Mixed Precision LU Factorization on GPU Tensor Cores: Reducing Data Movement and Memory Footprint

Florent Lopez, Théo Mary

Journal Title
XX(X):1–27

DOI: 10.1177/ToBeAssigned
www.sagepub.com/
SAGE

Mixed Precision LU Factorization on GPU Tensor Cores: Reducing Data Movement and Memory Footprint

Florent Lopez[1] and Theo Mary[2]

Abstract

Modern GPUs equipped with mixed precision tensor core units present great potential to accelerate dense linear algebra operations such as LU factorization. However, state-of-the-art mixed half/single precision LU factorization algorithms all require the matrix to be stored in single precision, leading to expensive data movement and storage costs. This is explained by the fact that simply switching the storage precision from single to half leads to significant loss of accuracy, forfeiting all accuracy benefits from using tensor core technology. In this article, we propose a new factorization algorithm that is able to store the matrix in half precision without incurring any significant loss of accuracy. Our approach is based on a left-looking scheme employing single precision buffers of controlled size and a mixed precision doubly partitioned algorithm exploiting tensor cores in the panel factorizations. Our numerical results show that compared with the state of the art, the proposed approach is of similar accuracy but with only half the data movement and memory footprint, and hence potentially much faster: it achieves up to $2\times$ and $3.5\times$ speedups on V100 and A100 GPUs, respectively.

Keywords

numerical linear algebra, mixed precision algorithms, high performance computing, LU factorization, tensor cores, NVIDIA GPU, rounding error analysis

[1]Livermore Software Technology, an ANSYS Company, Livermore, USA
[2]Sorbonne Université, CNRS, LIP6, Paris, France
Email: florent.lopez@ansys.com, theo.mary@lip6.fr

Prepared using sagej.cls *[Version: 2017/01/17 v1.20]*

1 Introduction

Until recently, the majority of scientific codes used to carry out floating-point computations in either IEEE fp64 or fp32 arithmetic, commonly known as double and single precisions. The emergence in hardware of lower precisions, such as the half precision fp16 and bfloat16 formats, creates new opportunities as these lower precisions provide significant performance benefits, such as higher FLOPS rates and reduced data movement costs. This has generated a growing interest in mixed precision algorithms, which combine different precisions to achieve both high performance and high accuracy. Indeed, the use of mixed precision arithmetic is becoming increasingly common in scientific computing, both in algorithms and in hardware, with the emergence of specialized units that exploit multiple precisions internally. Examples of such units include the NVIDIA GPUs equipped with tensor cores, the Google TPUs, and the ARMv8.6-A and Intel Cooper Lake CPUs.

This article focuses on the direct solution of dense linear systems $Ax = b$ by LU factorization, a key computational task at the heart of numerical linear algebra. The high potential of mixed precision arithmetic for solving linear systems is well established. For example, with the most recent variants of iterative refinement proposed by Carson and Higham [1], [2], an fp16 LU factorization of A may be enough to obtain a solution x at fp64 accuracy, even for relatively ill-conditioned matrices. Another approach to exploit multiple precisions in the solution of $Ax = b$ is to use a mixed precision algorithm for the LU factorization itself. This is most natural when targeting hardware equipped with mixed precision computing units, such as those previously mentioned. In particular, Haidar et al. [3], [4] and Blanchard et al. [5] have investigated mixed precision LU factorization algorithms exploiting GPU tensor cores that employ fp16 and fp32 precisions. Moreover, a mixed precision LU factorization can be combined with iterative refinement to accelerate the convergence of the latter [3].

In this article, we propose new mixed precision LU factorization algorithms that extend and improve previous algorithms in several ways, in terms of speed, accuracy, data movement, and memory footprint. Indeed, previously proposed algorithms require the matrix A to be entirely stored in fp32, thereby forfeiting any potential gains in memory consumption associated with the use of the lower fp16 precision, and requiring expensive data movements in fp32. We propose, to our knowledge, the first mixed precision algorithm based on GPU tensor cores that is able to accurately factorize a matrix stored in fp16. We explain why this is not immediate: simply switching the storage precision from fp32 to fp16 leads to a significant loss of accuracy. Our algorithm overcomes this issue with a left-looking update scheme that makes use of temporary fp32 buffers of modest and controlled size. We also investigate in what precision the panel factorizations should be carried out; we show that the approach achieving the best performance–accuracy tradeoff is to use two levels of partitioning so as to exploit tensor cores both in the updates and in the outer panel factorizations.

Throughout the article, we explore several variants of mixed precision LU factorization, and investigate their performance and accuracy, both via rounding error analysis (generalizing that of [5]) and via numerical experiments on random dense matrices using NVIDIA V100 GPUs. Overall, the best variant of our new algorithm achieves a better performance–accuracy tradeoff than previous state-of-the-art algorithms by achieving similar accuracy, half the memory footprint, and up to 3.5× higher performance thanks to reduced data movement. We have made our code publicly available*.

The rest of this article is organized as follows. We begin by providing technical background and by describing our experimental setting in section 2. We also review previous work and explain how we aim to improve it in section 3. Our core algorithmic contributions are found in sections 4 and 5. In section 4 we describe a left-looking mixed precision algorithm that factorizes a matrix stored in fp16 without losing all the accuracy benefits of tensor cores. In section 5 we investigate the choice of precision for the panel factorization. We provide additional experiments on a range of matrices coming from various applications in section 6. We also assess the performance of our algorithms on the latest A100 GPUs in section 7. Our conclusions are reported in section 8.

Throughout the article, we will denote by fl_{16} and fl_{32} the operations of rounding to the fp16 and fp32 formats, and by $u_{16} = 2^{-11}$ and $u_{32} = 2^{-24}$ their respective unit roundoffs. We also define $\gamma_n = nu/(1-nu)$ and use the superscript of γ to indicate that the unit roundoff u has the corresponding subscript. Hence,

$$\gamma^{(32)} = nu_{32}/(1-nu_{32}), \quad \gamma^{(16)} = nu_{16}/(1-nu_{16}), \quad \widetilde{\gamma} = n\widetilde{u}/(1-n\widetilde{u}).$$

2 Technical background and experimental setting

We provide some technical background on partitioned LU factorization and GPU tensor cores, and describe our experimental setting used throughout the article.

2.1 Partitioned LU factorization

High performance LU factorization algorithms are based on partitioning the matrix in $r \times r$ blocks so as to recast most of the operations as matrix–matrix (BLAS-3) operations. Given such a partitioning, the LU factorization amounts to performing a sequence of panel factorizations and updates. At each step k, the panel factorization computes the part of the LU factors associated with the current panel, while the update overwrites the trailing submatrix via matrix–matrix products.

We can distinguish two types of algorithms, right- and left-looking versions, depending on the order in which the operations are performed. In a right-looking factorization (Algorithm 2.1), at step k, the entire trailing submatrix (to the *right*)

*https://github.com/flipflapflop/remifa/

is updated with respect to the current panel *after* factorizing it. In a left-looking factorization (Algorithm 2.2), at step k, the current panel is updated with respect to the already computed LU factors (to the *left*) *before* being factorized.

Algorithm 2.1 Standard (uniform precision) LU factorization (right-looking version).

Input: a matrix $A \in \mathbb{R}^{n\times n}$ in precision u, partitioned into $r \times r$ blocks A_{ij}, where $q = n/r$ is assumed to be an integer.
Output: the LU factors of A in precision u (with L and U partitioned into $r \times r$ blocks).
for $k = 1\colon q$ **do**
    Factorize $L_{kk}U_{kk} = A_{kk}$.
    for $i = k+1\colon q$ **do**
        Solve $L_{ik}U_{kk} = A_{ik}$ for L_{ik}.
        Solve $L_{kk}U_{ki} = A_{ki}$ for U_{ki}.
    end for
    for $i = k+1\colon q$ **do**
        for $j = k+1\colon q$ **do**
            Update $A_{ij} \leftarrow A_{ij} - L_{ik}U_{kj}$.
        end for
    end for
end for

To achieve maximal performance, the matrix can be recursively partitioned so as to also exploit BLAS-3 operations within the panel factorization [6]. For example, LAPACK's dgetrf [7] uses two levels of partitioning. Algorithm 5.2, proposed in section 5 is based on LAPACK's algorithm and also uses two levels.

2.2 GPU tensor cores

The tensor cores in the NVIDIA Volta and Turing architectures are a special type of unit that carry out the operation $D = C + AB$, where all matrices are 4×4 [8]. The tensor cores are inherently mixed precision units: while the matrices A and B must be stored in fp16, C and D can be in fp16 or fp32. Pictorially, we have

$$\underbrace{\begin{bmatrix} \times & \times & \times & \times \\ \times & \times & \times & \times \\ \times & \times & \times & \times \\ \times & \times & \times & \times \end{bmatrix}}_{\text{fp16 or fp32}} = \underbrace{\begin{bmatrix} \times & \times & \times & \times \\ \times & \times & \times & \times \\ \times & \times & \times & \times \\ \times & \times & \times & \times \end{bmatrix}}_{\text{fp16 or fp32}} + \underbrace{\begin{bmatrix} \times & \times & \times & \times \\ \times & \times & \times & \times \\ \times & \times & \times & \times \\ \times & \times & \times & \times \end{bmatrix}}_{\text{fp16}} \underbrace{\begin{bmatrix} \times & \times & \times & \times \\ \times & \times & \times & \times \\ \times & \times & \times & \times \\ \times & \times & \times & \times \end{bmatrix}}_{\text{fp16}}$$

Tensor cores are an instance of what has been called a $t \times t$ block fused multiply-add (with $t = 4$) by Blanchard et al. [5]. In the context of both matrix

Algorithm 2.2 Standard (uniform precision) LU factorization (left-looking version).

