about 2%). Losing 75% (e.g., through up to 40 iterations) would lead to no acceleration compared to the FP64 solver. This overhead per iteration is very low, owing to fusing all data conversions with computational kernels. Without fusion, the overhead would have been easily about $3\times$ higher. Second, note that iterative refinement using the mixed-precision factorization has less than half of the overhead in terms of iterations to solution (three vs. seven 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.

Using mixed-precision Cholesky factorization in Algorithm 3.3, Abdelfattah et al. (2020) obtain speedups of up to 4.7 over a double-precision solver on an NVIDIA V100.

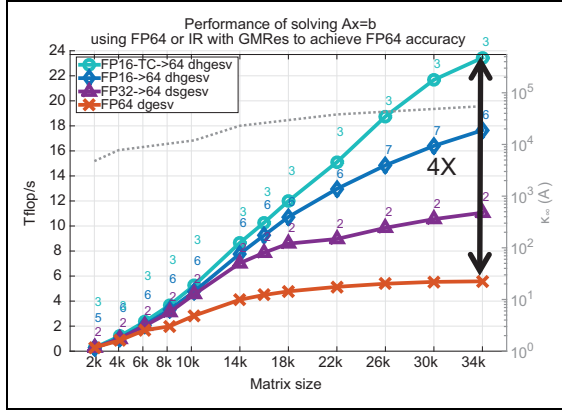


Figure 7. Mixed-precision iterative refinement in MAGMA and acceleration vs. FP64 solvers. Note $\approx 2\%$ overhead per iteration, and less than half the overhead in terms of iterations for mixed-precision LU vs. regular FP16 LU (the three vs. seven iterations until FP64 convergence). The condition numbers of the matrices are computed using FP64.

3.7. 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 $m \geq n$ having full rank. The idea of mixed-precision iterative refinement and GMRES-IR for square linear systems can be adapted to the least squares case. Least squares problems may be ill-conditioned in practice, and so rounding errors may result in an insufficiently accurate solution. In this case, iterative refinement may be used to improve accuracy, and it also improves stability.

3.7.1. Cholesky-based approach. The normal equations method solves:

$$A^T A x = A^T b$$

using the Cholesky factorization of $A^T A$ (Section 3.5). In general, this method is not recommended unless A is very well conditioned because it has a backward error bound of order $\kappa_2(A)u$ (Higham, 2002, sect. 20.4), and the Cholesky factorization can break down for $\kappa_2(A) > u^{-1/2}$, where $\kappa_2(A)$ is the ratio of the largest to the smallest singular value of A . Higham and Pranesh (2019) assume that A is well conditioned and propose the Cholesky and GMRES-IR-based least squares solver given in Algorithm 3.4.

Line 1 of Algorithm 3.4 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 3.3, but here the matrix C is already well scaled and in precision u_ℓ , so there is no need to scale C to have unit diagonal.

Note that although Algorithm 3.4 explicitly forms $C = B^{(h)T} B^{(h)}$ in Algorithm 3.4, C is used to form a preconditioner, so the usual problems with forming a

Algorithm 3.4. (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 $S = \text{diag}(1/\|a_j\|_2)$, with a_j the j th column of A .
- 2: $\mu = \theta x_{\max}$
- 3: $B^{(h)} = fl_\ell(\mu^{1/2} B)$
- 4: Compute $C = B^{(h)T} B^{(h)}$ in precision u_ℓ .
- 5: Compute the Cholesky factorization $C + cu_\ell \text{diag}(c_{ii}) = R^T R$ in precision u_ℓ .
- 6: **if** Cholesky factorization failed **then**
- 7: $c \leftarrow 2c$, goto line 5
- 8: **end if**
- 9: Form $b^{(h)} = fl_\ell(SA^T b)$.
- 10: Solve $R^T R y_0 = b^{(h)}$ in precision u_ℓ and form $x_0 = \mu S y_0$ at precision u .
- 11: **for** $i = 0 : i_{\max} - 1$ **do**
- 12: Compute $r_i = A^T(b - Ax_i)$ at precision u_r and round r_i to precision u .
- 13: Solve $MA^T A d_i = M r_i$ 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**

cross-product matrix (loss of significance and condition squaring) are less of a concern. Also note that 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. (2020).

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

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

so we are preconditioning with an approximation to the inverse of $A^T A$. For large n , as long as GMRES converges quickly, the cost of the refinement stage should be negligible compared with the cost of forming $A^T A$ and computing the Cholesky factorization.

We also mention the Cholesky-QR algorithm for computing a QR factorization $A = QR$. It forms the cross-product matrix $A^T A$, computes the Cholesky factorization $A^T A = R^T R$, and then obtains the orthogonal factor Q as $Q = AR^{-1}$; this process can be iterated for better numerical stability (Fukaya et al., 2020). Mixed precision can be exploited in this algorithm, as shown by Yamazaki et al. (2015, 2016).

3.7.2. Augmented matrix approach. As mentioned, the Cholesky-based approach described in the previous section is intended only for the case where the matrix is very well conditioned. Another approach to mixed-precision least-squares iterative refinement with a less stringent requirement on the condition number is presented by Carson et al. (2020). This approach is based on 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 $x = R^{-1} Q_1^T b$ with residual $\|b - Ax\|_2 = \|Q_2^T b\|_2$.

An iterative refinement approach for least squares systems was suggested by Björck (1967a). Refinement is performed on the augmented system

$$\begin{bmatrix} I & A \\ A^T & 0 \end{bmatrix} \begin{bmatrix} r \\ x \end{bmatrix} = \begin{bmatrix} b \\ 0 \end{bmatrix}, \quad (3.3)$$

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 (1967a) shows that this augmented 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 $u_r \leq u$ 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 Björck (1990) and Higham (2002, sect. 20.5) and the references therein.

Carson et al. (2020) show that the three-precision iterative refinement approach of Carson and Higham (2018) can be applied in this case; the theorems developed in Carson and Higham (2018) for 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 Carson et al. (2020) also extends the GMRES-based refinement scheme of Carson and Higham (2017) 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 Carson et al. (2020) for further details.

For details of convergence tests for iterative refinement of least squares problems see Demmel et al. (2006).

3.8. Eigenvalue problems

Newton's method can be used to refine an approximate eigenpair of a matrix by defining a function $F: \mathbb{R}^{n+1} \rightarrow \mathbb{R}^{n+1}$ that has as its first n components $(A - \lambda I)x$ and its last component $e_s^T x - 1$ for some unit vector e_s , with this last component serving to normalize x . If an initial eigen decomposition is available, it can be exploited to simplify the implementation of the Newton iteration. This idea was developed by Dongarra (1982) and Dongarra et al. (1983), building on a Schur decomposition and allowing the residual $(A - \lambda I)x$ to be computed in higher precision. Algorithm 3.5 implements this procedure, called SICE, which, in each iteration, solves a linear system resulting from a rank-1 update in order to refine a single eigenpair. 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 (Dongarra 1982) 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.

This idea has been extended to the generalized eigenvalue problem by Tisseur (2001) and Davies et al. (2001).

Algorithm 3.5. SICE algorithm for iteratively refining computed eigenvalue.

```

1: function  $[x, \lambda] \leftarrow \text{SICE}(A, x_0, \lambda_0)$ 
2:    $[Q, T] \leftarrow \text{schur}(A)$   $\triangleright$  Schur decomposition to find
    $A = QTQ^T$  where  $T$  is upper quasitriangular.
3:    $[m, s] \leftarrow \max(x_0)$   $\triangleright$  Find maximum value and
   index in the eigenvector.
4:    $x_0 \leftarrow x_0/m$   $\triangleright$  Normalize
5:   for  $i = 1, 2, \dots$  do
6:      $c \leftarrow -x_{i-1} - (A - \lambda_{i-1}I)[:, s]$   $\triangleright$  Column  $s$  of
        $A - \lambda_{i-1}I$ 
7:      $d \leftarrow Q^T c$ 
8:      $f \leftarrow e_s^T Q$   $\triangleright$  Row  $s$  of  $Q$ 
9:     Solve the rank-1 updated system
        $Q(T - \lambda_{i-1}I + df^T)Q^T y_i = Ax_{i-1} - \lambda_{i-1}x_{i-1}$ 
10:     $\lambda_i \leftarrow \lambda_{i-1} + y_i[s]$   $\triangleright$  Eigenvalue correction.
11:     $x_i \leftarrow x_{i-1} + y_i$   $\triangleright$  Eigenvector correction.
12:     $x_i[s] \leftarrow x_{i-1}[s]$   $\triangleright$  Restore  $x[s]$ .
13:    if  $2 \times y_i[s] > y_{i-1}[s]$  then
14:      Break from for loop.
15:    end if
16:  end for
17:   $x \leftarrow x_i$ 
18:   $\lambda \leftarrow \lambda_i$ 
19: end function

