*Acta Numerica* (2022), pp. 347–414 Printed in the United Kingdom
doi:10.1017/S0962492922000022

# Mixed precision algorithms in numerical linear algebra

Nicholas J. Higham
*Department of Mathematics, University of Manchester,*
*Manchester, M13 9PL, UK*
*E-mail: nick.higham@manchester.ac.uk*

Theo Mary
*Sorbonne Université, CNRS, LIP6,*
*Paris, F-75005, France*
*E-mail: theo.mary@lip6.fr*

Today's floating-point arithmetic landscape is broader than ever. While scientific computing has traditionally used single precision and double precision floating-point arithmetics, half precision is increasingly available in hardware and quadruple precision is supported in software. Lower precision arithmetic brings increased speed and reduced communication and energy costs, but it produces results of correspondingly low accuracy. Higher precisions are more expensive but can potentially provide great benefits, even if used sparingly. A variety of mixed precision algorithms have been developed that combine the superior performance of lower precisions with the better accuracy of higher precisions. Some of these algorithms aim to provide results of the same quality as algorithms running in a fixed precision but at a much lower cost; others use a little higher precision to improve the accuracy of an algorithm. This survey treats a broad range of mixed precision algorithms in numerical linear algebra, both direct and iterative, for problems including matrix multiplication, matrix factorization, linear systems, least squares, eigenvalue decomposition and singular value decomposition. We identify key algorithmic ideas, such as iterative refinement, adapting the precision to the data, and exploiting mixed precision block fused multiply–add operations. We also describe the possible performance benefits and explain what is known about the numerical stability of the algorithms. This survey should be useful to a wide community of researchers and practitioners who wish to develop or benefit from mixed precision numerical linear algebra algorithms.

© The Author(s), 2022. Published by Cambridge University Press.
This is an Open Access article, distributed under the terms of the Creative Commons Attribution licence (http://creativecommons.org/licenses/by/4.0/), which permits unrestricted re-use, distribution, and reproduction in any medium, provided the original work is properly cited.

https://doi.org/10.1017/S0962492922000022 Published online by Cambridge University Press

## CONTENTS

## 1. Introduction

Traditionally, scientific computing has been carried out in double precision arithmetic, which nowadays corresponds to a 64-bit floating-point number format. It has long been recognized that single precision computations have advantages over double precision ones, not just because single precision arithmetic is typically twice as fast as double precision arithmetic but also because single precision data requires half as much storage as double precision data and has half the memory transfer costs. Of course, single precision computations will generally provide only single precision accuracy. Whether this is sufficient for a given application depends on the application, and the answer can be different even for different computations within the same field: see Section 1.2.2.

Modern hardware increasingly supports half precision arithmetic, which is attractive compared with single and double precisions because of its speed, its lower energy usage and its reduced storage and data movement costs.

As we now have three precisions of floating-point arithmetic in hardware, as well as quadruple precision arithmetic in software, we are in an intrinsically mixed precision world, where precisions can be judiciously chosen in order to make the best use of our computational resources.

In this work we survey mixed precision algorithms in numerical linear algebra. Relevant work goes back to the beginning of the digital computer area, but most

https://doi.org/10.1017/S0962492922000022 Published online by Cambridge University Press

contributions in this area have been made in the last couple of decades. An earlier survey of the same areas is that of Abdelfattah *et al.* (2021*a*).

### *1.1. Mixed precision versus multiprecision*

We use the following terminology.

- A *mixed precision algorithm* uses two or more precisions chosen from a small number of available precisions, which are typically half, single and double precision, provided in hardware, and quadruple precision, provided in software.
- A *multiprecision algorithm* uses one or more arbitrary precisions, which may be problem-dependent and are provided in software. The precision at which results are returned may be fixed (for example, double precision) or may be a parameter. See Section 2.5 for details of some available multiprecision arithmetics. The term *variable precision* is sometimes used in place of multiprecision.

This survey is concerned with mixed precision algorithms, but we will briefly discuss some multiprecision algorithms in Section 15.2.

### *1.2. Applications*

Mixed precision algorithms are being used, or considered for use in a wide variety of applications, some of which involve computations at very large scale. We mention some examples here in order to illustrate the different motivations for using mixed precision arithmetic and the possible benefits in real-life applications.

#### *1.2.1. Simulations*

Idomura, Ina, Ali and Imamura (2020) carry out plasma turbulence simulations for the next generation experimental fusion reactor ITER on the Fugaku and Summit supercomputers. Their code for integrating the gyrokinetic partial differential equation (PDE) in double precision involves the solution of linear systems by a Krylov method. The authors show that using a communication-avoiding version of the Krylov method with a half precision (fp16) version of the preconditioner results in speedups over the original solver by a factor of approximately 2–3.

Yang *et al.* (2019) implement in TensorFlow a Monte Carlo simulation of the Ising model on a two-dimensional lattice and run it on Google Tensor Processing Units (TPUs). They find that single precision can be replaced by half precision (bfloat16) without any loss of accuracy, enabling larger lattices to be simulated because of the lower memory requirement of half precision.

#### *1.2.2. Climate modelling and weather forecasting*

In climate modelling and weather forecasting, codes have traditionally used double precision variables, but in recent years the use of lower precisions has been extensively investigated (Palmer 2014). The lesser data movement and faster execution

https://doi.org/10.1017/S0962492922000022 Published online by Cambridge University Press

of lower precision arithmetic offers the possibility of using refined spatial grids represented in lower precision, allowing higher resolution simulations with no increase in run time, potentially improving the output of a model. The observations on which a model is built have low precision, so it can be argued that variables do not need to be represented in double precision (Tintó Prims *et al.* 2019), and this argument is also supported by the notion that the grid parametrizations should be stochastic (Palmer 2020). Moreover, many model components have uncertainties that can be much larger than the level of double precision (Dawson, Düben, MacLeod and Palmer 2018). Codes in this area can consist of millions of lines of Fortran (Bauer *et al.* 2021), so changing the precisions of variables and assessing the effect of the changes is not an easy task.

Váňa *et al.* (2017) show that almost all double precision variables in the Integrated Forecast System of the European Centre for Medium-Range Weather Forecasts can be converted to single precision with no noticeable loss in accuracy and a gain in speed of about 40%. A benefit of running the code in lower precision was found to be that it revealed places where the code could be made more robust. Harvey and Verseghy (2015) had a different experience with their code for a land surface model, where running in single precision instead of double did not provide sufficient accuracy for some of the land depths and timescales of interest.

A weather and climate simulation code called the Unified Model (UM) is used by the Met Office for both operational numerical weather prediction and climate modelling. The code carries out time integration of a system of PDEs, which involves at each time step the solution of a linear system with a banded, time-varying non-symmetric matrix of size $3.5 \times 10^8$. The system is solved by the preconditioned BiCGstab algorithm, with a convergence test requiring a residual of norm $10^{-4}$ relative to the right-hand side. The UM is coded in double precision and is memory-bound (that is, its execution time is determined by the speed at which data is transferred from main memory to the arithmetic units rather than by the speed of the floating-point arithmetic). Maynard and Walters (2019) implemented the linear system solution almost entirely in single precision, with the same convergence tolerance, obtaining close to a factor 2 speedup of the solver, which they attribute to the reduction in data movement costs. To alleviate some stability issues they use a mixed precision summation algorithm that is essentially the FABsum algorithm of Blanchard, Higham and Mary (2020*a*), which is discussed in Section 4.2. The mixed precision solver is now used in operational forecasts.

### *1.2.3. Machine learning*

Low precision arithmetic has become widely used in machine learning in the last few years because it has been found experimentally that algorithms can run faster with certain parts executed in low precision, with little or no deterioration in the quality of the results. Dean (2020) gives three characteristics of deep learning models that make specialized hardware suitable for running them.

> First, they are very tolerant of reduced-precision computations. Second, the computations performed by most models are simply different compositions of a relatively small handful of operations like matrix multiplies, vector operations, application of convolutional kernels, and other dense linear algebra calculations . . . Third, many of the mechanisms developed over the past 40 years to enable general-purpose programs to run with high performance on modern CPUs . . . are unnecessary for machine learning computations. So, the opportunity exists to build computational hardware that is specialized for dense, low-precision linear algebra, and not much else, but is still programmable at the level of specifying programs as different compositions of mostly linear algebra-style operations.

The special-purpose hardware being referred to here can be classified as either field programmable gate arrays (FPGAs) or application-specific integrated circuits (ASICs), and it may use fixed-point arithmetic, floating-point arithmetic or an intermediate between the two called block floating-point arithmetic (which was in use in the 1960s (Wilkinson 1963, Wang *et al.* 2019)).

Variables of different precisions arise in machine learning algorithms from quantization, the process of reducing the number of bits per operand. The limiting case is binarization, in which a number has just two possible values, 0 and 1. Quantization is applied in various ways, including during training or on a trained model.

One of the first papers to popularize the use of low precision arithmetic in deep learning is by Courbariaux, Bengio and David (2015), who find that '*very low precision is sufficient* not just for running trained networks but *also for training them*.'

Several reasons have been suggested by numerical analysts for the success of low precision floating-point arithmetic in machine learning. Scheinberg (2016) argues that in machine learning we are solving the wrong problem, namely a surrogate for the original optimization problem, so we do not need an accurate solution. It can also be argued that low precision arithmetic provides regularization and that this is beneficial to machine learning, perhaps by leading to flat minima rather than narrow minima.

In machine learning one often updates a parameter $\phi$ by a sequence of small quantities $h_i$ through a recurrence $\phi^{(i+1)} \leftarrow \phi^{(i)} + h_i$, $i = 1\colon n$. If $h_i$ is of absolute value less than half the spacing of the floating-point numbers around $\phi^{(i)}$, which is more likely in low precision arithmetic, then $\phi^{(i)} + h_i$ rounds to $\phi^{(i)}$ with round to nearest, so the information in $h_i$ is lost, and if this happens for many $i$ then the error in $\phi^{(n+1)}$ can be large. This phenomenon is called *stagnation*. It can be avoided by using stochastic rounding in place of round to nearest (Connolly, Higham and Mary 2021, Croci *et al.* 2022). Stochastic rounding is a randomized form of rounding that rounds to the next larger or next smaller floating-point number with probability proportional to 1 minus the distance to those floating-point numbers. An early use in deep learning was by Gupta, Agrawal, Gopalakrishnan and Narayanan (2015), who find that 'deep networks can be trained using only 16-bit wide fixed-point number representation when using stochastic rounding, and incur little to no degradation in the classification accuracy.'

https://doi.org/10.1017/S0962492922000022 Published online by Cambridge University Press

### 1.2.4. *HPL-AI Mixed Precision Benchmark*

The HPL-AI Mixed Precision Benchmark[1] is intended to measure supercomputer performance on AI-type workloads. It solves a double precision non-singular linear system $Ax = b$ of order $n$ using an LU factorization without pivoting computed in half precision and it refines the solution using preconditioned GMRES in double precision. As of November 2021, the world record execution rate for the benchmark is 2.0 ExaFlop/s ($2 \times 10^{18}$ floating-point operations per second), where most of the operations are half precision ones, for a matrix of size 16 957 440, which was achieved by the Fugaku supercomputer in Japan (Kudo, Nitadori, Ina and Imamura 2020*a*,*b*). The choice of matrix $A$ for the benchmark is critical, as it must be cheap to compute, have a controlled condition number, and have a numerically stable LU factorization without pivoting; a class of matrices having these properties is derived by Fasi and Higham (2021).

## 2. Floating-point arithmetics

Support for more than one precision of floating-point arithmetic, provided in hardware or software, has existed throughout the history of digital computing. A landmark was the Fortran 66 standard (ANSI 1966), which included the real and double precision data types and so made it possible to write portable programs that used two precisions.

Some early machines that supported $d$-digit but not $2d$-digit arithmetic offered the ability to accumulate an inner product of $d$-digit vectors in a $2d$-digit accumulator, only rounding back to $d$ digits after the final addition. This mode of computation was discussed by von Neumann and Goldstine (1947, Section 2.3) and was exploited by Wilkinson (1948, 1961) on the ACE computer and by Moler (1967) on the IBM 7094. Even earlier, desk calculating machines such as the Brunsviga offered accumulators with more digits than the input or the registers (Croarken 1985, Section 1.2.1).

Up to the mid 1980s, most computers used for scientific computing offered both single precision and double precision floating-point arithmetic, but the formats of the precisions varied greatly between machines. For example, a double precision number had a 96-bit significand on the Cray-1, a 53-bit significand on the DEC VAX (G format) and a 14-hexadecimal digit significand on the IBM 3090. This lack of uniformity, and more importantly the differing properties of the arithmetics, hindered the development of software intended to perform consistently across different machines.

### 2.1. *IEEE arithmetics*

A major breakthrough for scientific computing was the publication of the ANSI/IEEE Standard 754-1985 for Binary Floating-Point Arithmetic (IEEE 1985), which

[1] https://icl.bitbucket.io/hpl-ai/

https://doi.org/10.1017/S0962492922000022 Published online by Cambridge University Press

Table 2.1. Parameters for five floating-point arithmetics: number of bits in significand (including implicit most significant bit) and exponent (sig, exp); unit roundoff $u$; smallest positive (subnormal) number $x^s_{\min}$; smallest positive normalized number $x_{\min}$; and largest finite number $x_{\max}$. The last four columns are given to three significant figures. In Intel's bfloat16 specification subnormal numbers are not supported (Intel Corporation 2018).

| | (sig, exp) | $u$ | $x^s_{\min}$ | $x_{\min}$ | $x_{\max}$ |
|---|---|---|---|---|---|
| bfloat16 | $(8, 8)$ | $3.91 \times 10^{-3}$ | $9.18 \times 10^{-41}$ | $1.18 \times 10^{-38}$ | $3.39 \times 10^{38}$ |
| fp16 | $(11, 5)$ | $4.88 \times 10^{-4}$ | $5.96 \times 10^{-8}$ | $6.10 \times 10^{-5}$ | $6.55 \times 10^{4}$ |
| fp32 | $(24, 8)$ | $5.96 \times 10^{-8}$ | $1.40 \times 10^{-45}$ | $1.18 \times 10^{-38}$ | $3.40 \times 10^{38}$ |
| fp64 | $(53, 11)$ | $1.11 \times 10^{-16}$ | $4.94 \times 10^{-324}$ | $2.22 \times 10^{-308}$ | $1.80 \times 10^{308}$ |
| fp128 | $(113, 15)$ | $9.63 \times 10^{-35}$ | $6.48 \times 10^{-4966}$ | $3.36 \times 10^{-4932}$ | $1.19 \times 10^{4932}$ |

provided binary floating-point formats and precise rules for carrying out arithmetic on them. The standard had been carefully designed over several years by a committee of experts from academia and industry, and it brought much-needed order to computer arithmetic (Kahan 1981). Within a few years virtually all computer manufacturers had adopted it.

The 1985 standard prescribed two floating-point number formats: 32-bit single precision and 64-bit double precision. A 2008 revision (IEEE 2008) added a 128-bit quadruple precision format and a 16-bit half precision format, the latter defined as a storage format only rather than for computation. The half precision format was motivated by the emergence of support for 16-bit formats on graphical processing units (GPUs), where these formats were used for graphics and gaming.

To define the IEEE formats we recall that a floating-point number system is a finite subset $F = F(\beta, t, e_{\min}, e_{\max})$ of $\mathbb{R}$ whose elements have the form

$$x = \pm m \times \beta^{e-t+1}. \tag{2.1}$$

Here, $\beta$ is the base, which is 2 on virtually all current computers. The integer $t$ is the precision and the integer $e$ is the exponent, which lies in the range $e_{\min} \le e \le e_{\max}$, and the IEEE standard requires that $e_{\min} = 1 - e_{\max}$. The significand $m$ is an integer satisfying $0 \le m \le \beta^t - 1$. To ensure a unique representation for each non-zero $x \in F$ it is assumed that $m \ge \beta^{t-1}$ if $x \ne 0$, so that the system is *normalized*.

The largest and smallest positive numbers in the system are $x_{\max} = \beta^{e_{\max}}(\beta - \beta^{1-t})$ and $x_{\min} = \beta^{e_{\min}}$, respectively. Two other important quantities are $u = \frac{1}{2}\beta^{1-t}$, the *unit roundoff*, and $\epsilon = \beta^{1-t}$, the *machine epsilon*, which is the distance from 1 to the next larger floating-point number.

Numbers with $e = e_{\min}$ and $0 < m < \beta^{t-1}$ are called *subnormal numbers*. They have the minimal exponent but fewer than $t$ digits of precision. They form an equally spaced grid between 0 and the smallest normalized number.

https://doi.org/10.1017/S0962492922000022 Published online by Cambridge University Press

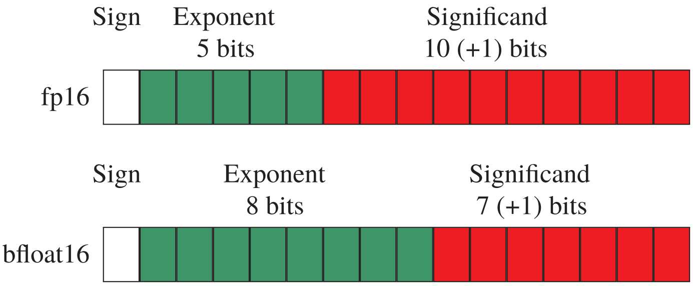

Figure 2.1. Comparison of the 16-bit bfloat16 and fp16 floating-point number formats. The '+1' refers to the implicit leading bit of the significand, which is not stored.

The parameters for the IEEE formats are given in Table 2.1. We refer to these formats as 'fpxy', where the integer 'xy' specifies the number of bits in a floating-point number (the IEEE standard uses the terminology 'binaryxy').

What is perhaps most striking is the great difference in ranges $[x_{\min}^s, x_{\max}]$ between the formats, and especially the narrow range $[x_{\min}, x_{\max}] \approx [6 \times 10^{-5}, 6 \times 10^4]$ for fp16. This means that a large proportion of fp32 and fp64 numbers are not representable as finite, non-zero fp16 numbers; as we will see, this means that careful scaling is needed in mixed precision algorithms that use fp16.

### 2.2. *Other arithmetics*

A floating-point number format called bfloat16 was proposed by researchers in the Google Brain artificial intelligence research group. Like fp16, it is a 16-bit format, but it allocates bits between the significand and exponent differently: as illustrated in Figure 2.1, bfloat16 allocates 8 bits for the significand and 8 bits for the exponent versus 11 bits for the significand 5 bits for the exponent for fp16. As shown in Table 2.1, the range of bfloat16 is very similar to that of fp32 (but not identical, because of its narrower significand), which means that overflow in converting to bfloat16 from higher precisions is much less likely than for fp16. The drawback of bfloat16 is its low precision: about three decimal digits of precision versus four for fp16. Bfloat16 has been taken up by Intel (Intel Corporation 2018), Arm and NVIDIA (beginning with the Ampere architecture).

There is no generally accepted 8-bit quarter precision format, though suggestions have been made by Moler (2017) (and implemented in MATLAB by Moler[2]), Tagliavini *et al.* (2018) and Wang *et al.* (2018).

Double-double arithmetic is a form of quadruple precision arithmetic in which a quadruple precision number is represented as the unevaluated sum of two double precision numbers, one representing the higher-order bits of the significand and

[2] http://mathworks.com/matlabcentral/fileexchange/59085-cleve-laboratory

https://doi.org/10.1017/S0962492922000022 Published online by Cambridge University Press

the other the lower-order bits (Muller *et al.* 2018, Section 14.1). Double-double arithmetic has a long history going back to the 1960s (Li *et al.* 2002). It is a 'poor man's quadruple precision' in that it is slightly less accurate than quadruple precision, has roughly the same range as double precision, and does not inherit all the desirable properties of the underlying double precision arithmetic (Joldes, Muller and Popescu 2017).

Other floating-point formats have been proposed for specific applications. For example, a 9-bit format with a 4-bit significand and a 5-bit exponent is proposed by O'uchi *et al.* (2018) for use in deep learning.

### *2.3. Availability in hardware and software*

IEEE single and double precision arithmetic began to be widely supported in hardware in the late 1980s and early 1990s. In fact, the Intel 8087 coprocessor, produced before the standard was published, partly supported it. In principle, single precision arithmetic operations should run twice as fast as their double precision counterparts, and single precision variables have the benefit of requiring half the storage of double precision ones, resulting in less data movement, but on Intel chips single precision had no speed advantage over double precision until the late 1990s, when Streaming SIMD Extensions (SSE) instructions were introduced (Langou *et al.* 2006).

IEEE fp16 arithmetic began to be supported on NVIDIA GPUs in the Maxwell architectures (Jetson TX1, 2014) and was included in the subsequent Pascal (P100, 2016), Volta (V100, 2017), Turing (T4, 2018) and Ampere (A100, 2020) architectures. Fp16 is also supported on AMD GPUs in the GCN and CDNA architectures.

Bfloat16 is supported on Google's TPUs (Norrie *et al.* 2021), the NVIDIA A100 GPU (Choquette *et al.* 2021, NVIDIA Corporation 2020), the ARM NEON architecture (ARM 2018) and Armv8-A architecture (ARM 2019), the Fujitsu A64FX ARM processor (Dongarra 2020, Sato *et al.* 2020) and the Intel Xeon Cooper Lake processors. It is not only high-end devices that support half precision: the Raspberry Pi, which uses the Armv8-A architecture, supports Bfloat16 (Groote, Morel, Schmaltz and Watkins 2021, Section 7.2.1).

The future Chinese Sunway exascale computer is scheduled to have double precision arithmetic running at 1 ExaFlop/s and half precision arithmetic running at 4 ExaFlop/s (Gao *et al.* 2021, Section 4).

Quadruple precision arithmetic is available almost exclusively in software. It is supported by some compilers, such as the GNU Compiler Collection (GCC),[3] and in MATLAB through the Symbolic Math Toolbox[4] and the Multiprecision Computing Toolbox.[5] Quadruple precision arithmetic is supported in hardware on

[3] https://gcc.gnu.org/

[4] http://www.mathworks.co.uk/products/symbolic/

[5] http://www.advanpix.com

https://doi.org/10.1017/S0962492922000022 Published online by Cambridge University Press

the IBM Power9 processor (Trader 2016) and the IBM z13 processor (Lichtenau, Carlough and Mueller 2016).

A set of extended and mixed precision Basic Linear Algebra Subprograms (BLAS) known as the XBLAS[6] provides extended and mixed precision counterparts of selected level 1, 2 and 3 BLAS (Li *et al.* 2002). They use extended precision internally, defined to mean a precision at least 1.5 times as accurate as double precision and wider than 80 bits. The input and output arguments remain single or double precision variables, but some arguments can be of mixed type (real or complex) as well as mixed precision (single or double), and the main visible difference is an extra input argument that specifies the precision at which internal computations are to be performed. A reference implementation is provided that employs the double-double format described in the previous subsection.

### *2.4. Block fused multiply–adds*

Since the 1990s some processors have provided a fused multiply–add (FMA) operation that computes $x + yz$ with just one rounding error instead of two: $x + yz$ is essentially computed exactly and then rounded. The motivation for an FMA is speed, as it can be implemented in a pipelined fashion so that it takes about the same time as a single multiplication or addition (Muller *et al.* 2018, Section 3.4.2).

In recent years, mixed precision block FMAs (also known as mixed precision matrix multiply–accumulate accelerators) have become available in hardware. In general, such a device takes as input matrices $A \in \mathbb{R}^{b_1 \times b}$, $B \in \mathbb{R}^{b \times b_2}$ and $C \in \mathbb{R}^{b_1 \times b_2}$, where $A$ and $B$ are provided in a given precision $u_{\text{low}}$ and $C$ is either in precision $u_{\text{low}}$ or in a higher precision $u_{\text{high}}$, and computes

$$\underbrace{D}_{u_{\text{low}} \text{ or } u_{\text{high}}} = \underbrace{C}_{u_{\text{low}} \text{ or } u_{\text{high}}} + \underbrace{A}_{u_{\text{low}}} \underbrace{B}_{u_{\text{low}}}, \tag{2.2}$$

returning $D$ in precision $u_{\text{low}}$ or $u_{\text{high}}$.