Input: a matrix $A \in \mathbb{R}^{n\times n}$ in precision u, partitioned into $r \times r$ blocks A_{ij}, where $q = n/r$ is assumed to be an integer.
Output: the LU factors of A in precision u (with L and U partitioned into $r \times r$ blocks).
for $k = 1\colon q$ **do**
  for $i = k\colon q$ **do**
    for $j = 1\colon k-1$ **do**
      Update $A_{ik} \leftarrow A_{ik} - L_{ij}U_{jk}$.
      if $i \neq k$ **then** Update $A_{ki} \leftarrow A_{ki} - L_{kj}U_{ji}$.
    end for
  end for
  Factorize $L_{kk}U_{kk} = A_{kk}$.
  for $i = k+1\colon q$ **do**
    Solve $L_{ik}U_{kk} = A_{ik}$ for L_{ik}.
    Solve $L_{kk}U_{ki} = A_{ki}$ for U_{ki}.
  end for
end for

multiplication and LU factorization, Blanchard et al. [5] show that storing C and D in fp32 (the so-called TC32 variant) yields much more accurate results than storing them in fp16 (the so-called TC16 variant). This is because with C and D in fp32, rounding error accumulation only affects the error term proportional to the fp32 unit roundoff u_{32}. For example, consider the matrix product $C = AB$ where $A \in \mathbb{R}^{m\times n}$ and $B \in \mathbb{R}^{n\times p}$. Blanchard et al. [5, Alg. 3.1] propose a blocked matrix–matrix product algorithm that can compute C using tensor cores, and [5, Thm. 3.2] shows that the computed $\widehat{C}$ satisfies:

$$|\widehat{C} - C| \lesssim c_n|A||B|, \quad c_n = \begin{cases} nu_{16} & \text{(TC16)}, \\ 2u_{16} + nu_{32} & \text{(TC32)}. \end{cases} \tag{2.1}$$

Furthermore, Blanchard et al. [5] also report that the TC16 and TC32 variants achieve similar performance.

2.3 Experimental setting

We use a range of different test matrices in our experiments. In sections 3 through 5, we use the same test matrices as the HPL-AI benchmark[†]. These are $n \times n$ dense matrices with off-diagonal entries randomly sampled from the

[†] https://icl.bitbucket.io/hpl-ai/

uniform $[0, 1]$ distribution and with diagonal entries set to n to make them diagonally dominant. Then, in section 6, we provide additional experiments with other matrices, including real-life ones from the SuiteSparse collection [9].

Our code implements the various LU factorization algorithms presented in this article for the NVIDIA GPU architecture. Following the guidelines of the HPL-AI benchmark, we do not perform any numerical pivoting. We however note that all the algorithms presented in this article are compatible with numerical pivoting. In all our experiments except those of section 7, we use a single V100 GPU unit equipped with 16 GB of memory and with a peak performance of 7.8 and 15.7 TFLOPS using double (fp64) and single (fp32) precision arithmetic. With the tensor cores, the peak performance increases to 125 TFLOPS. We use the cuBLAS v10.1 library for the matrix–matrix product operations in the updates, and hand-coded CUDA kernel for the panel factorizations.

We assess the stability of the different LU factorization algorithms by using them to solve a linear system $Ax = b$ by the substitutions $\widehat{L}y = b$ and $\widehat{U}x = y$, where $\widehat{L}$ and $\widehat{U}$ are the computed LU factors of A. The substitutions are performed in fp32. We generate the solution x as the vector of ones $[1 \dots 1]^T$, and thus the right-hand side is computed as $b = Ax$. Denoting the computed solution as $\widehat{x}$, we measure the componentwise backward error

$$\begin{aligned} \varepsilon_{\text{bwd}} &= \min\left\{\varepsilon > 0 \ : \ (A + \Delta A)\widehat{x} = b, \quad |\Delta A| \leq \varepsilon(|A| + |\widehat{L}||\widehat{U}|)\right\} \\ &= \max_i \frac{|A\widehat{x} - b|_i}{((|A| + |\widehat{L}||\widehat{U}|)|\widehat{x}|)_i}, \end{aligned} \tag{2.2}$$

where (2.2) follows from the Oettli–Prager theorem [10, Thm. 7.3], [11].

3 Contributions with respect to previous work

To our knowledge, the first LU factorization algorithm exploiting GPU tensor cores is that of Haidar et al. [3], that is shown to accelerate the solution of dense linear systems in the context of iterative refinement [2]. Their algorithm is a right-looking factorization where the tensor cores are exploited to accelerate the update operations. To do so, it suffices to convert to fp16 the LU factors computed by the panel factorization on the fly. This is described in Algorithm 3.1, where the highlighted lines correspond to the changes that need to be made to the standard LU factorization (Algorithm 2.1).

Blanchard et al. [5] give the rounding error analysis of Algorithm 3.1 (which is the equivalent of their Algorithm 4.1), and show that it can be significantly more accurate than a standard LU factorization in fp16 arithmetic (Algorithm 2.1 with $u = u_{16}$). Note that Blanchard et al. [5] assume the panel size r to match the dimension t of the block FMA (that is, $r = t = 4$ in the case of GPU tensor cores), but the extension to an arbitrary size r is straightforward.

Both previously cited works, as well as subsequent works following them [4], share the common weakness that the matrix A is assumed to be stored in fp32

Algorithm 3.1 State-of-the-art mixed precision LU factorization from [3, 5].

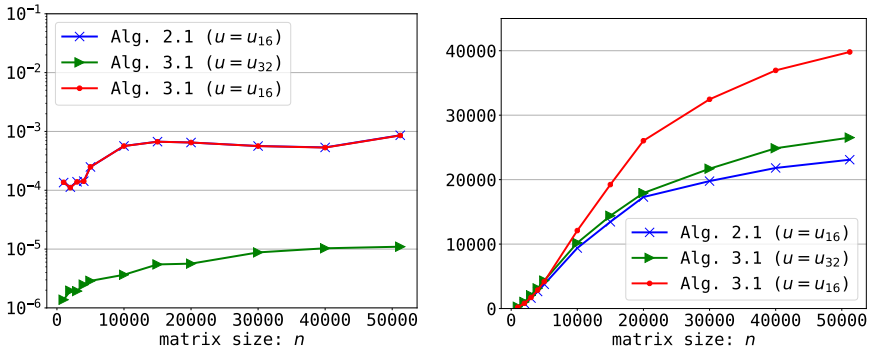
Input: a matrix $A \in \mathbb{R}^{n\times n}$ in precision $u = u_{32}$ or $u = u_{16}$, partitioned into $r \times r$ blocks A_{ij}, where $q = n/r$ is assumed to be an integer.
Output: the LU factors of A in precision u (with L and U partitioned into $r \times r$ blocks).
for $k = 1\colon q$ **do**
  Factorize $L_{kk}U_{kk} = A_{kk}$ (in precision u).
  for $i = k+1\colon q$ **do**
    Solve $L_{ik}U_{kk} = A_{ik}$ for L_{ik} (in precision u).
    Solve $L_{kk}U_{ki} = A_{ki}$ for U_{ki} (in precision u).
  end for
  for $i = k+1\colon q$ **do**
    for $j = k+1\colon q$ **do**
      Convert to fp16: $\widetilde{L}_{ik} \leftarrow \mathrm{fl}_{16}(L_{ik})$.
      Convert to fp16: $\widetilde{U}_{ki} \leftarrow \mathrm{fl}_{16}(U_{ki})$.
      Update $A_{ij} \leftarrow A_{ij} - \widetilde{L}_{ik}\widetilde{U}_{kj}$ using tensor cores.
    end for
  end for
end for

(that is, Algorithm 3.1 uses $u = u_{32}$). Therefore, the use of half precision in current state-of-the-art mixed precision LU algorithms does not reduce neither the memory consumption nor the volume of data movement. This represents a significant shortcoming since memory consumption can be the most critical resource in some applications, especially on GPUs, which have very limited memory. Moreover, since the FLOPS rate on GPU tensor cores is very high, data movements can be very expensive and can significantly hinder the overall performance of the factorization.

Before describing the contributions of this article, let us first explain why it is so challenging to store the matrix in fp16. To do so, we recall the main result of the error analysis of [5], which analyzes a version of Algorithm 3.1 with general $t \times t$ block FMA units in precisions u_{low} and u_{high}. We specialize that result to the case of GPU tensor cores where matrices C and D in the block FMA are stored in fp32 (TC32 variant).

Theorem 1. Specialization of [5, Thm. 4.3] to tensor cores. *Let $A \in \mathbb{R}^{n\times n}$ be partitioned in $r \times r$ blocks. If Algorithm 3.1 runs to completion then the computed LU factors $\widehat{L}$ and $\widehat{U}$ satisfy $A + \Delta A = \widehat{L}\widehat{U}$, where*

$$|\Delta A| \le f(n, r, u_{16}, u_{32}, u)\big(|A| + |\widehat{L}||\widehat{U}|\big) + O(u_{32}u_{16}), \tag{3.1}$$



(a) Componentwise backward error (2.2). **(b)** Performance (GFLOPS).