```

Algorithm 3.6. Iterative refinement for symmetric eigenvalue problem.

```

1: Input:  $A = A^T \in \mathbb{R}^{n \times n}$ ,  $\hat{X} \in \mathbb{R}^{n \times \ell}$ ,  $1 \leq \ell \leq n$ 
2: Output:  $X' \in \mathbb{R}^{n \times \ell}$ ,  $\tilde{D} = \text{diag}(\tilde{\lambda}_i) \in \mathbb{R}^{\ell \times \ell}$ ,
    $\tilde{E} \in \mathbb{R}^{\ell \times \ell}$ ,  $\omega \in \mathbb{R}$ 
3: function  $[X', \tilde{D}, \tilde{E}, \omega] \leftarrow \text{REFSYEV}(A, \hat{X})$ 
4:    $R \leftarrow \mathbb{I}_n - \hat{X}^T \hat{X}$ 
5:    $S \leftarrow \hat{X}^T A \hat{X}$ 
6:    $\tilde{\lambda}_i \leftarrow s_{ii} / (1 - r_{ii})$  for  $i = 1, \dots, \ell$   $\triangleright$  Compute
     approximate eigenvalues.
7:    $\tilde{D} \leftarrow \text{diag}(\tilde{\lambda}_i)$ 
8:    $\omega \leftarrow 2 \left( \|S - \tilde{D}\|_2 + \|A\|_2 \|R\|_2 \right)$ 
9:    $e_{ij} \leftarrow \begin{cases} \frac{s_{ij} + \tilde{\lambda}_j r_{ij}}{\tilde{\lambda}_j - \tilde{\lambda}_i} & \text{if } |\tilde{\lambda}_i - \tilde{\lambda}_j| > \omega \\ r_{ij}/2 & \text{otherwise} \end{cases}$  for  $1 \leq i, j \leq \ell$ 
      $\triangleright$  Compute the entries of the refinement
     matrix  $\tilde{E}$ .
10:   $X' \leftarrow \hat{X} + \hat{X} \tilde{E}$   $\triangleright$  Update  $\hat{X}$  by  $\hat{X}(\mathbb{I}_n + \tilde{E})$ 
11: end function

```

For the symmetric eigenvalue problem, Petschow et al. (2014) use extra precision to improve the accuracy of the multiple relatively robust representations (MRRR) algorithm, with little or no performance penalty.

Algorithm 3.6 shows another iterative refinement procedure from Ogita and Aishima (2018) for solving a symmetric eigenvalue problem. This method also succeeds for clustered eigenvalues (Ogita and Aishima, 2019). Lines 4, 5, and 10 represent the compute-intensive parts of the algorithm, which amounts to 4 calls to the matrix-matrix multiply function xGEMM. Line 8 uses the 2-norm, but it is recommended to approximate 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 the matrix norm. Even though the algorithm may be applied for only a subset of ℓ eigenvectors, the refinement iterations are confined to the corresponding subspace and its approximation provided on input. Hence, the refinement process could be limited: the desired accuracy might be unattainable if only a part of the spectrum is refined in higher precision and the subspace spanned by input approximate eigenvectors does not cover the real eigenvector close enough. It is designed to be applied on the clustered eigenvalues after the eigenspace has been accurately captured. The clustered eigenvalues' subspace is not very sensitive to perturbations, provided that the gap between the clustered eigenvalues and all the others is sufficiently large. Thus, eigenvalue gaps must be expanded in each clustering subproblem by using diagonal shifts as proposed by Ogita and Aishima (2019) in addition to a modification for improved orthogonality of the refined eigenvectors. Figure 8 shows the convergence behavior of Algorithm 3.6 on a real symmetric

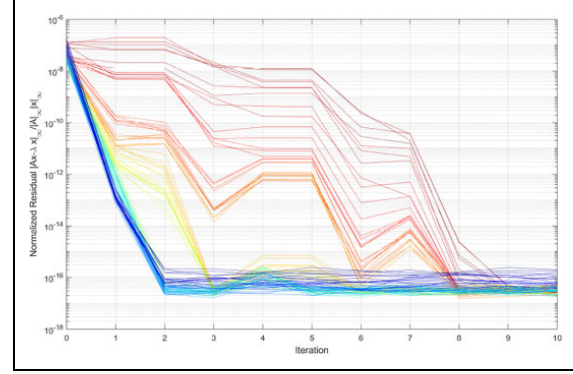


Figure 8. Convergence of eigenvalue refinement using Algorithm 3.6 from single to double precision for a real symmetric matrix of size $n = 100$. The matrix is generated with random eigenvectors and geometrically distributed eigenvalues between 1 (blue) and 10^{-5} (red).

matrix of size $n = 100$ when refining the entire spectrum from single precision toward double precision. The matrix is generated with random eigenvectors and geometrically distributed eigenvalues between 1 and 10^{-5} . Each line represents the convergence of one eigenvalue, and the normalized residual $\|Ax - \lambda x\| / \|A\| \|x\|$ is plotted against subsequent iteration numbers. The color indicates the value of eigenvector from blue as 1 toward red as 10^{-7} . Here we can see the condition of solving the eigenvector is related to the gap between the eigenvalue and its neighborhoods. The blue ones are well conditioned and can converge toward double-precision accuracy in two iterations while the red ones requires up to 8 iterations. This is an important computational aspect of the convergence rate. The larger the iteration count, the more time will be spent in line 4, 5, and 10 of Algorithm 3.6 that have to use higher precision matrix product.

4. Sparse linear algebra

In contrast to dense linear algebra, sparse linear algebra operations are typically memory bound, and the primary goal of mixed-precision sparse linear algebra algorithms is to reduce the memory access and communication volume. However, for sparse direct solvers, blocking strategies and fill-in during the factorization can result in matrices for which the factorization step becomes compute bound. This makes mixed-precision iterative refinement strategies like those presented in Section 3.3 also attractive for sparse linear systems (see Section 4.1). For 2-D and 3-D problems, the matrix representation of a finite element discretization has a larger matrix bandwidth, and the fill-in arising in the factorization often exceeds the hardware capabilities in terms of memory and computational cost. Consequently, many applications rely on iterative solvers and preconditioned iterative solvers to handle sparse linear systems. Mixed-precision strategies to accelerate iterative sparse linear algebra methods utilize components that are not critical

to the final accuracy (e.g., preconditioners or individual operations in a larger algorithm) in lower precision than working precision, or trade low-precision memory access against additional iteration steps. In Section 4.2, we present a theoretical analysis of the rounding effects low-precision computations have on the accuracy of Krylov solvers. However, as previously mentioned, it is usually not the arithmetic computations that limit the performance of iterative algorithms for sparse problems, but rather it is the communication and memory bandwidth. In Section 4.3, we present the idea of radically decoupling the format used for arithmetic computations from the format that is used for communication and memory operations. This concept can span from using lower precision for memory accesses to using dedicated compression techniques before invoking communication operations. Examples of how this concept of format decoupling and compression helps accelerate sparse linear algebra include preconditioners for iterative solvers (Section 4.4) and multigrid methods (Section 4.5).

4.1. Mixed-precision sparse direct solvers

The factorization process of a sparse matrix usually generates fill-in elements, significantly increasing the number of nonzero elements in the factors. The fill-in is usually structured, and the fill-in locations can be predicted from the sparsity pattern of the original system matrix. To improve performance and memory efficiency, factorization-based sparse solvers typically operate in a block-sparse fashion: forming blocks covering the nonzero elements reduces the indexing information to index the blocks, and storing the elements as small dense blocks allows for the application of highly efficient dense linear algebra operations. There exist two options for realizing the concept of block-sparse factorizations. One is to convert the system matrix into a block-sparse matrix prior to the factorization process. The other, more popular, one is based on forming the dense blocks “on-the-fly” in registers/fast memory during the factorization process, **(block-) sparse factorization**:

1. gather the values from sparse data structures into dense blocks in registers/fast memory;
2. invoke dense linear algebra kernels on dense blocks; and
3. scatter results into the sparse output data structure.

Similar to dense linear solvers, sparse direct solvers can also benefit from the mixed-precision iterative refinement framework presented in Section 3.3: The (block-) sparse factorization is computed in low precision, thereby leveraging the corresponding high compute power, and the iterative refinement process recovers a high-accuracy solution. Contrary to the dense case, the low-precision (block-) sparse factorizations not only benefit from higher arithmetic performance in the invocation of the compute-bound dense linear algebra kernels, but they also benefit

from the reduced memory access volume in the memory-bound gather and scatter operations.