The tensor cores in the NVIDIA Volta and Turing architectures have $b_1 = b = b_2 = 4$. They require the matrices $A$ and $B$ to be in the fp16 format, $C$ and the result $D$ can be in fp16 or fp32, and internal computations are done in fp32 (Appleyard and Yokim 2017). Pictorially, we have

$$\overset{D}{\underbrace{\begin{bmatrix} \times & \times & \times & \times \\ \times & \times & \times & \times \\ \times & \times & \times & \times \\ \times & \times & \times & \times \end{bmatrix}}_{\text{fp16 or fp32}}} \overset{=}{=} \overset{C}{\underbrace{\begin{bmatrix} \times & \times & \times & \times \\ \times & \times & \times & \times \\ \times & \times & \times & \times \\ \times & \times & \times & \times \end{bmatrix}}_{\text{fp16 or fp32}}} \overset{+}{+} \overset{A}{\underbrace{\begin{bmatrix} \times & \times & \times & \times \\ \times & \times & \times & \times \\ \times & \times & \times & \times \\ \times & \times & \times & \times \end{bmatrix}}_{\text{fp16}}} \overset{B}{\underbrace{\begin{bmatrix} \times & \times & \times & \times \\ \times & \times & \times & \times \\ \times & \times & \times & \times \\ \times & \times & \times & \times \end{bmatrix}}_{\text{fp16}}}.$$

[6] https://netlib.org/xblas/

https://doi.org/10.1017/S0962492922000022 Published online by Cambridge University Press

Table 2.2. Processing units or architectures equipped with mixed precision block fused multiply–add accelerators. Matrix dimensions are expressed as $b_1 \times b \times b_2$, where the matrix product is of a $b_1 \times b$ matrix with a $b \times b_2$ matrix. The input and output precisions $u_{\text{low}}$ and $u_{\text{high}}$ are defined in (2.2). Sources: ARM (2020), Choquette *et al.* (2021), Jouppi *et al.* (2020, 2021), Norrie *et al.* (2021).

| Year of release | Device | Matrix dimensions | $u_{\text{low}}$ | $u_{\text{high}}$ |
|---|---|---|---|---|
| 2016 | Google TPU v2 | $128 \times 128 \times 128$ | bfloat16 | fp32 |
| 2017 | Google TPU v3 | $128 \times 128 \times 128$ | bfloat16 | fp32 |
| 2020 | Google TPU v4i | $128 \times 128 \times 128$ | bfloat16 | fp32 |
| 2017 | NVIDIA V100 | $4 \times 4 \times 4$ | fp16 | fp32 |
| 2018 | NVIDIA T4 | $4 \times 4 \times 4$ | fp16 | fp32 |
| 2019 | ARMv8.6-A | $2 \times 4 \times 2$ | bfloat16 | fp32 |
| 2020 | NVIDIA A100 | $8 \times 8 \times 4$ | bfloat16 | fp32 |
| | | $8 \times 8 \times 4$ | fp16 | fp32 |
| | | $8 \times 4 \times 4$ | TensorFloat-32 | fp32 |
| | | $2 \times 4 \times 2$ | fp64 | fp64 |

The Ampere architecture offers a wider choice of input data types for the tensor cores, including bfloat16 and fp32 (NVIDIA Corporation 2020).

Other instances of block FMAs are the matrix units (MXUs) available on Google TPUs (Jouppi *et al.* 2020, 2021, Wang and Kanwar 2019). They use bfloat16 rather than fp16 as the low precision format and operate on square matrices of dimension 128. Google TPUs are not commercially available.

Table 2.2 summarizes the properties of block FMAs available in some current hardware. We note that the details of the computations, such as rounding modes, normalization of intermediate results and whether subnormal numbers are supported, are generally not available and so must be inferred from experimentation. Fasi, Higham, Mikaitis and Pranesh (2021) investigate NVIDIA tensor cores; among their findings is that the inner products within the matrix multiplications use round towards zero for the additions and can be non-monotonic.

## 2.5. *Multiprecision arithmetic*

Multiprecision floating-point arithmetic is a built-in feature of Maple[7] and Mathematica[8] as well as the open-source PARI/GP[9] and Sage[10] computer algebra systems.

[7] http://www.maplesoft.com
[8] http://www.wolfram.com
[9] http://pari.math.u-bordeaux.fr
[10] http://www.sagemath.org

https://doi.org/10.1017/S0962492922000022 Published online by Cambridge University Press

It is available in MATLAB through the Symbolic Math Toolbox and the Multiprecision Computing Toolbox. The programming language Julia[11] (Bezanson, Edelman, Karpinski and Shah 2017) supports multiprecision floating-point numbers by means of the built-in data type `BigFloat`. For other languages third-party libraries are available.

Python: mpmath[12] (Johansson *et al.* 2013) and SymPy[13] (Meurer *et al.* 2017).

C: the GNU Multiple Precision Arithmetic Library[14] and the GNU MPFR Library[15] (Fousse *et al.* 2007).

C++: the BOOST libraries.[16]

C++ and Fortran: the ARPREC library (Bailey, Hida, Li and Thompson 2002).

C++: the MPFUN2020 library (Bailey 2021).[17]

The GNU MPFR Library is used in some of the software mentioned above, and interfaces to it are available for several programming languages. It was originally intended for high precisions, though recent work has improved its efficiency for fp64 and fp128 (Lefèvre and Zimmermann 2017). As the documentation notes,[18] the default exponent range is wide and ‘subnormal numbers are not implemented (but can be emulated).’

Nakata (2021) has produced a multiprecision version of the LAPACK library called MPLAPACK by translating the LAPACK source code from Fortran to C++. MPLAPACK has several options for the underlying arithmetic, including quadruple precision provided by GCC, double-double arithmetic, quad-double arithmetic (which represents a number as the unevaluated sum of four double precision numbers, so has about twice the precision of quadruple precision), the GNU Multiple Precision Arithmetic Library and the GNU MPFR Library. The test results reported in Nakata (2021) indicate a roughly 1:5:10 ratio of the time for double precision arithmetic, double-double arithmetic and quadruple precision arithmetic for matrix multiplication on an AMD Ryzen multicore CPU.

### *2.6. Simulating different precisions*

When developing mixed precision algorithms one may not have access in hardware to all the precisions of interest. Or one may wish to experiment with floating-point formats not yet supported in hardware. It is therefore useful to be able

[11] http://julialang.org
[12] http://mpmath.org
[13] http://www.sympy.org
[14] http://gmplib.org/
[15] http://www.mpfr.org
[16] http://www.boost.org
[17] https://www.davidhbailey.com/dhbsoftware/
[18] https://www.mpfr.org/mpfr-current/mpfr.html

https://doi.org/10.1017/S0962492922000022 Published online by Cambridge University Press

to simulate arithmetics of different precisions using arithmetic of a given higher precision available in hardware. This capability has proved particularly useful for half precision arithmetic, since initially fp16 was available only on GPUs and bfloat16 on Google TPUs. In addition to half precision, support for other binary formats specified by the user via the number of bits in the significand and the exponent is desirable, as well as support for different rounding modes. These features differentiate the simulations from the multiprecision arithmetics described in the previous section, some of which are parametrized by the number of base 10 digits.

The MATLAB function chop[19] of Higham and Pranesh (2019) rounds the elements of a matrix stored in single precision or double precision to a lower precision using one of several forms of rounding, with the result stored in the original precision. The target format for the rounding is specified by the number of bits in the significand and the maximum value of the exponent. The bfloat16, fp16 and fp32 formats are built-in. Subnormal numbers can be included or not. Six rounding modes are supported: round to nearest using round to even last bit to break ties (the default), round towards plus infinity, round towards minus infinity, round towards zero, and two forms of stochastic rounding. The chop function makes it easy to adapt existing codes to mixed precision by wrapping statements in calls to chop, and since the chop function is vectorized, few calls to it are typically needed for linear algebra codes.

The library CPFloat[20] by Fasi and Mikaitis (2020) offers similar functionality to chop for C. It comes with a MEX interface to MATLAB, and calling CPFloat can be faster than calling chop for large matrices. Fasi and Mikaitis (2020) offer a comparison with some other available packages for simulating low precision floating-point arithmetics.

Another approach to simulation is to provide a new storage class and overload operators to do arithmetic on the class. The `fp16` half precision MATLAB class of Moler (2017)[21] introduces a new data type fp16 that implements the fp16 storage format and overloads some basic functions for fp16 arguments. Arithmetic in this class is slow because of both the overhead of object orientation in MATLAB and the cost of converting to and from the fp16 storage format. Moler has also written a class `vfp16` that allows the partitioning of a 16-bit word between significand and exponent to be varied, in particular allowing bfloat16 to be simulated (Moler 2019). This class also allows subnormals to be included or not and FMAs to be done within the inner products inside a matrix multiplication.

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

[20] https://github.com/mfasi/cpfloat

[21] http://mathworks.com/matlabcentral/fileexchange/59085-cleve-laboratory

https://doi.org/10.1017/S0962492922000022 Published online by Cambridge University Press

Half precision simulations are available in some other languages.

Julia: the built-in float16 (fp16) datatype and the bfloat16 package[22] provide simulations of these half precision arithmetics.
C++: a header file for fp16 is available.[23]
Python: NumPy provides a float16 data type.[24]

The rpe (reduced floating-point precision) library of Dawson and Düben (2017) provides a derived type and overloaded operators for Fortran and was developed for use in weather and climate modelling. It emulates the specified precision but in general uses the exponent range of double precision.

With all these simulations it is important to realize that one might obtain more accurate results than for a true low precision computation because certain operations may be done in higher precision. For example, a language that supports matrix operations and provides a half precision data type may implement half precision matrix operations by doing them at higher precision and rounding to half precision. For a detailed discussion of the resulting differences in accuracy, see Higham and Pranesh (2019, Section 3).

## 3. Rounding error analysis model

We denote by fl the operator that rounds a real number into the floating-point number system $F$ whose elements are given by (2.1). We recall that if $x$ is in the range of $F$,

$$\mathrm{fl}(x) = x(1+\delta), \qquad |\delta| \le u,$$

where $u$ is the unit roundoff (Higham 2002, Theorem 2.2). Unless otherwise stated, when the argument of fl is an expression *expr*, fl(*expr*) denotes the result of evaluating that expression in floating-point arithmetic.

We will use the standard model of floating-point arithmetic (Higham 2002, Sec 2.2), which states that

$$\mathrm{fl}(x \mathbin{\mathrm{op}} y) = (x \mathbin{\mathrm{op}} y)(1+\delta), \qquad |\delta| \le u, \qquad \mathrm{op} = +, -, *, /. \tag{3.1}$$

This model is certainly satisfied by IEEE arithmetic (in the absence of underflow or overflow), which defines $\mathrm{fl}(x \mathbin{\mathrm{op}} y)$ to be the rounded exact value.

A constant that appears in rounding error analyses is

$$\gamma_n = \frac{nu}{1-nu} \qquad (nu < 1).$$

We will use the notation $u_{16}$, $u_{32}$, $u_{64}$ and $u_{128}$ to denote the unit roundoffs corresponding to IEEE arithmetics with the indicated word sizes. These values are given in the third column of Table 2.1.

[22] https://github.com/JuliaMath/BFloat16s.jl
[23] http://half.sourceforge.net/
[24] https://numpy.org/

https://doi.org/10.1017/S0962492922000022 Published online by Cambridge University Press

The rounding error bounds we state in this paper are mostly worst-case bounds and can be very pessimistic. For blocked algorithms, worst-case bounds that are smaller by a factor equal to the block size can be obtained for many algorithms, as explained by Higham (2021). Moreover, under suitable assumptions on the rounding errors, probabilistic bounds with constants that are the square roots of the constants in the worst-case bounds can be obtained; see Section 4.3. These observations are important because for low precisions a constant $nu$ (say) in a worst-case rounding error bound can exceed 1 even for very modest $n$.

## 4. Matrix multiplication

In this section we consider the computation of $C = AB$, where $A \in \mathbb{R}^{m\times n}$ and $B \in \mathbb{R}^{n\times p}$ are two general matrices. If all operations are carried out in a uniform precision $u$, the computed $\widehat{C}$ satisfies the standard bound (Higham 2002, Section 3.5)

$$|\widehat{C} - C| \le \gamma_n |A||B|, \tag{4.1}$$

where $|A|$ denotes the matrix of absolute values, $(|a_{ij}|)$.

The presence of the dimension $n$ in bound (4.1), which reflects the fact that rounding errors accumulate along the inner dimension, may prevent the computation from achieving sufficient accuracy when $n$ is large or $u$ is large. Various approaches have therefore been proposed to reduce the effect of error accumulation, and mixed precision arithmetic is at the heart of several of them.

### *4.1. Using block FMAs*

A matrix product can be computed with the aid of a block FMA (2.2). We will assume that the internal computations are done at precision $u_{\text{high}}$.

Block FMAs can be chained together by taking the output $D$ at precision $u_{\text{high}}$ and using it as the input $C$ to a subsequent FMA. Block FMAs thereby provide a natural way to mitigate error accumulation, as accumulation occurs at the level of $u_{\text{high}}$, not $u_{\text{low}}$. The required conversion of $A$ and $B$ to $u_{\text{low}}$ is the only source of error of order $u_{\text{low}}$, and it does not depend on the matrix dimensions.

Algorithm 4.1 shows how to use a block FMA to compute a general matrix product. Three precisions are in play: the working precision $u$ and the precisions $u_{\text{low}}$ and $u_{\text{high}}$, where $u_{\text{high}} \le u_{\text{low}}$.

The following error bound, a special case of Blanchard *et al.* (2020*b*, Theorem 3.2), describes the result of Algorithm 4.1.

**Theorem 4.1.** Let the product $C = AB$ of $A \in \mathbb{R}^{m\times n}$ and $B \in \mathbb{R}^{n\times t}$, given in precision $u$, be evaluated by Algorithm 4.1, where $q = n/b$. The computed $\widehat{C}$ satisfies

$$|\widehat{C} - C| \le (2u_{\text{low}} + u_{\text{low}}^2 + nu_{\text{high}} + O(u_{\text{high}}u_{\text{low}}))|A||B|. \tag{4.2}$$

https://doi.org/10.1017/S0962492922000022 Published online by Cambridge University Press

**Algorithm 4.1.** Let $A \in \mathbb{R}^{m\times n}$ and $B \in \mathbb{R}^{n\times t}$, given in precision $u$, be partitioned into $b_1 \times b$ blocks $A_{ij}$ and $b \times b_2$ blocks $B_{ij}$, respectively, where $p = m/b_1$, $q = n/b$ and $r = t/b_2$ are assumed to be integers. This algorithm performs the matrix multiplication $C = AB$ using a block FMA.

```
$\widetilde{A} \leftarrow \mathrm{fl}_{\mathrm{low}}(A)$, $\widetilde{B} \leftarrow \mathrm{fl}_{\mathrm{low}}(B)$
for $i = 1\colon p$
    for $j = 1\colon r$
        $C_{ij} = 0$
        for $\ell = 1\colon q$
            Compute $C_{ij} = C_{ij} + \widetilde{A}_{i\ell}\widetilde{B}_{\ell j}$ using a block FMA with output
            at precision $u_{\mathrm{high}}$.
        end
        Convert $C_{ij}$ to precision $u$.
    end
end
```

Theorem 4.1 is applicable to NVIDIA Volta and Turing tensor cores with $b = 4$, $u_{\mathrm{low}} = u_{16}$ and $u_{\mathrm{high}} = u_{32}$. The theorem is also applicable to the latest Ampere generation of NVIDIA GPUs, where $A$ and $B$ can also be stored in bfloat16 or tfloat32 arithmetics.[25]

We note that a more general error analysis is given in Blanchard *et al.* (2020*b*, Theorem 3.2) that allows for a different precision in the internal block FMA evaluation.

Optimized low precision BLAS are available in vendor libraries, such as in NVIDIA's cuBLAS library. Open-source implementations are also available, such as that of San Juan *et al.* (2021), who target the ARM v8.2 architecture, and Abdelfattah, Tomov and Dongarra (2019*a*), who provide batched multiplication routines for NVIDIA GPUs. A batched operation is one in which many independent operations on small matrices are grouped together and carried out by a single routine, and the batched BLAS standard described by Abdelfattah *et al.* (2021*b*) includes half precision and quadruple precision data types.

### 4.2. *Blocked summation*

In the absence of block FMA hardware with internal computations in higher precision, reduced error bounds can still be achieved by changing the summation algorithm used to compute each element $c_{ij} = \sum_{k=1}^{n} a_{ik}b_{kj}$. In particular, blocked algorithms, which are widely used in numerical linear algebra, compute the sum $s = \sum_{k=1}^{n} x_k$ by grouping summands $x_k$ into blocks of size $b$. Partial sums of $b$

[25] Tfloat32 is a format introduced by NVIDIA for use in tensor cores that has the range of fp32 and the precision of fp16.

summands are first computed independently, before being combined into the final result. By doing so, the term $\gamma_n$ in the error bound (4.1) is reduced to $\gamma_{b+n/b-1}$, because rounding errors incurred in different blocks do not accumulate. Indeed, in forming $c_{ij}$, precisely $(n/b)(b-1) = n - n/b$ of the additions are carried out in computing the partial sums, and these account for the term $bu$ in the bound. Only the last $n/b - 1$ additions account for the error term $(n/b-1)u$. This observation creates an opportunity for mixed precision arithmetic: by computing these last $n/b - 1$ additions in higher precision (say, in precision $u^2$), we can obtain an error bound independent of $n$ to first order. The next result is a special case of Blanchard *et al.* (2020*a*, Theorem 4.2).

**Theorem 4.2.** Let the product $C = AB$ of $A \in \mathbb{R}^{m\times n}$ and $B \in \mathbb{R}^{n\times p}$ be evaluated by computing the inner products $c_{ij} = \sum_{k=1}^{n} a_{ik}b_{kj}$ by blocks of size $b$, where partial sums of each block are computed in precision $u$ before being combined in precision $u^2$. The computed $\widehat{C}$ satisfies

$$|\widehat{C} - C| \le ((b+1)u + (n/b + b^2 - 1)u^2 + O(u^3))|A||B|. \tag{4.3}$$

This mixed precision summation algorithm is an instance of the FABsum algorithm (Blanchard *et al.* 2020*a*), which computes the partial sums with a fast summation and combines them with an accurate summation. The reduction of the error bound is achieved at a modest extra cost, because most of the additions are still carried out in precision $u$.

Fasi *et al.* (2022) implement FABsum on NVIDIA GPUs using the CUTLASS[26] library to improve the accuracy of multiword matrix multiplication (see Section 13). Their implementation achieves an improved performance–accuracy tradeoff compared with cuBLAS: depending on the choice of block size and precisions, FABsum can be either as fast as cuBLAS with fp16 tensor cores, but more accurate, or as accurate as cuBLAS with fp32 arithmetic, but faster.

### 4.3. *Probabilistic analyses*

The bounds (4.1)–(4.3) are worst-case bounds and they do not reflect the fact that rounding errors of opposite signs can partially cancel each other. Under some assumptions on the rounding errors, probabilistic error analyses (Connolly *et al.* 2021, Connolly and Higham 2022, Higham and Mary 2019*a*, 2020, Ipsen and Zhou 2020) show that the dimension-dependent constants in the bounds can be replaced by their square roots. The underlying assumptions of these analyses are not always satisfied; one way to enforce them is to use stochastic rounding (Connolly *et al.* 2021).

In the case where the matrices $A$ and $B$ are generated by sampling their entries from a random distribution, the sharper analysis of Higham and Mary (2020) shows that the means of the entries play an important role. Specifically, for zero-mean

[26] https://github.com/nvidia/cutlass

https://doi.org/10.1017/S0962492922000022 Published online by Cambridge University Press

data, the error bound is independent of $n$. Therefore, given a general matrix, a natural idea is to shift its entries so that they have zero mean. Computing the product $C = AB$ of the shifted matrices and shifting back the result can provide a much more accurate result (Higham and Mary 2020, Theorem 4.2). Shifting back the result has negligible cost for large dimensions, but it must be carried out in higher precision.

### *4.4. Multiword matrix multiplication with block FMAs*

The emergence of block FMA hardware that allows for accumulation in higher precision (see Section 2.4) has provided new opportunities to efficiently implement matrix multiplication using multiword arithmetic, such as double-fp16 arithmetic, which approximates fp32 arithmetic by representing numbers as the unevaluated sum of two fp16 numbers. These approaches are described in Section 13.

### *4.5. Data-driven matrix–vector product*

Recently, various inner product and matrix–vector product algorithms have been proposed based on the idea of storing each element in a precision adapted to its magnitude. These approaches are described in Section 14.4.

## 5. Non-linear equations

Consider a system of non-linear algebraic equations

$$F(x) = 0, \quad F : \mathbb{R}^n \to \mathbb{R}^n. \tag{5.1}$$

Many problems of interest can be formulated in this form, so we consider the use of mixed precision arithmetic in this general context before specializing to particular problems.

Suppose we have an iteration $x_{k+1} = g(x_k)$ that generates a sequence of vectors $x_0, x_1, \dots$ converging to a solution $x_*$. An obvious idea is to use on each iteration arithmetic of the lowest available precision that equals or exceeds the accuracy of the iterates. Therefore we use low precision arithmetic for the early iterations and increase the precision as the iteration proceeds, until the last few iterations are done at the working precision. The justification is that the rounding errors committed on the early iterations should be dominated by the inherent iteration errors. If the iteration is globally convergent this approach should produce a solution of quality as good as if the working precision were used throughout, because such an iteration damps out errors, and each iterate $x_k$ can be regarded as restarting the iteration with a new starting value.

The possible gain in speed from using mixed precision arithmetic in this way depends on the number of iterations required, which in turn depends on the rate of convergence and on the cost per iteration. Consider a quadratically convergent

https://doi.org/10.1017/S0962492922000022 Published online by Cambridge University Press

iteration and a working precision of double. If at the end of the first iteration the error is $10^{-1}$, the subsequent errors will ideally be $10^{-2}$, $10^{-4}$, $10^{-8}$ and $10^{-16}$. We might carry out the first three iterations at half precision, the fourth at single precision and the fifth at double precision. Assuming each iteration requires the same number of operations and a ratio of 1:2:4 for the costs of half, single and double precision arithmetic, the overall cost will be a fraction $(3/4 + 1/2 + 1)/5 = 9/20$ of the cost of carrying out all the iterations in double precision. A greater speedup will be obtained if a greater proportion of the iterations are carried out at lower precisions, as will be the case if the initial non-asymptotic convergence phase is long, but in this case alternative iterations or methods might be more efficient. The assumption that each iteration requires the same number of operations may not hold, as we will see in Section 6, and this is why greater speedups are possible.

Varying the precisions in the way just described does not always work. In particular, it fails for iterations for matrix functions that are not self-correcting, such as the Newton iteration for the unitary polar factor (a solution to $X^*X = I$): $X_{k+1} = (X_k + X_k^{-*})/2$, $X_0 = A \in \mathbb{C}^{n\times n}$ (Higham 1986; Higham 2008, Chapter 8). The iteration formula is independent of $A$, so if we perturb $X_k \to X_k + E_k$ with a general $E_k$ with $\|E_k\| \gg u$ (as opposed to the specially structured errors that appear in the exact arithmetic iteration), then important information about $A$ has been lost and convergence to a matrix with error of order $u$ cannot be obtained.

### 5.1. Newton's method

Newton's method is an excellent method for solving (5.1) and it includes various particular methods of interest as special cases. Therefore we will carry out an analysis of Newton's method in mixed precision arithmetic. Because the analysis is general, it may be suboptimal for particular problems, but it will reveal features common to all.

We suppose that $F$ is continuously differentiable and denote by $J$ its Jacobian matrix $(\partial F_i/\partial x_j)$. Given a starting vector $x_0$, Newton's method for (5.1) generates a sequence $\{x_i\}$ defined by

$$J(x_i)(x_{i+1} - x_i) = -F(x_i), \quad i \geq 0. \tag{5.2}$$

As is well known, under appropriate conditions $x_i$ converges to a solution $x_*$ from any starting vector $x_0$ sufficiently close to $x_*$, and the rate of convergence is quadratic if $J(x_*)$ is non-singular (Dennis and Schnabel 1983, Theorem 5.2.1). We consider the mixed precision implementation of Newton's method given in Algorithm 5.1. Here, we evaluate $f$ in a possibly higher precision $u_r$ and solve for the update $d_i$ at a possibly lower precision $u_\ell$ (hoping that the resulting errors are damped out). In this and the later algorithms, $i_{\max}$ is a limit on the number of iterations and 'or until converged' means that the iteration will be terminated if an unspecified convergence test based on the residual or an estimate of the forward error is satisfied.

https://doi.org/10.1017/S0962492922000022 Published online by Cambridge University Press

**Algorithm 5.1.** Newton's method for $F(x) = 0$ with starting vector $x_1$, in precisions $u_\ell$, $u$ and $u_r$ ($u_r \le u \le u_\ell$).

```
for $i = 1 : i_{\max}$ or until converged
    Compute $f_i = F(x_i)$ in precision $u_r$.
    Solve $J(x_i)d_i = -f_i$ in precision $u_\ell$.
    $x_{i+1} = x_i + d_i$ at precision $u$.
end
```

Accounting for rounding and approximation errors, we can write the computed iterates $\widehat{x}_i$ as

$$\widehat{x}_{i+1} = \widehat{x}_i - (J(\widehat{x}_i) + E_i)^{-1}(F(\widehat{x}_i) + e_i) + \epsilon_i. \tag{5.3}$$

The error terms are explained as follows.