Figure 1. Accuracy and performance of Algorithm 3.1 with $u = u_{16}$ or $u = u_{32}$ for diagonally dominant matrices. The panel size is $r = 256$.

with

$$f(n, r, u_{16}, u_{32}, u) = 2u_{16} + u_{16}^2 + \max(\gamma_r, \gamma_{n-r+1}^{(32)} + \widetilde{\gamma}_{(n-r)/4})(1 + u_{16})^2$$

and with $\widetilde{u} = u$ *if* $u > u_{32}$ *and* $\widetilde{u} = 0$ *otherwise.*

Note how Algorithm 3.1 and Theorem 1 allow the matrix to be stored in a precision u that can be either u_{32} or u_{16}. In their experiments, Blanchard et al. [5] however only investigate the case $u = u_{32}$, for which bound (3.1) becomes

$$|\Delta A| \lesssim (2u_{16} + nu_{32})(|A| + |\widehat{L}||\widehat{U}|). \tag{3.2}$$

Crucially, the term proportional to u_{16} in (3.2) does not grow with n. Storing the matrix in fp16 gives a very different result. Indeed, setting $u = u_{16}$ in (3.1) yields

$$|\Delta A| \lesssim 0.25nu_{16}(|A| + |\widehat{L}||\widehat{U}|). \tag{3.3}$$

The term proportional to u_{16} now grows linearly with n. Hence, bound (3.3) is roughly $\min(n/8, u_{16}/4u_{32}) = \min(n/8, 2048)$ larger than bound (3.2). Hence, for large n, bound (3.3) is about three orders of magnitude larger than bound (3.2).

This increase is confirmed experimentally. Figure 1a shows that switching from $u = u_{32}$ to $u = u_{16}$ leads indeed to an error increase of about two orders of magnitude. This is less than could be expected from the theoretical bounds, but still represents a significant loss of accuracy. Moreover, for reference, we also compare Algorithm 3.1 with a standard LU factorization in fp16 arithmetic (Algorithm 2.1 with $u = u_{16}$). Figure 1a shows that, with $u = u_{16}$, both Algorithms 2.1 and 3.1 achieve the same error: any accuracy gain associated with the use of tensor cores is therefore lost by storing the matrix in fp16.

This behavior is explained by the term $\widetilde{\gamma}_{(n-r)/4}$ in Theorem 1, which accounts for the need to round the output D of the block FMA to the working precision u after each call. With the TC32 variant, the output D is in fp32 and therefore, if $u = u_{32}$ no extra rounding is necessary and this term is equal to zero. However, with $u = u_{16}$, it becomes equal to $nu_{16}/4 + o(nu_{16})$ due to the need to round to fp16 every $t = 4$ additions.

In this article, we introduce new algorithms that overcome this issue and are able to use an fp16 matrix (therefore halving the memory footprint and, as we will show, the volume of data movement) with no significant loss of accuracy. We begin, in the next section, by introducing a left-looking factorization with fp32 buffers.

4 A left-looking factorization using fp16 LU factors with fp32 buffers

As explained in the previous section, using Algorithm 3.1 with the matrix stored in fp16 ($u = u_{16}$) leads to a significant loss of accuracy due to the need to repeatedly round to fp16 every $t = 4$ additions. To be precise, the update operation (line 13)

$$A_{ij} \leftarrow A_{ij} - \widetilde{L}_{ik}\widetilde{U}_{kj} \tag{4.1}$$

takes the form of several block FMAs $D \leftarrow C - AB$ chained one after the other. Here, A and B are 4×4 subblocks of $\widetilde{L}_{ik}$ are $\widetilde{U}_{kj}$, stored in fp16, and C and D are 4×4 subblocks of A_{ij}, stored in precision u. In the case $u = u_{16}$, the block FMA is therefore being used with fp16 output (the so-called TC16 variant by Blanchard et al. [5]), which has indeed been shown to be much less accurate than with an fp32 output (TC32 variant), as explained in section 2.2.

It is important to realize that this issue is therefore not intrinsic to the tensor cores themselves, which are naturally able to handle matrix–matrix products of large dimensions without rounding to fp16 every 4 additions (by using the TC32 variant). The core issue lies instead with Algorithm 3.1, which forces the use of the much less accurate TC16 variant.

In this section, we present a reformulation of the algorithm that is able to exploit the TC32 variant instead of the TC16 one, while still storing the matrix in fp16. The key idea is to introduce temporary fp32 buffers that we use to chain several block FMAs by accumulating their output in fp32. That is, we replace the updates

$$A_{ij} \leftarrow A_{ij} - \widetilde{L}_{ik}\widetilde{U}_{kj}, \quad \text{for } k = 1\colon \min(i,j) - 1, \tag{4.2}$$

by

$$B_{ij} = \mathrm{fl}_{32}(A_{ij}), \tag{4.3a}$$

$$B_{ij} \leftarrow B_{ij} - \widetilde{L}_{ik}\widetilde{U}_{kj}, \quad \text{for } k = 1\colon \min(i,j) - 1, \tag{4.3b}$$

$$A_{ij} \leftarrow \mathrm{fl}_{16}(B_{ij}). \tag{4.3c}$$

Whereas (4.2) forces the use of the TC16 variant, (4.3) allows for the use of the TC32 variant.

However, simply replacing (4.2) by (4.3) is not satisfactory: while this is likely to recover a good accuracy, it requires an fp32 buffer of the same dimensions as the trailing submatrix, which contains $(n-kr)^2$ entries at step k. Therefore, at the early steps, this buffer consists of almost the entire matrix, and so this approach does not achieve a significant reduction of the memory footprint or the volume of data movement compared with Algorithm 3.1 with $u = u_{32}$ (which requires n^2 fp32 entries).

The key to overcome this issue is to switch from a right-looking algorithm to a left-looking one. Indeed, as explained in section 2.1, in a left-looking factorization, all the updates associated with a given block A_{ij} are performed together one after the other. Therefore, we only need the fp32 buffer B_{ij} for a short period of time, and so we can reuse the same buffer to update different blocks. We present in Algorithm 4.1 a new mixed precision left-looking LU factorization algorithm, where the highlighted lines correspond to the changes that need to be made to the standard left-looking LU factorization (Algorithm 2.2).

Algorithm 4.1 A new left-looking mixed precision LU factorization.

Input: a matrix $A \in \mathbb{R}^{n\times n}$ in precision u_{16}, partitioned into $r \times r$ blocks A_{ij}, where $q = n/r$ is assumed to be an integer.
Output: the LU factors of A in precision u_{16} (with L and U partitioned into $r \times r$ blocks).
for $k = 1 : q$ **do**
  for $i = k : q$ **do**
    Convert to fp32: $B_{ik} = \mathrm{fl}_{32}(A_{ik})$.
    if $i \neq k$ **then** Convert to fp32: $B_{ki} \leftarrow \mathrm{fl}_{32}(A_{ki})$.
    for $j = 1 : k-1$ **do**
      Update $B_{ik} \leftarrow B_{ik} - L_{ij}U_{jk}$ using tensor cores.
      if $i \neq k$ **then** Update $B_{ki} \leftarrow B_{ki} - L_{kj}U_{ji}$ using tensor cores.
    end for
    Convert back to fp16: $A_{ik} \leftarrow \mathrm{fl}_{16}(B_{ik})$.
    if $i \neq k$ **then** Convert back to fp16: $A_{ki} \leftarrow \mathrm{fl}_{16}(B_{ki})$.
  end for
  Factorize $L_{kk}U_{kk} = A_{kk}$ (in precision u_{16}).
  for $i = k+1 : q$ **do**
    Solve $L_{ik}U_{kk} = A_{ik}$ for L_{ik} (in precision u_{16}).
    Solve $L_{kk}U_{ki} = A_{ki}$ for U_{ki} (in precision u_{16}).
  end for
end for

There is some freedom in the choice of the fp32 buffer size. The absolute minimum buffer size we need is only of $t^2 = 16$ entries, because we technically could update each $r \times r$ block A_{ij} one 4×4 subblock at a time. However, it is desirable to update multiple $r \times r$ blocks at the same time to achieve increased parallelism and performance. In our GPU implementation of Algorithm 4.1, we use a buffer of the same dimension as either the L or U part of the current panel at step k (that is, a buffer formed of either all blocks A_{ik} for $i = k\colon q$ or all blocks A_{ki} for $i = k+1\colon q$). The algorithm therefore requires an fp32 buffer of at most nr entries.

We note that, at step k of Algorithm 4.1, we update both the L part of the panel (A_{ik} for $i = k\colon q$) and the U part (A_{ki} for $i = k+1\colon q$). This is sometimes referred to as a Crout factorization, as opposed to a "pure" left-looking factorization which only updates the L part and delays the update of A_{ki} to step i. In our context, there is no significant difference between a Crout or a pure left-looking factorization, neither in terms of accuracy, memory footprint, or volume of data movement.

Algorithm 4.1 possesses two key properties. First, we show in section 4.1 that, compared with Algorithm 3.1 with $u = u_{32}$, Algorithm 4.1 not only halves the memory footprint, but also the volume of data movement: as a result, we can expect it to be up to twice faster. Second, we prove in section 4.2 that, compared with Algorithm 3.1 with $u = u_{16}$, Algorithm 4.1 achieves a significantly smaller error bound. We show experimentally in section 4.3 that these two key properties allow for a significantly better performance–accuracy tradeoff.

4.1 Data movement analysis of Algorithm 4.1

Here we carry out a data movement analysis to show that Algorithm 4.1 reduces the volume of data movement compared with Algorithm 3.1 with $u = u_{32}$. We use a simplified model of memory architecture with two levels: a limited, fast memory and an unlimited, slower memory. We assume that the fast memory is large enough to accomodate an $r \times r$ block of the matrix and its associated buffer B_{ik} and LU factors. Hence, for example, we assume that the operation $B_{ik} \leftarrow B_{ik} - L_{ij}U_{jk}$ can directly be performed within the fast memory after all the required data has been loaded.

Under this model, at each step k, line 13 of Algorithm 3.1 requires to load all the blocks A_{ij} of the trailing submatrix, of which there are $(q-k)^2$. With $u = u_{32}$, these blocks are stored in fp32 and so the volume, in bytes, of data moved is

$$\sum_{k=1}^{q} (q-k)^2 r^2 \times 4 \text{ bytes} = \frac{4}{3} q n^2 + O(n^2) \text{ bytes}. \tag{4.4}$$

The other steps of the algorithm require data movement at most in $O(n^2)$ bytes.