The iterative refinement process for recovering high-precision solutions for a sparse linear system is conceptually identical to the dense case: like in Algorithm 3.1, an error correction equation computed in high precision is solved using the low-precision factorization. However, the triangular factors are (block-) sparse, and for solving the triangular systems, the same strategy of gathering data from the sparse data structures into contiguous memory proves successful,

(block-) sparse triangular solve:

1. gather the values from the sparse triangular structures into dense blocks in registers / fast memory and
2. invoke dense linear algebra kernels to solve for the right-hand side.

Again, gathering the data in dense blocks enables the use of efficient dense linear algebra kernels. Using low precision, the memory-bound “gather” step benefits from a reduced memory access volume, and the dense linear algebra kernels benefit from the higher performance in low precision.

For dense linear algebra, the performance benefits of mixed-precision iterative refinement over high-precision dense direct solvers mostly correlate with the hardware-specific arithmetic performance limits in the different precision formats. In particular, the performance benefits are mostly independent of the problem characteristics. This is different when using mixed-precision iterative refinement for the direct solution of sparse problems, as the matrix structure determines the amount and structure of the fill-in, the efficiency of the dense linear algebra kernels operating on the induced dense blocks, and the ratio between memory operations and arithmetic operations. As a result, it is much harder to predict whether the mixed-precision iterative refinement variant of a sparse direct solver provides performance benefits over the execution of a sparse direct solver in high precision.

4.2. Mixed-precision Krylov solvers

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 both the Arnoldi-QR algorithm with the modified Gram-Schmidt implementation of the GMRES Krylov subspace method for iteratively solving linear systems of equations as well as Flexible GMRES (FGMRES). The emphasis here is to examine the approaches employed to date that incorporate mixed-precision floating point arithmetic to speedup computations while retaining some or all of the numerical properties of the original algorithms in FP64 arithmetic (i.e., representation error and loss of orthogonality).

4.2.1. Lanczos-CG. First, we briefly summarize 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 also note that the literature on finite precision behavior of Lanczos-based methods is expansive, and we cannot hope to fully describe it here. For a more thorough account and historical references, we point the reader to the survey of Meurant and Strakoš (2006).

Fundamental relations dealing with the loss of orthogonality and other important quantities in finite precision Lanczos have been derived by Paige (1980). These results were subsequently used by Greenbaum to prove backward stability-like results for the CG method (Greenbaum, 1989); namely, Greenbaum showed that CG in finite precision can be seen as an 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, that is, the limiting value of $\|x_k - x\| / \|x\|$ for approximate solutions x_k and true solution x , in CG and other “recursively computed residual methods” (Greenbaum, 1997). The results of Paige and Greenbaum have also been extended to s -step Lanczos/CG variants in Carson (2015), where it is shown that s -step Lanczos in finite precision behaves like a 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.

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 Simoncini and Szyld, 2003; van den Eshof and Sleijpen, 2004). 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.

For comprehensiveness, 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. (2010), which uses mixed-precision CG and BICGSTAB methods to implement the “reliable update” strategy of Sleijpen and van der Vorst (1996) 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 (FP64) precision only in the true residual computations and group updates (and FP32 or FP16 for the rest of the computation), the authors claim they are able to achieve FP64 accuracy. This deserves further theoretical study, which we believe can be achieved by extending the results in Sleijpen and van der Vorst (1996) and the related work of Van Der Vorst and Ye (2000) to the mixed-precision setting.

4.2.2. Flexible GMRES. Much work has been done involving the use of lower precision preconditioners within iterative solvers, in particular, GMRES and FGMRES run in a higher precision.

Arioli and Duff (2009) rigorously prove, that using a triangular factorization computed as a preconditioner in FP32, FGMRES run in FP64 produces a solution with backward error to FP64 accuracy. In contrast, they demonstrate that using FP64 iterative refinement as the solver may fail in such cases. They provide numerical experiments which support their theoretical analysis. This builds on the previous work of Arioli et al. (2007), in which it is proved that FGMRES is backward stable.

Building on the work of Arioli and Duff (2009), Hogg and Scott (2010) develop a single-precision (FP32) implementation of an LDL^T factorization method for solving sparse-symmetric linear systems. This FP32 factorization is then used as a preconditioner within FP64 iterative solvers, including iterative refinement and FGMRES, effectively creating a mixed-precision solver. They demonstrate that for linear systems that are sufficiently well conditioned, the mixed-precision approach was sufficient for obtaining FP64 accuracy; the remaining cases required a full FP64 implementation. Additionally, it is shown that the mixed-precision approach is beneficial in terms of performance for sufficiently large problems.

4.2.3. Arnoldi-QR MGS-GMRES. For MGS-GMRES the mixed-precision work by Gratton et al. (2020) is the most recent and appropriate—and in particular the loss-of-orthogonality relations due to Björck (1967b) and Paige (1980), later refined by Paige et al. (2006), are employed in order to provide tolerances for mixed FP32–FP64 computations. MGS-GMRES convergence stalls (the normwise

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 (Paige 2018) is derived from the lower triangular T matrix appearing in the rounding error analyses by Giraud et al. (2004).

To summarize, Gratton et al. (2020) postulate 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 nonsymmetric matrix A produces the matrix factorization, with loss of orthogonality F_k .

$$AV_k = V_{k+1} H_k, \quad V_{k+1}^T V_{k+1} = I + F_k \quad (4.4)$$

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 $O(\varepsilon)\kappa(A)$, as was derived by Björck (1967b) and $O(\varepsilon)\kappa([r_0, AV_k])$ for Arnoldi-QR—which is described in Paige and Strakoš (2002), Paige et al. (2006), and related work. The inexact inner products are given by:

$$h_{ij} = v_i^T w_j + \eta_{ij}, \quad (4.5)$$

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

$$[r_0, AV_k] = V_{k+1} [\beta e_1, H_k]. \quad (4.6)$$

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

$$F_k = \bar{U}_k + \bar{U}_k^T, \quad U_k = \begin{bmatrix} v_1^T v_2 & \cdots & v_1^T v_{k+1} \\ & \ddots & \\ & & v_k^T v_{k+1} \end{bmatrix} \quad (4.7)$$

Define the matrices as below.

$$N_k = \begin{bmatrix} \eta_{11} & \cdots & \eta_{1k} \\ & \ddots & \\ & & \eta_{kk} \end{bmatrix} R_k = \begin{bmatrix} h_{21} & \cdots & h_{2k} \\ & \ddots & \\ & & h_{k+1,k} \end{bmatrix} \quad (4.8)$$

The loss of orthogonality relation, derived by Björck (1967b) for the $A = QR$ factorization via the modified Gram-Schmidt algorithm, can be applied to the Arnoldi-QR algorithm to obtain:

$$N_k = -[0, U_k] H_k = -U_k R_k. \quad (4.9)$$

The complete loss of orthogonality (triggers a loss of 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 normwise relative backward error reaches $O(\varepsilon)$. Gratton et al. (2020) show how to maintain sufficient orthogonality to achieve a desired relative residual error level by switching the inner products from FP64 to FP32 at certain tolerance levels and combine this with inexact matrix-

vector products as in van den Eshof and Sleijpen (2004) and Simoncini and Szyld (2003).

In practice, the restarted variant of GMRES is often employed to reduce memory requirements. The algorithm produces both implicit (iteratively computed) and explicit residuals. Thus, we might ask whether either can be performed in reduced precision. The work described herein on iterative refinement by Carson and Higham for mixed precision can be applied to analyze 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 normwise backward error close to FP64 machine roundoff.

4.2.4. Alternative approaches. Although somewhat outside the scope of this review, we can demonstrate that it is possible to modify the Gratton et al. (2020) analysis based on the inverse compact WY form of the MGS algorithm, introduced by Świrydowicz et al. (2020). 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 a single-precision (FP32) $L_{k-1,1:k-2} = (Q_{1:k-2}^T q_{k-1})^T$ and an FP64 triangular solve $r_j = (I + L_{j-1})^{-1} Q_{j-1}^T a_j$ to update R , as this would directly employ the forward error analysis of Higham (1989). 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 FP32.

Evidence for maintaining orthogonality is provided in Figure 9, with $\|I - Q^T Q\|$ plotted for $A = QR$ using the inner products in standard MGS (blue) in FP64 versus the

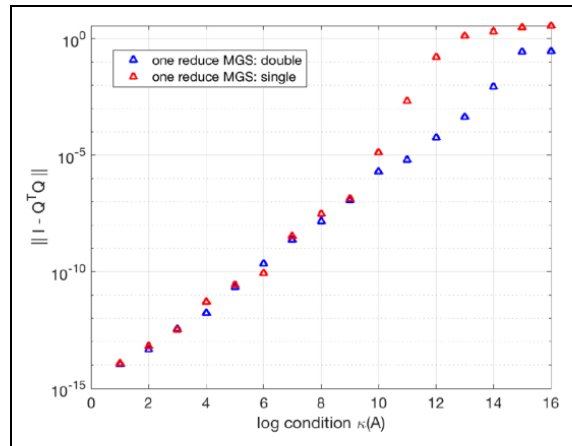


Figure 9. Loss of orthogonality for mixed single-double MGS algorithm.

inverse compact WY MGS (red) with $Q_{j-1}^T q_{j-1}$ in FP32 (simulated in MATLAB), and we observe at least the same or slightly higher error levels. The x -axis is the log condition number for randomly generated matrices. The lower triangular solve is computed in FP64.