- $e_i$ is the error made in computing $F(\widehat{x}_i)$, and we assume that there is a function $\psi$ depending on $F$, $\widehat{x}_i$, $u$ and $u_r$ such that

$$\|e_i\| \le u\|F(\widehat{x}_i)\| + \psi(F, \widehat{x}_i, u, u_r). \tag{5.4}$$

- $E_i$ combines the error incurred in forming $J(\widehat{x}_i)$ with the backward error for solving the linear system for $d_i$. We assume that

$$\|E_i\| \le \phi(F, \widehat{x}_i, n, u_\ell, u), \tag{5.5}$$

  for some function $\phi$ that reflects both the (in)stability of the linear system solver and the error made when approximating or forming $J(\widehat{x}_i)$. In practice, we certainly have $\phi(F, \widehat{x}_i, n, u_\ell, u) \ge u\|J(\widehat{x}_i)\|$.

- $\epsilon_i$ is the rounding error made when adding the correction $\widehat{d_i}$ to $\widehat{x}_i$, so

$$\|\epsilon_i\| \le u(\|\widehat{x}_i\| + \|\widehat{d_i}\|).$$

The norm is any absolute vector norm (one for which $\||v|\| = \|v\|$ for all $v$) and the corresponding subordinate matrix norm.

Note that (5.3) is a very general model, and with a suitable choice of $E_i$ it yields modified Newton methods, in which the Jacobian is held constant for several iterations in order to reduce the cost of the method.

We wish to know how the precisions affect (a) sufficient conditions for convergence and (b) the limiting accuracy and limiting residual, that is, how small $\|x_* - \widehat{x}_i\|$ and $\|F(\widehat{x}_i)\|$ are guaranteed to become as $i$ increases, where $x_*$ is the solution to which the iteration would converge in the absence of errors.

We will assume that $J$ is Lipschitz continuous with constant $\theta_L$, that is,

$$\|J(v) - J(w)\| \le \theta_L\|v - w\| \quad \text{for all } v, w \in \mathbb{R}^n.$$

Analyses of the effects of different sources of error on Newton's method are available in the literature, for example in Kelley (1995, Section 5.4) and Kelley

(2022). Most useful for our purposes are results of Tisseur (2001). The results were originally stated for the situation where just two precisions are in use ($u_\ell = u$), but they are general enough to support a third precision $u_\ell$ as well. The first result bounds the limiting accuracy. Here we use the condition number $\kappa(A) = \|A\|\|A^{-1}\|$.

**Theorem 5.1 (Tisseur).** Assume that there is an $x_*$ such that $F(x_*) = 0$ and $J_* = J(x_*)$ is non-singular with

$$\kappa(J_*)u \le \frac{1}{8}. \tag{5.6}$$

Assume also that for $\phi$ in (5.5),

$$\|J(\widehat{x}_i)^{-1}\|\,\phi(F, \widehat{x}_i, n, u_\ell, u) \le \frac{1}{8} \quad \text{for all } i. \tag{5.7}$$

Then, for all $x_0$ such that

$$\theta_L \|J_*^{-1}\|\,\|x_0 - x_*\| \le \frac{1}{8}, \tag{5.8}$$

Newton's method in floating-point arithmetic generates a sequence $\{\widehat{x}_i\}$ satisfying

$$\|\widehat{x}_{i+1} - x_*\| \le \alpha_i \|\widehat{x}_i - x_*\| + \beta_i, \tag{5.9}$$

where

$$\begin{aligned}\alpha_i &\approx \|J(\widehat{x}_i)^{-1}E_i\| + \|J_*^{-1}\|\|\widehat{x}_i - x_*\| + \kappa(J_*)u,\\ \beta_i &\approx \|J_*^{-1}\|\|\psi(F, \widehat{x}_i, u, u_r)\| + u\|x_*\|,\end{aligned}$$

and the normwise relative error decreases until the first $i$ for which

$$\frac{\|\widehat{x}_{i+1} - x_*\|}{\|x_*\|} \approx \frac{\|J_*^{-1}\|}{\|x_*\|}\psi(F, x_*, u, u_r) + u. \tag{5.10}$$

As a check, we note that in the absence of errors, the terms $u$, $\psi(F, v, u, u_r)$ and $\phi(F, v, n, u_\ell, u)$ are all zero and thus Theorem 5.1 implies local quadratic convergence of Newton's method.

In words, Theorem 5.1 says that if $J(x_*)$ is not too ill-conditioned, the Jacobian evaluation and the solver are not too inaccurate, the Lipschitz constant $\theta_L$ is not too large and the initial guess $x_0$ is not too far from $x_*$, then the limiting accuracy is proportional to the condition of the Jacobian at the solution and the accuracy with which the residual is evaluated. Note that the function $\phi$ does not appear in (5.10), which shows that errors in forming $J$ and solving the linear system do not affect the limiting accuracy, provided they are not too large. The $\alpha_i$ term in (5.9) shows that these errors do, however, affect the rate of convergence, and that this rate is essentially independent of $u_r$.

The next result bounds the limiting residual.

**Theorem 5.2 (Tisseur).** Under the assumptions of Theorem 5.1, if

$$\theta_L \|J_*^{-1}\|(\|J_*^{-1}\|\psi(F, x_*, u, u_r) + u\|x_*\|) \le \frac{1}{8},$$

https://doi.org/10.1017/S0962492922000022 Published online by Cambridge University Press

then, for all $x_0$ such that (5.8) holds, the sequence $\{\|F(\widehat{x}_i)\|\}$ of residual norms generated by Newton's method in floating-point arithmetic decreases until

$$\|F(\widehat{x}_{i+1})\| \approx \psi(F, \widehat{x}_i, u, u_r) + u\|J(\widehat{x}_i)\| \, \|\widehat{x}_i\|. \tag{5.11}$$

Theorem 5.2 shows that, under very similar conditions to those in Theorem 5.1, the limiting residual is at the level of the error made in computing the residual plus the term $u\|J(\widehat{x}_i)\| \, \|\widehat{x}_i\|$. This latter term is inevitable: from the Taylor series

$$F(x_* + \Delta x_*) = F(x_*) + J(x_*)\Delta x_* + O(\|\Delta x_*\|^2),$$

we see that merely rounding the exact solution to $\widetilde{x}_* = x_* + \Delta x_*$, so that $\|\Delta x_*\| \le u\|x_*\|$, gives

$$\|F(\widetilde{x}_*)\| \lesssim \|J(x_*)\| \, \|\Delta x_*\| \le u\|J(x_*)\| \, \|x_*\|.$$

Just as for the limiting accuracy, the limiting residual does not depend on the errors in evaluating $J$ or in solving the linear systems.

Since the limiting accuracy and limiting residual both depend on $\psi$, Theorems 5.1 and 5.2 confirm the folklore that Newton's method must be provided with good function values if it is to work well in practice.

As an application, we consider a linear system $F(x) = b - Ax = 0$, where $A \in \mathbb{R}^{n\times n}$ is non-singular. In principle, Newton's method converges in one step, but in the presence of errors it becomes an iterative method, namely iterative refinement. Here, we have $\theta_L = 0$. Computing $F$ at precision $u_r$ and rounding to precision $u$ gives

$$\psi(F, \widehat{x}_i, u, u_r) \approx \gamma_{n+1}^r(\|b\| + \|A\|\|\widehat{x}_i\|),$$

where $\gamma_{n+1}^r = (n+1)u_r/(1-(n+1)u_r)$. Hence Theorem 5.1 shows a limiting accuracy

$$\begin{aligned}\frac{\|\widehat{x}_{i+1} - x_*\|}{\|x_*\|} &\approx \frac{\|A^{-1}\|}{\|x^*\|}\gamma_{n+1}^r(\|b\| + \|A\|\|\widehat{x}_i\|) + u \\ &\lesssim 2\kappa(A)\gamma_{n+1}^r + u,\end{aligned}$$

since $\|\widehat{x}_i\| \approx \|x_*\|$ and $\|b\| \le \|A\|\|x_*\|$. Hence, if $u_r = u^2$, the limiting accuracy is of order $u$ for $\kappa(A) < u^{-1}$. Theorem 5.2 gives a limiting residual

$$\begin{aligned}\|b - A\widehat{x}_{i+1}\| &\approx \gamma_{n+1}^r(\|b\| + \|A\|\|\widehat{x}_i\|) + u\|A\|\|\widehat{x}_i\| \\ &\lesssim (nu_r + u)(\|b\| + \|A\|\|\widehat{x}_i\|),\end{aligned}$$

which means a backward error of order $nu_r + u$. Both theorems require (5.7) to hold, and since we expect $\phi$ to be proportional to $u_\ell$ for a solver in precision $u_\ell$, this condition is essentially of the form $c_n\kappa(A)u_\ell < 1$ for some constant $c_n$.

This very general analysis of Newton's method for $Ax = b$ provides significant insight into mixed precision iterative refinement, even though we have not specified the details of the solver. We will derive more specific and detailed results in the next section.

https://doi.org/10.1017/S0962492922000022 Published online by Cambridge University Press

## 6. Iterative refinement for $Ax = b$

We consider the general iterative refinement algorithm given in Algorithm 6.1 for solving a non-singular linear system $Ax = b$, based on the use of precisions $u_r \le u$ and $u_\ell \ge u$ in addition to the working precision $u$. The method used to solve for the update vector $d_i$ on line 3 is arbitrary; we will specialize to particular solvers in the following sections.

For a discussion of stopping tests see Higham (2002, Section 12.3).

**Algorithm 6.1.** Given a non-singular matrix $A \in \mathbb{R}^{n\times n}$, $b \in \mathbb{R}^n$ and an initial approximation $x_1$, this algorithm uses iterative refinement to solve $Ax = b$. The algorithm uses three precisions satisfying $u_r \le u \le u_\ell$.

```
1  for $i = 1 : i_{\max}$ or until converged
2      Compute $r_i = b - Ax_i$ in precision $u_r$.
3      Solve $Ad_i = r_i$ at precision $u_\ell$.
4      Update $x_{i+1} = x_i + d_i$ in precision $u$.
5  end
```

We denote the relative error in the solution computed on line 3 by

$$\xi_i = \frac{\|d_i - \widehat{d}_i\|_\infty}{\|d_i\|_\infty}. \tag{6.1}$$

Let

$$\mu_i = \frac{\|A(x_i - \widehat{x}_i)\|_\infty}{\|A\|_\infty \|x_1 - \widehat{x}_i\|_\infty} \le 1, \tag{6.2}$$

$$\phi_i = 2\min(\operatorname{cond}(A), \kappa_\infty(A)\mu_i)u_\ell + \xi_i, \tag{6.3}$$

where the condition number $\operatorname{cond}(A) = \|\,|A^{-1}||A|\,\|_\infty$. We also need the condition number

$$\operatorname{cond}(A, x) = \frac{\|\,|A^{-1}||A||x|\,\|_\infty}{\|x\|_\infty}.$$

Note that $\operatorname{cond}(A, x) \le \operatorname{cond}(A) \le \kappa_\infty(A)$. The next result is by Carson and Higham (2018, Corollary 3.3).

**Theorem 6.1.** Let Algorithm 6.1 be applied with any $x_1$ to a linear system $Ax = b$, where $A \in \mathbb{R}^{n\times n}$ is non-singular. As long as $\phi_i$ in (6.3) is sufficiently less than 1, the forward error is reduced on the $i$th iteration by a factor of approximately $\phi_i$ until an iterate $\widehat{x}$ is produced for which

$$\frac{\|\widehat{x} - x\|_\infty}{\|x\|_\infty} \lesssim u + 4p\operatorname{cond}(A, x)u_r, \tag{6.4}$$

where $p$ is the maximum number of non-zeros in any row of $[A\ b]$.

Theorem 6.1 shows that the limiting accuracy (6.4) depends on the precisions $u$ and $u_r$ but does not depend on the precision $u_\ell$, on $x_1$, or on how the system $Ad_i = r_i$ at line 3 is solved, provided that it is solved with some relative accuracy $\xi_i \ll 1$.

The limiting accuracy (6.4) motivates the use of extended precision in the computation of the residual. Indeed, if we set $u_r = u^2$, we obtain a limiting accuracy of order $u$, independent of the conditioning of the problem as long as $\operatorname{cond}(A, x)u \leq 1$.

## 6.1. *Historical development*

### 6.1.1. *Traditional iterative refinement*

Iterative refinement was programmed on a digital computer by Wilkinson (1948, p. 111 ff.), using LU factorization with partial pivoting as the solver. Wilkinson, and subsequent authors, took advantage in computing the residual of the ability of many machines of the time to accumulate inner products at twice the working precision at little or no extra cost (as discussed at the start of Section 2). The method was also used by Wilkinson and colleagues on desk calculating machines, making use of their extra length accumulators in computing residuals (Fox, Huskey and Wilkinson 1948).

Iterative refinement with extra precision residuals fell out of favour in the 1970s because machines began to lack the ability to accumulate inner products in extra precision. Indeed the LINPACK library did not include it because it could not be implemented in a portable way in Fortran (Dongarra, Bunch, Moler and Stewart 1979).

### 6.1.2. *Fixed precision iterative refinement*

As the traditional form of iterative refinement declined in popularity, another usage came to the fore: fixed precision refinement, in which only one precision is used. Jankowski and Woźniakowski (1977) proved that an arbitrary linear equation solver is made normwise backward stable by the use of fixed precision iterative refinement, as long as the solver is not too unstable to begin with and $A$ is not too ill-conditioned. Skeel (1980) analysed fixed precision iterative refinement for LU factorization with partial pivoting and showed that one step of refinement yields a small componentwise backward error under suitable conditions. Higham (1991) extended the componentwise backward error analysis of fixed precision iterative refinement to a general solver, and Higham (1997) gave an analysis that covers the traditional and fixed precision forms and a general solver.

### 6.1.3. *Iterative refinement with lower precision solves*

In the 2000s, hardware emerged in which fp32 arithmetic was much faster than fp64 arithmetic, such as Intel chips with SSE instructions (a factor about 2) and the Sony/Toshiba/IBM (STI) Cell processor (a factor up to 14) (Kurzak and Dongarra

https://doi.org/10.1017/S0962492922000022 Published online by Cambridge University Press

2007). Motivated by this speed difference, Langou *et al.* (2006) proposed a new usage of iterative refinement in which LU factors computed at a precision *lower* than the working precision (specifically, single versus double precision) are used to solve on line 3 in Algorithm 6.1.

Carson and Higham (2017) showed how preconditioned GMRES can be exploited on line 3 in Algorithm 6.1, giving an algorithm called GMRES-based iterative refinement (GMRES-IR). Carson and Higham (2018) proposed a three-precision version of iterative refinement, essentially Algorithm 6.1, and gave detailed convergence analysis for the backward error (normwise and componentwise) and the forward error.

Amestoy *et al.* (2021*b*) extended the analysis of Carson and Higham (2018) to a five-precision form of GMRES-IR.

More details of these works are given in Sections 7 and 8.

### *6.2. Specialized applications*

Much work has been done on specializing iterative refinement to particular contexts. We mention just a few examples.

Govaerts and Pryce (1990) develop and analyse an iterative refinement-based algorithm for solving bordered linear systems $Ax = b$ of order $n$ in which a black box solver is assumed to be available for systems involving the submatrix $A(1\colon n-1, 1\colon n-1)$. One application is to numerical continuation problems.

In some structured problems the elements of $A$ are never formed and so residuals cannot be computed in the usual way via matrix–vector multiplication. An example is when $A$ is a Vandermonde matrix and a fast $O(n^2)$ flops algorithm tailored to the structure is being used. Higham (1988) develops algorithms for solving Vandermonde-like systems where $a_{ij} = p_{i-1}(\alpha_j)$, with $\{p_i(x)\}_{i=0}^{n-1}$ a set of polynomials satisfying a three-term recurrence relation, such as orthogonal polynomials. The algorithms are numerically unstable for the Chebyshev polynomials, but one step of iterative refinement at the working precision is found to give stability. The residual is evaluated by a nested multiplication algorithm for orthogonal polynomials.

By steadily increasing the precision during the iterative refinement process it is possible to compute solutions to arbitrarily high accuracy, assuming that arithmetic of suitable precisions is available. This idea, first suggested in an exercise by Stewart (1973, pp. 206–207) has been investigated by Kiełbasiński (1981) and Smoktunowicz and Sokolnicka (1984).

## 7. Direct methods for $Ax = b$

In this section we discuss the solution of linear systems by direct methods based on a factorization of the matrix.

https://doi.org/10.1017/S0962492922000022 Published online by Cambridge University Press

### *7.1. LU factorization-based iterative refinement*

Algorithm 7.1 is a version of Algorithm 6.1 based on an LU factorization of $A$, hereinafter referred to as LU-IR. The LU factorization is computed in precision $u_\ell$ and is used to compute the initial solution $x_1$ and solve the update equation on line 5.

The only line of the algorithm that costs $O(n^3)$ flops is the first line, as the substitutions cost only $O(n^2)$ flops. The factorization is carried out at precision $u_\ell$, so if $u_\ell \gg u$, then if the iteration converges quickly the algorithm is potentially significantly faster than solving $Ax = b$ by LU factorization at precision $u$.

**Algorithm 7.1 (LU-IR).** Given a non-singular matrix $A \in \mathbb{R}^{n\times n}$ and $b \in \mathbb{R}^n$, this algorithm uses LU factorization-based iterative refinement with three precisions satisfying $u_r \le u \le u_\ell$, to solve $Ax = b$.

```
1  Compute the factorization $A = LU$ in precision $u_\ell$.
2  Solve $LUx_1 = b$ by substitution in precision $u_\ell$.
3  for $i = 1 : i_{\max}$ or until converged
4      Compute $r_i = b - Ax_i$ in precision $u_r$.
5      Solve $LUd_i = r_i$ by substitution in precision $u_\ell$.
6      Update $x_{i+1} = x_i + d_i$ in precision $u$.
7  end
```

Standard error analysis (Higham 2002, Theorem 9.4) shows that solving a linear system by substitution in precision $u_\ell$ with LU factors computed in precision $u_\ell$ achieves a relative error bounded approximately by $3n\||A^{-1}||\widehat{L}||\widehat{U}|\|u_\ell$. Theorem 6.1 therefore yields the following result.

**Theorem 7.1.** Let LU-IR (Algorithm 7.1) be applied to a linear system $Ax = b$, where $A \in \mathbb{R}^{n\times n}$ is non-singular. If $\||A^{-1}||\widehat{L}||\widehat{U}|\|u_\ell$ is sufficiently less than 1, then the algorithm produces an iterate $\widehat{x}$ satisfying (6.4).

As mentioned in Section 6, the traditional and fixed precision forms of iterative refinement use an LU factorization computed by LU factorization with partial pivoting with $u_\ell = u$. With such a stable LU factorization, the convergence condition in Theorem 7.1 reduces to $\kappa(A)u \ll 1$. However, as already mentioned, unstable solvers can still lead to convergence. In the context of LU-IR, two main approaches have been proposed that introduce potential instability in an attempt to increase speed.

The first approach is to use a potentially unstable LU factorization in precision $u$, where the instability can come from different sources. For example, using a weaker form of pivoting to accelerate the factorization and preserve the sparsity of the matrix, such as static pivoting (Li and Demmel 1998, Arioli, Duff, Gratton and Pralet 2007), can still lead to the convergence of LU-IR. Several sparse direct solvers incorporate static pivoting strategies as an option, such as MUMPS (Amestoy, Duff, L'Excellent and Koster 2001, Amestoy, Buttari, L'Excellent and Mary 2019), which implements the approach proposed by Duff and Pralet (2007), or as the default, such as SuperLU_DIST (Li and Demmel 2003) and PARDISO (Schenk, Gärtner, Fichtner and Stricker 2001). Other potentially unstable, but faster, factorizations have been combined with iterative refinement to remedy their instability, such as incomplete LU factorization (Zlatev 1982) or Cholesky factorization for quasidefinite systems (Gill, Saunders and Shinnerl 1996).

The second approach is to use an LU factorization in lower precision $u_\ell > u$. If the LU factorization algorithm is numerically stable, convergence is guaranteed provided that $\kappa(A)u_\ell \ll 1$, as noted above. This approach is attractive because most of the work ($O(n^3)$ flops for dense systems) is done in the factorization phase; the iterative phase ($O(n^2)$ flops) has negligible cost for large $n$, as long as the number of iterations remains reasonable. Thus, asymptotically, we may expect the speed of the entire solution to be determined by the speed of the lower precision arithmetic. Using the Cell processor, Langou *et al.* (2006) solve double precision linear systems ($u = u_{64}$) with speedups of up to a factor of 8 over a double precision LU factorization by using LU-IR (Algorithm 7.1) with $u_\ell = u_{32}$ and $u_r = u$. Further experimental results are reported by Buttari *et al.* (2007) for dense linear systems and by Buttari *et al.* (2008) for sparse ones. See Baboulin *et al.* (2009) for an overview of the methods developed in this period.

Iterative refinement with LU factorization in lower precision has also been exploited on FPGAs (Sun, Peterson and Storaasli 2008).

The popularity of LU-IR with a lower precision factorization grew again with the emergence of half precision arithmetic (fp16 and bfloat16). Indeed, half precision arithmetic is at least four times faster than double precision arithmetic, and possibly much more than that on some hardware, notably on NVIDIA tensor cores (see Section 7.3). Haidar, Wu, Tomov and Dongarra (2017) provide the first evaluation of the potential of half precision for iterative refinement, obtaining speedups of up to 2.7 on an NVIDIA P100 GPU using LU-IR with $u = u_r = u_{64}$ and $u_\ell = u_{16}$. Kudo *et al.* (2020*a*,*b*) implement LU-IR on the Fugaku supercomputer, which is equipped with ARM-based Fujitsu A64FX processors that support fp16 arithmetic, for use in the HPL-AI Mixed Precision Benchmark (see Section 1.2.4).

LU-IR with a half precision factorization can only guarantee convergence for well-conditioned problems: the condition $\kappa(A)u_\ell \ll 1$ translates to $\kappa(A) \ll 2000$ in fp16 and $\kappa(A) \ll 300$ in bfloat16. Two main approaches have been proposed to extend the applicability of half precision iterative refinement to a wider range of problems: the first uses a more accurate solver on line 5 of Algorithm 7.1 (see Section 7.2) and the second uses more accurate hardware such as tensor cores (see Section 7.3).

### *7.2. GMRES-based iterative refinement*

GMRES-IR (Carson and Higham 2017), mentioned in Section 6.1.3, is described in Algorithm 7.2.

https://doi.org/10.1017/S0962492922000022 Published online by Cambridge University Press

**Algorithm 7.2 (GMRES-IR).** Given a non-singular matrix $A \in \mathbb{R}^{n\times n}$ and $b \in \mathbb{R}^n$, this algorithm solves $Ax = b$ using by GMRES-IR in five precisions: $u_r$, $u_g$, $u_p$, $u$ and $u_\ell$.

```
1 Compute the factorization $A = LU$ in precision $u_\ell$.
2 Solve $LUx_1 = b$ by substitution in precision $u_\ell$.
3 for $i = 1 : i_{\max}$ or until converged
4     Compute $r_i = b - Ax_i$ in precision $u_r$.
5     Solve $U^{-1}L^{-1}Ad_i = U^{-1}L^{-1}r_i$ by GMRES in precision $u_g$, performing
      the products with $U^{-1}L^{-1}A$ in precision $u_p$.
6     Compute $x_{i+1} = x_i + d_i$ in precision $u$.
7 end
```

Note that the application of the preconditioner on line 5 involves a multiplication by $A$ and substitutions with the LU factors. The results quoted below assume the use of a backward stable implementation of GMRES, such as MGS-GMRES (Paige, Rozložník and Strakoš 2006).

### *7.2.1. High accuracy solution of ill-conditioned systems*

Carson and Higham (2017) proposed GMRES-IR with two precisions, $u = u_\ell$ and $u_r = u_g = u_p = u^2$, and were interested in solving ill-conditioned systems to high accuracy. They showed that the quantity $\mu_i$ defined in (6.2) tends to be small in the early iterations and gradually grows to order 1 as the iteration proceeds. This means that in $\phi_i$ in (6.3) the min term is negligible in the early iterations and so $\phi_i \approx \xi_i$. Carson and Higham (2017) also show that $\xi_i \approx u$ as long as $\kappa(A)$ is not much larger than $u^{-1}$. Hence Theorem 6.1 guarantees that a limiting accuracy of order $u$ will be achieved. In other words, by using a small amount of computation at twice the working precision it is possible to solve $Ax = b$ to full accuracy even if $A$ is numerically singular!