In comparison, at step k, lines 8 of Algorithm 4.1 requires to load all L_{ij}, U_{jk} for $i \geq k$ and $j < k$, which amounts to $(q-k)(k-1)$ blocks at step k. Similarly,

line 9 also requires the same volume of data movement. They key difference with the previous case is that these entries are now stored in fp16, and so require 2 bytes per entry instead of 4. Summing over k, the volume of data moved by the update steps of Algorithm 4.1 is

$$2\sum_{k=1}^{q}(q-k)(k-1)\times 2 \text{ bytes} = \frac{2}{3}qn^2 + O(n^2) \text{ bytes}. \tag{4.5}$$

Algorithm 4.1 does require some fp32 data movement, but in much lower volume: at step k, only one panel (all blocks B_{ik} and B_{ki} for $i \geq k$) needs to be loaded, which amounts to $O(n^2)$ bytes moved. For large matrices, $q = n/r$ is also large, and so Algorithm 4.1 requires about half the data movement of Algorithm 3.1 with $u = u_{32}$.

4.2 Rounding error analysis of Algorithm 4.1

Since Algorithm 4.1 carries out all updates by chaining block FMAs with fp32 output, we can expect it to be significantly more accurate than Algorithm 3.1 with $u = u_{16}$. To formally prove this result, we now perform the rounding error analysis of Algorithm 4.1.

Lemma 1. *Let $B = A - \sum_{j=1}^{q} X_j Y_j$, where $A, B \in \mathbb{R}^{r\times r}$ are given in precision u_{32} and $X_j, Y_j \in \mathbb{R}^{r\times r}$ are given in precision u_{16}, be computed with GPU tensor cores using the TC32 variant. The computed $\widehat{B}$ satisfies*

$$|\widehat{B} - B| \leq \gamma_{n+1}^{(32)}\left(|A| + \sum_{j=1}^{q}|X_j||Y_j|\right). \tag{4.6}$$

Proof. This is a both a specialization and an extension of [5, Corollary 4.1]. The result is specialized to tensor cores with the TC32 variant, for which $\overline{\gamma} = \gamma^{(32)}$ and $\widetilde{\gamma}^{\mathrm{FMA}} = 0$. It is also trivially extended to general block size r possibly different from $t = 4$.

Theorem 2. Backward error bound for Algorithm 4.1. *Let $A \in \mathbb{R}^{n\times n}$ be partitioned in $r \times r$ blocks. If Algorithm 4.1 runs to completion, the computed LU factors $\widehat{L}$ and $\widehat{U}$ satisfy $A + \Delta A = \widehat{L}\widehat{U}$, where*

$$|\Delta A| \leq u_{16}|A| + f(n, r, u_{16}, u_{32})\big((1+u_{16})|A| + |\widehat{L}||\widehat{U}|\big), \tag{4.7}$$

with

$$f(n, r, u_{16}, u_{32}) = \max(u_{16} + \gamma_{n-r+1}^{(32)} + u_{16}\gamma_{n-r+1}^{(32)}, \gamma_r^{(16)}).$$

Proof. Algorithm 4.1 requires the input matrix A to be in precision u_{16} so we must first account for this conversion: let $\widetilde{A} = \mathrm{fl}_{16}(A) = A + E$, with $|E| \leq u_{16}|A|$. We now apply the algorithm to $\widetilde{A}$. The (i, k) block of the L factor is computed

on line 16 by solving

$$L_{ik}\widehat{U}_{kk} = R_{ik}, \quad R_{ik} = B_{ik} - \sum_{j=1}^{k-1} \widehat{L}_{ij}\widehat{U}_{jk}. \quad i > k,$$

where $B_{ik} = \mathrm{fl}_{32}(\widetilde{A}_{ik}) = \widetilde{A}_{ik}$. By Lemma 1, the computed $\widehat{R}_{ik}$ satisfies, since $k \le q$,

$$|R_{ik} - \widehat{R}_{ik}| \le \gamma_{n-r+1}^{(32)} \Bigl(|\widetilde{A}_{ik}| + \sum_{j=1}^{k-1} |\widehat{L}_{ij}||\widehat{U}_{jk}| \Bigr). \tag{4.8}$$

We then convert $\widehat{R}_{ik}$ back to fp16 (lines 11 and 12), obtaining $\widetilde{R}_{ik} = \mathrm{fl}_{16}(\widehat{R}_{ik}) = \widehat{R}_{ik} + F_{ik}$, with

$$|F_{ik}| \le u_{16}|\widehat{R}_{ik}| \le u_{16}(1 + \gamma_{n-r+1}^{(32)}) \Bigl(|\widetilde{A}_{ik}| + \sum_{j=1}^{k-1} |\widehat{L}_{ij}||\widehat{U}_{jk}| \Bigr). \tag{4.9}$$

By [10, Thm. 8.5] we have

$$|\widehat{L}_{ik}\widehat{U}_{kk} - \widetilde{R}_{ik}| \le \gamma_r^{(16)} |\widehat{L}_{ik}||\widehat{U}_{kk}|. \tag{4.10}$$

Combining the three inequalities (4.8)–(4.10), we conclude that for $i > k$,

$$\begin{aligned} \Bigl| \widetilde{A}_{ik} - \sum_{j=1}^{k} \widehat{L}_{ij}\widehat{U}_{jk} \Bigr| &\le |R_{ik} - \widehat{R}_{ik}| + |F_{ik}| + |\widetilde{R}_{ik} - \widehat{L}_{ik}\widehat{U}_{kk}|, \\ &\le f(n, r, u_{16}, u_{32}) \Bigl(|\widetilde{A}_{ik}| + \sum_{j=1}^{k} |\widehat{L}_{ij}||\widehat{U}_{jk}| \Bigr), \end{aligned} \tag{4.11}$$

where $f(n, r, u_{16}, u_{32}) = \max(u_{16} + \gamma_{n-r+1}^{(32)} + u_{16}\gamma_{n-r+1}^{(32)}, \gamma_r^{(16)})$. For $i = k$, L_{kk} is determined with U_{kk} on line 14 of Algorithm 4.1, and by [10, Thm. 9.3] we have $|\widehat{L}_{kk}\widehat{U}_{kk} - \widehat{R}_{kk}| \le \gamma_r^{(16)} |\widehat{L}_{kk}||\widehat{U}_{kk}|$. Therefore (4.10) holds for $i = k$, too, and hence so does (4.11). In a similar way, the inequality (4.11) can be shown to hold for $i < k$. We have thus proved $\widetilde{A} + \Delta\widetilde{A} = \widehat{L}\widehat{U}$ with

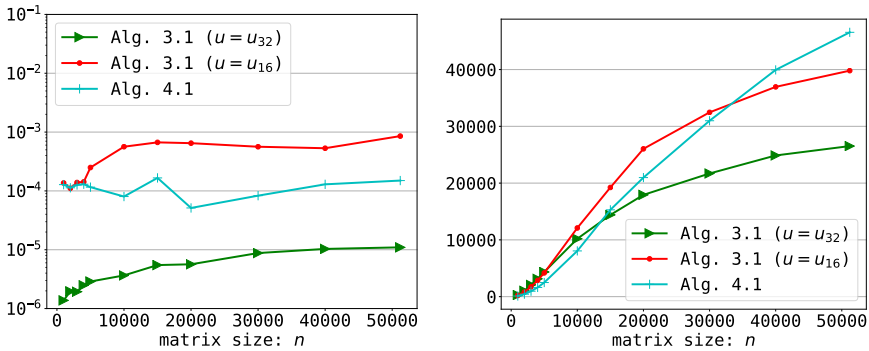
$$|\Delta\widetilde{A}| \le f(n, r, u_{16}, u_{32}) \bigl(|\widetilde{A}| + |\widehat{L}||\widehat{U}| \bigr).$$

Replacing $\widetilde{A}$ by $A + E$ yields the bound on $|\Delta A|$ and concludes the proof.

Theorem 2 proves that Algorithm 4.1 is indeed potentially much more accurate than Algorithm 3.1 with $u = u_{16}$. We summarize the dominant term in the error bounds of each algorithm in Table 1.

Table 1. Dominant term in the expression of f in the error bound $A + \Delta A = \widehat{L}\widehat{U}$, $|\Delta A| \leq f(n, r, u_{16}, u_{32})(|A| + |\widehat{L}||\widehat{U}|)$ and number of fp32 entries that need to be stored for various algorithms.

	fp32 storage (# of entries)	$f(n, r, u_{16}, u_{32})$
Alg. 3.1 ($u = u_{32}$)	n^2	$2u_{16} + nu_{32}$
Alg. 3.1 ($u = u_{16}$)	0	$nu_{16}/4$
Alg. 4.1	nr	$u_{16} + \max(nu_{32}, ru_{16})$



(a) Componentwise backward error (2.2). **(b)** Performance (GFLOPS).

Figure 2. Accuracy and performance of Algorithms 3.1 and 4.1 for diagonally dominant matrices. The panel size is $r = 256$.

4.3 Numerical experiments with Algorithm 4.1

Figure 2a shows that our new Algorithm 4.1 is indeed more accurate than Algorithm 3.1 with $u = u_{16}$, reducing the error by about an order of magnitude for large n. Moreover, Figure 2b shows that this accuracy improvement is accompanied by a significant performance boost. Indeed Algorithm 4.1 achieves a peak performance of 46 TFLOPS, instead of 26 TFLOPS for Algorithm 3.1 with $u = u_{32}$. This improved performance is explained by two reasons. First, storing the matrix in fp16 instead of fp32 not only halves the memory footprint, but also significantly reduces data movement, which in turn improves performance. In fact, Algorithm 3.1 with $u = u_{16}$ is itself faster than with $u = u_{32}$ (achieving a peak performance of 40 TFLOPS). Second, a left-looking factorization involves matrix–matrix products with different shapes than those of a right-looking factorization, that lead to a higher FLOPS rate for large matrices.