Barlow (2019) contains similar if not the same algorithm formulations in block form. His work is related to Björck’s 1994 paper (Björck 1994, 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 T for the compact WY form of MGS, Malard and Paige (1994) 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 $O(\varepsilon)\kappa^2(A)$ loss of orthogonality. An interesting observation from Leon et al. (2013) 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 et al. (2013).

Barlow (2019) employs the Householder compact WY representation of reflectors and also refers to the work of Puglisi (1992)—discussed in Joffrain et al. (2006)—and this is referred to as the “inverse compact WY” representation of Householder; this originally comes from Walker’s work on Householder GMRES Walker (1988). Barlow then extends this approach to the block compact WY form of MGS; see also the technical report by Sun (1996). The contribution by Świrydowicz et al. (2020) was to note that there exists an inverse compact WY representation for MGS—having the projector P with lower triangular correction matrix T :

$$\begin{aligned} P &= I - Q_{j-1} T Q_{j-1}^T \\ &= I - Q_{j-1} (I + L_{j-1})^{-1} Q_{j-1}^T \end{aligned}$$

—and to “lag” the norm $\|q_{j-1}\|_2$ so that these can be computed in one global reduction. Barlow (2019) makes this connection for blocks, and in effect this is given in his equation (3.10), and references (Puglisi, 1992).

Björck and Paige (1992) made the link between Householder and MGS based on the observation made by Shefffield. Paige defines this to be augmentation, and Gratton et al. (2020) also references this work. Paige has also recently extended these augmentation ideas to Lanczos. The T matrix appears in Paige and Wüling (2014) and then later in Paige (2018) 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 et al. (2004); Langou also worked with Smoktunowicz et al. (2006) 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 FP32, but then we would still have to form $Q_{j-1}^T a_j$, and store Q_{j-1} in FP64. By examining the cost trade-offs a bit further, we can instead use a form of re-

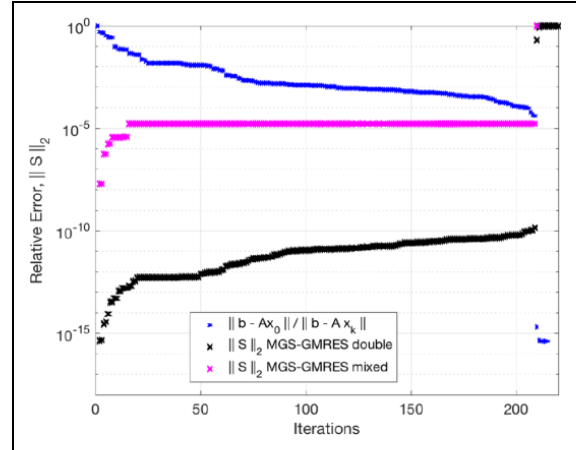


Figure 10. GMRES residuals and loss of orthogonality $\|S\|_2$ for `impcol_e` matrix.

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, displayed in Figure 10, demonstrate this works well, driving the relative residual and, more importantly, the normwise relative backward error to $O(\varepsilon)$ in FP64, with orthogonality maintained to $O(\varepsilon)$ in FP32 as indicated by the magenta curve. Here, the black curve is the FP64 loss of orthogonality metric given by $\|S\|_2$.

The representation error (backward error) for $A + E = QR$ computed by MGS is not affected by FP32 inner products and remains $O(\varepsilon)$. We are not aware of whether or not this was previously known.

4.3. Memory format decoupling

We already elaborated on sparse linear algebra operations being memory bound across the complete hardware technology food chain. Additionally, we are witnessing a widening gap between the compute power (in terms of FLOP/s) on the one side and the communication power (in terms of memory bandwidth) on the other. 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. 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

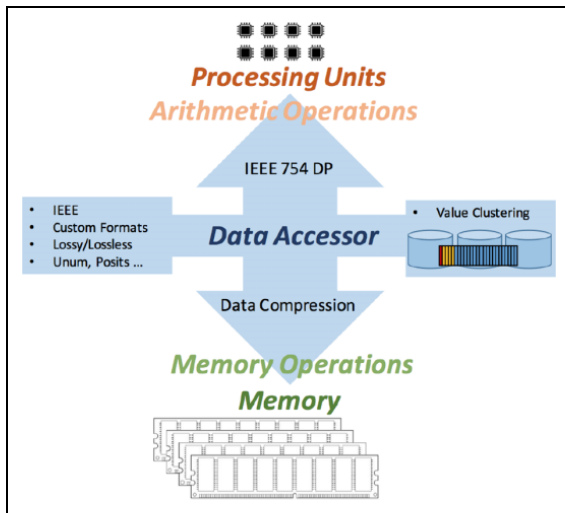


Figure 11. Accessor separating the memory format from the arithmetic format and realizing on-the-fly data conversion in each memory access.

precision from the arithmetic precision, employ high precision only in the computations, and lower the precision as much as possible when accessing data in main memory or communicating with remote processors (Anzt et al., 2019b). 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 (Figure 11). 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 (Grützmacher et al., 2019) or even nonstandard formats like Gustafson’s Posits (Unum type III, Gustafson, 2015). On an abstract level, the idea is to compress data before and after memory operations and only use the working precision in the arithmetic operations. While one generally distinguishes between “lossy compression” and “lossless compression” (Sayood, 2012), significant bandwidth reduction usually requires the loss of some information. How much information can be disregarded without endangering the numerical stability heavily depends on the algorithm and the problem characteristics. Thus, the choice of the memory format requires careful consideration (e.g., in the form of automated format select); see Section 4.4.

4.4. Mixed-precision preconditioning

In the iterative solution process of large sparse systems (e.g., when using Krylov solvers) preconditioners are an important building block for facilitating satisfactory convergence. The concept of preconditioning is to turn an ill-conditioned linear system $Ax = b$ into a (left-) preconditioned system $MAx = Mb$ (or $AMy = b$, $x = My$ for right-preconditioning), which allows for faster convergence of

the iterative solver (Anzt et al., 2018). 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 cannot 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 impacts neither the preconditioner accuracy nor its regularity. For example, Trilinos (The Trilinos Project Team, 2020) allows the use of low-precision preconditioners inside high-precision iterative solvers. 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 (Anzt et al., 2019a). 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 additional cost.

An alternative workaround is to decouple the memory precision from the arithmetic precision (see Section 4.3) and only store the preconditioner in low precision but apply it in high precision (Anzt et al., 2019a). 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 on-the-fly conversion of in-between formats 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 (Flegar et al., 2021). A production-ready implementation of an adaptive-precision block-Jacobi preconditioner decoupling memory precision from arithmetic precision is available in the Ginkgo library (Anzt et al., 2020).

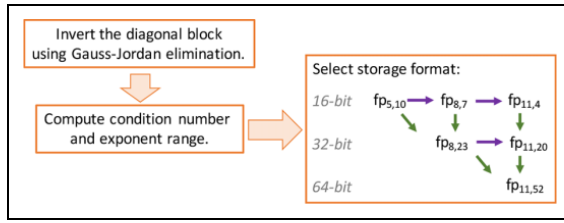


Figure 12. Storage format optimization for block-Jacobi: starting from the most compact storage (left top), the format is extended in exponent bits to fit the data range (rightward) and to preserve regularity (downward) until the range is fit and regularity is satisfied. Note that these configurations are chosen to reflect the hardware characteristics (16/32/64 bit access) and significant-truncated IEEE standard precision formats.

4.4.1. Adaptive-precision block-Jacobi preconditioning. The adaptive-precision block-Jacobi preconditioner realizes the concept of decoupling arithmetic precision from memory precision proposed in Section 4.3 for a block-Jacobi preconditioner (Anzt et al., 2019a). 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 (Figure 12).

This storage format is chosen for each diagonal block individually, respectively reflecting the numerical characteristics like condition number and value range. Figure 13 (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 precision 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 therefore, by design, only have very limited accuracy. Experimental results reveal that the use of a lower precision format for storing the inverted diagonal 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, thereby accelerating the bandwidth bound iterative solution process (Figure 13). For the adaptive-precision block-Jacobi preconditioner, it 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 preconditioner application preserves the preconditioner as a constant operator, and applying a preconditioner in lower precision would result in a non-constant preconditioner and require the use of a (more expensive) flexible iterative solver (Anzt et al., 2019a).