It is important to emphasize that standard methods, such as even the singular value decomposition (SVD), will not in general yield an accurate solution to an ill-conditioned system. GMRES-IR computes an accurate solution to the update equation, which is relatively well-conditioned thanks to the preconditioning and which has an accurate right-hand side. The behaviour of $\mu_i$ is also crucial, and it had not been previously been proved or exploited, though Wilkinson (1977) did make an observation that is equivalent to saying that the $\mu_i$ increase with $i$.

### *7.2.2. Exploiting low precision LU factors of an ill-conditioned matrix*

As mentioned in Section 7.1, one of the main limitations of LU-IR (Algorithm 7.1) is that its success is guaranteed only when $\kappa(A)u_\ell \ll 1$. If the LU factorization is computed in low precision, LU-IR is therefore limited to well-conditioned matrices. In this setting, GMRES-IR becomes particularly useful. Indeed, even though

GMRES-IR was originally intended to solve linear systems nearly singular to the working precision $u$, as described in the previous subsection, Carson and Higham (2018) subsequently proposed using it to exploit LU factors computed in a precision $u_\ell$ (potentially much) lower than the working precision $u$. They assume that the preconditioner is applied in precision $u_g = u_p = u^2$. Amestoy *et al.* (2021*b*) relax this requirement by allowing the products with $U^{-1}L^{-1}A$ to be carried out in a precision $u_p$ possibly lower than $u^2$, and additionally allow the rest of the GMRES computations to be performed in a precision $u_g$ possibly lower than $u$. This results in the five-precision algorithm described in Algorithm 7.2. To obtain convergence guarantees for this algorithm, Amestoy *et al.* (2021*b*) generalize the analysis of Paige *et al.* (2006) on the backward stability of GMRES to a two-precision GMRES with LU preconditioning, and they prove the following theorem.

**Theorem 7.2.** Let GMRES-IR (Algorithm 7.2) be applied to a linear system $Ax = b$, where $A \in \mathbb{R}^{n\times n}$ is non-singular. If $\kappa(A)^2u_\ell^2(u_g + \kappa(A)u_p)$ is sufficiently less than 1, then the algorithm produces an iterate $\widehat{x}$ satisfying (6.4).

Amestoy *et al.* (2021*b*) consider the thousands of possible combinations of the five precisions in Algorithm 7.2 and narrow down the choice to a few combinations of practical interest, from among which one can balance accuracy, robustness and performance. The bounds on $\kappa(A)$ for convergence guarantees are not always sharp, so it can be difficult to decide which variant should be preferred for a particular problem. To address this issue, Oktay and Carson (2022) propose a multistage iterative refinement that switches to increasingly robust but also more expensive variants by monitoring key quantities during the iterative process.

Haidar, Tomov, Dongarra and Higham (2018*b*) and Haidar *et al.* (2020) implement GMRES-IR with just two precisions: the factorization is in half precision ($u_\ell$) and the rest of the operations are in double precision ($u_g = u_p = u_r = u$). They show that for several matrices where LU-IR takes a large number of iterations to converge, GMRES-IR can still converge in a small number of iterations and thus retains an attractive performance boost compared with LU-IR with a single precision factorization.

Higham and Mary (2019*b*) propose a new preconditioner that builds upon the low precision LU factors and exploits a low-rank approximation to speed up GMRES-IR.

### 7.3. *Harnessing tensor cores*

NVIDIA tensor cores present two benefits for iterative refinement compared with standard half precision arithmetic on NVIDIA GPUs. The first is that they are significantly faster and so computing the LU factorization with tensor cores provides more room to amortize the cost of the iterations in the iterative phase. The second benefit is their improved accuracy since, as discussed in Section 2.4, tensor cores accumulate intermediate operations in fp32 arithmetic.

https://doi.org/10.1017/S0962492922000022 Published online by Cambridge University Press

Tensor cores can carry out arbitrarily sized matrix products using Algorithm 4.1 and so can be naturally exploited by standard blocked LU factorization algorithms, which mostly consist of matrix–matrix products. Haidar *et al.* (2018*b*) propose an algorithm that harnesses tensor cores to accelerate the updates of the trailing submatrix, which account for the $O(n^3)$ flops of the factorization; the remaining $O(n^2)$ flops are carried out by standard floating-point units in fp32 arithmetic. Blanchard *et al.* (2020*b*, Theorem 4.4) analyse this algorithm and prove that it possesses a reduced backward error bound of order $u_{16} + nu_{32}$ instead of the standard bound $nu_{16}$ of an LU factorization entirely in fp16 arithmetic.

Using their mixed precision LU factorization algorithm within LU-IR or GMRES-IR, Haidar *et al.* (2018*b*) are able to solve linear systems with fp64 accuracy at a speed of up to 24 TFLOPS on an NVIDIA V100 GPU, which represents a speedup of 4 over a double precision solver. Moreover, for some of the more difficult matrices in their test set, the solution entirely in fp16 arithmetic requires many iterations or does not converge at all, whereas the algorithm using tensor cores maintains a fast convergence. This shows that the accuracy boost of tensor cores can strongly improve the convergence of iterative refinement. The use of half precision and/or tensor cores also improves the energy efficiency of the solution, reducing power consumption by up to a factor of 5 (Haidar *et al.* 2018*a*). See Haidar *et al.* (2020) for a more complete discussion of iterative refinement with tensor cores, and Abdelfattah, Tomov and Dongarra (2019*b*) for an extension of these approaches to complex matrices. These ideas are implemented in the MAGMA library,[27] in the NVIDIA cuSOLVER library[28] and also in the SLATE library[29] (Charara *et al.* 2020), which targets machines with large numbers of cores and multiple hardware accelerators per node.

In addition to its speed and energy benefits, fp16 arithmetic can also be used to reduce memory consumption and data movement. However, special care has to be taken not to lose the accuracy boost of tensor cores. Indeed, tensor cores carry out computations internally in fp32 arithmetic, and so to benefit from their improved accuracy the input matrix $C$ in (2.2) needs to be stored in fp32. Lopez and Mary (2020) propose a modification of the above approaches of Haidar *et al.*, based on a left-looking Crout factorization that allows them to store the matrix in fp16 while accumulating computations in fp32 buffers of controlled size. As a result, memory consumption is halved and data movement costs are greatly reduced, making the factorization faster by up to a factor of 2 on NVIDIA V100 GPUs.

### 7.4. *Scaling strategies*

A limitation of iterative refinement with fp16 as the low precision arithmetic is the narrow range of the arithmetic: as seen in Table 2.1, numbers of magnitude outside

[27] https://icl.utk.edu/magma/

[28] https://developer.nvidia.com/cusolver

[29] https://icl.utk.edu/slate/

https://doi.org/10.1017/S0962492922000022 Published online by Cambridge University Press

the interval $[x_{\min}^s, x_{\max}] = [5.96 \times 10^{-8}, 6.55 \times 10^4]$ are not representable and will underflow or overflow when converted to fp16. Moreover, numbers of magnitude smaller than $x_{\min} = 6.10 \times 10^{-5}$ are often flushed to zero in practice, to avoid the possibly heavy performance penalty of handling subnormal numbers.

In the LU factorization of a matrix $A$ in fp16 arithmetic, overflow and underflow may occur during the initial conversion of $A$ to fp16 and also during the LU factorization itself. In particular, note that even LU factorization algorithms using tensor cores that keep the original matrix in fp32 are not immune to overflow and underflow, since the LU factors must be converted to fp16.

One way to deal with overflow in rounding $A$ to fp16 is to replace any element $a_{ij}$ that overflows by $\operatorname{sign}(a_{ij})\theta x_{\max}$, where $\theta \in (0, 1]$ is a parameter. We will refer to this as the overflow mapping strategy. This approach is used in Haidar *et al.* (2017, 2018*a*,*b*).

Higham, Pranesh and Zounon (2019) suggest Algorithm 7.3, which uses a two-sided diagonal scaling at the working precision and only rounds to fp16 once all the matrix elements do not exceed $x_{\max}$. The algorithm applies row and column equilibration, which produces a matrix $\widetilde{A}$ in which every row and column has maximum element in modulus equal to 1. Then it scales $\widetilde{A}$ so that the maximum element in modulus of the scaled matrix is $\theta x_{\max}$ and rounds to fp16. Here, $\theta$ is intended to be reasonably close to 1, in order to maximize the use of the limited fp16 range and keep the numbers away from the subnormal zone. If $A$ is symmetric, a symmetry-preserving two-sided scaling of Knight, Ruiz and Uçar (2014) can be used instead of row and column equilibration.

**Algorithm 7.3.** 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.

```
$R = \operatorname{diag}(\|A(i,:)\|_\infty^{-1})$
$\widetilde{A} = RA$          % $\widetilde{A}$ is row equilibrated.
$S = \operatorname{diag}(\|\widetilde{A}(:,j)\|_\infty^{-1})$
$\widetilde{A} = \widetilde{A}S$          % $\widetilde{A}$ is now row and column equilibrated.
Let $\beta$ be the maximum magnitude of any entry of $\widetilde{A}$.
$\mu = \theta x_{\max}/\beta$
$A^{(h)} = \mathrm{fl}_h(\mu\widetilde{A})$
```

How should $\theta$ be chosen? The main requirement is that there is no overflow in the LU factorization, which means that we need $\theta \le \rho_n^{-1}$, where $\rho_n$ is the growth factor for LU factorization on $A$. With partial pivoting, $\rho_n$ is typically not large, so one might take $\theta = 0.1$, as used in Higham *et al.* (2019). However, large growth factors can occur, notably for 'randsvd matrices' having one small singular value (Higham, Higham and Pranesh 2021), and this led to poor performance with $\theta = 0.1$ in one of the experiments in Haidar *et al.* (2020, Section 14(b)).

https://doi.org/10.1017/S0962492922000022 Published online by Cambridge University Press

Higham *et al.* (2019) show experimentally that compared with the overflow mapping strategy, Algorithm 7.3 leads to faster and more reliable convergence of GMRES-IR on badly scaled matrices.

Unless the LU factors are converted to fp32 precision at the end of the factorization (or are already available in fp32, such as when using tensor cores), the substitution operations must also be performed in fp16 arithmetic, and are therefore vulnerable to overflow and underflow, especially as the elements of the residual vector $r_i$ on line 4 of Algorithm 7.1 must eventually become of order $u(\|A\|\|x\| + \|b\|)$, and so are likely to underflow in fp16. Techniques for avoiding overflow in solving triangular systems can be found in Anderson (1991) and Demmel and Li (1994) (these are used in the LAPACK subroutine `xLATRS`), and they can be combined with the simple scaling suggested by Carson and Higham (2018, Section 6) and Luszczek, Yamazaki and Dongarra (2019).

### 7.5. *Exploiting symmetry and positive definiteness*

Suppose, now, that $A \in \mathbb{R}^{n\times n}$ is symmetric positive definite. In principle, LU-IR can be adapted in a straightforward way by replacing LU factorization with Cholesky factorization. However, there is a problem to overcome: a matrix that has elements stored in a given precision and is symmetric positive definite may lose definiteness when rounded to a lower precision, and in Algorithm 7.1 we round $A$ to precision $u_\ell$ on the first step. We can guarantee to preserve definiteness in the rounding only if $\kappa_2(A)u_\ell < 1$, which is a severe restriction if we are using half precision. Higham and Pranesh (2021) suggest Algorithm 7.4, which scales and shifts in order to ensure a successful Cholesky factorization. The two-sided scaling $H = D^{-1}AD^{-1}$, where $D = \mathrm{diag}(a_{ii}^{1/2})$, produces a unit diagonal matrix with off-diagonal elements bounded in magnitude by 1. This matrix is then shifted by an amount intended to lift the smallest eigenvalue sufficiently above zero, and a multiplicative factor $\theta$ is applied that plays the same role as that in Algorithm 7.3. As explained by Higham and Pranesh (2021), shifting $H$ by a multiple of $I$ is better than shifting $A$ by a multiple of $I$, as it is equivalent to shifting $A$ by a multiple of $\mathrm{diag}(a_{ii})$ and so it makes the same relative perturbation to each diagonal element of $A$.

Higham and Pranesh (2021) give perturbation analysis and error analysis that suggests taking $c \approx n^2$ in Algorithm 7.4, but they find this is too pessimistic in practice. They recommend taking $c$ as a small constant and found $c = 2$ to work well in practice, with no need for the doubling on line 8. They use this idea with an appropriate modification of GMRES-IR in which GMRES is applied to the preconditioned update equation $MAd_i = Mr_i$, where $M = \mu D^{-1}R^{-1}R^{-T}D^{-1}$.

Note that since $A$ is symmetric positive definite it is more natural to use the conjugate gradient (CG) method instead of GMRES, but the supporting rounding error analysis works only for GMRES, because it relies on the backward stability of GMRES and preconditioned CG is not guaranteed to be backward stable

https://doi.org/10.1017/S0962492922000022 Published online by Cambridge University Press

**Algorithm 7.4.** Given a symmetric positive definite $A \in \mathbb{R}^{n\times n}$ in precision $u$, this algorithm computes an approximate Cholesky factorization $R^TR \approx \mu D^{-1}AD^{-1}$ at precision $u_\ell > u$, where $D = \mathrm{diag}(a_{ii}^{1/2})$. The scalar $\theta \in (0, 1]$ and the positive integer $c$ are parameters.

```
1  $D = \mathrm{diag}(a_{ii}^{1/2})$, $H = D^{-1}AD^{-1}$     % Set $h_{ii} \equiv 1$ instead of computing it.
2  $G = H + cu_\ell I$
3  $\beta = 1 + cu_\ell$
4  $\mu = \theta x_{\max}/\beta$
5  $A_\ell = \mathrm{fl}_\ell(\mu G)$
6  Attempt Cholesky factorization $A_\ell = R^TR$ in precision $u_\ell$.
7  if Cholesky factorization failed
8     $c \leftarrow 2c$, goto line 2
9  end
```

(Greenbaum 1997, eq. (34)). However, Higham and Pranesh (2021) find that in their experiments CG works as well as GMRES.

Algorithm 7.4 has been implemented on an NVIDIA V100 GPU by Abdelfattah, Tomov and Dongarra (2020), effectively taking $u_\ell = u_{16}$, $u = u_r = u_{64}$, with Cholesky factorization computed in mixed fp16 and fp32 precisions. With matrices of dimensions up to 42 000, they obtained speedups of up to 4.7 over a double precision solver.

### *7.6. Sparse matrix considerations*

Sparsity presents both opportunities and obstacles to the use of iterative refinement.

On the one hand, while LU factorization of dense matrices tends to run twice as fast in single precision as in double precision, this speedup may not be attained for sparse matrices, for two reasons explained by Zounon, Higham, Lucas and Tisseur (2022). The first reason is that real-life sparse double precision matrices, such as many of those in the SuiteSparse Matrix Collection[30] (Davis and Hu 2011), can have elements of widely varying magnitudes. While the matrix elements usually fit into the range of single precision numbers, LU factorization can generate cascading fill-ins in which small multipliers combine to produce subnormal numbers. This can cause a significant performance loss because floating-point operations on subnormal numbers can be very slow. A cure is to set a compiler flag to flush subnormal numbers to zero. The second reason why LU factorization of a sparse matrix in single precision may not give the expected speedup over double precision is that the reordering and analysis phase of the algorithm does not involve floating-point arithmetic (Duff, Erisman and Reid 2017) and so does not benefit from reducing the precision. Moreover, if the reordering and analysis is sequential rather than

[30] https://sparse.tamu.edu/. Previously known as the University of Florida Sparse Matrix Collection.

parallelized, then increasing the number of cores increases the proportion of time spent on non-floating-point arithmetic computations.

On the other hand, some of the features of iterative refinement are especially attractive when the matrix is sparse. First, as explained by Amestoy *et al.* (2022), iterative refinement with a lower precision LU factorization can lead to significant memory savings due to the fact that the LU factors of a sparse matrix are typically much denser, and unlike for dense matrices, the overhead of keeping a high precision copy of the original matrix is negligible. Second, to best preserve the sparsity of the matrix, sparse direct solvers often employ relaxed pivoting strategies, such as threshold partial pivoting (Duff *et al.* 2017, Chapter 7) or the more aggressive static pivoting (Li and Demmel 1998), which can lead to large growth factors; iterative refinement can overcome any resulting numerical instability.

Amestoy *et al.* (2022) develop implementations of LU-IR and GMRES-IR based on a single precision sparse LU factorization computed with the multifrontal solver MUMPS and use them to solve with double precision accuracy a range of large and ill-conditioned sparse systems coming from a variety of applications. They obtain reductions of up to a factor of 2 in both execution time and memory consumption over the double precision MUMPS solver, with LU-IR being usually faster than GMRES-IR, although the latter is more robust and successfully converged for all test problems.

### *7.7. Exploiting data sparsity*

In many applications, the matrix possesses a so-called data sparse structure: many of its off-diagonal blocks have low numerical rank. In the last two decades, several approaches have been devised to leverage this property to accelerate the solution of linear solvers, such as hierarchical ($\mathcal{H}$) or block low-rank (BLR) methods.

The low-rank approximations are computed with a truncation threshold parameter $\varepsilon$, which controls the accuracy of these data sparse solvers, as proved by Higham and Mary (2021) in the case of BLR solvers. Thus data sparse solvers can be used either as direct solvers (setting $\varepsilon$ to the target accuracy) or as preconditioners to iterative methods. In particular, they can be used in conjunction with iterative refinement. Amestoy *et al.* (2022) use the BLR sparse solver MUMPS (Amestoy *et al.* 2019) at low accuracy with LU-IR and GMRES-IR, and obtain large performance gains with respect to the double precision solver, reducing execution time by up to 5.6× and memory consumption by up to 4.4×. Moreover, GMRES-IR can converge for larger values of the parameter $\varepsilon$ than LU-IR, which leads to increased performance in some cases.

In addition to their use in lower precision, data sparse solvers can also benefit from mixed precision. Indeed, data sparse matrices exhibit blocks of highly unequal importance: those that correspond to weak interactions (and that are usually far away from the diagonal) contain less significant information and are more resilient to the use of reduced precision. As a result, Abdulah *et al.* (2019) propose to store

https://doi.org/10.1017/S0962492922000022 Published online by Cambridge University Press

blocks that are sufficiently far way from the diagonal in single precision instead of double precision. They apply this strategy to the Cholesky factorization of data sparse covariance matrices arising in geostatistical modelling, obtaining an average 1.6× speedup. The approach is extended to also include half precision in Abdulah *et al.* (2022), leading to an improved 2.6× speedup. Doucet, Ltaief, Gratadour and Keyes (2019) use the same approach in a different application in computational astronomy (tomographic reconstructors) using only single and half precisions on NVIDIA V100 GPUs.

To go even further, in addition to storing different blocks in different precisions, each block can also use a mixed precision representation. Since most of the blocks of data sparse matrices exhibit rapidly decaying singular values, they are amenable to the mixed precision low-rank representation proposed by Amestoy *et al.* (2021*a*) and described in Section 12.2. Amestoy *et al.* (2021*a*) apply this approach to the LU factorization of BLR matrices and obtain storage and flops reductions of up to a factor of 3 using fp64, fp32 and bfloat16 arithmetics.

## 8. Iterative methods for $Ax = b$

We outline three classes of approaches to exploit mixed precision arithmetic in iterative methods. The first approach is to use an inner–outer scheme such as GMRES-IR, where the low precision is used by the inner scheme (Section 8.1). The second approach is to use low precision arithmetic to compute and/or apply the preconditioner in a higher precision iterative method (Section 8.2). The third approach is to intrinsically use mixed precision within the iterative method, such as for inexact Krylov methods (Section 8.3). Finally, we also comment on specific methods such as communication-avoiding or multigrid methods.

### 8.1. GMRES-IR without an LU factorization

Computing an LU factorization can be expensive, especially for large, sparse matrices. GMRES-IR can also be effective with a cheaper preconditioner $M^{-1}$, or with no preconditioner at all. In this latter case, Algorithm 7.2 reduces to Algorithm 8.1, which has the form of an inner–outer scheme: the outer loop for iterative refinement (in precision $u$, with the residual computed at a possibly higher precision $u_r$) and the inner loop for solving the correction equations with GMRES (assumed backward stable) in lower precision $u_\ell$. By Theorem 6.1 (or indeed Theorem 5.2), convergence to an iterate satisfying (6.4) is guaranteed as long as $\kappa(A)u_\ell \ll 1$. Note that inner solvers other than GMRES can be used, and, as long as they are backward stable, the convergence condition $\kappa(A)u_\ell \ll 1$ still holds.

Algorithm 8.1 is one form of mixed precision restarted GMRES, although to guarantee convergence the GMRES call on line 4 must not terminate after a fixed number of iterations, but rather when a sufficiently small residual has been achieved.

Algorithm 8.1 was first described by Turner and Walker (1992), who perform the inner loop in single precision ($u_\ell$) and the outer loop in double precision

https://doi.org/10.1017/S0962492922000022 Published online by Cambridge University Press

**Algorithm 8.1.** GMRES-based iterative refinement in three precisions for the solution of $Ax = b$ with no preconditioner.

```
Choose an initial $x_1$.
for $i = 1 : i_{\max}$ or until converged
    Compute $r_i = b - Ax_i$ in precision $u_r$.
    Solve $Ad_i = r_i$ by GMRES in precision $u_\ell$.
    Compute $x_{i+1} = x_i + d_i$ in precision $u$.
end
```

($u$ and $u_r$), and use a fixed number of inner GMRES iterations. Buttari *et al.* (2008) implement several inner–outer iterative algorithms similar to GMRES-IR employing single and double precisions for the solution of sparse linear systems. In particular, one version uses GMRES for the inner loop and FGMRES for the outer loop; this version is also studied by Baboulin *et al.* (2009).

More recent implementations of these methods, still using only single and double precisions, are described by Lindquist, Luszczek and Dongarra (2020, 2022) for CPUs and by Loe *et al.* (2021*a*,*b*) for GPUs. Iwashita, Suzuki and Fukaya (2020) propose a restarted GMRES where the inner loop uses integer arithmetic and the outer loop uses floating-point arithmetic.

To find a compromise between computing an LU factorization and using no preconditioner at all, cheaper preconditioners can be considered. Algorithm 8.2 is obtained by replacing $U^{-1}L^{-1}$ in Algorithm 7.2 with a general preconditioner $M^{-1}$.

**Algorithm 8.2.** GMRES-based iterative refinement in five precisions for the solution of $Ax = b$ with a general preconditioner $M^{-1} \approx A^{-1}$ stored in precision $u_\ell$.

```
Compute $x_1 = M^{-1}b$ in precision $u_\ell$.
for $i = 1 : i_{\max}$ or until converged
    Compute $r_i = b - Ax_i$ in precision $u_r$.
    Solve $M^{-1}Ad_i = M^{-1}r_i$ by GMRES in precision $u_g$, performing the
    products with $M^{-1}A$ in precision $u_p$.
    Compute $x_{i+1} = x_i + d_i$ in precision $u$.
end
```

There is a tradeoff involved, since a better quality preconditioner will lead to faster convergence but will be more expensive to compute. More subtly, the closer $M^{-1}$ is to $A^{-1}$, the more significant the rounding errors incurred in the matrix–vector products with $M^{-1}A$ become. Indeed, with $M^{-1} = U^{-1}L^{-1}$ (LU factorization-based preconditioner), we have explained in Section 7.2.2 that the products with $U^{-1}L^{-1}A$ introduce an extra $\kappa(A)$ term in the convergence condition, which can be attenuated by performing them in higher precision $u_p$. This error analysis has not been extended to a general preconditioner $M^{-1}$ in the literature, but we can expect

https://doi.org/10.1017/S0962492922000022 Published online by Cambridge University Press

$\kappa(A)$ in the error bound to be replaced by a more general term depending on both $A$ and $M^{-1}$.

Examples of implementations that use a preconditioner other than a low precision LU factorization are found in Lindquist *et al.* (2020, 2022), who use GMRES-IR preconditioned by an incomplete LU factorization, or in Loe *et al.* (2021*a*,*b*), where block Jacobi and polynomial preconditioners are used.

### 8.2. *Iterative methods with low or mixed precision preconditioner*

Another approach to exploiting mixed precision arithmetic in iterative methods is to use low precision to compute and/or apply the preconditioner. If the iterative method is iterative refinement, and the preconditioner is a low precision LU factorization, this corresponds to LU-IR (Algorithm 7.1). The idea can be extended to other iterative methods or preconditioners.

For example, Arioli and Duff (2009) show that FGMRES implemented in double precision and preconditioned with an LU factorization computed in single precision can give backward stability at double precision, even for ill-conditioned systems. Building on this work, Hogg and Scott (2010) implement an algorithm for symmetric indefinite systems that computes a solution using a direct solver in single precision, performs iterative refinement using the factorization of $A$, and then uses mixed precision FGMRES preconditioned by the direct solver to solve the original system.