Despite these positive results, Figure 2a also shows that Algorithm 4.1 is not able to fully recover the same accuracy as Algorithm 3.1 (which stores the matrix in fp32), with an error about an order of magnitude larger. This is explained by

the term $\max(ru_{16}, nu_{32})$ in the error bound of Algorithm 4.1. For matrix sizes n such that ru_{16} dominates over nu_{32}, the ratio between the dominant terms in the error bounds of Algorithm 4.1 and Algorithm 3.1 with $u = u_{32}$ is thus $ru_{16}/(2u_{16} + nu_{32})$, which can be as large as $r/2$. In the experiments of Figure 2, we have used a panel size $r = 256$, so that $nu_{32} \geq ru_{16}$ for $n \gtrsim 2.1 \times 10^6$. The available memory on a single GPU device limits the size n that we can test to $n \approx 5 \times 10^4$, so we are clearly in the regime where the ru_{16} term dominates.

We also note that, due to statistical effects in the rounding errors, probabilistic analyses [12], [13], [14] have shown that constants depending on the problem dimensions can usually be replaced by their square root to obtain more realistic bounds. The smallest n such that the term $\sqrt{n}u_{32}$ dominates over the term $\sqrt{r}u_{16}$ is even larger and equal to about 1.7×10^{10}, which is about three orders of magnitude larger than the largest linear system ever solved by the TOP500 ranking as of June 2022.

It is therefore clear that the term limiting the accuracy of Algorithm 4.1 for practical values of n is the term ru_{16}, which comes from the panel factorization (lines 14 to 18). In the next section we turn our attention to this step and investigate how to perform it more accurately.

5 Mixed precision algorithms for the panel factorization

We first present in section 5.1 a simple modification of Algorithm 4.1 which performs the panel factorization entirely in fp32 arithmetic (Algorithm 5.1). We show that this is enough to recover almost the same accuracy as Algorithm 3.1 with fp32 LU factors, but at the price of lesser performance. Therefore, we then propose in section 5.2 a doubly partitioned factorization algorithm that exploits GPU tensor cores in the panel factorization in order to achieve both high accuracy and high performance (Algorithm 5.2).

5.1 Mixed precision LU factorization with fp32 panel factorization

Algorithm 4.1 can be easily modified to perform the panel factorizations in fp32: it suffices to convert the panels back to fp16 after the panel factorization, instead of before. The fp32 buffer in which we accumulate the updates is thus also used for the panel factorization. The resulting algorithm is described in Algorithm 5.1, where the conversion that has been moved is highlighted.

Theorem 3. Backward error bound for Algorithm 5.1. *Let $A \in \mathbb{R}^{n\times n}$ be partitioned in $r \times r$ blocks. If Algorithm 5.1 runs to completion, the computed LU factors $\widetilde{L}$ and $\widetilde{U}$ satisfy $A + \Delta A = \widetilde{L}\widetilde{U}$, where*

$$|\Delta A| \leq u_{16}|A| + f(n, r, u_{16}, u_{32})\big((1 + u_{16})|A| + |\widetilde{L}||\widetilde{U}|\big), \tag{5.1}$$

with

$$f(n, r, u_{16}, u_{32}) = \max(\gamma^{(32)}_{n-r+1}, 2u_{16} + u_{16}^2 + \gamma^{(32)}_r(1 + u_{16})^2).$$

Algorithm 5.1 A modified version of Algorithm 4.1 that computes the panel factorization in fp32 instead of fp16.

1: **Input:** a matrix $A \in \mathbb{R}^{n\times n}$ in precision u_{16}, partitioned into $r \times r$ blocks A_{ij}, where $q = n/r$ is assumed to be an integer.
2: **Output:** the LU factors of A in precision u_{16} (with L and U partitioned into $r \times r$ blocks).
3: **for** $k = 1\colon q$ **do**
4: **for** $i = k\colon q$ **do**
5: Convert $B_{ik} \leftarrow \mathrm{fl}_{32}(A_{ik})$.
6: **if** $i \neq k$ **then** Convert $B_{ki} \leftarrow \mathrm{fl}_{32}(A_{ki})$.
7: **for** $j = 1\colon k-1$ **do**
8: Update $B_{ik} \leftarrow B_{ik} - \widetilde{L}_{ij}\widetilde{U}_{jk}$ using tensor cores.
9: **if** $i \neq k$ **then** Update $B_{ki} \leftarrow B_{ki} - \widetilde{L}_{kj}\widetilde{U}_{ji}$ using tensor cores.
10: **end for**
11: **end for**
12: Factorize $L_{kk}U_{kk} = B_{kk}$ (in precision u_{32}).
13: **for** $i = k+1\colon q$ **do**
14: Solve $L_{ik}U_{kk} = B_{ik}$ for L_{ik} (in precision u_{32}).
15: Solve $L_{kk}U_{ki} = B_{ki}$ for U_{ki} (in precision u_{32}).
16: **end for**
17: **for** $i = k\colon q$ **do**
18: Convert $\widetilde{L}_{ik} \leftarrow \mathrm{fl}_{16}(L_{ik})$.
19: Convert $\widetilde{U}_{ki} \leftarrow \mathrm{fl}_{16}(U_{ki})$.
20: **end for**
21: **end for**

Proof. Algorithm 5.1 requires the input matrix A to be in precision u_{16} so we must first account for this conversion: let $\widetilde{A} = \mathrm{fl}_{16}(A) = A + E$, with $|E| \le u_{16}|A|$. We now apply the algorithm to $\widetilde{A}$. The (i,k) block of the L factor is computed on line 14 by solving

$$L_{ik}\widehat{U}_{kk} = R_{ik}, \quad R_{ik} = \widetilde{A}_{ik} - \sum_{j=1}^{k-1} \widetilde{L}_{ij}\widetilde{U}_{jk}, \quad i > k,$$

where $\widehat{U}_{kk}$ is the computed U_{kk} in precision fp32, and $\widetilde{L}_{ij}\widetilde{U}_{jk}$ are the computed LU factors converted to fp16. By Lemma 1, the computed $\widehat{R}_{ik}$ satisfies, since $k \le q$,

$$|R_{ik} - \widehat{R}_{ik}| \le \gamma^{(32)}_{n-r+1}\left(|\widetilde{A}_{ik}| + \sum_{j=1}^{k-1} |\widetilde{L}_{ij}||\widetilde{U}_{jk}|\right). \tag{5.2}$$

Unlike in Algorithm 4.1, $\widehat{R}_{ik}$ is now in fp32, and so by [10, Thm. 8.5], the computed $\widehat{L}_{ik}$ satisfies

$$|\widehat{L}_{ik}\widehat{U}_{kk} - \widehat{R}_{ik}| \leq \gamma_r^{(32)}|\widehat{L}_{ik}||\widehat{U}_{kk}|. \tag{5.3}$$

We then convert $\widehat{L}_{ik}$ back to fp16 (line 18), obtaining $\widetilde{L}_{ik} = \mathrm{fl}_{16}(\widehat{L}_{ik}) = \widehat{L}_{ik} + F_{ik}$, with $|F_{ik}| \leq u_{16}|\widetilde{L}_{ik}|$ (using the alternative form [10, Eq. (2.5)] of the model of floating-point arithmetic to have $|\widetilde{L}_{ik}|$ rather than $|\widehat{L}_{ik}|$ on the right-hand side). Similarly the conversion of $\widehat{U}_{kk}$ (line 19) yields $\widetilde{U}_{kk} = \widehat{U}_{kk} + F_{kk}$, with $|F_{kk}| \leq u_{16}|\widetilde{U}_{kk}|$. Replacing $\widehat{L}_{ik}$ by $\widetilde{L}_{ik} - F_{ik}$ and $\widehat{U}_{kk}$ by $\widetilde{U}_{kk} - F_{kk}$ in (5.3) yields

$$|\widetilde{L}_{ik}\widetilde{U}_{kk} - \widehat{R}_{ik}| \leq \left(2u_{16} + u_{16}^2 + \gamma_r^{(32)}(1+u_{16})^2\right)|\widetilde{L}_{ik}||\widetilde{U}_{kk}|. \tag{5.4}$$

Combining (5.4) with (5.2), we conclude that for $i > k$,

$$\begin{aligned}\left|\widetilde{A}_{ik} - \sum_{j=1}^{k}\widetilde{L}_{ij}\widetilde{U}_{jk}\right| &\leq |R_{ik} - \widehat{R}_{ik}| + |\widehat{R}_{ik} - \widetilde{L}_{ik}\widetilde{U}_{kk}|, \\ &\leq f(n,r,u_{16},u_{32})\left(|\widetilde{A}_{ik}| + \sum_{j=1}^{k}|\widetilde{L}_{ij}||\widetilde{U}_{jk}|\right).\end{aligned} \tag{5.5}$$

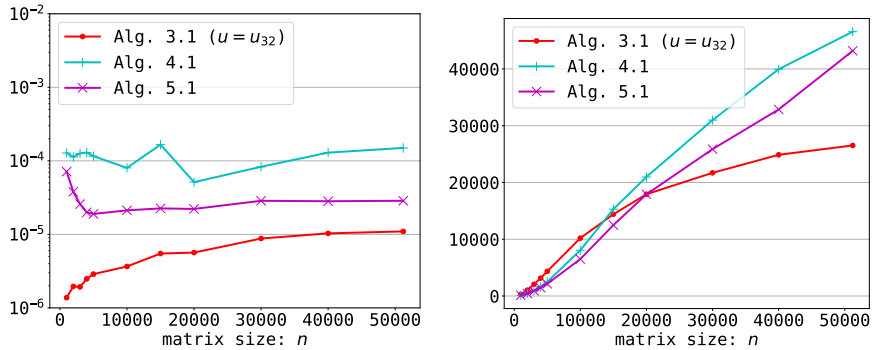
where $f(n,r,u_{16},u_{32}) = \max(\gamma_{n-r+1}^{(32)}, 2u_{16} + u_{16}^2 + \gamma_r^{(32)}(1+u_{16})^2)$. In a similar way, the inequality (5.5) can be shown to hold for $i < k$ and $i = k$. We have thus proved $\widetilde{A} + \Delta\widetilde{A} = \widetilde{L}\widetilde{U}$ with

$$|\Delta\widetilde{A}| \leq f(n,r,u_{16},u_{32})\left(|\widetilde{A}| + |\widetilde{L}||\widetilde{U}|\right).$$

Replacing $\widetilde{A}$ by $A + E$ yields the bound on $|\Delta A|$ and concludes the proof.