4.5. Mixed-precision 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 the 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 that 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 FP64 solver.

Ljungkvist and Kronbichler (2017, 2019) successfully used 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 to perform the multigrid V-cycle in FP32 with an outer correction in FP64, thereby increasing throughput by up to 83%.

Similarly, Glimberg et al. (2011) use a FP32 multigrid to precondition a FP64 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 $1.6\times$ for the mixed-precision version over the FP64 version and a speedup of $1.9\times$ for a purely FP32 version.

Yamagishi and Matsumura (2016) also apply a FP32 multigrid to a FP64 conjugate gradient solver to the Poisson/Helmholtz problem within their non-hydrostatic ocean model. They report a speedup up to $2\times$ for a FP32 Matvec over a FP64 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 FP32 AMG preconditioner to a FP64 solver.

Emans and van der Meer (2012) 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 $1.1\times$ and $1.5\times$, can potentially carry over to the whole solver and lead to improvements of runtimes within computational fluid dynamics applications.



Figure 13. Top: distribution of floating point formats among the distinct blocks when inverting the blocks in FP64 and preserving one-digit accuracy of the values in each inverted diagonal block when writing to main memory. Each column represents one symmetric positive definite matrix 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, and almost all preconditioner applications are faster, resulting in an average 20% runtime reduction.

Sumiyoshi et al. (2014) 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 $1.2\times$ and $1.6\times$ on the GPU-CPU architectures for the mixed-precision implementation over the FP64 version. These speedups are related to SpMV performance (between

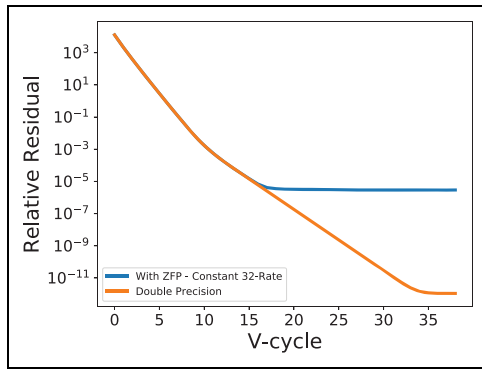


Figure 14. Comparison between the relative residual for a MG method, where the approximation vector is represented in FP64 (orange) or as a ZFP fixed-rate array with a rate of 32 (blue).

1.6 \times and 1.8 \times) 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. (2014) examine the performance of a FP32 AMG preconditioner (ML and PETSc) applied to a FP64 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% of the time of the FP64 version.

An approach described in a presentation by Clark (2019) takes the use of mixed precision even further to involve half precision. Clark and collaborators achieved good results using a FP64 defect correction approach with a FP32 Krylov solver and a half-precision AMG preconditioner.

Another interesting related study by Fox and Kolasinski (2019) 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 or other traditional sources of error (e.g., discretization error).

ZFP decomposes the field of interest into smaller pieces, called blocks, that are then compressed and decompressed independently. ZFP compressed arrays implemented in Lindstrom (2018) are C++ classes that enable random-accessible arrays whose storage size is specified by the user. In particular, ZFP fixed-rate arrays specify a rate used to compress each block of the data field to a finite number of bits. The study uses ZFP fixed-rate arrays to represent the approximation vector in MG on a 2-D Poisson problem with Dirichlet boundary conditions when the number of interior nodes of the finest grid is $(2^8 - 1)^2$. Figure 14 presents the relative residual for a V-cycle with or without ZFP fixed-rate arrays. The orange line represents the relative residual with respect to the FP64 solution, while the blue line represents the relative residual with respect to the solution with ZFP fixed-rate arrays with a rate of 32.

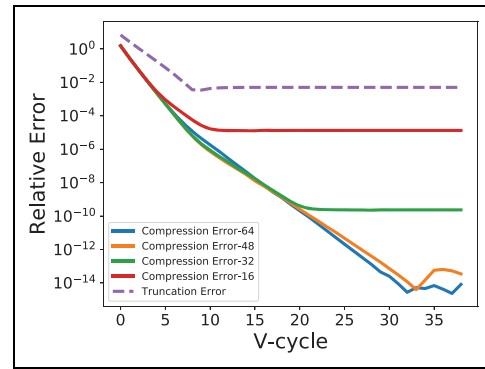


Figure 15. Relative compression error for an adaptive-rate ZFP solution, where the rate for the approximation vector is sequentially lowered on the coarser grids. The purple line represents the truncation error for the double-precision solution.

As the number of V-cycles increase, the relative residual between the two solutions matches until the relative residual for the ZFP solution approximately reaches machine unit roundoff u for FP32.

Figure 15 displays a similar study for when the rate used for the ZFP fixed-rate arrays is adapted depending on the level within the V-cycle. It is assumed that the ZFP fixed-rate arrays have a fixed set of possible rates, 64, 48, 32, or 16. The blue line in Figure 15 depicts the relative compression error (i.e., the error between the FP64 solution and the ZFP fixed-rate solution, where the finest level has a rate of 64, and the rate for the coarser levels is sequentially lowered). That is, if we consider a six-level V-cycle for which the finest level has a fixed rate of 64, then the second finest level has a fixed rate of 48, then 32, and then 16 for the remaining coarse grids. The orange, green, and red lines depict the relative compression error for a rate of 48, 32, and 16, respectively for the finest level. The purple dashed line depicts the relative truncation error for the FP64 solution. Each ZFP fixed-rate solution remains below the truncation error and the compression error continuously decreases until the relative error for the ZFP solution approximately reaches the respective machine unit round-off u dependent on the rate of the finest level.

This study shows that, for MG on a Poisson problem, applying ZFP to the approximation vector can significantly decrease memory use and is expected to decrease run times, while the generated errors stay below the discretization error. Since a hardware version of ZFP is not available yet, no actual runs were possible; however, the results show good potential for using GMG and/or AMG as a preconditioner.

Currently, Tamstorf et al. (2020a) appear to be the only ones who investigated the theory of multiprecision 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 (2017) but ultimately provides tighter bounds for many PDEs. Tamstorf et al. (2020b) 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 discretization 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.

4.6. Eigenvalue problems

Little work appears to have been done on exploiting mixed-precision arithmetic in algorithms for sparse eigenvalue problems. The ESSEX-II project (Alvermann et al., 2019) is developing eigensolvers based on Jacobi-Davidson, subspace iteration, and other methods, and is using lower precision in early iterations for speed and higher precision within orthogonalization for robustness.

5. Summary and outlook on the potential of mixed-precision technology

We have presented mixed-precision algorithms for dense and sparse linear algebra that are outperforming traditional algorithms operating in high precision. For performance-bound dense linear algebra algorithms, mixed-precision iterative refinement that employs a low-precision error correction solver remains the first-choice algorithm to exploit the compute power in low precision. For sparse linear algebra, the memory-bound nature of the the algorithms makes the concept of decoupling memory precision from arithmetic precision attractive. Furthermore, preconditioners with limited approximation accuracy are natural targets for the use of lower precision. Carefully adjusting the preconditioner precision to the numerical requirements and the approximation accuracy can render run time savings without impacting the iterative solver's convergence.

As AI and deep learning are currently driving the hardware market, we expect a large number of processors and accelerators featuring low-precision special function units and support for nonstandard precision formats and integer operations. For numerical linear algebra, we anticipate

significant potential in the use of integer arithmetic for numerical calculations and the low-precision function units designed for deep learning. As we see the machine imbalance continuing to grow, we also expect format decoupling and compression techniques to become essential and are eager to see hardware support for data compression.

Authors' note

This document was prepared as an account of work sponsored by an agency of the United States government. Neither the United States government nor Lawrence Livermore National Security, LLC, nor any of their employees makes any warranty, expressed or implied, or assumes any legal liability or responsibility for the accuracy, completeness, or usefulness of any information, apparatus, product, or process disclosed, or represents that its use would not infringe privately owned rights. Reference herein to any specific commercial product, process, or service by trade name, trademark, manufacturer, or otherwise does not necessarily constitute or imply its endorsement, recommendation, or favoring by the United States government or Lawrence Livermore National Security, LLC. The views and opinions of authors expressed herein do not necessarily state or reflect those of the United States government or Lawrence Livermore National Security, LLC, and shall not be used for advertising or product endorsement purposes. Sandia National Laboratories is a multimission laboratory managed and operated by National Technology and Engineering Solutions of Sandia, LLC., a wholly owned subsidiary of Honeywell International, Inc., for the U.S. Department of Energy's National Nuclear Security Administration under contract DE-NA-0003525.




Declaration of conflicting interests

The author(s) declared no potential conflicts of interest with respect to the research, authorship, and/or publication of this article.