Giraud, Haidar and Watson (2008) propose an fp32 domain decomposition preconditioner applied to an fp64 CG solver. Similarly, Emans and van der Meer (2012) propose the use of an fp32 algebraic multigrid method as preconditioner for an fp64 CG.

Anzt *et al.* (2019*a*) and Flegar, Anzt, Cojean and Quintana-Ortí (2021) implement a block Jacobi preconditioner within the preconditioned conjugate gradient method and store the explicitly inverted diagonal blocks of the preconditioner in half, single or double precision arithmetic according to a criterion based on the condition number of each block. In experiments that use the preconditioner within a conjugate gradient solver, Flegar *et al.* (2021) report reductions in run time of 10%–30% compared with a full precision implementation. Göbel, Grützmacher, Ribizel and Anzt (2021) apply the same idea to sparse approximate inverse preconditioning with a BiCGSTAB solver. It is worth noting that in these papers the preconditioner does not simply use low precision throughout but is itself in mixed precision.

### 8.3. *Mixed precision GMRES*

The previously described approaches introduce mixed precision in GMRES either in the preconditioner or via an inner–outer iteration scheme. However, there are opportunities to exploit multiple precisions even within a non-restarted, unpreconditioned GMRES.

https://doi.org/10.1017/S0962492922000022 Published online by Cambridge University Press

A first approach is to use lower precision in the matrix–vector products with $A$, based on the theory of inexact Krylov methods (Giraud, Gratton and Langou 2007, Simoncini and Szyld 2003, van den Eshof and Sleijpen 2004), which proves that an increasing level of inexactness as the iteration proceeds can be tolerated in the matrix–vector products without degrading the achievable accuracy. This was first experimentally observed by Bouras, Frayssé and Giraud (Bouras, Frayssé and Giraud 2000, Bouras and Frayssé 2005). The effect of inexactness on the convergence rate of the method is, however, not well understood.

In addition, Gratton, Simon, Titley-Peloquin and Toint (2019) prove that the orthonormalization of the Krylov basis can also be performed inexactly. This observation is leveraged by Aliaga *et al.* (2020), who propose to store the Krylov basis in lower precision.

### 8.4. *Communication-avoiding iterative methods*

On modern computers, communication has become a significant performance bottleneck. Communication-avoiding (CA) methods seek to reduce the communication costs, sometimes at the expense of additional flops, in order to achieve higher performance, especially when scaling to large numbers of processors. In particular, CA iterative methods often compute blocks of $s$ iterations at a time to reduce synchronization costs. However, these $s$-step approaches are known to be sometimes unstable. Mixed precision arithmetic has been used to overcome this potential instability.

Yamazaki, Tomov, Dong and Dongarra (2015*b*) and Yamazaki, Tomov and Dongarra (2015*a*) propose a mixed precision Cholesky–QR orthonormalization (described in Section 9) that they use to stabilize CA-GMRES. They show that the use of this stabilized orthonormalization avoids the need to orthogonalize twice and speeds up the convergence of GMRES.

Carson, Gergelits and Yamazaki (2022*a*) propose mixed precision $s$-step Lanczos and conjugate gradient methods that compute the Gram matrix in higher precision. This allows for reducing the loss of orthogonality by a factor relating to the condition number of the $s$-step Krylov bases, speeding up the convergence of the method at the expense of an increase of the per-iteration cost that is expected to be small in latency-bound applications.

### 8.5. *Multigrid iterative refinement*

In addition to Krylov methods, mixed precision has also been investigated for multigrid methods. The most popular approach has been to use a multigrid method as the inner solver for iterative refinement, that is, to use Algorithm 6.1 with a multigrid solver on line 3, usually run in lower precision. Single precision multigrid methods have, for example, been used within double precision iterative refinement algorithms by Göddeke, Strzodka and Turek (2007), Goddeke and Strzodka (2011), Sumiyoshi, Fujii, Nukada and Tanaka (2014) and Kronbichler and Ljungkvist

https://doi.org/10.1017/S0962492922000022 Published online by Cambridge University Press

(2019). More recently, Oo and Vogel (2020) also used fp16 arithmetic on V100 GPUs. The first error analysis of multigrid methods in this context was performed by McCormick, Benzaken and Tamstorf (2021), who observed that different levels in the grid hierarchy should use different precisions: coarser grids are more resilient to lower precisions. This 'progressive precision' approach was applied to the solution of elliptic PDEs by Tamstorf, Benzaken and McCormick (2021).

For more details of mixed precision multigrid algorithms see Abdelfattah *et al.* (2021*a*).

### 8.6. *Other iterative solvers*

Clark *et al.* (2010) gave an early investigation into mixed precision implementation of conjugate gradients (CG) and BiCGstab solvers on GPUs, for a lattice quantum chromodynamics application. They used half precision for storage only, since half precision computation was not available to them.

Anzt, Dongarra and Quintana-Ortí (2015) carried out the Jacobi iterative method with different solution components represented in different precisions, using an inexpensive test to decide when to increase precisions during the iteration.

### 8.7. *Decoupling formats for data storage and processing*

One specific feature of exploiting reduced precision in GMRES and iterative methods more generally is that performance is often limited by the memory bandwidth. This leads to the idea of storing the data in compressed form and uncompressing it before performing arithmetic operations on the processor. The aim is that the compression reduces the data movement costs enough to outweigh the costs of compressing and uncompressing. Anzt, Flegar, Grützmacher and Quintana-Ortí (2019*b*) propose this approach of decoupling the data storage format from the processing format, and they focus on storing the data at a lower precision than that at which the computations are performed. This approach is used in the papers mentioned at the end of Section 8.2 and for level 1 and level 2 BLAS by Grützmacher, Anzt and Quintana-Ortí (2021). Agullo *et al.* (2020) propose a similar approach for flexible GMRES, using as compression either reduced precision or the lossy floating-point SZ compressor (Di and Cappello 2016).

## 9. Mixed precision orthogonalization and QR factorization

There exist many algorithms to orthogonalize a set of vectors and to carry out the related task of computing the QR factorization of a matrix $A \in \mathbb{R}^{m\times n}$, where we assume $m \geq n$. Householder QR factorization is the most widely used and is unconditionally stable: it achieves a backward error and a loss of orthogonality of the computed $\widehat{Q}$ (if it is explicitly formed) both of order the unit roundoff $u$ (Higham 2002, Section 19.3). However, Householder QR factorization offers relatively little parallelism and requires expensive synchronizations. Alternative

https://doi.org/10.1017/S0962492922000022 Published online by Cambridge University Press

algorithms that are more suitable for parallel computers are unfortunately also less stable: for example, the classical and modified Gram–Schmidt algorithms (CGS and MGS) lead to a loss of orthogonality of order $\kappa(A)^2 u$ (Giraud, Langou, Rozložník and van den Eshof 2005) and $\kappa(A)u$, respectively. Developing orthogonalization algorithms that are both parallel and stable is an active field of research. In this section we discuss how mixed precision arithmetic can be used to stabilize or accelerate these algorithms. We refer to the recent survey by Carson, Lund, Rozložník and Thomas (2022*b*) for a description of what is known about the stability of block Gram–Schmidt algorithms.

Yang, Fox and Sanders (2021) perform rounding error analysis of Householder QR factorization under a model of mixed precision computation that assumes that inner products are computed in high precision $u_{\text{high}}$ and then rounded to lower precision $u_{\text{low}}$. This model is thus applicable to the use of block FMAs. They show that the bound for the backward error, which is of order $mnu$ in uniform precision $u$ (Higham 2002, p. 361), becomes of order $nu_{\text{low}} + mnu_{\text{high}}$ under this mixed precision model. Unlike the error bound $2u_{\text{low}} + nu_{\text{high}}$ of Blanchard *et al.* (2020*b*) for LU factorization with block FMAs (see Section 7.3), their bound for QR factorization still grows with $n$ at the $u_{\text{low}}$ level. This is because the model assumes the result of the inner products to be rounded to precision $u_{\text{low}}$ at each step of the factorization. Yang *et al.* (2021) also analyse a blocked version of Householder QR assuming that the rounding to precision $u_{\text{low}}$ takes place only once per block-column. They show that the term $nu_{\text{low}}$ can then be replaced by $Nu_{\text{low}}$, where $N$ is the number of block-columns. Taking advantage of the capability of some block FMAs (such as NVIDIA tensor cores) of keeping the result in high precision, one can also imagine a Householder QR factorization which starts with the original matrix $A$ in high precision and rounds its QR factors to low precision on the fly, similarly to the LU factorization algorithm proposed by Haidar *et al.* (2018*b*) and analysed by Blanchard *et al.* (2020*b*). We expect this algorithm to further reduce the constant in the error bound at the $u_{\text{low}}$ level by dropping the dependence on $N$, although this is not covered by the analysis of Yang *et al.* (2021).

Zhang, Baharlouei and Wu (2020) implement Householder QR factorization using NVIDIA tensor cores to accelerate the matrix–matrix products, but obtain only modest speedups due to the panel factorization being the performance bottleneck. For this reason, they propose to switch to a recursive QR factorization employing MGS for the orthogonalization, which requires more flops and is potentially less stable but makes a more intensive use of matrix–matrix operations. They also propose to use communication-avoiding QR (CAQR) for the panel factorizations. These two modifications allow them to efficiently leverage the tensor core performance, significantly accelerating the factorization. They demonstrate experimentally that their algorithm can solve least squares problems with double precision accuracy by using the computed QR factors to precondition the CGLS iterative method.

The Cholesky–QR algorithm computes the QR factorization of $A \in \mathbb{R}^{m\times n}$ by a three-step process.

https://doi.org/10.1017/S0962492922000022 Published online by Cambridge University Press

1 Compute the matrix product $B = A^T A$.

2 Compute the Cholesky factorization $R^T R = B$.

3 Compute $Q = AR^{-1}$ by a multiple right-hand side triangular solve.

For tall, thin matrices ($m \gg n$), most of the flops take place at steps 1 and 3, which are very parallel and make intensive use of BLAS-3 operations. However, Cholesky–QR in uniform precision $u$ leads to a loss of orthogonality of order $\kappa(A)^2 u$ (Stathopoulos and Wu 2002), and can fail if the Cholesky factorization at step 2 breaks down. Cholesky–QR can be partially stabilized by using mixed precision arithmetic: Yamazaki *et al.* (2015*a*) show that, if the first two steps above are carried out at precision $u_{\text{high}}$ and the third step is carried out at precision $u$, the loss of orthogonality can be bounded by (Yamazaki *et al.* 2015*a*, Theorem 3.2)

$$\|I - \widehat{Q}^T \widehat{Q}\| = O\big(\kappa(A)^2 (u_{\text{high}} + u^2) + \kappa(A)u\big). \tag{9.1}$$

Thus, by using doubled precision (that is, $u_{\text{high}} = u^2$) for the first two steps, the loss of orthogonality is $O(\kappa(A)^2 u^2 + \kappa(A)u)$ and is therefore of order $O(\kappa(A)u)$ as long as $\kappa(A) < u^{-1}$. In this context, mixed precision arithmetic can therefore be used not to accelerate the algorithm but to (partially) stabilize it, by reducing the loss of orthogonality by a factor of $\kappa(A)$. Yamazaki *et al.* (2015*a*) implement this mixed precision Cholesky–QR algorithm with fp64 as the working precision $u$, and employ double-double arithmetic for the first two steps. Despite requiring 8.5× more flops due to the use of software-emulated arithmetic, they show that the mixed precision Cholesky–QR algorithm can be only moderately slower than in uniform precision (about 1.4× slower in the best case) when the number of columns $n$ to orthogonalize is small, because in this case the performance of Cholesky–QR is memory-bound. They apply this mixed precision Cholesky–QR algorithm to the solution of linear systems with a communication-avoiding GMRES method, and show that the use of this more stable Cholesky–QR algorithm avoids the need for reorthogonalization and allows GMRES to converge faster, leading to significant speedups. See also Yamazaki *et al.* (2015*b*) for early results on this approach. When the number of columns to orthogonalize is larger, the performance of Cholesky–QR tends to become compute-bound and the overhead associated with the use of double-double arithmetic becomes more significant. To overcome this issue, Yamazaki *et al.* (2015*c*) propose a block MGS method that partitions the matrix into block-columns of smaller size and uses the mixed precision Cholesky–QR to orthogonalize each block. This method can be up to seven times faster than applying mixed precision Cholesky–QR to the entire matrix, and numerical experiments show that it can also be as stable, despite the lack of error analysis bounding the loss of orthogonality.

A drawback of Cholesky–QR-based methods is that they can fail if the Cholesky factorization breaks down (because it encounters a non-positive pivot). Breakdown can be avoided by shifting the matrix to ensure the success of the Cholesky factorization (Fukaya *et al.* 2020).

Another solution is the singular value QR (SVQR) factorization (Stathopoulos and Wu 2002), which takes the following steps.

1 Compute the matrix product $B = A^T A$.

2 Compute the singular value decomposition $U\Sigma U^T = B$.

3 Compute the QR factorization $\bar{Q}R = \Sigma^{1/2}U^T$.

4 Compute $Q = AR^{-1}$ by multiple right-hand side triangular solve.

Steps 2 and 3 require more flops than simply computing the Cholesky factorization of $B$, but if $m \gg n$ the overhead is negligible compared with the flops required by steps 1 and 4. The advantage of SVQR is that when $B$ is singular to the working precision, step 2 will directly identify the entire subspace of the nearly dependent columns and one can replace all the associated singular values with an appropriately large value. Stathopoulos and Wu (2002) suggest replacing singular values smaller than $u\sigma_1$ with $u\sigma_1$, where $\sigma_1$ is the largest singular value of $B$. In uniform precision $u$, SVQR also suffers from a loss of orthogonality proportional to $\kappa(A)^2 u$. Yamazaki, Tomov and Dongarra (2016) propose a mixed precision version of SVQR analogous to their mixed precision Cholesky–QR (Yamazaki *et al.* 2015*a*), where step 4 is carried out in halved precision $u^{1/2}$ compared with the first two steps. The loss of orthogonality can then be bounded by (Yamazaki *et al.* 2016, Theorem 5.1)

$$\|I - \widehat{Q}^T\widehat{Q}\| = O(\kappa(A)^2 u + \kappa(A)u^{1/2}). \tag{9.2}$$

When $\kappa(A)$ is larger than $u^{-1/2}$, the use of halved precision in step 4 therefore does not significantly impact the loss of orthogonality. For smaller values of $\kappa(A)$, the loss of orthogonality is increased but remains a factor $\kappa(A)$ smaller than if SVQR were carried out entirely in halved precision.

## 10. Least squares problems

Consider the linear least squares (LS) problem $\min_x \|Ax - b\|_2$, where $A \in \mathbb{R}^{m\times n}$ with $m \geq n$ has full rank. Recall that the unique LS solution is the solution of the normal equations

$$A^T Ax = A^T b \tag{10.1}$$

and that the normal equations can be rewritten as the $(m + n) \times (m + n)$ augmented system

$$\begin{bmatrix} I & A \\ A^T & 0 \end{bmatrix} \begin{bmatrix} r \\ x \end{bmatrix} = \begin{bmatrix} b \\ 0 \end{bmatrix}. \tag{10.2}$$

Björck (1967) proposed refining an approximate LS solution by applying iterative refinement to the augmented system, with residuals calculated at twice the working precision, and he showed how to efficiently solve the augmented system given a QR

https://doi.org/10.1017/S0962492922000022 Published online by Cambridge University Press

factorization of $A$. He also gave rounding error analysis for the method. Björck's method and analysis was extended to constrained and weighted LS problems by Gulliksson (1994).

Demmel, Hida, Riedy and Li (2009) discuss practical implementation details such as convergence tests and how to compute error bounds, and they exploit the XBLAS.

For more on traditional and fixed precision forms of iterative refinement for the LS problem, see Björck (1996) and Higham (2002, Chapter 20).

Recently, mixed precision algorithms for solving the LS problem have been developed by building on GMRES-IR for square linear systems.

Higham and Pranesh (2021) assume that $A$ is well-conditioned and make use of the normal equations (10.1). Their idea is a modification of the algorithm of Section 7.5 that uses GMRES-IR with Cholesky preconditioning. It chooses a diagonal matrix $S$ so that $B = AS$ has columns of unit 2-norm, forms $C = B^T B$ at precision $u_\ell$, computes the Cholesky factorization of a shifted $C$ at precision $u_\ell$, then applies GMRES-IR to the normal equations, computing the residual in precision $u_r$ as $r_i = A^T(b-Ax_i)$ and applying GMRES to the preconditioned update equation $MA^TAd_i = Mr_i$, where $M = SR^{-1}R^{-T}S$. Solving the normal equations is usually avoided by numerical analysts because it gives a backward error bound of order $\kappa_2(A)u$ (Higham 2002, Section 20.4) and the Cholesky factorization can break down for $\kappa_2(A) > u^{-1/2}$. Its use here is justified by the facts that $A$ is assumed to be well-conditioned, the Cholesky factorization of the cross-product matrix is being used as a preconditioner rather than to compute the solution directly, and if a block FMA is available it can be exploited in forming $C$, boosting the speed and accuracy.

Carson, Higham and Pranesh (2020) make use of the augmented system (10.2). Their method computes a QR factorization at precision $u_\ell$, then applies GMRES-IR to the augmented system with a left preconditioner constructed in one of two possible ways from the QR factors. Backward error analysis given in Carson *et al.* (2020), combined with the analysis of Carson and Higham (2017, 2018) and Amestoy *et al.* (2021*b*), shows that the method yields a forward error, and a backward error for the augmented system, of order the working precision under reasonable assumptions. Numerical experiments in Carson *et al.* (2020) with various combinations of the three precisions show that the method behaves as predicted by the theory.

## 11. Eigenvalue decomposition

A natural way to refine approximate solutions to the eigenvalue problem is by Newton's method, and it presents opportunities for exploiting different arithmetic precisions. Early references developing Newton's method for mixed precision iterative refinement for the standard eigenvalue problem are Dongarra (1980, 1982) and Dongarra, Moler and Wilkinson (1983).

https://doi.org/10.1017/S0962492922000022 Published online by Cambridge University Press

We consider the generalized eigenvalue problem $Ax = \lambda Bx$, where $A, B \in \mathbb{R}^{n\times n}$. Setting $B = I$ gives the standard eigenvalue problem. We suppose that we have an approximate eigenpair that we wish to improve. We will use Newton's method, so we need to put the problem in the form of a non-linear system.

Since an eigenvector remains an eigenvector when multiplied by a non-zero scalar, we need to normalize $x$, which we will do by requiring that $e_s^T x = 1$ for some chosen $s$, where $e_s$ is the unit vector with a 1 in position $s$. Define

$$F(v) = \begin{bmatrix} (A - \lambda B)x \\ e_s^T x - 1 \end{bmatrix} : \mathbb{C}^{n+1} \to \mathbb{C}^{n+1}, \quad v = \begin{bmatrix} x \\ \lambda \end{bmatrix}.$$

The Jacobian is

$$J(v) = \left(\frac{\partial F_i}{\partial v_j}\right) = \begin{bmatrix} A - \lambda B & -Bx \\ e_s^T & 0 \end{bmatrix}.$$

It is easy to see that $\|J(w) - J(v)\|_\infty \le 2\|B\|_\infty \|w - v\|_\infty$, so $J$ is Lipschitz continuous with constant $2\|B\|_\infty$. Moreover, it can be shown that $J$ is non-singular when $\lambda$ is a simple (non-multiple) eigenvalue (Tisseur 2001, Lemma 3.3).

By applying Theorems 5.1 and 5.2, Tisseur (2001, Section 3.2) shows that if $(x_0, \lambda_0)$ is a sufficiently good approximation to an eigenpair $(x_*, \lambda_*)$, $\lambda_*$ is simple, $J$ is not too ill-conditioned at $(x_*, \lambda_*)$, and the linear system solver is not too unstable, then Newton's method is well-defined and the limiting forward error is bounded by

$$\frac{\|(\widehat{x}^T, \lambda)^T - (x_*^T, \lambda_*)^T\|_\infty}{\|(x_*^T, \lambda_*)^T\|_\infty} \lesssim cnu_r \|J(v_*)^{-1}\|_\infty \max(\|A\|_\infty, \|B\|_\infty) + u,$$

where $c$ is a small integer constant and $u_r$ is the precision in which the residual $F(v)$ is evaluated. If $u_r = u^2$ this bound can be shown to reduce to $cnu$. Moreover, the limiting backward error is bounded by

$$\eta_\infty(\widehat{x}, \widehat{\lambda}) \lesssim cnu_r + u(3 + |\lambda|) \max\left(\frac{\|A\|_\infty}{\|B\|_\infty}, \frac{\|B\|_\infty}{\|A\|_\infty}\right). \qquad (11.1)$$

Note that as for linear systems, instability in the linear system solver does not affect the bounds for the limiting forward error and backward error.

Each Newton iteration involves the solution of a linear system with the Jacobian matrix evaluated at the current iterate. If this is done using LU factorization of $J(v)$ it costs $O(n^3)$ flops per step, which is expensive. If an approximate eigendecomposition is available then this cost can be reduced to $O(n^2)$ flops per iteration. We specialize to the symmetric definite generalized eigenvalue problem in which $A$ is symmetric and $B$ is symmetric positive definite. Algorithm 11.1 is given by Tisseur (2001, Algorithm 4.2) and is used by Davies, Higham and Tisseur (2001) to refine solutions from the Cholesky–Jacobi method, which uses a Cholesky decomposition of $B$ to reduce the problem to a standard symmetric eigenvalue problem and then applies the Jacobi method.

https://doi.org/10.1017/S0962492922000022 Published online by Cambridge University Press

**Algorithm 11.1.** Given a symmetric $A \in \mathbb{R}^{n\times n}$, a symmetric positive definite $B \in \mathbb{R}^{n\times n}$, $X \in \mathbb{R}^{n\times n}$ and a diagonal $\Lambda \in \mathbb{R}^{n\times n}$ such that $X^T AX \approx \Lambda$ and $X^T BX \approx I$, and an approximate eigenpair $(x, \lambda)$ with $\|x\|_\infty = x_s = 1$, this algorithm applies iterative refinement to $\lambda$ and $x$ at a cost of $O(n^2)$ flops per iteration. Computations are at precision $u$ unless otherwise stated.

```
repeat until converged
    Compute $r = \lambda Bx - Ax$ in precision $u_r$.
    $D_\lambda = \Lambda - \lambda I$
    $d = -Bx - c_s$, where $c_s$ is the $s$th column of $A - \lambda B$.
    $v = X^T d$, $f = X^T e_s$
    Compute Givens rotations $J_k$ in the $(k, k+1)$ plane, such that
        $Q_1^T v : = J_1^T \dots J_{n-1}^T v = \|v\|_2 e_1$.
    Compute orthogonal $Q_2$ such that
        $T = Q_2^T Q_1^T (D_\lambda + v f^T)$ is upper triangular.
    $z = Q_2^T Q_1^T X^T r$
    Solve $Tw = z$ for $w$.
    $\delta = Xw$
    $\lambda = \lambda + \delta_s$, $\delta_s = 0$
    $x = x + \delta$
end
```

Newton's method is well suited to refining a small number of eigenpairs but not a complete eigensystem, as in the latter case it is expensive and may not converge for all eigenpairs.

Tsai, Luszczek and Dongarra (2021) revisit the Newton method for the standard symmetric eigenvalue problem and develop a mixed precision algorithm that transforms the matrix to tridiagonal form in single precision, computes the eigensystem by divide and conquer in double precision, then refines the eigenpairs in double precision.

Ogita and Aishima (2018) develop an iteration for refining the whole eigensystem of a symmetric matrix. It requires four matrix multiplications per iteration, all executed in a higher precision than the working precision. Quadratic convergence is proved for sufficiently good initial approximations. The algorithm does not work well when there are nearly multiple eigenvalues. The latter limitation is addressed in Ogita and Aishima (2019) by using further steps that work with clusters of eigenvalues.

Petschow, Quintana-Ortí and Bientinesi (2014) use extra precision to improve the accuracy of the multiple relatively robust representations (MRRR) method for the symmetric tridiagonal eigenvalue problem without sacrificing performance.

Ralha (2018) considers carrying out the bisection method for symmetric tridiagonal matrices with early iterations in single precision before switching to the

https://doi.org/10.1017/S0962492922000022 Published online by Cambridge University Press

working precision of double, and develops criteria for deciding when to make the switch.

Stor, Slapničar and Barlow (2015) give an algorithm for the eigendecomposition of symmetric arrowhead matrices that employs bisection and a shift and invert technique, and in the latter it uses arithmetic at twice the working precision for one element of the inverse in order to ensure forward stability.

Tsuchida and Choe (2012) consider a trace minimization method for computing the complete eigensystem of a symmetric matrix and explore running different parts of the method at half the working precision. Gains of over 30% in execution time are reported with little loss of accuracy.