Theorem 3 shows that carrying out the panel factorization in fp32 arithmetic drops from the error bound (5.1) any term proportional to the panel size r times the fp16 unit roundoff u_{16}. For matrix sizes n such that the term ru_{16} dominates over nu_{32} (which, as discussed in the previous section, covers most practical values of n), Algorithm 5.1 therefore achieves an error bound roughly r times smaller than Algorithm 4.1.

We now assess experimentally the accuracy and performance of Algorithm 5.1 in Figure 3. We see that Algorithm 5.1 is indeed more accurate than Algorithm 4.1, up to about an order of magnitude in some cases. This confirms that the precision of the panel factorization is indeed the limiting factor for the accuracy of Algorithm 4.1. There still remains a small gap of about a factor three between the errors of Algorithms 5.1 and 3.1 with $u = u_{32}$. This small gap is not fully explained by the theoretical error bounds. A possible explanation may be that Algorithm 3.1 with $u = u_{32}$ does not need to convert the initial matrix A nor the diagonal blocks of its LU factors L_{kk} and U_{kk}.



(a) Componentwise backward error (2.2). **(b)** Performance (GFLOPS).

Figure 3. Accuracy and performance of Algorithms 3.1, 4.1 and 5.1 for diagonally dominant matrices. The panel size is $r = 256$.

While the accuracy of Algorithm 5.1 is therefore more or less satisfactory, the same cannot be said about its performance. Indeed, Figure 3b shows that switching the panel factorization precision from fp16 to fp32 may have a significant impact on the performance. For large n, the FLOPS rate may decrease by up to 20%.

This observation motivates the idea of exploiting the tensor cores not only in the update operations, but also in the panel factorization.

5.2 *Doubly partitioned algorithm that exploits GPU tensor cores in the panel factorization*

To our knowledge, all previous works on mixed precision LU factorization algorithms exploiting GPU tensor cores only used them in the updates, but not in the panel factorizations.

For example, Haidar et al. [3] explain this design choice with two reasons. First, the panel factorizations are said to occupy only "a small portion of the total time" [3, p. 3]; while that is usually true for monoprecision LU factorizations, the performance experiments shown in Figure 3b show that this is not the case in mixed precision arithmetic. Indeed, when the update operation is accelerated with tensor cores, the relative cost of the panel factorization becomes more important and cannot be neglected any more. The second reason given by [3] not to exploit tensor cores in the panel factorizations is that these operations are "numerically sensitive" [3, p. 4]. However, using a doubly partitioned matrix, the panel factorizations are nothing else than partitioned LU factorizations themselves, so it stands that we can recursively benefit from tensor cores without endangering the stability of the algorithm. Our numerical experiments will in fact show that exploiting the tensor cores within the panel factorizations is numerically harmless.

As described in section 2.2, tensor cores operate on matrices of dimensions 4×4. Therefore, in order to exploit tensor cores within the panel factorizations, a natural strategy is to partition again these panels, just as LAPACK's dgetrf routine [7]. The matrix is thus doubly partitioned into *outer panels* of size r, themselves partitioned into *inner panels* of size s. The panel factorizations then amount to a sequence of (inner) panel factorizations and updates (of the trailing outer panel), where the latter can be accelerated with tensor cores, provided that $s \geq 4$ (and for practical reasons, s should preferably be a multiple of 4).

The resulting algorithm is described in Algorithm 5.2, which is a modified version of Algorithm 4.1 where the panel factorization (lines 14 to 18) has been replaced by a `PanelFactor` algorithm that exploits tensor cores. Since the role of `PanelFactor` is simply to compute a partitioned LU factorization, any of the previously presented algorithms can be used, such as Algorithms 3.1, 4.1, or 5.1. (Strictly, these algorithms must be modified since they perform the LU factorization of a square matrix, whereas the panels are rectangular, but this modification is trivial: it suffices to stop the main loop on k at r/s instead of n/s).

Algorithm 5.2 A doubly partitioned LU factorization algorithm that exploits tensor cores in both the updates and panel factorizations.

1: **Input:** a matrix $A \in \mathbb{R}^{n \times n}$ in precision u_{16}, doubly partitioned into $r \times r$ blocks A_{ij}, themselves partitioned into $s \times s$ blocks, where $q = n/r$ and r/s are assumed to be integers, and a partitioned LU factorization algorithm `PanelFactor` (that potentially exploits tensor cores).
2: **Output:** the LU factors of A in precision u_{16} (with L and U partitioned into $r \times r$ blocks).
3: **for** $k = 1\colon q$ **do**
4: **for** $i = k\colon q$ **do**
5: Convert to fp32: $B_{ik} = \mathrm{fl}_{32}(A_{ik})$.
6: **if** $i \neq k$ **then** Convert to fp32: $B_{ki} \leftarrow \mathrm{fl}_{32}(A_{ki})$.
7: **for** $j = 1\colon k-1$ **do**
8: Update $B_{ik} \leftarrow B_{ik} - L_{ij}U_{jk}$ using tensor cores.
9: **if** $i \neq k$ **then** Update $B_{ki} \leftarrow B_{ki} - L_{kj}U_{ji}$ using tensor cores.
10: **end for**
11: **end for**
12: If necessary, convert B_{ik} and B_{ki}, $i = k\colon q$, to fp16.
13: Compute the LU factors L_{ik} and U_{ki}, $i = k\colon q$, using `PanelFactor`.
14: If necessary, convert L_{ik} and U_{ki}, $i = k\colon q$, to fp16.
15: **end for**

Theorem 4. Backward error bound for Algorithm 5.2. *Let $A \in \mathbb{R}^{n \times n}$ be doubly partitioned in $r \times r$ blocks themselves partitioned in $s \times s$ blocks. If Algorithm 5.2*

Table 2. Dominant term in the expression of g and f in the error bound (5.6) of Algorithm 5.2, depending on the `PanelFactor` algorithm.

`PanelFactor`	$\widetilde{u}_{\text{in}}$	$\widetilde{u}_{\text{out}}$	$g(r,s,u_{16},u_{32})$	$f(n,r,s,u_{16},u_{32})$
Alg. 2.1 ($u=u_{16}$)	u_{16}	0	ru_{16}	$ru_{16}+nu_{32}$
Alg. 2.1 ($u=u_{32}$)	0	u_{16}	ru_{32}	$2u_{16}+nu_{32}$
Alg. 3.1 ($u=u_{16}$)	u_{16}	0	$ru_{16}/4$	$ru_{16}/4+nu_{32}$
Alg. 3.1 ($u=u_{32}$)	0	0	$2u_{16}+ru_{32}$	$2u_{16}+nu_{32}$
Alg. 4.1	u_{16}	0	$\max(u_{16}+ru_{32}, su_{16})$	$su_{16}+nu_{32}$
Alg. 5.1	u_{16}	0	$\max(ru_{32}, 2u_{16}+su_{32})$	$2u_{16}+nu_{32}$

runs to completion, the computed LU factors $\widehat{L}$ *and* $\widehat{U}$ *satisfy* $A+\Delta A=\widehat{L}\widehat{U}$, *where*

$$|\Delta A| \le u_{16}|A| + f(n,r,s,u_{16},u_{32})\big((1+u_{16})|A| + |\widehat{L}||\widehat{U}|\big), \tag{5.6}$$

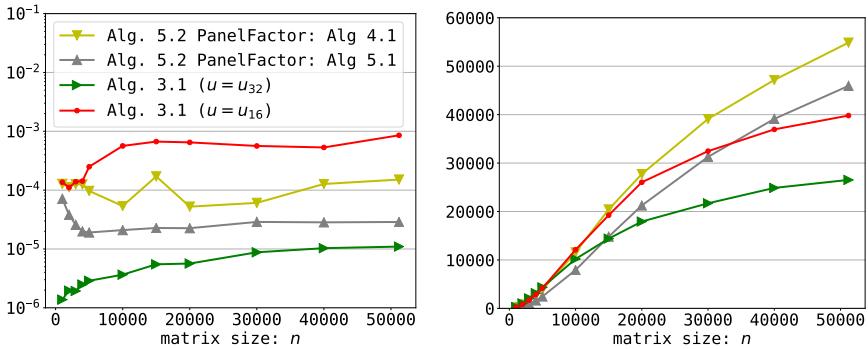
with

$$f(n,r,s,u_{16},u_{32}) = \max\big(\widetilde{u}_{\text{in}} + \gamma^{(32)}_{n-r+1} + \widetilde{u}_{\text{in}}\gamma^{(32)}_{n-r+1}, \\ g(r,s,u_{16},u_{32})(1+\widetilde{u}_{\text{out}})^2 + 2\widetilde{u}_{\text{out}} + \widetilde{u}^2_{\text{out}}\big), \tag{5.7}$$

where $g(r,s,u_{16},u_{32})$ *depends on the panel factorization algorithm* `PanelFactor` *(see Table 2), where* $\widetilde{u}_{\text{in}}=u_{16}$ *or* $\widetilde{u}_{\text{in}}=0$ *depending on whether* `PanelFactor` *takes fp32 or fp16 input, respectively, and* $\widetilde{u}_{\text{out}}=u_{16}$ *or* $\widetilde{u}_{\text{out}}=0$ *depending on whether it returns fp32 or fp16 output, respectively.*

Proof. The proof is derived by generalizing the proofs of Theorems 2 and 3 to a general panel factorization algorithm `PanelFactor`. In the proof of Theorem 2, (4.9) holds with u_{16} replaced by $\widetilde{u}_{\text{in}}$, since the conversion of R_{ik} is not needed if `PanelFactor` takes fp32 input, and (4.10) holds with $\gamma_r^{(16)}$ replaced by $g(r,s,u_{16},u_{32})$, whose value depends on `PanelFactor` and is given in Table 2. Finally, if `PanelFactor` returns fp32 output, the conversion of its output to fp16 must be accounted for, as done in the proof of Theorem 3, where (5.4) holds with u_{16} replaced by $\widetilde{u}_{\text{out}}$.