Funding

The author(s) disclosed receipt of the following financial support for the research, authorship, and/or publication of this article: 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.

ORCID iD

Hartwig Anzt  <https://orcid.org/0000-0003-2177-952X>
 Jennifer Loe  <https://orcid.org/0000-0002-3018-7190>
 Tobias Ribizel  <https://orcid.org/0000-0003-3023-1849>

Notes

1. <https://www.olcf.ornl.gov/summit/>.
2. <https://icl.bitbucket.io/hpl-ai/>.

3. Note that some hardware architectures, e.g. NVIDIA Tensor Cores, perform computations and accumulations in higher precision, and only truncate down to FP16 when writing the results to main memory.

References

- Abdelfattah A, Tomov S and Dongarra JJ (2019) 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, 20–24 May 2019, pp. 111–122. IEEE.
- Abdelfattah A, Tomov S and Dongarra J (2020) Investigating the benefit of FP16-enabled mixed-precision solvers for symmetric positive definite matrices using GPUs. In: Krzhizhanovskaya VV, Závodszy G, Lees MH, Dongarra JJ, Sloot PMA, and Teixeira SBJ (eds) *Computational Science—ICCS 2020*, Lecture Notes in Computer Science, Vol. 12138. New York, NY: Springer International Publishing, pp. 237–250. DOI: 10.1007/978-3-030-50417-5_18.
- Agullo E, Demmel J, Dongarra J, et al. (2009) Numerical linear algebra on emerging architectures: the PLASMA and MAGMA projects. *Journal of Physics: Conference Series* 180: 012037.
- Alvermann A, Basermann A, Bungartz HJ, et al. (2019) Benefits from using mixed precision computations in the ELPA-AEO and ESSEX-II eigensolver projects. *Japan Journal of Industrial and Applied Mathematics* 36(2): 699–717.
- Anderson E, Bai Z, Bischof CH, et al. (1999) *LAPACK Users' Guide*. 3rd edn. Philadelphia, PA: Society for Industrial and Applied Mathematics. ISBN 0-89871-447-8. Available at: <http://www.netlib.org/lapack/lug/> (accessed 2020).
- Anzt H, Cojean T, Chen YC, et al. (2020) Ginkgo: A high performance numerical linear algebra library. *Journal of Open Source Software* 5(52): 2260. DOI: <https://doi.org/10.21105/joss.02260>.
- Anzt H, Dongarra J, Flegar G, et al. (2019a) Adaptive precision in block-jacobi preconditioning for iterative sparse linear system solvers. *Concurrency and Computation: Practice and Experience* 31(6): e4460.
- Anzt H, Flegar G, Grützmacher T, et al. (2019b) Toward a modular precision ecosystem for high-performance computing. *The International Journal of High Performance Computing Applications* 33(6): 1069–1078.
- Anzt H, Huckle TK, Bräckle J, et al. (2018) Incomplete sparse approximate inverses for parallel preconditioning. *Parallel Computing* 71: 1–22.
- Arioli M and Duff IS (2009) Using FGMRES to obtain backward stability in mixed precision. *Electronic Transactions on Numerical Analysis* 33: 31–44.
- Arioli M, Duff IS, Gratton S, et al. (2007) A note on GMRES preconditioned by a perturbed LDL^T decomposition with static pivoting. *SIAM Journal on Scientific Computing* 29(5): 2024–2044.
- Barlow JL (2019) Block modified Gram–Schmidt algorithms and their analysis. *SIAM Journal on Matrix Analysis and Applications* 40(4): 1257–1290.
- Björck Å (1967a) Iterative refinement of linear least squares solutions I. *BIT Numerical Mathematics* 7(4): 257–278.
- Björck Å (1967b) Solving linear least squares problems by Gram–Schmidt orthogonalization. *BIT Numerical Mathematics* 7(1): 1–21.
- Björck Å (1990) Iterative refinement and reliable computing. In: MG Cox, and SJ Hammarling (eds) *Reliable Numerical Computation*. Oxford: Oxford University Press, pp. 249–266.
- Björck Å (1994) Numerics of Gram–Schmidt orthogonalization. *Linear Algebra and its Applications* 197: 297–316.
- Björck Å and Paige CC (1992) Loss and recapture of orthogonality in the modified Gram–Schmidt algorithm. *SIAM Journal on Matrix Analysis and Applications* 13(1): 176–190.
- Blanchard P, Higham NJ, Lopez F, et al. (2020) Mixed precision block fused multiply-add: error analysis and application to GPU tensor cores. *SIAM Journal on Scientific Computing* 42(3): C124–C141.
- Carson E and Higham NJ (2017) 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.
- Carson E and Higham NJ (2018) Accelerating the solution of linear systems by iterative refinement in three precisions. *SIAM Journal on Scientific Computing* 40(2): A817–A847.
- Carson E, Higham NJ and Pranesh S (2020) Three-precision GMRES-based iterative refinement for least squares problems. *SIAM Journal on Scientific Computing* 42(6): A4063–A408.
- Carson EC (2015) *Communication-avoiding Krylov subspace methods in theory and practice*. PhD Thesis, University of California, Berkeley.
- Clark K (2019) Effective use of mixed precision for hpc. In: *Smoky Mountain Conference 2019*, Tennessee, 3–6 September 2019.
- Clark MA, Babich R, Barros K, et al. (2010) Solving lattice QCD systems of equations using mixed precision solvers on GPUs. *Computer Physics Communications* 181(9): 1517–1528.
- Davies PI, Higham NJ and Tisseur F (2001) Analysis of the Cholesky method with iterative refinement for solving the symmetric definite generalized eigenproblem. *SIAM Journal on Matrix Analysis and Applications* 23(2): 472–493.
- Demmel JW, Hida Y, Kahan W, et al. (2006) Error bounds from extra-precise iterative refinement. *ACM Transactions on Mathematical Software* 32(2): 325–351.
- Dongarra JJ (1982) Algorithm 589 SICEDR: a FORTRAN subroutine for improving the accuracy of computed matrix eigenvalues. *ACM Transactions on Mathematical Software* 8(4): 371–375.
- Dongarra JJ, Moler CB and Wilkinson JH (1983) Improving the accuracy of computed eigenvalues and eigenvectors. *SIAM Journal on Numerical Analysis* 20(1): 23–45.
- Elble JM and Sahinidis NV (2012) Scaling linear optimization problems prior to application of the simplex method. *Computational Optimization and Applications* 52(2): 345–371.
- Emans M and van der Meer A (2012) Mixed-precision AMG as linear equation solver for definite systems. In: *Proceedings of International Conference on Computational Science, ICCS*