Alvermann *et al.* (2019) report on two projects that are developing eigensolvers based on the (block) Jacobi–Davidson method, subspace iteration and other methods, and are using lower precision in early iterations for speed and higher precision within the orthogonalizations for robustness.

## 12. Singular value decomposition

We now consider the singular value decomposition (SVD) of $A \in \mathbb{R}^{m\times n}$ with $m \geq n$: $A = U\Sigma V^T$ with $U \in \mathbb{R}^{m\times m}$ and $V \in \mathbb{R}^{n\times n}$ orthogonal and $\Sigma = \operatorname{diag}(\sigma_i) \in \mathbb{R}^{m\times n}$.

### *12.1. Iterative refinement*

The Newton approach to refining eigenpairs can be extended to singular value triples of $A \in \mathbb{R}^{m\times n}$ by using the function

$$F(x) = \begin{bmatrix} Av - \mu_1 u \\ A^T u - \mu_2 v \\ u^T u - 1 \\ v^T v - 1 \end{bmatrix}, \quad x = \begin{bmatrix} u \\ v \\ \mu_1 \\ \mu_2 \end{bmatrix}.$$

The Jacobian of $f$ is

$$J(x) = \begin{bmatrix} -\mu_1 I & A & -u & 0 \\ A^T & -\mu_2 I & 0 & -v \\ 2u^T & 0 & 0 & 0 \\ 0 & 2v^T & 0 & 0 \end{bmatrix}.$$

The approximate singular value is updated by $(\mu_1 + \mu_2)/2$. Dongarra (1983), extending the work in Dongarra *et al.* (1983), shows how to solve systems with $J(x)$ in $O(mn)$ flops, given an SVD or bidiagonal factorization of $A$. Again, the Newton theory of Section 5 applies.

Ogita and Aishima (2020) extend their algorithm for the symmetric eigenvalue problem, mentioned in the previous section, to the SVD in order to refine the complete SVD; the algorithm uses higher precision and is dominated by matrix multiplication.

https://doi.org/10.1017/S0962492922000022 Published online by Cambridge University Press

### *12.2. SVD with rapidly decaying singular values*

Another opportunity for mixed precision arithmetic arises in the case of matrices with rapidly decaying singular values. Given a target accuracy $\varepsilon$, it is well known that singular values smaller than $\varepsilon$ and the corresponding singular vectors can be dropped to provide a low-rank approximation to the matrix with an error bound of order $\varepsilon$. Amestoy *et al.* (2021*a*) explain that among the singular values that remain, those that are small enough can be represented, along with their associated singular vectors, in lower precision. For example, singular vectors associated with singular values less than $\varepsilon/u_s$, where $u_s = 2^{-24}$ is the unit roundoff for single precision, can be stored in single precision, even when $\varepsilon \ll u_s$. They introduce a mixed precision SVD representation that uses $p$ precisions,

$$A = U\Sigma V^T = [U_1\ U_2 \dots U_p]\Sigma[V_1\ V_2 \dots V_p]^T, \tag{12.1}$$

where $U_i$ and $V_i$ are stored in precision $u_i$, with $u_1 < u_2 < \dots < u_p$. They give an explicit rule on how to partition $U$ and $V$ in order to guarantee an overall accuracy of order $\varepsilon$ (Amestoy *et al.* 2021*a*, Theorem 2.2). Note that this approach is applicable not only to the SVD but also to other types of rank-revealing decompositions, such as QR factorization with column pivoting.

Ooi *et al.* (2020) propose three different methods to introduce mixed precision arithmetic in the product of a low-rank matrix with a vector. Their method 3 is similar to the representation (12.1), which they use with fp64 and fp32 arithmetics. They apply this approach to the solution of linear systems exploiting products of a hierarchical ($\mathcal{H}$) matrix with a vector, using the iterative BiCGstab solver.

## 13. Multiword arithmetic

Multiword arithmetic is a well-known approach to enhance the accuracy of computations while employing fast arithmetic supported in hardware. It consists of representing high precision numbers by the unevaluated sum of lower precision numbers. An example is double-double arithmetic which, as mentioned in Section 2.2, approximates an fp128 number as the sum of two fp64 numbers and replaces fp128 operations with fp64 operations.

The emergence of specialized hardware supporting low precision matrix multiplication with high precision accumulators, such as the NVIDIA GPU tensor cores, provides new opportunities for multiword arithmetic. Indeed, these units are much faster than standard fp32 arithmetic (up to 8 and 16 times faster on the Volta and Ampere GPUs, for example). Therefore an approach to accelerate the computation of an fp32 matrix product $C = AB$ is to approximate $A \approx A_1 + A_2$ as the sum of two fp16 matrices, and similarly $B \approx B_1 + B_2$. Then $C$ can be computed as $C \approx A_1B_1 + A_1B_2 + A_2B_1 + A_2B_2$ using block FMAs to compute each of the $A_iB_j$ terms using internal tensor core arithmetic at fp32 accuracy. Since there are only

https://doi.org/10.1017/S0962492922000022 Published online by Cambridge University Press

four terms (and in fact, we can reduce that number to three, as explained below), this approach can potentially be much faster than standard fp32 arithmetic.

This approach was first used with NVIDIA tensor cores by Markidis *et al.* (2018) to accelerate matrix products, and by Sorna *et al.* (2018) to accelerate the fast Fourier transform (FFT). Pisha and Ligowski (2021) similarly use the TensorFloat32 format in computing the FFT on the NVIDIA A100 GPUs. Henry, Tang and Heinecke (2019) describe an approach based on block FMA hardware using the bfloat16 format instead of the fp16 one, where $A$ and $B$ are split into three bfloat16 matrices, which requires nine products to compute $C = AB$. Finally, Mukunoki, Ozaki, Ogita and Imamura (2020) explain how to achieve not only fp32 accuracy but also fp64 accuracy with this approach, by using the Ozaki scheme. Their approach, however, requires splitting both $A$ and $B$ a large number of times, which leads to several dozens if not hundreds of products. Their algorithm is therefore only beneficial on GPUs on which fp64 arithmetic is very slow, such as some of the Turing models.

Fasi *et al.* (2022) generalize these approaches by considering any low precision $u_{\text{low}}$ and any number of splits $p$. They give Algorithm 13.1.

**Algorithm 13.1 (multiword matrix multiplication).** This algorithm computes the matrix–matrix product $C = AB$ using $p$-word arithmetic with a mixed precision block FMA with precisions $u_{\text{low}}$ and $u_{\text{high}}$.

```
for $i = 1\colon p$
    $A_i = \mathrm{fl}_{\text{low}}(A - \sum_{k=1}^{i-1} A_k)$
    $B_i = \mathrm{fl}_{\text{low}}(B - \sum_{k=1}^{i-1} B_k)$
end
for $i = 1\colon p$
    for $j = 1\colon p$
        Compute $C_{ij} = A_i B_j$ with Algorithm 4.1.
        $C \leftarrow C + C_{ij}$
    end
end
```

The algorithm recursively computes $A_i$ (and similarly $B_j$) as the residual from the $(i-1)$-way split $A \approx A_1 + \cdots + A_{i-1}$ and rounds it to precision $u_{\text{low}}$, that is,

$$\left.\begin{aligned} A_i &= \mathrm{fl}_{\text{low}}\left(A - \sum_{k=1}^{i-1} A_k\right) \\ B_i &= \mathrm{fl}_{\text{low}}\left(B - \sum_{k=1}^{i-1} B_k\right) \end{aligned}\right\} \quad i = 1\colon p. \tag{13.1}$$

This gives the approximations

$$A = \sum_{i=1}^{p} A_i + \varDelta A, \quad |\varDelta A| \le u_{\mathrm{low}}^{p} |A|, \tag{13.2}$$

$$B = \sum_{i=1}^{p} A_i + \varDelta B, \quad |\varDelta B| \le u_{\mathrm{low}}^{p} |B|. \tag{13.3}$$

Then, if $C$ is approximated by the sum of the $p^2$ products $A_i B_j$, which are computed by chaining calls to a block FMA with internal precision $u_{\mathrm{high}}$, by Theorem 4.1 we obtain a computed $\widehat{C}$ satisfying (Fasi *et al.* 2022)

$$\widehat{C} = AB + E, \quad |E| \lesssim \left(2u_{\mathrm{low}}^{p} + u_{\mathrm{low}}^{2p} + (n + p^2 - 1)u_{\mathrm{high}}\right)|A||B|. \tag{13.4}$$

Clearly, for practical choices of $u_{\mathrm{low}}$ and $u_{\mathrm{high}}$ a small value of $p$ is sufficient. For example, for fp16 ($u_{\mathrm{low}} = 2^{-11}$) and fp32 ($u_{\mathrm{high}} = 2^{-24}$), $p = 2$ is enough since in this case $u_{\mathrm{low}}^2 = 4u_{\mathrm{high}}$. Taking larger values of $p$ will not significantly improve the bound (13.4) since the term $(n + p^2)u_{\mathrm{high}}$ will then dominate. For bfloat16 ($u_{\mathrm{low}} = 2^{-8}$) and fp32, the case $p = 3$ is also of interest because the significand of one fp32 number fits exactly into the significands of three bfloat16 numbers.

Importantly, in practice not all $p^2$ products $A_i B_j$ need be computed. As a result of the construction (13.1), the magnitude of the elements of $A_i$ and $B_j$ rapidly decreases as we increase $i$ and $j$. More precisely, we have

$$|A_i| \le u_{\mathrm{low}}^{i-1}(1 + u_{\mathrm{low}})|A|, \quad |B_i| \le u_{\mathrm{low}}^{i-1}(1 + u_{\mathrm{low}})|B|, \quad i = 1\colon p,$$

and thus

$$|A_i||B_j| \le u_{\mathrm{low}}^{i+j-2}(1 + u_{\mathrm{low}})^2|A||B|. \tag{13.5}$$

Therefore, ignoring any product $A_i B_j$ such that $i + j > p + 1$ only introduces an error of order $u_{\mathrm{low}}^{p}$ or higher, which does not significantly impact the bound (13.4). Indeed, by only computing the products $A_i B_j$ such that $i + j \le p + 1$, we obtain the modified bound

$$\widehat{C} = AB + E,$$

$$|E| \lesssim \left(2u_{\mathrm{low}}^{p} + u_{\mathrm{low}}^{2p} + (n + p^2)u_{\mathrm{high}} + \sum_{i=1}^{p-1}(p - i)u_{\mathrm{low}}^{p+i-1}(1 + u_{\mathrm{low}})^2\right)|A||B|.$$

The constant in this bound is $(p + 1)u_{\mathrm{low}}^{p}$ plus higher-order terms, so to order $u_{\mathrm{low}}^{p}$ we have only increased the constant 2 from (13.4) to $p+1$, and we have reduced the number of products from $p^2$ to $p(p+1)/2$. Concretely, with fp32 and fp16 ($p = 2$), we only need three products, which is less than the four used by Markidis *et al.* (2018), and with bfloat16 and fp32 ($p = 3$), we can reduce the number of products from nine to six, as already suggested by Henry *et al.* (2019).

Note that further reducing the number of products (such as using two products for $p = 2$, as attempted by Markidis *et al.* 2018) is possible, but the analysis tells us

https://doi.org/10.1017/S0962492922000022 Published online by Cambridge University Press

it should not be beneficial. Indeed, ignoring any product $A_i B_j$ such that $i + j \le p + 1$ would introduce an error of order at least $u_{\text{low}}^{p-1}$, and so could not be significantly more accurate than simply using $p - 1$ splits rather than $p$.

The above analysis encompasses previously proposed algorithms, and also includes new cases. For example, we may use a 2-way split ($p = 2$) with bfloat16 and fp32, which requires three products rather than six (when $p = 3$) and delivers an accuracy of order $2^{-16}$ rather than $2^{-24}$.

Note that this analysis deals with worst-case error bounds and so does not guarantee that multiword arithmetic with low precision block FMAs will be as accurate as higher precision standard arithmetic in the case where the latter does not attain its worst-case error. In fact, in their experiments with NVIDIA tensor cores, Fasi *et al.* (2022) find that double-fp16 arithmetic can be much less accurate than fp32 arithmetic due to the rounding mode of these devices, which can make the worst-case bounds for double-fp16 sharp. To overcome this issue, Fasi *et al.* (2022) propose the use of FABsum (see Section 4.2) to reduce the worst-case error bound.

## 14. Adaptive precision algorithms

Several mixed precision algorithms described in the previous sections share the same foundation: adapt the precision to the data by using lower precisions to represent the less important or significant parts of the data. As an example, consider the computation of the sum $a + b$, where $|b| \ll |a|$. Because of the widely different magnitudes of $a$ and $b$, the least significant bits of $b$ do not play a significant role in the computed value of the result. Indeed if we round $b$ to $\widetilde{b} = \text{fl}_{\text{low}}(b) = b(1 + \delta_{\text{low}})$, where $|\delta_{\text{low}}| \le u_{\text{low}}$, then

$$\begin{aligned} \text{fl}(a + \widetilde{b}) &= (a + \widetilde{b})(1 + \delta) \quad (|\delta| \le u) \\ &= (a + b(1 + \delta_{\text{low}}))(1 + \delta) \\ &= (a + b)(1 + \delta)\left(1 + \frac{b}{a + b}\delta_{\text{low}}\right), \end{aligned}$$

and so we have an extra term $1 + b\delta_{\text{low}}/(a + b)$, which is insignificant as long as $|b|u_{\text{low}} \ll |a + b|u$. Therefore $b$ can be stored in lower precision without significantly impacting the accuracy of the computation. Moreover, if $b$ is the result of a previous computation, that computation can also be carried out in lower precision. This example illustrates that computations performed on data of small magnitude need not use very high precision. This is a simple but fundamental observation that has given birth to several adaptive precision algorithms. The object of this section is to show that these algorithms share strong connections.

Adaptive precision algorithms seek to exploit this observation by adapting the precision to be inversely proportional to the weight of the data, where the weight is defined by some metric such as the maximum magnitude or the norm of the data.

https://doi.org/10.1017/S0962492922000022 Published online by Cambridge University Press

In numerical linear algebra algorithms, this can be done at different levels of the computation: at the element, block, column/row or matrix levels.

### *14.1. At the matrix level*

In computations involving several matrices, we may choose to compute and store some of them in lower precision. For example, in computing $C = A_1B_1 + A_2B_2$ where $|A_1| \geq |A_2|$ and $|B_1| \geq |B_2|$, if $|A_2||B_2| \ll |A_1||B_1|$ then the matrix product $A_2B_2$ can be computed in lower precision than $A_1B_1$. An example where this situation arises is the use of multiword arithmetic, as illustrated by (13.5). In fact we have already explained that the products $A_iB_j$ of highest order can be ignored; data-driven analysis also shows that most of the products that cannot be ignored can however be computed in lower precision. For example, with $u_{\text{low}}$ as fp16 and $p = 2$, the products $A_1B_2$ and $A_2B_1$ can be computed in fp16 arithmetic, because the magnitude of their entries is proportional to $u_{\text{low}}$. Only the first term $A_1B_1$ actually needs to be computed in fp32 arithmetic. This observation is especially important when implementing multiword arithmetic on GPU tensor cores, which lead to heavy rounding error accumulation in the products $A_iB_j$ because of their rounding mode: Fasi *et al.* (2022) explain that it is only necessary to take care of reducing the effect of error accumulation on the $A_1B_1$ term.

### *14.2. At the column level (or, equivalently, at the row level)*

Given a matrix, we may think of storing each of its columns (or rows) in a different precision. This approach makes the most sense when dealing with matrices that can be decomposed as low-rank components of rapidly decreasing norm. This can be the case, for example, of SVDs or rank-revealing factorizations. In fact, the mixed precision truncated SVD approaches described in Section 12.2 (Amestoy *et al.* 2021*a*, Ooi *et al.* 2020) are precisely based on this property: rounding errors introduced by converting singular vectors to lower precision are demagnified by the associated singular value, and so the precision of each vector should be selected based on its associated singular value.

Note that, given the SVD $U\Sigma V^T$ of a matrix $A$, we can express the matrix as $A = \sum_i A_i$ with $A_i = u_i\sigma_i v_i^T$, where $\|A_{i+1}\|_F \leq \|A_i\|_F$. Thus, even though the matrices $A_i$ are never formed or manipulated explicitly, the link with the matrix-level case is clear.

### *14.3. At the block level*

In some applications it pays to partition a matrix into several blocks and adapt the precision to each block. For example, in Section 7.7 we described approaches where the precision of each block is based on its distance to the diagonal (Abdulah *et al.* 2019, 2022, Doucet *et al.* 2019). The success of these approaches is explained by the fact that, for many data-sparse matrices, blocks distant from the diagonal tend to have smaller norm. Indeed, storing each block in a precision inversely proportional to its norm can allow for significant gains with potentially little accuracy loss. As an example, consider a matrix $A \in \mathbb{R}^{pb \times pb}$ partitioned into $p^2$ blocks $A_{ij} \in \mathbb{R}^{b \times b}$ and assume we have two precisions $u_{\text{high}}$ and $u_{\text{low}}$ at our disposal. Then the matrix $\widehat{A}$ obtained by storing blocks $A_{ij}$ of Frobenius norm less than $u_{\text{high}}\|A\|_F/(pu_{\text{low}})$ in precision $u_{\text{low}}$ satisfies $\|\widehat{A} - A\|_F \le u_{\text{high}}\|A\|_F$. Thus we can store selected blocks of $A$ in precision $u_{\text{low}}$ and still recover a global approximation at accuracy $u_{\text{high}}$. This example trivially extends to more than two precisions.

Another example of an adaptive precision algorithm at the block level is the adaptive precision block Jacobi preconditioner discussed in Section 8.2 (Anzt *et al.* 2019*a*, Flegar *et al.* 2021). In this case the precisions of the blocks are selected based on their condition number rather than their norm, because this is the relevant metric when applying the inverse of the blocks as part of the preconditioner.

### *14.4. At the element level*

The adaptive precision algorithms described above seek to exploit the underlying structure of the data. However, the question arises as to whether it can be beneficial to adapt the precision at the element level: that is, to allow each variable in the computation to have its own precision, without any special structure (by blocks or by columns, for instance). This is similar in goal to transprecision computing and precision auto-tuning tools, which we briefly discuss in Section 15.1.

While this approach maximizes the use of reduced precision, it also destroys the granularity of the computation and should therefore only be used for memory-bound applications, such as for sparse matrix–vector products (SpMV) $y = Ax$. In particular, Ahmad, Sundar and Hall (2019) propose to split $A$ as $A_d + A_s$, where $A_s$ contains the small non-zero elements of $A$ and is stored in single precision, whereas $A_d$ is kept in double precision.

More generally, given $p$ precisions, one could split the elements of $A$ into $p$ different matrices and compute $p$ independent products in the corresponding precision. This idea is then similar to bucket summation (Demmel and Hida 2004, Zhu and Hayes 2009), in which summands are split into buckets based on their exponent. The novelty comes from summing each bucket in a different precision. Diffenderfer, Osei-Kuffuor and Menon (2021) propose such a bucket algorithm for the inner product that uses the four IEEE arithmetics as well as ‘perforation’, that is, the option to ignore some of the smallest summands. Graillat, Jézéquel, Mary and Molina (2022) propose an adaptive precision sparse matrix–vector product algorithm of similar spirit that, given $p$ precisions $u_1 < \cdots < u_p$, splits $A$ into $p$ buckets based on the magnitude of the elements: bucket number $i$ contains all the elements whose absolute value lies in the interval $[\varepsilon/u_i, \varepsilon/u_{i+1}]$, for a given target accuracy $\varepsilon$. They obtain speedups of up to an order of magnitude compared with a standard product in uniform precision.

https://doi.org/10.1017/S0962492922000022 Published online by Cambridge University Press

## 15. Miscellany

### *15.1. Tuning precisions*

A very different approach to mixed precision computing is to focus on the code rather than the algorithm. Given a code to solve a particular problem and a set of arithmetics of different precisions, one can ask what is the best selection of precision in which to store each variable. Here, 'best' means a choice that minimizes some suitable performance metric subject to achieving a computed result of acceptable quality. The motivation is the assumption that reducing precisions means faster execution and lower storage and energy requirements, though conversion between different precisions is required and adds overhead costs.

This problem is clearly combinatorial in nature, as if there are $n$ variables and $p$ precisions there are $p^n$ possible implementations. Ensuring results of acceptable quality requires, in principle, a parametrized rounding error analysis that encapsulates all the possible input data.

Much research has been done on algorithms that attempt to solve this problem. Usually, optimization of code is done for a 'representative' data set, with the assumption that the code will be used on related data for which the quality of the results will be similar. At best a local minimum of the objective function can be expected. No guarantee is provided that the code with the chosen precisions will satisfy error requirements across all possible input data.

Tools can be categorized as using static analysis (carried out before the code is run) or dynamic analysis. Dynamic analysis tools typically instrument the compiled code in order to try different combinations of precisions, and a popular way to do so is via the LLVM compiler infrastructure.[31]

Any attempt to reduce precisions of variables must ensure that sufficient range is maintained to avoid overflow and harmful underflow, which is particularly important if fp16 is one of the formats, given its narrow range.

An example of such work is the tool Precimonious, by Rubio-González *et al.* (2013), which uses execution time as the performance metric. It takes a C program as input and outputs a description of the precisions to be assigned to the variables. The experiments in Rubio-González *et al.* (2013) demonstrate a speedup of up to a factor of 1.4 by replacing certain double precision variables with single precision ones in the benchmark codes tested.

A more recent example, focused on GPUs, is GRAM, which chooses the precisions at run time (Ho, De Silva and Wong 2021). Each block of threads is kept at the same precision and a proportion $\alpha$ of the blocks is assigned a lower precision, with a binary search used to select $\alpha$. Speedups on an NVIDIA GPU of up to 1.8 over single precision are reported, by exploiting half precision arithmetic. GRAM does not support the use of tensor cores.

[31] https://llvm.org/

https://doi.org/10.1017/S0962492922000022 Published online by Cambridge University Press

Brun *et al.* (2021) develop a tool that uses a heuristic search strategy to select the precisions at which elementary functions are evaluated in a code, aiming to minimize the precisions subject to achieving output of a given accuracy. They use the Intel Vector Mathematics functions (VM) in the Intel oneAPI Math Kernel Library, which have an input argument that allows the user to select 'high accuracy', 'low accuracy' or 'enhanced performance accuracy' modes for the functions. They are able to obtain up to an approximate halving of the execution time on a Monte Carlo code that spends 70% of its time in mathematical library functions.

Precision tuning has also been used in climate and weather models. Tintó Prims *et al.* (2019) use the rpe Fortran library that emulates reduced precision, which we mentioned in Section 2.6. For two widely used ocean model codes they use a divide and conquer approach to find assignments of precisions to variables, finding that half precision or single precision can be used for large portions of the codes. Similar findings were made by Düben, Subramanian, Dawson and Palmer (2017) for a cloud-resolving model within a general circulation model.

It needs to be kept in mind that simply lowering the precisions of variables in a code may not be all that can be done. In some problems the choice of algorithm, or the algorithm itself, is precision-dependent. For example, an algorithm for computing an elementary function may be built upon a rational approximation that depends on the target accuracy, so that different approximations can be used for half, single and double precision.

### *15.2. Multiprecision algorithms*

Multiprecision algorithms for the matrix logarithm and the matrix exponential are developed by Fasi and Higham (2018, 2019). These algorithms take as input the unit roundoff $u$ of the arithmetic and then determine a suitable level of (inverse) scaling and squaring transformations and degree of Taylor or Padé approximants such that the functions are approximated to precision $u$. The key algorithmic parameters are determined at run time, which contrasts with the state-of-the-art algorithms for double precision arithmetic, where some of the parameters have been determined in advance. A similar strategy is followed by Al-Mohy, Higham and Liu (2022) in a multiprecision algorithm for computing for the matrix cosine and its Fréchet derivative.

Higham and Liu (2021) develop a multiprecision version of the Schur–Parlett algorithm for computing general analytic functions at a matrix argument. It avoids the need for derivatives by computing the function of the diagonal blocks of the reordered and blocked Schur form by diagonalizing, at a suitable precision, a small random perturbation of each block.

https://doi.org/10.1017/S0962492922000022 Published online by Cambridge University Press

## Acknowledgements

The work of the first author was supported by Engineering and Physical Sciences Research Council grant EP/P020720/1, the Royal Society and the Exascale Computing Project (17-SC-20-SC), a collaborative effort of the US Department of Energy Office of Science and the National Nuclear Security Administration. The work of the second author was supported by the InterFLOP project (ANR-20-CE46-0009) of the French National Agency for Research.

We thank Massimiliano Fasi, Sven Hammarling, Claude-Pierre Jeannerod, Mantas Mikaitis and Françoise Tisseur for their comments on a draft manuscript.