Theorem 4 generalizes both Theorems 2 and 3 to mixed precision LU factorization with an arbitrary panel factorization algorithm `PanelFactor`. Table 2 analyzes six possible choices for `PanelFactor`. For the first two options, Algorithm 2.1 in precision u is used, which does not exploit tensor cores; in this case Algorithm 5.2 reduces to either Algorithm 4.1 ($u=u_{16}$) or Algorithm 5.1 ($u=u_{32}$). The third option, Algorithm 3.1 with $u=u_{16}$, is not very interesting since the error bound still exhibits a term proportional to ru_{16}. The last three options are the most attractive ones, since they all achieve a bound whose term proportional to u_{16} is independent of the outer panel size r.



(a) Componentwise backward error (2.2). **(b)** Performance (GFLOPS).

Figure 4. Accuracy and performance of Algorithm 5.2 with `PanelFactor` 4.1 or `PanelFactor` 5.1, compared with Algorithm 3.1 with $u = u_{32}$ or $u = u_{16}$, for diagonally dominant matrices. The panel size is $r = 256$ and inner panel size $s = 8$.

In Figure 4, we compare the accuracy and performance of the last two options of Table 2), that is, Algorithms 4.1 or 5.1 as `PanelFactor`, with Algorithm 3.1 with $u = u_{32}$ or $u = u_{16}$.

- Figure 4a shows that Algorithm 5.2 with Algorithm 5.1 as `PanelFactor` achieves high accuracy, similar to that of Algorithm 5.1 in Figure 3a and almost comparable to Algorithm 3.1 with $u = u_{32}$. Moreover, Figure 4b shows that it also achieves high performance, peaking at around 47 TFLOPS, which is slightly better than the 43 TFLOPS performance of Algorithm 5.1 (shown in Figure 3b) and almost twice better than the 26 TFLOPS of Algorithm 3.1 with $u = u_{32}$.
- As for Algorithm 5.2 with Algorithm 4.1 as `PanelFactor`, Figure 4a shows that it achieves a lesser accuracy than with Algorithm 5.1 as `PanelFactor`. Its accuracy is similar to that of Algorithm 4.1 (shown in Figure 3a), even though the theoretical error bounds suggests it should be a factor r/s smaller. In any case, it remain significantly more accurate than Algorithm 3.1 with $u = u_{16}$. Figure 4b also shows that this algorithm achieves the highest performance of all variants considered in this article, peaking at around 56 TFLOPS, which is significantly better than the 46 TFLOPS of Algorithm 4.1 (shown in Figure 3b) and the 40 TFLOPS of Algorithm 3.1 with $u = u_{16}$.

To summarize, comparing Figures 3 and 4 shows that we have obtained two variants of Algorithm 5.2 (with different `PanelFactor`) that achieve similar accuracy to Algorithms 4.1 and 5.1, respectively, but significantly outperform them. This confirms that exploiting tensor cores in the panel factorizations allows to achieve the best performance–accuracy tradeoff. Moreover, Figure 4 also shows

that Algorithm 5.2 achieves a much better tradeoff than Algorithm 3.1, since it can be both faster and more accurate. Finally, there is no clear winner between the two choices of `PanelFactor` algorithm, since they each achieve a different compromise between performance and accuracy.

We make one final comment. In general the inner panel size s can be larger than the size of the block FMA $t = 4$. Therefore, we could further partition the inner panel into smaller panels of size t, obtaining a triply partitioned matrix overall. This could be useful in the cases where `PanelFactor` factorizes the inner panel in fp16 arithmetic, such as Algorithm 4.1, because it would reduce the error bound $su_{16} + nu_{32}$ (penultimate row of Table 2) to $tu_{16} + nu_{32}$. In the experiments of Figure 4, we have set the inner panel size to $s = 8$ (which we have observed to be the optimal value in terms of speed), so in our case the improvement of the error bound does not seem worth seeking, but different environments (much larger matrix, different block FMA unit) could lead to a different conclusion.

6 Numerical experiments on a range of matrices

In this section we compare four mixed precision LU factorization algorithms on a range of matrices coming from various applications: two versions of Algorithm 3.1 (with $u = u_{32}$ and $u = u_{16}$), and two versions of Algorithm 5.2 (with either Algorithm 4.1 or Algorithm 5.1 as `PanelFactor`).

6.1 Matrices from Fasi and Higham [15]

We first test our algorithms with the matrix generator proposed by Fasi and Higham [15], which was specially designed for the HPL-AI benchmark: it generates random unsymmetric matrices with no need for pivoting, and both the matrix dimension n and its condition number $\kappa(A)$ can be freely chosen.

In Figure 5 we plot the backward error for increasing values of n and for $\kappa(A) = 10^3$. We have observed results with other values of $\kappa(A)$ (not shown) to be similar. We also do not plot the performance, which does not depend on the matrix values and is therefore identical to Figure 4b. The behavior of the four algorithms on these matrices is similar to the one previously analyzed: Algorithm 5.2 achieves a significantly improved performance–accuracy tradeoff compared with Algorithm 3.1.

6.2 Matrices from SuiteSparse [9]

We now test our algorithms on some real-life matrices taken from the SuiteSparse collection[‡] [9]. Although these matrices are sparse, we treat them as dense for the purpose of our study. Note that most of these matrices are ill conditioned: ten ouf of the fifteen matrices have a condition number greater than 10^6, with one as large as 10^{20}. These matrices are therefore at risk of becoming singular when converted

[‡] https://sparse.tamu.edu/

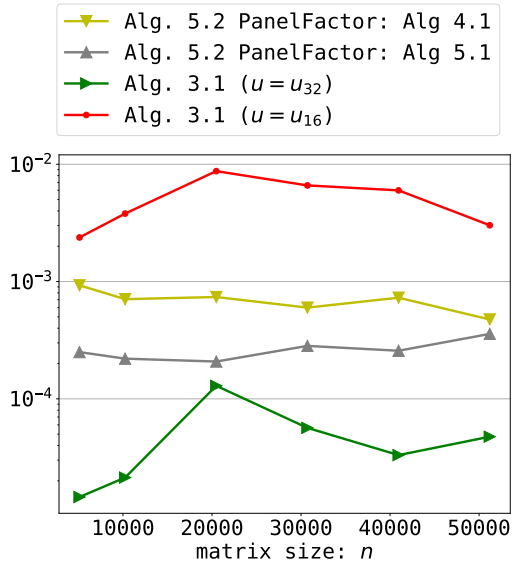


Figure 5. Componentwise backward error (2.2) for matrices generated with the Fasi and Higham [15] generator, with $\kappa(A) = 10^3$.

to half precision. In order to reduce the chance of that happening, we preprocess them using row and column scaling techniques as suggested in [16]. As a result, we were able to successfully factorize most of the test matrices. Nevertheless, for some matrices, the LU factorization failed and/or produced singular LU factors for some of the algorithms considered (indicated by a "—" in the table).

We compare the accuracy and performance of the four algorithms in Table 3. Algorithm 3.1 with $u = u_{32}$ is the most accurate of all four, but it is also the only algorithm that requires the matrix to be stored in fp32. Algorithm 3.1 with $u = u_{16}$ is the least accurate of all four. For these matrices, the error increase remains less than about an order of magnitude, but this is a still significant loss of accuracy. Most importantly, Algorithm 5.2 achieves a much better performance–accuracy tradeoff than Algorithm 3.1 with $u = u_{16}$. Indeed, when Algorithm 5.1 is used as `PanelFactor`, Algorithm 5.2 achieves a similar performance but a better accuracy than Algorithm 3.1 with $u = u_{16}$. Conversely, when Algorithm 4.1 is used as `PanelFactor`, then Algorithm 5.2 achieves an accuracy no worse than Algorithm 3.1 with $u = u_{16}$ (and often noticeably better), while achieving a much higher performance.

7 Performance experiments with A100 GPUs

All the previous experiments have been performed on the Volta architecture of the NVIDIA GPUs. In this last section, we provide some additional performance

Table 3. Accuracy and performance of four variants of mixed precision LU factorization on a set of SuiteSparse matrices. Algorithm 3.1 (with $u = u_{32}$ or $u = u_{16}$) is compared with Algorithm 5.2 (with two choices of `PanelFactor`, Algorithm 4.1 or 5.1). The performance ("perf.") is measured in TFLOPS. The accuracy is assessed with the componentwise backward error ε_{bwd} given by (2.2), and is scaled by 10^4.