- 2010, Vol. 1, The Netherlands 31 May–2 June 2010, pp. 175–183.
- Flegar G, Anzt H, Cojean T, et al. (2021) Adaptive precision block-Jacobi for high performance preconditioning in the ginkgo linear algebra software. *ACM Transaction on Mathematical Software* 47(2): e4460.
- Fox A and Kolasinski A (2019) Error analysis of inline ZFP compression for multigrid methods. In: *2019 Copper Mountain Conference for Multigrid Methods*, Copper Mountain, CO, March 25, 2019.
- Fukaya T, Kannan R, Nakatsukasa Y, et al. (2020) Shifted CholeskyQR for computing the QR factorization of ill-conditioned matrices. *SIAM Journal on Scientific Computing* 42(1): A477–A503.
- Giraud L, Gratton S and Langou J (2004) A rank- k update procedure for reorthogonalizing the orthogonal factor from modified Gram–Schmidt. *SIAM Journal on Matrix Analysis and Applications* 25(4): 1163–1177.
- Glimberg SL, Engsig-Karup AP and Madsen MG (2011) A fast GPU-accelerated mixed-precision strategy for fully nonlinear-water wave computations. In: *Proceedings of ENUMATH 2011*, University of Leicester.
- Gratton S, Simon E, Titley-Peloquin D and Toint P (2019) Exploiting variable precision in GMRES. *arXiv preprint arXiv:1907.10550*.
- Greenbaum A (1989) Behavior of slightly perturbed Lanczos and conjugate-gradient recurrences. *Linear Algebra and its Applications* 113: 7–63.
- Greenbaum A (1997) Estimating the attainable accuracy of recursively computed residual methods. *SIAM Journal on Matrix Analysis and Applications* 18(3): 535–551.
- Grützmacher T, Cojean T, Flegar G, et al. (2019) A customized precision format based on mantissa segmentation for accelerating sparse linear algebra. *Concurrency and Computation: Practice and Experience* 32: e5418.
- Gupta S, Agrawal A, Gopalakrishnan K, et al. (2015) Deep learning with limited numerical precision. In: *Proceedings of the 32nd International Conference on International Conference on Machine Learning*, Vol. 37, ICML’15. JMLR.org, pp. 1737–1746. Available at: <http://dl.acm.org/citation.cfm?id=3045118.3045303> (accessed 2020).
- Gustafson J (2015) *The End of Error: Unum Computing*. Milton Park: Chapman & Hall/CRC Computational Science. Taylor & Francis. ISBN 9781482239867. Available at: <https://books.google.de/books?id=W2ThoAEACAAJ> (accessed 2020).
- Haidar A, Abdelfattah A, Zounon M, et al. (2018a) The design of fast and energy-efficient linear solvers: on the potential of half-precision arithmetic and iterative refinement techniques. In: Shi Y, Fu H, Tian Y, Krzhizhanovskaya VV, Lees MH, Dongarra J, and Sloot PMA (eds) *Computational Science—ICCS 2018*. Cham: Springer International Publishing, pp. 586–600. DOI: 10.1007/978-3-319-93698-7_45.
- Haidar A, Bayraktar H, Tomov S, et al. (2020) *Mixed-precision solution of linear systems using accelerator-based computing*. Mixed-precision iterative refinement using tensor cores on GPUs to accelerate solution of linear systems. *Proceedings of the Royal Society A* 476(2243): 20200110, 2020.
- Haidar A, Tomov S, Dongarra J, et al. (2018b) 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, Dallas, TX, USA, 11–16 November 2018. Piscataway, NJ, USA: IEEE Press, pp. 47: 1–47:11.
- Haidar A, Wu P, Tomov S, et al. (2017) 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*, New York, NY, November 2017, pp. 1–8.
- Higham NJ (1989) The accuracy of solutions to triangular systems. *SIAM Journal on Numerical Analysis* 26(5): 1252–1265.
- Higham NJ (1997) Iterative refinement for linear systems and LAPACK. *IMA Journal of Numerical Analysis* 17(4): 495–509.
- Higham NJ (2002) *Accuracy and Stability of Numerical Algorithms*. 2nd edn. Philadelphia, PA: Society for Industrial and Applied Mathematics. ISBN 0-89871-521-0.
- Higham NJ (2019) *Error analysis for standard and GMRES-based iterative refinement in two and three-precisions*. MIMS EPrint 2019.19, Manchester: Manchester Institute for Mathematical Sciences, The University of Manchester. Available at: <http://eprints.maths.manchester.ac.uk/2735/> (accessed 2020).
- Higham NJ and Mary T (2019) A new approach to probabilistic rounding error analysis. *SIAM Journal on Scientific Computing* 41(5): A2815–A2835.
- Higham NJ and Mary T (2020) Sharper probabilistic backward error analysis for basic linear algebra kernels with random data. *SIAM Journal on Scientific Computing* 42(5): A3427–A3446.
- Higham NJ and Pranesh S (2021) Exploiting lower precision arithmetic in solving symmetric positive definite linear systems and least squares problems. *SIAM Journal on Scientific Computing* 43(1): A258–A277.
- Higham NJ and Pranesh S (2019) Simulating low precision floating-point arithmetic. *SIAM Journal on Scientific Computing* 41(5): C585–C602.
- Higham NJ, Pranesh S and Zounon M (2019) Squeezing a matrix into half precision, with an application to solving linear systems. *SIAM Journal on Scientific Computing* 41(4): A2536–A2551.
- Hogg JD and Scott JA (2010) A fast and robust mixed-precision solver for the solution of sparse symmetric linear systems. *ACM Transactions on Mathematical Software* 37(2): 17: 1–17:24.
- IEEE (2019) *IEEE Standard for Floating-Point Arithmetic, IEEE Std 754-2019 (Revision of IEEE 754-2008)*. New York, NY: The Institute of Electrical and Electronics Engineers. ISBN 978-1-5044-5924-2. DOI:10.1109/IEEESTD.2019.8766229.
- Joffrain T, Low TM, Quintana-Ort ES, et al. (2006) Accumulating Householder transformations, revisited. *ACM Transactions on Mathematical Software (TOMS)* 32(2): 169–179.
- Knight PA, Ruiz D and Uçar B (2014) A symmetry preserving algorithm for matrix scaling. *SIAM Journal on Matrix Analysis and Applications* 35(3): 931–955.
- Langou J, Langou J, Luszczek P, et al. (2006) Exploiting the performance of 32 bit floating point arithmetic in obtaining

- 64 bit accuracy (revisiting iterative refinement for linear systems). In: *Proceedings of the 2006 ACM/IEEE Conference on Supercomputing*, Tampa, FL, USA, 11–17 November 2006.
- Leon SJ, Björck Å and Gander W (2013) Gram-Schmidt orthogonalization: 100 years and more. *Numerical Linear Algebra with Applications* 20(3): 492–532.
- Lindstrom P (2018) Zfp version 0.5.3. Available at: <https://zfp.readthedocs.io/en/release0.5.3/index.htm> (accessed 2020).
- Ljungkvist K and Kronbichler M (2017) *Multigrid for matrix-free finite element computations on graphics processors*. Technical report, Department of Information Technology, Uppsala University.
- Ljungkvist K and Kronbichler M (2019) Multigrid for matrix-free high-order finite element computations on graphics processors. *ACM Transactions on Parallel Processing* 6: 3322813.
- Malard J and Paige C (1994) Efficiency and scalability of two parallel QR factorization algorithms. In: *Proceedings of IEEE Scalable High Performance Computing Conference*. New York, NY: IEEE, pp. 615–622.
- Meurant G and Strakoš Z (2006) The Lanczos and conjugate gradient algorithms in finite precision arithmetic. *Acta Numerica* 15: 471–542.
- Moler CB (1967) Iterative refinement in floating point. *Journal of the ACM (JACM)* 14(2): 316–321.
- NVIDIA (2017) Nvidia Tesla V100 GPU Architecture. Available at: <https://images.nvidia.com/content/volta-architecture/pdf/volta-architecture-whitepaper.pdf> (accessed 2020).
- Ogita T and Aishima K (2018) Iterative refinement for symmetric eigenvalue decomposition. *Japan Journal of Industrial and Applied Mathematics* 35(3): 1007–1035.
- Ogita T and Aishima K (2019) Iterative refinement for symmetric eigenvalue decomposition II: clustered eigenvalues. *Japan Journal of Industrial and Applied Mathematics* 36: 435–459.
- Paige CC (1980) Accuracy and effectiveness of the Lanczos algorithm for the symmetric eigenproblem. *Linear Algebra and its Applications* 34: 235–258.
- Paige CC (2018) The effects of loss of orthogonality on large scale numerical computations. In: *International Conference on Computational Science and Its Applications*. Cham: Springer, pp. 429–439.
- Paige CC and Strakoš Z (2002) Residual and backward error bounds in minimum residual Krylov subspace methods. *SIAM Journal on Scientific Computing* 23(6): 1898–1923.
- Paige CC and Wülling W (2014) Properties of a unitary matrix obtained from a sequence of normalized vectors. *SIAM Journal on Matrix Analysis and Applications* 35(2): 526–545.
- Paige CC, Rozložník M and Strakoš Z (2006) Modified gram-schmidt MGS, least squares, and backward stability of MGS-GMRES. *SIAM Journal on Matrix Analysis and Applications* 28(1): 264–284.
- Petschow M, Quintana-Ort E and Bientinesi P (2014) Improved accuracy and parallelism for MRRR-based eigensolvers—a mixed precision approach. *SIAM Journal on Scientific Computing* 36(2): C240–C263.
- Puglisi C (1992) Modification of the Householder method based on the compact WY representation. *SIAM Journal on Scientific Computing* 13(3): 723–726.
- Richter C, Schops S and Clemens M (2014) GPU-accelerated mixed precision algebraic multigrid preconditioners for discrete elliptic field problems. *9th IET International Conference on Computation in Electromagnetics (CEM 2014)*, London, UK, 2014, pp. 1–2, DOI: 10.1049/cp.2014.0185.
- Saad Y and Schultz MH (1986) GMRES: a generalized minimal residual algorithm for solving nonsymmetric linear systems. *SIAM Journal on scientific and statistical computing* 7(3): 856–869.
- Sayood K (2012) *Introduction to Data Compression, Fourth Edition*. 4th edn. San Francisco, CA: Morgan Kaufmann Publishers Inc. ISBN 0124157963.
- Simoncini V and Szyld DB (2003) Theory of inexact Krylov subspace methods and applications to scientific computing. *SIAM Journal on Scientific Computing* 25(2): 454–477.
- Skeel RD (1980) Iterative refinement implies numerical stability for Gaussian elimination. *Mathematics of Computation* 35(151): 817–832.
- Sleijpen GL and van der Vorst HA (1996) Reliable updated residuals in hybrid Bi-CG methods. *Computing* 56(2): 141–163.
- Smoktunowicz A, Barlow JL and Langou J (2006) A note on the error analysis of classical Gram-Schmidt. *Numerische Mathematik* 105(2): 299–313.
- Stewart GW (1973) *Introduction to Matrix Computations*. New York, NY: Academic Press. ISBN 0-12-670350-7.
- Sumiyoshi Y, Fujii A, Nukada A, et al. (2014) Mixed-precision amg method for many core accelerators. In: *EUROMPI/ASIA 14: Proceedings of the 21st European MPI Users' Group Meeting*. September 2014, pp. 127–132.
- Sun X (1996) *Aggregations of elementary transformations*. Technical Report DUKE-TR-1996-03, Durham, NC: Duke University.
- Świrydowicz K, Langou J, Ananthan S, et al. (2020) Low synchronization Gram-Schmidt and generalized minimal residual algorithms. *Numerical Linear Algebra with Applications* 28: e2343.
- Tamstorf R, Benzaken J and McCormick S (2020a) Algebraic error analysis for mixed precision multigrid solvers. *SIAM Journal on Scientific Computing* Submitted.
- Tamstorf R, Benzaken J and McCormick S (2020b) Discretization-error-accurate mixed precision multigrid solvers. *SIAM Journal on Scientific Computing* Submitted.
- The Trilinos Project Team (2020) The Trilinos Project Website. Available at: <https://trilinos.github.io> (accessed 2020).
- Tisseur F (2001) Newton's method in floating point arithmetic and iterative refinement of generalized eigenvalue problems. *SIAM Journal on Matrix Analysis and Applications* 22(4): 1038–1057.
- van den Eshof J and Sleijpen GL (2004) Inexact Krylov subspace methods for linear systems. *SIAM Journal on Matrix Analysis and Applications* 26(1): 125–153.
- Van Der Vorst HA and Ye Q (2000) Residual replacement strategies for Krylov subspace iterative methods for the convergence of true residuals. *SIAM Journal on Scientific Computing* 22(3): 835–852.
- Walker HF (1988) Implementation of the GMRES method using Householder transformations. *SIAM Journal on Scientific Computing* 9(1): 152–163.