## References

A. Abdelfattah, H. Anzt, E. G. Boman, E. Carson, T. Cojean, J. Dongarra, A. Fox, M. Gates, N. J. Higham, X. S. Li, J. Loe, P. Luszczek, S. Pranesh, S. Rajamanickam, T. Ribizel, B. F. Smith, K. Swirydowicz, S. Thomas, S. Tomov, Y. M. Tsai and U. M. Yang (2021*a*), A survey of numerical linear algebra methods utilizing mixed-precision arithmetic, *Int. J. High Perform. Comput. Appl.* **35**, 344–369.

A. Abdelfattah, T. Costa, J. Dongarra, M. Gates, A. Haidar, S. Hammarling, N. J. Higham, J. Kurzak, P. Luszczek, S. Tomov and M. Zounon (2021*b*), A set of Batched Basic Linear Algebra Subprograms and LAPACK routines, *ACM Trans. Math. Software* **47**, 21.

A. Abdelfattah, S. Tomov and J. Dongarra (2019*a*), Fast batched matrix multiplication for small sizes using half-precision arithmetic on GPUs, in *2019 IEEE International Parallel and Distributed Processing Symposium (IPDPS)*, IEEE, pp. 111–122.

A. Abdelfattah, S. Tomov and J. Dongarra (2019*b*), Towards half-precision computation for complex matrices: A case study for mixed-precision solvers on GPUs, in *2019 IEEE/ACM 10th Workshop on Latest Advances in Scalable Algorithms for Large-Scale Systems (ScalA)*, IEEE, pp. 17–24.

A. Abdelfattah, S. Tomov and J. Dongarra (2020), Investigating the benefit of FP16-enabled mixed-precision solvers for symmetric positive definite matrices using GPUs, in *Computational Science – ICCS 2020* (V. V. Krzhizhanovskaya *et al.*, eds), Vol. 12138 of Lecture Notes in Computer Science, Springer, pp. 237–250.

S. Abdulah, Q. Cao, Y. Pei, G. Bosilca, J. Dongarra, M. G. Genton, D. E. Keyes, H. Ltaief and Y. Sun (2022), Accelerating geostatistical modeling and prediction with mixed-precision computations: A high-productivity approach with PaRSEC, *IEEE Trans. Parallel Distrib. Syst.* **33**, 964–976.

S. Abdulah, H. Ltaief, Y. Sun, M. G. Genton and D. E. Keyes (2019), Geostatistical modeling and prediction using mixed precision tile Cholesky factorization, in *2019 IEEE 26th International Conference on High Performance Computing, Data, and Analytics (HiPC)*, IEEE, pp. 152–162.

E. Agullo, F. Cappello, S. Di, L. Giraud, X. Liang and N. Schenkels (2020), Exploring variable accuracy storage through lossy compression techniques in numerical linear algebra: A first application to flexible GMRES. Research report RR-9342, Inria Bordeaux Sud-Ouest. Available at hal-02572910v2.

K. Ahmad, H. Sundar and M. Hall (2019), Data-driven mixed precision sparse matrix vector multiplication for GPUs, *ACM Trans. Archit. Code Optim.* **16**, 51.

https://doi.org/10.1017/S0962492922000022 Published online by Cambridge University Press

A. H. Al-Mohy, N. J. Higham and X. Liu (2022), Arbitrary precision algorithms for computing the matrix cosine and its Fréchet derivative, *SIAM J. Matrix Anal. Appl.* **43**, 233–256.

J. I. Aliaga, H. Anzt, T. Grützmacher, E. S. Quintana-Ortí and A. E. Tomás (2020), Compressed basis GMRES on high performance GPUs. Available at arXiv:2009.12101.

A. Alvermann, A. Basermann, H.-J. Bungartz, C. Carbogno, D. Ernst, H. Fehske, Y. Futamura, M. Galgon, G. Hager, S. Huber, T. Huckle, A. Ida, A. Imakura, M. Kawai, S. Köcher, M. Kreutzer, P. Kus, B. Lang, H. Lederer, V. Manin, A. Marek, K. Nakajima, L. Nemec, K. Reuter, M. Rippl, M. Röhrig-Zöllner, T. Sakurai, M. Scheffler, C. Scheurer, F. Shahzad, D. Simoes Brambila, J. Thies and G. Wellein (2019), Benefits from using mixed precision computations in the ELPA-AEO and ESSEX-II eigensolver projects, *Japan J. Indust. Appl. Math.* **36**, 699–717.

P. Amestoy, O. Boiteau, A. Buttari, M. Gerest, F. Jézéquel, J.-Y. L'Excellent and T. Mary (2021*a*), Mixed precision low rank approximations and their application to block low rank LU factorization. Available at hal-03251738.

P. Amestoy, A. Buttari, N. J. Higham, J.-Y. L'Excellent, T. Mary and B. Vieublé (2021*b*), Five-precision GMRES-based iterative refinement. MIMS EPrint 2021.5, Manchester Institute for Mathematical Sciences, The University of Manchester, UK.

P. Amestoy, A. Buttari, N. J. Higham, J.-Y. L'Excellent, T. Mary and B. Vieublé (2022), Combining sparse approximate factorizations with mixed precision iterative refinement. MIMS EPrint 2022.2, Manchester Institute for Mathematical Sciences, The University of Manchester, UK.

P. Amestoy, A. Buttari, J.-Y. L'Excellent and T. Mary (2019), Performance and scalability of the block low-rank multifrontal factorization on multicore architectures, *ACM Trans. Math. Software* **45**, 2.

P. Amestoy, I. S. Duff, J.-Y. L'Excellent and J. Koster (2001), A fully asynchronous multifrontal solver using distributed dynamic scheduling, *SIAM J. Matrix Anal. Appl.* **23**, 15–41.

E. Anderson (1991), Robust triangular solves for use in condition estimation. Technical report CS-91-142, Department of Computer Science, The University of Tennessee, Knoxville, TN, USA. LAPACK Working Note 36.

ANSI (1966), *American National Standard FORTRAN*, American National Standards Institute, New York.

H. Anzt, J. Dongarra and E. S. Quintana-Ortí (2015), Adaptive precision solvers for sparse linear systems, in *Proceedings of the 3rd International Workshop on Energy Efficient Supercomputing (E2SC '15)*, ACM Press, article 2.

H. Anzt, J. Dongarra, G. Flegar, N. J. Higham and E. S. Quintana-Ortí (2019*a*), Adaptive precision in block-Jacobi preconditioning for iterative sparse linear system solvers, *Concurrency Comput. Pract. Exper.* **31**, e4460.

H. Anzt, G. Flegar, T. Grützmacher and E. S. Quintana-Ortí (2019*b*), Toward a modular precision ecosystem for high-performance computing, *Int. J. High Perform. Comput. Appl.* **33**, 1069–1078.

J. Appleyard and S. Yokim (2017), Programming tensor cores in CUDA 9. Available at https://devblogs.nvidia.com/programming-tensor-cores-cuda-9/.

M. Arioli and I. S. Duff (2009), Using FGMRES to obtain backward stability in mixed precision, *Electron. Trans. Numer. Anal.* **33**, 31–44.

https://doi.org/10.1017/S0962492922000022 Published online by Cambridge University Press

M. Arioli, I. S. Duff, S. Gratton and S. Pralet (2007), A note on GMRES preconditioned by a perturbed $LDL^T$ decomposition with static pivoting, *SIAM J. Sci. Comput.* **29**, 2024–2044.

ARM (2018), *ARM Architecture Reference Manual. ARMv8, for ARMv8-A Architecture Profile*, ARM Limited, Cambridge, UK. Version dated 31 October 2018. Original release dated 30 April 2013.

ARM (2019), *Arm A64 Instruction Set Architecture Armv8, for Armv8-A Architecture Profile*, ARM Limited, Cambridge, UK.

ARM (2020), *Arm Architecture Reference Manual. Armv8, for Armv8-A Architecture Profile*, ARM Limited, Cambridge, UK. ARM DDI 0487F.b (ID040120).

M. Baboulin, A. Buttari, J. Dongarra, J. Kurzak, J. Langou, J. Langou, P. Luszczek and S. Tomov (2009), Accelerating scientific computations with mixed precision algorithms, *Comput. Phys. Comm.* **180**, 2526–2533.

D. H. Bailey (2021), MPFUN2020: A new thread-safe arbitrary precision package (full documentation). Available at https://www.davidhbailey.com/dhbpapers/mpfun2020.pdf.

D. H. Bailey, Y. Hida, X. S. Li and B. Thompson (2002), ARPREC: An arbitrary precision computation package. Technical report LBNL-53651, Lawrence Berkeley National Laboratory, Berkeley, CA, USA.

P. Bauer, P. D. Dueben, T. Hoefler, T. Quintino, T. C. Schulthess and N. P. Wedi (2021), The digital revolution of earth-system science, *Nature Comput. Sci.* **1**, 104–113.

J. Bezanson, A. Edelman, S. Karpinski and V. B. Shah (2017), Julia: A fresh approach to numerical computing, *SIAM Rev.* **59**, 65–98.

Å. Björck (1967), Iterative refinement of linear least squares solutions I, *BIT* **7**, 257–278.

Å. Björck (1996), *Numerical Methods for Least Squares Problems*, SIAM.

P. Blanchard, N. J. Higham and T. Mary (2020*a*), A class of fast and accurate summation algorithms,, *SIAM J. Sci. Comput.* **42**, A1541–A1557.

P. Blanchard, N. J. Higham, F. Lopez, T. Mary and S. Pranesh (2020*b*), Mixed precision block fused multiply-add: Error analysis and application to GPU tensor cores, *SIAM J. Sci. Comput.* **42**, C124–C141.

A. Bouras and V. Frayssé (2005), Inexact matrix-vector products in Krylov methods for solving linear systems: A relaxation strategy, *SIAM J. Matrix Anal. Appl.* **26**, 660–678.

A. Bouras, V. Frayssé and L. Giraud (2000), A relaxation strategy for inner–outer linear solvers in domain decomposition methods. Technical report TR/PA/00/17, CERFACS, Toulouse, France.

E. Brun, D. Defour, P. De Oliveira Castro, M. Iştoan, D. Mancusi, E. Petit and A. Vaquet (2021), A study of the effects and benefits of custom-precision mathematical libraries for HPC codes, *IEEE Trans. Emerg. Topics Comput.* **9**, 1467–1478.

A. Buttari, J. Dongarra, J. Kurzak, P. Luszczek and S. Tomov (2008), Using mixed precision for sparse matrix computations to enhance the performance while achieving 64-bit accuracy, *ACM Trans. Math. Software* **34**, 17.

A. Buttari, J. Dongarra, J. Langou, J. Langou, P. Luszczek and J. Kurzak (2007), Mixed precision iterative refinement techniques for the solution of dense linear systems, *Int. J. High Perform. Comput. Appl.* **21**, 457–466.

E. Carson and N. J. Higham (2017), A new analysis of iterative refinement and its application to accurate solution of ill-conditioned sparse linear systems, *SIAM J. Sci. Comput.* **39**, A2834–A2856.

https://doi.org/10.1017/S0962492922000022 Published online by Cambridge University Press

E. Carson and N. J. Higham (2018), Accelerating the solution of linear systems by iterative refinement in three precisions, *SIAM J. Sci. Comput.* **40**, A817–A847.

E. Carson, T. Gergelits and I. Yamazaki (2022*a*), Mixed precision *s*-step Lanczos and conjugate gradient algorithms, *Numer. Linear Algebra Appl.* **29**, e2425.

E. Carson, N. J. Higham and S. Pranesh (2020), Three-precision GMRES-based iterative refinement for least squares problems, *SIAM J. Sci. Comput.* **42**, A4063–A4083.

E. Carson, K. Lund, M. Rozložník and S. Thomas (2022*b*), Block Gram–Schmidt algorithms and their stability properties, *Linear Algebra Appl.* **638**, 150–195.

A. Charara, M. Gates, J. Kurzak, A. YarKhan and J. Dongarra (2020), SLATE developers' guide. SLATE Working Note 11, Innovative Computing Laboratory, The University of Tennessee, Knoxville, TN, US.

J. Choquette, W. Gandhi, O. Giroux, N. Stam and R. Krashinsky (2021), NVIDIA A100 tensor core GPU: Performance and innovation, *IEEE Micro* **41**, 29–35.

M. A. Clark, R. Babich, K. Barros, R. C. Brower and C. Rebbi (2010), Solving lattice QCD systems of equations using mixed precision solvers on GPUs, *Comput. Phys. Comm.* **181**, 1517–1528.

M. P. Connolly and N. J. Higham (2022), Probabilistic rounding error analysis of Householder QR factorization. MIMS EPrint 2022.5, Manchester Institute for Mathematical Sciences, The University of Manchester, UK.

M. P. Connolly, N. J. Higham and T. Mary (2021), Stochastic rounding and its probabilistic backward error analysis, *SIAM J. Sci. Comput.* **43**, A566–A585.

M. Courbariaux, Y. Bengio and J.-P. David (2015), Training deep neural networks with low precision multiplications. Available at arXiv:1412.7024v5.

M. G. Croarken (1985), The centralization of scientific computation in Britain 1925–1955. PhD thesis, University of Warwick, Coventry, UK.

M. Croci, M. Fasi, N. J. Higham, T. Mary and M. Mikaitis (2022), Stochastic rounding: Implementation, error analysis, and applications, *Roy. Soc. Open Sci.* **9**, 1–25.

P. I. Davies, N. J. Higham and F. Tisseur (2001), Analysis of the Cholesky method with iterative refinement for solving the symmetric definite generalized eigenproblem, *SIAM J. Matrix Anal. Appl.* **23**, 472–493.

T. A. Davis and Y. Hu (2011), The University of Florida Sparse Matrix Collection, *ACM Trans. Math. Software* **38**, 1.

A. Dawson and P. D. Düben (2017), rpe v5: An emulator for reduced floating-point precision in large numerical simulations, *Geosci. Model Dev.* **10**, 2221–2230.

A. Dawson, P. D. Düben, D. A. MacLeod and T. N. Palmer (2018), Reliable low precision simulations in land surface models, *Climate Dynam.* **51**, 2657–2666.

J. Dean (2020), The deep learning revolution and its implications for computer architecture and chip design, in *2020 IEEE International Solid-State Circuits Conference (ISSCC)*, IEEE, pp. 8–14.

J. Demmel and Y. Hida (2004), Accurate and efficient floating point summation, *SIAM J. Sci. Comput.* **25**, 1214–1248.

J. Demmel and X. Li (1994), Faster numerical algorithms via exception handling, *IEEE Trans. Comput.* **43**, 983–992.

J. Demmel, Y. Hida, E. J. Riedy and X. S. Li (2009), Extra-precise iterative refinement for overdetermined least squares problems, *ACM Trans. Math. Software* **35**, 28.

J. E. Dennis, Jr and R. B. Schnabel (1983), *Numerical Methods for Unconstrained Optimization and Nonlinear Equations*, Prentice Hall. Reprinted by SIAM, 1996.

https://doi.org/10.1017/S0962492922000022 Published online by Cambridge University Press

S. Di and F. Cappello (2016), Fast error-bounded lossy HPC data compression with SZ, in *2016 IEEE International Parallel and Distributed Processing Symposium (IPDPS)*, IEEE, pp. 730–739.

J. Diffenderfer, D. Osei-Kuffuor and H. Menon (2021), QDOT: Quantized dot product kernel for approximate high-performance computing. Available at arXiv:2105.00115.

J. J. Dongarra (1980), Improving the accuracy of computed matrix eigenvalues. Preprint ANL-80-84, Mathematics and Computer Science Division, Argonne National Laboratory, Argonne, IL, USA.

J. J. Dongarra (1982), Algorithm 589 SICEDR: A FORTRAN subroutine for improving the accuracy of computed matrix eigenvalues, *ACM Trans. Math. Software* **8**, 371–375.

J. J. Dongarra (1983), Improving the accuracy of computed singular values, *SIAM J. Sci. Statist. Comput.* **4**, 712–719.

J. J. Dongarra (2020), Report on the Fujitsu Fugaku system. Technical report ICL-UT-20-06, Innovative Computing Laboratory, The University of Tennessee, Knoxville, TN, USA.

J. J. Dongarra, J. R. Bunch, C. B. Moler and G. W. Stewart (1979), *LINPACK Users' Guide*, SIAM.

J. J. Dongarra, C. B. Moler and J. H. Wilkinson (1983), Improving the accuracy of computed eigenvalues and eigenvectors, *SIAM J. Numer. Anal.* **20**, 23–45.

N. Doucet, H. Ltaief, D. Gratadour and D. Keyes (2019), Mixed-precision tomographic reconstructor computations on hardware accelerators, in *2019 IEEE/ACM 9th Workshop on Irregular Applications: Architectures and Algorithms (IA3)*, IEEE, pp. 31–38.

P. D. Düben, A. Subramanian, A. Dawson and T. N. Palmer (2017), A study of reduced numerical precision to make superparameterization more competitive using a hardware emulator in the OpenIFS model, *J. Adv. Model. Earth Syst.* **9**, 566–584.

I. S. Duff and S. Pralet (2007), Towards stable mixed pivoting strategies for the sequential and parallel solution of sparse symmetric indefinite systems, *SIAM J. Matrix Anal. Appl.* **29**, 1007–1024.

I. S. Duff, A. M. Erisman and J. K. Reid (2017), *Direct Methods for Sparse Matrices*, second edition, Oxford University Press.

M. Emans and A. van der Meer (2012), Mixed-precision AMG as linear equation solver for definite systems, *Procedia Comput. Sci.* **1**, 175–183.

M. Fasi and N. J. Higham (2018), Multiprecision algorithms for computing the matrix logarithm, *SIAM J. Matrix Anal. Appl.* **39**, 472–491.

M. Fasi and N. J. Higham (2019), An arbitrary precision scaling and squaring algorithm for the matrix exponential, *SIAM J. Matrix Anal. Appl.* **40**, 1233–1256.

M. Fasi and N. J. Higham (2021), Matrices with tunable infinity-norm condition number and no need for pivoting in LU factorization, *SIAM J. Matrix Anal. Appl.* **42**, 417–435.

M. Fasi and M. Mikaitis (2020), CPFloat: A C library for emulating low-precision arithmetic. MIMS EPrint 2020.22, Manchester Institute for Mathematical Sciences, The University of Manchester, UK.

M. Fasi, N. J. Higham, F. Lopez, T. Mary and M. Mikaitis (2022), Matrix multiplication in multiword arithmetic: Error analysis and application to GPU tensor cores. MIMS EPrint 2022.3, Manchester Institute for Mathematical Sciences, The University of Manchester, UK.

M. Fasi, N. J. Higham, M. Mikaitis and S. Pranesh (2021), Numerical behavior of NVIDIA tensor cores, *PeerJ Comput. Sci.* **7**, e330.

https://doi.org/10.1017/S0962492922000022 Published online by Cambridge University Press

G. Flegar, H. Anzt, T. Cojean and E. S. Quintana-Ortí (2021), Adaptive precision block-Jacobi for high performance preconditioning in the Ginkgo linear algebra software, *ACM Trans. Math. Software* **47**, 1–28.

L. Fousse, G. Hanrot, V. Lefèvre, P. Pélissier and P. Zimmermann (2007), MPFR: A multiple-precision binary floating-point library with correct rounding, *ACM Trans. Math. Software* **33**, 13.

L. Fox, H. D. Huskey and J. H. Wilkinson (1948), The solution of algebraic linear simultaneous equations by punched card methods. Report, Mathematics Division, Department of Scientific and Industrial Research, National Physical Laboratory, Teddington, UK.

T. Fukaya, R. Kannan, Y. Nakatsukasa, Y. Yamamoto and Y. Yanagisawa (2020), Shifted Cholesky QR for computing the QR factorization of ill-conditioned matrices, *SIAM J. Sci. Comput.* **42**, A477–A503.

J. Gao, F. Zheng, F. Qi, Y. Ding, H. Li, H. Lu, W. He, H. Wei, L. Jin, X. Liu, D. Gong, F. Wang, Y. Zheng, H. Sun, Z. Zhou, Y. Liu and H. You (2021), Sunway supercomputer architecture towards exascale computing: Analysis and practice, *Sci. China Inform. Sci.* **64**, 141101.

P. E. Gill, M. A. Saunders and J. R. Shinnerl (1996), On the stability of Cholesky factorization for symmetric quasidefinite systems, *SIAM J. Matrix Anal. Appl.* **17**, 35–46.

L. Giraud, S. Gratton and J. Langou (2007), Convergence in backward error of relaxed GMRES, *SIAM J. Sci. Comput.* **29**, 710–728.

L. Giraud, A. Haidar and L. T. Watson (2008), Mixed-precision preconditioners in parallel domain decomposition solvers, in *Domain Decomposition Methods in Science and Engineering XVII* (U. Langer *et al.*, eds), Vol. 60 of Lecture Notes in Computational Science and Engineering, Springer, pp. 357–364.

L. Giraud, J. Langou, M. Rozložník and J. van den Eshof (2005), Rounding error analysis of the classical Gram–Schmidt orthogonalization process, *Numer. Math.* **101**, 87–100.

F. Göbel, T. Grützmacher, T. Ribizel and H. Anzt (2021), Mixed precision incomplete and factorized sparse approximate inverse preconditioning on GPUs, in *Euro-Par 2021: Parallel Processing*, Vol. 12820 of Lecture Notes in Computer Science, Springer, pp. 550–564.

D. Goddeke and R. Strzodka (2011), Cyclic reduction tridiagonal solvers on GPUs applied to mixed-precision multigrid, *IEEE Trans. Parallel Distrib. Syst.* **22**, 22–32.

D. Göddeke, R. Strzodka and S. Turek (2007), Performance and accuracy of hardware-oriented native-, emulated- and mixed-precision solvers in FEM simulations, *Int. J. Parallel Emergent Distrib. Syst.* **22**, 221–256.

W. Govaerts and J. D. Pryce (1990), Block elimination with one iterative refinement solves bordered linear systems accurately, *BIT* **30**, 490–507.

S. Graillat, F. Jézéquel, T. Mary and R. Molina (2022), Adaptive precision matrix–vector product. Available at hal-03561193.

S. Gratton, E. Simon, D. Titley-Peloquin and P. Toint (2019), Exploiting variable precision in GMRES. Available at arXiv:1907.10550.

A. Greenbaum (1997), Estimating the attainable accuracy of recursively computed residual methods, *SIAM J. Matrix Anal. Appl.* **18**, 535–551.

J. F. Groote, R. Morel, J. Schmaltz and A. Watkins (2021), *Logic Gates, Circuits, Processors, Compilers and Computers*, Springer.

T. Grützmacher, H. Anzt and E. S. Quintana-Ortí (2021), Using Ginkgo's memory accessor for improving the accuracy of memory-bound low precision BLAS, *Software Pract. Exper.* Available at doi:10.1002/spe.3041.

https://doi.org/10.1017/S0962492922000022 Published online by Cambridge University Press

M. Gulliksson (1994), Iterative refinement for constrained and weighted linear least squares, *BIT* **34**, 239–253.

S. Gupta, A. Agrawal, K. Gopalakrishnan and P. Narayanan (2015), Deep learning with limited numerical precision, in *Proceedings of the 32nd International Conference on Machine Learning* (F. Bach and D. Blei, eds), Vol. 37 of Proceedings of Machine Learning Research, PMLR, pp. 1737–1746.

A. Haidar, A. Abdelfattah, M. Zounon, P. Wu, S. Pranesh, S. Tomov and J. Dongarra (2018*a*), The design of fast and energy-efficient linear solvers: On the potential of half-precision arithmetic and iterative refinement techniques, in *Computational Science – ICCS 2018* (Y. Shi *et al.*, eds), Vol. 10860 of Lecture Notes in Computer Science, Springer, pp. 586–600.

A. Haidar, H. Bayraktar, S. Tomov, J. Dongarra and N. J. Higham (2020), Mixed-precision iterative refinement using tensor cores on GPUs to accelerate solution of linear systems, *Proc. Roy. Soc. London A* **476** (2243), 20200110.

A. Haidar, S. Tomov, J. Dongarra and N. J. Higham (2018*b*), 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 (SC18)*, IEEE, article 47.

A. Haidar, P. Wu, S. Tomov and J. Dongarra (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 (ScalA '17)*, ACM Press, article 10.

R. Harvey and D. L. Verseghy (2015), The reliability of single precision computations in the simulation of deep soil heat diffusion in a land surface model, *Climate Dynam.* **16**, 3865–3882.

G. Henry, P. T. P. Tang and A. Heinecke (2019), Leveraging the bfloat16 artificial intelligence datatype for higher-precision computations, in *2019 IEEE 26th Symposium on Computer Arithmetic (ARITH)*, IEEE, pp. 69–76.