		Alg. 3.1				Alg. 5.2 with `PanelFactor`...			
		$u = u_{32}$		$u = u_{16}$		Alg. 4.1		Alg. 5.1	
Matrix	n	ε_{bwd}	perf.	ε_{bwd}	perf.	ε_{bwd}	perf.	ε_{bwd}	perf.
gyro	17,361	0.4	16.4	4.5	22.8	3.9	28.2	2.2	18.7
FEM_3D_thermal1	17,880	0.4	16.9	4.0	24.9	3.7	27.4	2.4	18.9
Goodwin_040	17,922	0.8	17.0	3.0	24.6	1.7	28.0	0.9	19.5
nmos3	18,588	2.5	16.9	—	—	3.7	29.8	3.3	20.6
crystm03	24,696	0.5	20.3	3.4	29.4	2.9	37.2	2.7	27.3
smt	25,710	0.4	20.2	2.2	30.3	1.1	37.8	0.8	28.3
sme3Db	29,067	0.4	21.8	5.4	32.0	2.0	40.4	1.2	30.7
ship_001	29,736	3.2	23.0	—	—	—	—	2.6	35.9
nd12k	36,000	0.9	23.6	5.3	35.9	4.8	46.7	2.8	37.0
pdb1HYS	36,417	0.3	23.2	5.0	35.3	4.0	46.5	1.4	37.0
wathen120	36,441	0.7	23.7	2.9	35.8	2.7	46.5	1.9	37.0
jnlbrng1	40,000	1.2	24.9	2.8	37.1	2.8	49.5	2.4	39.6
torsion1	40,000	0.3	24.9	2.4	37.0	2.6	49.4	1.6	39.7
sme3Dc	42,930	0.5	27.0	6.4	38.7	1.6	52.6	1.3	43.9
Goodwin_071	56,021	0.7	27.3	—	—	2.4	54.8	1.5	46.6

measurements on the (more recent) Ampere A100 architecture, for which the peak performance of fp16 arithmetic with tensor cores is 312 TFLOPS, about 2.5× higher than on V100 GPUs.

We have observed the accuracy of the LU factorization on A100 (not shown) to be similar than on V100, so we focus here on performance only. Figure 6 plots the performance of the same four algorithms tested in the previous section on square matrices of order n up to $80,000$. The performance of our new Algorithm 5.2 compares again favorably with respect to the state-of-the-art Algorithm 3.1, and the results are even more positive than on V100. Algorithm 5.2 achieves up to 140 or even 170 TFLOPS (with Algorithm 5.1 or 4.1 as `PanelFactor`, respectively), which is much higher than Algorithm 3.1, which peaks at 52 and 92 TFLOPS with $u = u_{32}$ and $u = u_{16}$, respectively.

8 Conclusion

Modern GPUs equipped with mixed precision tensor core units can be exploited to significantly accelerate LU factorization of dense matrices. Existing approaches (Algorithm 3.1) however assume the matrix to be stored in fp32 precision ($u = u_{32}$), thereby preventing any memory consumption or data movement reductions associated with the use of fp16 arithmetic. This is explained by the fact that naively converting the matrix to fp16 (setting $u = u_{16}$ in Algorithm 3.1) leads to

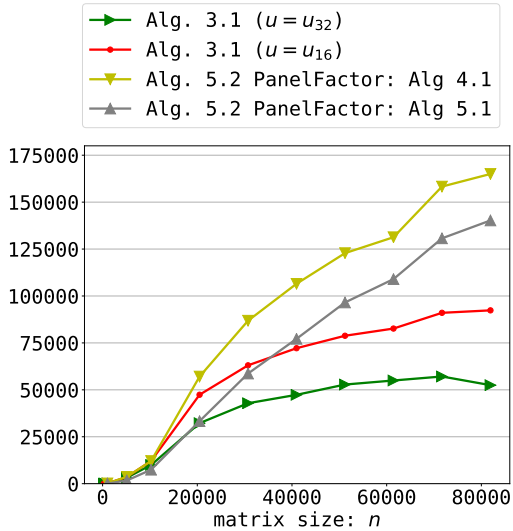


Figure 6. Performance (GFLOPS) on NVIDIA A100 GPUs.

a significant loss of accuracy, negating the entire accuracy boost delivered by the tensor cores (see Figure 1).

We have proposed a new mixed precision LU factorization algorithm that is able to store the matrix in fp16 without incurring such a significant loss of accuracy. The first step is to switch to a left looking scheme employing fp32 buffers of controlled size to accumulate the updates (Algorithm 4.1). This is however not sufficient, because carrying out the panel factorizations in fp16 arithmetic introduces an error term that in practice dominates the overall error. A possible solution is to carry out the panel factorizations in fp32 arithmetic instead (Algorithm 5.1), but this leads to a significant performance penalty. To overcome this issue, we propose a doubly partitioned factorization that also exploits the tensor cores in the panel factorizations (Algorithm 5.2). This is our final algorithm, and we show that it achieves the best performance–accuracy tradeoff of all the variants considered.

Throughout the article, we have assessed the performance and accuracy of all algorithms both theoretically and experimentally, by performig their rounding error analyses in Theorems 2, 3, and 4, and by providing numerical experiments on both random dense matrices and real life ones. Overall, we have shown on a number of different matrices that, compared with the state of the art, our new Algorithm 5.2 can be of similar accuracy, but requires only half the memory footprint and volume of data movement. As a result, it is potentially much faster: for large matrices, it achieves up to 50 TFLOPS on V100 and 170 TFLOPS on A100, which represents a 2× and 3.5× speedup, respectively.

We expect our algorithms and analysis to be applicable to other block FMA units similar to GPU tensor cores. Moreover, while we have focused here on using a single GPU unit, we expect our conclusions to remain relevant in a distributed environment with multiple GPUs.

In future work, we would like to investigate how the different algorithms analyzed in this article impact the convergence of iterative refinement [1, 2] or other similar iterative methods preconditioned by LU factorization. It would also be worth investigating the extension of this work to other matrix factorization algorithms, in particular QR factorization. Indeed, to our knowledge, existing mixed precision QR algorithms [17], [18] also assume the matrix to be stored in fp32 precision; the ideas proposed here could be extended so as to store them in fp16.

Acknowledgments

We thank Nicholas J. Higham for his feedback on a draft of this article.

References

[1] Carson E and Higham NJ. A new analysis of iterative refinement and its application to accurate solution of ill-conditioned sparse linear systems. *SIAM J Sci Comput* 2017; 39(6): A2834–A2856. DOI:10.1137/17M1122918.

[2] Carson E and Higham NJ. Accelerating the solution of linear systems by iterative refinement in three precisions. *SIAM J Sci Comput* 2018; 40(2): A817–A847. DOI:10.1137/17M1140819.

[3] Haidar A, Tomov S, Dongarra J et al. 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 (Dallas, TX), Piscataway, NJ, USA, pp. 47:1–47:11. DOI:10.1109/SC.2018.00050.

[4] Haidar A, Bayraktar H, Tomov S et al. Mixed-precision iterative refinement using tensor cores on GPUs to accelerate solution of linear systems. *Proc Roy Soc London Ser A* 2020; 476(2243): 20200110. DOI:10.1098/rspa.2020.0110.

[5] Blanchard P, Higham NJ, Lopez F et al. Mixed precision block fused multiply-add: Error analysis and application to GPU tensor cores. *SIAM J Sci Comput* 2020; 42(3): C124–C141. DOI:10.1137/19M1289546.

[6] Gustavson FG. Recursion leads to automatic variable blocking for dense linear-algebra algorithms. *IBM J Res Dev* 1997; 41(6): 737–755. DOI: 10.1147/rd.416.0737.

[7] Anderson E, Bai Z, Bischof C et al. *LAPACK Users' Guide*. Third ed. Philadelphia, PA: SIAM Press, 1995.

[8] Appleyard J and Yokim S. Programming tensor cores in CUDA 9. https://devblogs.nvidia.com/programming-tensor-cores-cuda-9/, 2017. Accessed March 25, 2019.

[9] Davis TA and Hu Y. The University of Florida sparse matrix collection. *ACM Trans Math Software* 2011; 38(1): 1:1–1:25. URL http://doi.acm.org/10.1145/2049662.2049663.

[10] Higham NJ. *Accuracy and Stability of Numerical Algorithms*. Second ed. Philadelphia, PA, USA: Society for Industrial and Applied Mathematics, 2002. ISBN 0-89871-521-0. DOI:10.1137/1.9780898718027.

[11] Oettli W and Prager W. Compatibility of approximate solution of linear equations with given error bounds for coefficients and right-hand sides. *Numer Math* 1964; 6: 405–409. DOI:10.1007/BF01386090.

[12] Higham NJ and Mary T. A new approach to probabilistic rounding error analysis. *SIAM J Sci Comput* 2019; 41(5): A2815–A2835. DOI: 10.1137/18M1226312.

[13] Higham NJ and Mary T. Sharper probabilistic backward error analysis for basic linear algebra kernels with random data. *SIAM J Sci Comput* 2020; 42(5): A3427–A3446. DOI:10.1137/20M1314355.

[14] Connolly MP, Higham NJ and Mary T. Stochastic rounding and its probabilistic backward error analysis. *SIAM J Sci Comput* 2021; 43(1): A566–A585. DOI:10.1137/20m1334796.

[15] Fasi M and Higham NJ. Matrices with tunable infinity-norm condition number and no need for pivoting in LU factorization. *SIAM J Matrix Anal Appl* 2021; 42(1): 417–435. DOI:10.1137/20m1357238.

[16] Higham NJ, Pranesh S and Zounon M. Squeezing a matrix into half precision, with an application to solving linear systems. *SIAM J Sci Comput* 2019; 41(4): A2536–A2551. DOI:10.1137/18M1229511.

[17] Yang LM, Fox A and Sanders G. Rounding error analysis of mixed precision block Householder QR algorithms. *SIAM J Sci Comput* 2021; 43(3): A1723–A1753. DOI:10.1137/19M1296367.

[18] Zhang S, Baharlouei E and Wu P. 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. DOI:10.1145/3369583.3392685.