Wilkinson JH (1963) *Rounding Errors in Algebraic Processes*. Notes on Applied Science No. 32, Her Majesty's Stationery Office. ISBN 0-486-67999-3. Also published by Prentice-Hall, Englewood Cliffs, NJ, USA. Reprinted by Dover, New York, 1994.

Yamagishi T and Matsumura Y (2016) GPU acceleration of a non-hydrostatic ocean model with a multigrid poisson/helmholtz solver. *Procedia Computer Science* 80: 1658–1669.

Yamazaki I, Tomov S and Dongarra J (2015) Mixed-precision Cholesky QR factorization and its case studies on multicore CPU with multiple GPUs. *SIAM Journal on Scientific Computing* 37(1): C307–C330.

Yamazaki I, Tomov S and Dongarra J (2016) Stability and performance of various singular value QR implementations on multicore CPU with a GPU. *ACM Transactions on Mathematical Software* 43(2): 10:1–10:18.

Author biographies

Ahmad Abdelfattah is a research scientist at the Innovative Computing Laboratory, University of Tennessee. He received his PhD in computer science from King Abdullah University of Science and Technology (KAUST) in 2015. His research interests include parallel numerical algorithms, performance optimization, and accelerator-based computing.

Hartwig Anzt is a research group leader at the Karlsruhe Institute of Technology (KIT) and holds a research consultant position at the University of Tennessee. He leads the multiprecision focus effort within the US Exascale Computing project, a cross-laboratory effort to develop and deploy mixed precision algorithms for accelerating scientific computing applications.

Erik G Boman is a scientist at Center for Computing Research at Sandia National Laboratories USA. He holds a PhD in Scientific Computing from Stanford University. His research interests are in numerical linear algebra, combinatorial scientific computing, and high-performance computing.

Erin Carson is a research scientist at Charles University. She received a Ph.D. in Computer Science in 2015 from the University of California-Berkeley. Her research focuses on numerical linear algebra, parallel algorithms, and high performance computing.

Terry Cojean received his PhD at Inria Bordeaux in 2018 and is currently a post-doctoral researcher in Hartwig Anzt's research group at the Karlsruhe Institute of Technology. Currently, he is a lead developer of the Ginkgo sparse linear algebra software.

Jack Dongarra holds an appointment at the University of Tennessee, Oak Ridge National Laboratory, and the University of Manchester. He specializes in numerical algorithms in linear algebra, parallel computing, use of advanced-computer architectures, programming methodology, and tools for parallel computers.

Alyson Fox is a Computational Mathematician in the Center for Applied Scientific Computing at Lawrence Livermore National Laboratory and a SIAM Science Policy Fellow. Her interests include lossy compression algorithms, collaborative autonomy, network analysis, numerical analysis, graph theory, data mining, and scientific computing.

Mark Gates is a Research Assistant Professor at the University of Tennessee, Knoxville, where he specializes in high-performance computing, concentrating on parallel algorithms for linear algebra.

Nicholas J Higham is Royal Society Research Professor and Richardson Professor of Applied Mathematics in the Department of Mathematics at the University of Manchester. He received his PhD in 1985 from the University of Manchester. He is a Fellow of the Royal Society, an ACM Fellow, a SIAM Fellow, and a Member of Academia Europaea.

Xiaoye S Li is a Senior Scientist at the Lawrence Berkeley National Laboratory. She has worked on diverse problems in high performance scientific computations, including parallel computing, sparse matrix computations, high-precision arithmetic, and combinatorial scientific computing. She is the lead developer of SuperLU, a widely used sparse direct solver, and has contributed to the development of several other mathematical libraries, including ARPREC, LAPACK, PDSLin, STRUMPACK, and XBLAS. She is a SIAM Fellow.

Jennifer Loe completed her PhD in mathematics at Baylor University. She is now a postdoc at Sandia National Laboratories in the Scalable Algorithms department. Her primary research is in Krylov solvers.

Piotr Luszczek is a Research Assistant Professor at the Innovative Computing Laboratory in the University of Tennessee, Knoxville's Tickle College of Engineering. His research interests include benchmarking, numerical linear algebra for high-performance computing, automatic performance tuning for modern hardware, and stochastic models for performance.

Srikara Pranesh is a research associate in the department of Mathematics, University of Manchester. His main research

interests are numerical analysis, numerical linear algebra, and mathematical software.

Siva Rajamanickam is a principal member of technical staff in the Scalable Algorithms department at the Center for Computing Research at Sandia National Laboratories. He has a PhD in Computer Science and Engineering from the University of Florida. His focus is in the intersection of high performance computing, combinatorial scientific computing, performance portability, graph algorithms and machine learning. Dr Rajamanickam leads the linear solvers product of the Trilinos library and the Kokkos Kernels library.

Tobias Ribizel received the master's degree in computer science and mathematics from Karlsruhe Institute of Technology (KIT) in 2019, where he now works as doctoral researcher in the FiNE junior research group. His research interests include software engineering and low-level performance optimization for HPC.

Barry F Smith received his mathematics degree from the Courant Institute in 1990. Since then, much of the time, he has been developing mathematical software libraries at Argonne National Laboratory.

Kasia Swirydowicz is a Computational Scientist in Pacific Northwest National Laboratory in Richland, WA. Her research interest include numerical linear algebra and Krylov subspace solvers, and their efficient implementation on High Performance Computing systems.

Stephen Thomas is an applied mathematician with the US DOE National Renewable Energy Laboratory in Colorado.

Throughout his graduate studies and research career, Dr. Thomas has focused on the intersection of high-performance computing and iterative solvers for large sparse linear systems with applications in climate, geoscience, and renewable energy.

Stanimire Tomov is Research Assistant Professor at the University of Tennessee, Knoxville. He specializes in parallel algorithms, data analytics, and high-performance scientific computing. His current work is concentrated on the development of numerical linear algebra software for new architectures.

Yaohung M Tsai received his PhD degree in Computer Science at the University of Tennessee. His research focused on mixed precision numerical methods that exploited contemporary and emerging hardware platforms with the new accuracy options for floating-point processing. His work also involved automated performance engineering approach to achieve efficient implementations on various hardware architectures in a portable fashion.

Ulrike Meier Yang leads the Mathematical Algorithms & Computing group in the Center for Applied Scientific Computing at Lawrence Livermore National Laboratory, the xSDK (Extreme-scale Scientific Software Kit) project in the Exascale Computing Project (ECP) and the Linear and Nonlinear Solvers Topical Area in the SciDAC FAS-TMath Institute. Her research interests are numerical algorithms, particularly multigrid methods, high performance computing, parallel algorithms, performance evaluation and scientific software design.