D. J. Higham, N. J. Higham and S. Pranesh (2021), Random matrices generating large growth in LU factorization with pivoting, *SIAM J. Matrix Anal. Appl.* **42**, 185–201.

N. J. Higham (1986), Computing the polar decomposition: With applications, *SIAM J. Sci. Statist. Comput.* **7**, 1160–1174.

N. J. Higham (1988), Fast solution of Vandermonde-like systems involving orthogonal polynomials, *IMA J. Numer. Anal.* **8**, 473–486.

N. J. Higham (1991), Iterative refinement enhances the stability of $QR$ factorization methods for solving linear equations, *BIT* **31**, 447–468.

N. J. Higham (1997), Iterative refinement for linear systems and LAPACK, *IMA J. Numer. Anal.* **17**, 495–509.

N. J. Higham (2002), *Accuracy and Stability of Numerical Algorithms*, second edition, SIAM.

N. J. Higham (2008), *Functions of Matrices: Theory and Computation*, SIAM.

N. J. Higham (2021), Numerical stability of algorithms at extreme scale and low precisions. MIMS EPrint 2021.14, Manchester Institute for Mathematical Sciences, The University of Manchester, UK. To appear in *Proc. Int. Cong. Math.*

N. J. Higham and X. Liu (2021), A multiprecision derivative-free Schur–Parlett algorithm for computing matrix functions, *SIAM J. Matrix Anal. Appl.* **42**, 1401–1422.

https://doi.org/10.1017/S0962492922000022 Published online by Cambridge University Press

N. J. Higham and T. Mary (2019*a*), A new approach to probabilistic rounding error analysis, *SIAM J. Sci. Comput.* **41**, A2815–A2835.

N. J. Higham and T. Mary (2019*b*), A new preconditioner that exploits low-rank approximations to factorization error, *SIAM J. Sci. Comput.* **41**, A59–A82.

N. J. Higham and T. Mary (2020), Sharper probabilistic backward error analysis for basic linear algebra kernels with random data, *SIAM J. Sci. Comput.* **42**, A3427–A3446.

N. J. Higham and T. Mary (2021), Solving block low-rank linear systems by LU factorization is numerically stable, *IMA J. Numer. Anal.* Available at doi:10.1093/imanum/drab020.

N. J. Higham and S. Pranesh (2019), Simulating low precision floating-point arithmetic, *SIAM J. Sci. Comput.* **41**, C585–C602.

N. J. Higham and S. Pranesh (2021), Exploiting lower precision arithmetic in solving symmetric positive definite linear systems and least squares problems, *SIAM J. Sci. Comput.* **43**, A258–A277.

N. J. Higham, S. Pranesh and M. Zounon (2019), Squeezing a matrix into half precision, with an application to solving linear systems, *SIAM J. Sci. Comput.* **41**, A2536–A2551.

N.-M. Ho, H. De Silva and W.-F. Wong (2021), GRAM: A framework for dynamically mixing precisions in GPU applications, *ACM Trans. Archit. Code Optim.* **18**, 1–24.

J. D. Hogg and J. A. Scott (2010), A fast and robust mixed-precision solver for the solution of sparse symmetric linear systems, *ACM Trans. Math. Software* **37**, 17.

Y. Idomura, T. Ina, Y. Ali and T. Imamura (2020), Acceleration of fusion plasma turbulence simulations using the mixed-precision communication-avoiding Krylov method, in *International Conference for High Performance Computing, Networking, Storage and Analysis (SC20)*, IEEE, pp. 1–13.

IEEE (1985), *IEEE Standard for Binary Floating-Point Arithmetic, ANSI/IEEE Standard* 754-1985, Institute of Electrical and Electronics Engineers.

IEEE (2008), *IEEE Standard for Floating-Point Arithmetic, IEEE Std* 754-2008 (*Revision of IEEE* 754-1985), Institute of Electrical and Electronics Engineers.

Intel Corporation (2018), BFLOAT16: Hardware Numerics Definition. White paper. Document number 338302-001US.

I. C. F. Ipsen and H. Zhou (2020), Probabilistic error analysis for inner products, *SIAM J. Matrix Anal. Appl.* **41**, 1726–1741.

T. Iwashita, K. Suzuki and T. Fukaya (2020), An integer arithmetic-based sparse linear solver using a GMRES method and iterative refinement, in *2020 IEEE/ACM 11th Workshop on Latest Advances in Scalable Algorithms for Large-Scale Systems (ScalA)*, IEEE, pp. 1–8.

M. Jankowski and H. Woźniakowski (1977), Iterative refinement implies numerical stability, *BIT* **17**, 303–311.

F. Johansson *et al.* (2013), Mpmath: A Python library for arbitrary-precision floating-point arithmetic. Available at http://mpmath.org.

M. Joldes, J.-M. Muller and V. Popescu (2017), Tight and rigorous error bounds for basic building blocks of double-word arithmetic, *ACM Trans. Math. Software* **44**, 15res.

N. P. Jouppi, D. H. Yoon, M. Ashcraft, M. Gottscho, T. B. Jablin, G. Kurian, J. Laudon, S. Li, P. Ma, X. Ma, T. Norrie, N. Patil, S. Prasad, C. Young, Z. Zhou and D. Patterson (2021), Ten lessons from three generations shaped Google's TPUv4i: Industrial product, in *2021 ACM/IEEE 48th Annual International Symposium on Computer Architecture (ISCA)*, IEEE, pp. 1–14.

https://doi.org/10.1017/S0962492922000022 Published online by Cambridge University Press

N. P. Jouppi, D. H. Yoon, G. Kurian, S. Li, N. Patil, J. Laudon, C. Young and D. Patterson (2020), A domain-specific supercomputer for training deep neural networks, *Comm. Assoc. Comput. Mach.* **63**, 67–78.

W. Kahan (1981), Why do we need a floating-point arithmetic standard? Technical report, University of California, Berkeley, CA, USA.

C. T. Kelley (1995), *Iterative Methods for Linear and Nonlinear Equations*, SIAM.

C. T. Kelley (2022), Newton's method in mixed precision, *SIAM Rev.* **64**, 191–211.

A. Kiełbasiński (1981), Iterative refinement for linear systems in variable-precision arithmetic, *BIT* **21**, 97–103.

P. A. Knight, D. Ruiz and B. Uçar (2014), A symmetry preserving algorithm for matrix scaling, *SIAM J. Matrix Anal. Appl.* **35**, 931–955.

M. Kronbichler and K. Ljungkvist (2019), Multigrid for matrix-free high-order finite element computations on graphics processors, *ACM Trans. Parallel Comput.* **6**, 2.

S. Kudo, K. Nitadori, T. Ina and T. Imamura (2020*a*), Implementation and numerical techniques for one EFlop/s HPL-AI benchmark on Fugaku, in *Proceedings of the 11th IEEE/ACM Workshop on Latest Advances in Scalable Algorithms for Large-Scale*, Vol. 1, IEEE, pp. 69–76.

S. Kudo, K. Nitadori, T. Ina and T. Imamura (2020*b*), Prompt report on exa-scale HPL-AI benchmark, in *2020 IEEE International Conference on Cluster Computing (CLUSTER)*, IEEE, pp. 418–419.

J. Kurzak and J. Dongarra (2007), Implementation of mixed precision in solving systems of linear equations on the Cell processor, *Concurrency Comput. Pract. Exper.* **19**, 1371–1385.

J. Langou, J. Langou, P. Luszczek, J. Kurzak, A. Buttari and J. Dongarra (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 (SC '06)*, IEEE.

V. Lefèvre and P. Zimmermann (2017), Optimized binary64 and binary128 arithmetic with GNU MPFR, in *2017 IEEE 24th Symposium on Computer Arithmetic (ARITH)*, IEEE, pp. 18–26.

X. S. Li and J. W. Demmel (1998), Making sparse Gaussian elimination scalable by static pivoting, in *Proceedings of the 1998 ACM/IEEE Conference on Supercomputing*, IEEE, pp. 1–17.

X. S. Li and J. W. Demmel (2003), SuperLU_DIST: A scalable distributed-memory sparse direct solver for unsymmetric linear systems, *ACM Trans. Math. Software* **29**, 110–140.

X. S. Li, J. W. Demmel, D. H. Bailey, G. Henry, Y. Hida, J. Iskandar, W. Kahan, S. Y. Kang, A. Kapur, M. C. Martin, B. J. Thompson, T. Tung and D. J. Yoo (2002), Design, implementation and testing of extended and mixed precision BLAS, *ACM Trans. Math. Software* **28**, 152–205.

C. Lichtenau, S. Carlough and S. M. Mueller (2016), Quad precision floating point on the IBM z13, in *2016 IEEE 23rd Symposium on Computer Arithmetic (ARITH)*, IEEE, pp. 87–94.

N. Lindquist, P. Luszczek and J. Dongarra (2020), Improving the performance of the GMRES method using mixed-precision techniques, in *Communications in Computer and Information Science* (J. Nichols *et al.*, eds), Springer, pp. 51–66.

N. Lindquist, P. Luszczek and J. Dongarra (2022), Accelerating restarted GMRES with mixed precision arithmetic, *IEEE Trans. Parallel Distrib. Syst.* **33**, 1027–1037.

https://doi.org/10.1017/S0962492922000022 Published online by Cambridge University Press

J. A. Loe, C. A. Glusa, I. Yamazaki, E. G. Boman and S. Rajamanickam (2021*a*), Experimental evaluation of multiprecision strategies for GMRES on GPUs, in *2021 IEEE International Parallel and Distributed Processing Symposium Workshops (IPDPSW)*, IEEE, pp. 469–478.

J. A. Loe, C. A. Glusa, I. Yamazaki, E. G. Boman and S. Rajamanickam (2021*b*), A study of mixed precision strategies for GMRES on GPUs. Available at arXiv:2109.01232.

F. Lopez and T. Mary (2020), Mixed precision LU factorization on GPU tensor cores: Reducing data movement and memory footprint. MIMS EPrint 2020.20, Manchester Institute for Mathematical Sciences, The University of Manchester, UK.

P. Luszczek, I. Yamazaki and J. Dongarra (2019), Increasing accuracy of iterative refinement in limited floating-point arithmetic on half-precision accelerators, in *2019 IEEE High Performance Extreme Computing Conference (HPEC)*, IEEE, pp. 1–6.

S. Markidis, S. Wei Der Chien, E. Laure, I. B. Peng and J. S. Vetter (2018), NVIDIA tensor core programmability, performance & precision, in *2018 IEEE International Parallel and Distributed Processing Symposium Workshops (IPDPSW)*, IEEE, pp. 522–531.

C. M. Maynard and D. N. Walters (2019), Mixed-precision arithmetic in the ENDGame dynamical core of the unified model, a numerical weather prediction and climate model code, *Comput. Phys. Comm.* **244**, 69–75.

S. F. McCormick, J. Benzaken and R. Tamstorf (2021), Algebraic error analysis for mixed-precision multigrid solvers, *SIAM J. Sci. Comput.* **43**, S392–S419.

A. Meurer, C. P. Smith, M. Paprocki, O. Čertik, S. B. Kirpichev, M. Rocklin, A. Kumar, S. Ivanov, J. K. Moore, S. Singh, T. Rathnayake, S. Vig, B. E. Granger, R. P. Muller, F. Bonazzi, H. Gupta, S. Vats, F. Johansson, F. Pedregosa, M. J. Curry, A. R. Terrel, Š. Roučka, A. Saboo, I. Fernando, S. Kulal, R. Cimrman and A. Scopatz (2017), SymPy: Symbolic computing in Python, *PeerJ Comput. Sci.* **3**, e103.

C. B. Moler (1967), Iterative refinement in floating point, *J. Assoc. Comput. Mach.* **14**, 316–321.

C. B. Moler (2017), 'Half precision' 16-bit floating point arithmetic. Available at http://blogs.mathworks.com/cleve/2017/05/08/half-precision-16-bit-floating-point-arithmetic/.

C. B. Moler (2019), Variable format half precision floating point arithmetic. Available at https://blogs.mathworks.com/cleve/2019/01/16/variable-format-half-precision-floating-point-arithmetic/.

D. Mukunoki, K. Ozaki, T. Ogita and T. Imamura (2020), DGEMM using tensor cores, and its accurate and reproducible versions, in *High Performance Computing* (P. Sadayappan *et al.*, eds), Springer, pp. 230–248.

J.-M. Muller, N. Brunie, F. de Dinechin, C.-P. Jeannerod, M. Joldes, V. Lefèvre, G. Melquiond, N. Revol and S. Torres (2018), *Handbook of Floating-Point Arithmetic*, second edition, Birkhäuser.

M. Nakata (2021), MPLAPACK version 1.0.0 user manual. Available at arXiv:2109.13406.

T. Norrie, N. Patil, D. H. Yoon, G. Kurian, S. Li, J. Laudon, C. Young, N. Jouppi and D. Patterson (2021), The design process for Google's training chips: TPUv2 and TPUv3, *IEEE Micro* **41**, 56–63.

NVIDIA Corporation (2020), NVIDIA A100 Tensor Core GPU Architecture, v1.0.

T. Ogita and K. Aishima (2018), Iterative refinement for symmetric eigenvalue decomposition, *Japan J. Indust. Appl. Math.* **35**, 1007–1035.

https://doi.org/10.1017/S0962492922000022 Published online by Cambridge University Press

T. Ogita and K. Aishima (2019), Iterative refinement for symmetric eigenvalue decomposition II: Clustered eigenvalues, *Japan J. Indust. Appl. Math.* **36**, 435–459.

T. Ogita and K. Aishima (2020), Iterative refinement for singular value decomposition based on matrix multiplication, *J. Comput. Appl. Math.* **369**, 112512.

E. Oktay and E. Carson (2022), Multistage mixed precision iterative refinement, *Numer. Linear Algebra Appl.* Available at doi:10.1002/nla.2434.

K. L. Oo and A. Vogel (2020), Accelerating geometric multigrid preconditioning with half-precision arithmetic on GPUs. Available at arXiv:2007.07539.

R. Ooi, T. Iwashita, T. Fukaya, A. Ida and R. Yokota (2020), Effect of mixed precision computing on H-matrix vector multiplication in BEM analysis, in *Proceedings of the International Conference on High Performance Computing in Asia-Pacific Region*, ACM Press.

S.-i. O'uchi, H. Fuketa, T. Ikegami, W. Nogami, T. Matsukawa, T. Kudoh and R. Takano (2018), Image-classifier deep convolutional neural network training by 9-bit dedicated hardware to realize validation accuracy and energy efficiency superior to the half precision floating point format, in *2018 IEEE International Symposium on Circuits and Systems (ISCAS)*, IEEE, pp. 1–5.

C. C. Paige, M. Rozložník and Z. Strakoš (2006), Modified Gram–Schmidt (MGS), least squares, and backward stability of MGS-GMRES, *SIAM J. Matrix Anal. Appl.* **28**, 264–284.

T. N. Palmer (2014), More reliable forecasts with less precise computations: A fast-track route to cloud-resolved weather and climate simulators?, *Phil. Trans. R. Soc. A* **372** (2018), 1–14.

T. N. Palmer (2020), The physics of numerical analysis: A climate modelling case study, *Phil. Trans. R. Soc. A* **378** (2166), 1–6.

M. Petschow, E. Quintana-Ortí and P. Bientinesi (2014), Improved accuracy and parallelism for MRRR-based eigensolvers: A mixed precision approach, *SIAM J. Sci. Comput.* **36**, C240–C263.

L. Pisha and L. Ligowski (2021), Accelerating non-power-of-2 size Fourier transforms with GPU tensor cores, in *2021 IEEE International Parallel and Distributed Processing Symposium (IPDPS)*, IEEE, pp. 507–516.

R. Ralha (2018), Mixed precision bisection, *Math. Comput. Sci.* **12**, 173–181.

C. Rubio-González, C. Nguyen, H. D. Nguyen, J. Demmel, W. Kahan, K. Sen, D. H. Bailey, C. Iancu and D. Hough (2013), Precimonious: Tuning assistant for floating-point precision, in *Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis (SC '13)*, ACM Press, article 27.

P. San Juan, R. Rodríguez-Sánchez, F. D. Igual, P. Alonso-Jordá and E. S. Quintana-Ortí (2021), Low precision matrix multiplication for efficient deep learning in NVIDIA carmel processors, *J. Supercomput.* **77**, 11257–11269.

M. Sato, Y. Ishikawa, H. Tomita, Y. Kodama, T. Odajima, M. Tsuji, H. Yashiro, M. Aoki, N. Shida, I. Miyoshi, K. Hirai, A. Furuya, A. Asato, K. Morita and T. Shimizu (2020), Co-design for A64FX manycore processor and 'Fugaku', in *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis (SC '20)*, IEEE.

K. Scheinberg (2016), Evolution of randomness in optimization methods for supervised machine learning, *SIAG/OPT Views and News* **24**, 1–8.

https://doi.org/10.1017/S0962492922000022 Published online by Cambridge University Press

O. Schenk, K. Gärtner, W. Fichtner and A. Stricker (2001), PARDISO: A high-performance serial and parallel sparse linear solver in semiconductor device simulation, *Future Gener. Comput. Syst.* **18**, 69–78.

V. Simoncini and D. B. Szyld (2003), Theory of inexact Krylov subspace methods and applications to scientific computing, *SIAM J. Sci. Comput.* **25**, 454–477.

R. D. Skeel (1980), Iterative refinement implies numerical stability for Gaussian elimination, *Math. Comp.* **35**, 817–832.

A. Smoktunowicz and J. Sokolnicka (1984), Binary cascades iterative refinement in doubled-mantissa arithmetics, *BIT* **24**, 123–127.

A. Sorna, X. Cheng, E. D'Azevedo, K. Won and S. Tomov (2018), Optimizing the fast Fourier transform using mixed precision on tensor core hardware, in *2018 IEEE 25th International Conference on High Performance Computing Workshops (HiPCW)*, IEEE, pp. 3–7.

A. Stathopoulos and K. Wu (2002), A block orthogonalization procedure with constant synchronization requirements, *SIAM J. Sci. Comput.* **23**, 2165–2182.

G. W. Stewart (1973), *Introduction to Matrix Computations*, Academic Press.

N. J. Stor, I. Slapničar and J. L. Barlow (2015), Accurate eigenvalue decomposition of real symmetric arrowhead matrices and applications, *Linear Algebra Appl.* **464**, 62–89.

Y. Sumiyoshi, A. Fujii, A. Nukada and T. Tanaka (2014), Mixed-precision AMG method for many core accelerators, in *Proceedings of the 21st European MPI Users' Group Meeting (EuroMPI/ASIA '14)*, ACM Press, pp. 127–132.

J. Sun, G. D. Peterson and O. O. Storaasli (2008), High-performance mixed-precision linear solver for FPGAs, *IEEE Trans. Comput.* **57**, 1614–1623.

G. Tagliavini, S. Mach, D. Rossi, A. Marongiu and L. Benin (2018), A transprecision floating-point platform for ultra-low power computing, in *2018 Design, Automation and Test in Europe Conference and Exhibition (DATE)*, pp. 1051–1056.

R. Tamstorf, J. Benzaken and S. F. McCormick (2021), Discretization-error-accurate mixed-precision multigrid solvers, *SIAM J. Sci. Comput.* **43**, S420–S447.

O. Tintó Prims, M. C. Acosta, A. M. Moore, M. Castrillo, K. Serradell, A. Cortés and F. J. Doblas-Reyes (2019), How to use mixed precision in ocean models: Exploring a potential reduction of numerical precision in NEMO 4.0 and ROMS 3.6, *Geosci. Model Dev.* **12**, 3135–3148.

F. Tisseur (2001), Newton's method in floating point arithmetic and iterative refinement of generalized eigenvalue problems, *SIAM J. Matrix Anal. Appl.* **22**, 1038–1057.

T. Trader (2016), IBM advances against x86 with Power9. Available at https://www.hpcwire.com/2016/08/30/ibm-unveils-power9-details/.

Y. M. Tsai, P. Luszczek and J. Dongarra (2021), Mixed-precision algorithm for finding selected eigenvalues and eigenvectors of symmetric and Hermitian matrices. Technical report ICL-UT-21-05, Innovative Computing Laboratory, The University of Tennessee, Knoxville, TN, USA.

E. Tsuchida and Y.-K. Choe (2012), Iterative diagonalization of symmetric matrices in mixed precision and its application to electronic structure calculations, *Comput. Phys. Comm.* **183**, 980–985.

K. Turner and H. F. Walker (1992), Efficient high accuracy solutions with GMRES($m$), *SIAM J. Sci. Statist. Comput.* **12**, 815–825.

J. van den Eshof and G. L. G. Sleijpen (2004), Inexact Krylov subspace methods for linear systems, *SIAM J. Matrix Anal. Appl.* **26**, 125–153.

https://doi.org/10.1017/S0962492922000022 Published online by Cambridge University Press

F. Váňa, P. Düben, S. Lang, T. Palmer, M. Leutbecher, D. Salmond and G. Carver (2017), Single precision in weather forecasting models: An evaluation with the IFS, *Mon. Weather Rev.* **145**, 495–502.

J. von Neumann and H. H. Goldstine (1947), Numerical inverting of matrices of high order, *Bull. Amer. Math. Soc.* **53**, 1021–1099.

E. Wang, J. J. Davis, R. Zhao, H.-C. Ng, X. Niu, W. Luk, P. Y. K. Cheung and G. A. Constantinides (2019), Deep neural network approximation for custom hardware, *ACM Comput. Surv.* **52**, 1–39.

N. Wang, J. Choi, D. Brand, C.-Y. Chen and K. Gopalakrishnan (2018), Training deep neural networks with 8-bit floating point numbers, in *Advances in Neural Information Processing Systems 31* (S. Bengio *et al.*, eds), Curran Associates, pp. 7686–7695.

S. Wang and P. Kanwar (2019), BFloat16: The secret to high performance on cloud TPUs. Available at https://cloud.google.com/blog/products/ai-machine-learning/bfloat16-the-secret-to-high-performance-on-cloud-tpus.

J. H. Wilkinson (1948), Progress report on the Automatic Computing Engine. Report MA/17/1024, Mathematics Division, Department of Scientific and Industrial Research, National Physical Laboratory, Teddington, UK.

J. H. Wilkinson (1961), Error analysis of direct methods of matrix inversion, *J. Assoc. Comput. Mach.* **8**, 281–330.

J. H. Wilkinson (1963), *Rounding Errors in Algebraic Processes*, Notes on Applied Science No. 32, Her Majesty's Stationery Office. Also published by Prentice Hall, USA. Reprinted by Dover, 1994.

J. H. Wilkinson (1977), The use of the single-precision residual in the solution of linear systems. Unpublished manuscript.

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

I. Yamazaki, S. Tomov and J. Dongarra (2016), Stability and performance of various singular value QR implementations on multicore CPU with a GPU, *ACM Trans. Math. Software* **43**, 10.

I. Yamazaki, S. Tomov, T. Dong and J. Dongarra (2015*b*), Mixed-precision orthogonalization scheme and adaptive step size for improving the stability and performance of CA-GMRES on GPUs, in *High Performance Computing for Computational Science (VECPAR 2014)* (M. Daydé *et al.*, eds), Vol. 8969 of Lecture Notes in Computer Science, Springer, pp. 17–30.

I. Yamazaki, S. Tomov, J. Kurzak, J. Dongarra and J. Barlow (2015*c*), Mixed-precision block Gram Schmidt orthogonalization, in *Proceedings of the 6th Workshop on Latest Advances in Scalable Algorithms for Large-Scale Systems (ScalA '15)*, ACM Press.

K. Yang, Y.-F. Chen, G. Roumpos, C. Colby and J. Anderson (2019), High performance Monte Carlo simulation of Ising model on TPU clusters, in *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis (SC '19)*, ACM Press.

L. M. Yang, A. Fox and G. Sanders (2021), Rounding error analysis of mixed precision block Householder QR algorithms, *SIAM J. Sci. Comput.* **43**, A1723–A1753.

S. Zhang, E. Baharlouei and P. Wu (2020), High accuracy matrix computations on neural engines: A study of QR factorization and its applications, in *Proceedings of the 29th International Symposium on High-Performance Parallel and Distributed Computing*, ACM Press.

https://doi.org/10.1017/S0962492922000022 Published online by Cambridge University Press

Y.-K. Zhu and W. B. Hayes (2009), Correct rounding and a hybrid approach to exact floating-point summation, *SIAM J. Sci. Comput.* **31**, 2981–3001.

Z. Zlatev (1982), Use of iterative refinement in the solution of sparse linear systems, *SIAM J. Numer. Anal.* **19**, 381–399.

M. Zounon, N. J. Higham, C. Lucas and F. Tisseur (2022), Performance impact of precision reduction in sparse linear systems solvers, *PeerJ Comput. Sci.* **8**, e778.

https://doi.org/10.1017/S0962492922000022 Published online by Cambridge University Press