Mixed Precision Block Fused Multiply-Add: Error Analysis and Application to GPU Tensor Cores

Blanchard, Pierre and Higham, Nicholas J. and Lopez, Florent and Mary, Theo and Pranesh, Srikara

2019

MIMS EPrint: **2019.18**

Manchester Institute for Mathematical Sciences
School of Mathematics

The University of Manchester

Reports available from: http://eprints.maths.manchester.ac.uk/
And by contacting: The MIMS Secretary
School of Mathematics
The University of Manchester
Manchester, M13 9PL, UK

ISSN 1749-9097

MIXED PRECISION BLOCK FUSED MULTIPLY-ADD: ERROR ANALYSIS AND APPLICATION TO GPU TENSOR CORES*

PIERRE BLANCHARD†, NICHOLAS J. HIGHAM‡, FLORENT LOPEZ§, THEO MARY¶, AND SRIKARA PRANESH‖

Abstract. Computing units that carry out a fused multiply-add (FMA) operation with matrix arguments, referred to as tensor units by some vendors, have great potential for use in scientific computing. However, these units are inherently mixed precision and existing rounding error analyses do not support them. We consider a mixed precision block FMA that generalizes both the usual scalar FMA and existing tensor units. We describe how to exploit such a block FMA in the numerical linear algebra kernels of matrix multiplication and LU factorization and give rounding error analyses of both kernels. An important application is to GMRES-based iterative refinement with block FMAs, for which our analysis provides new insight. Our framework is applicable to the tensor core units in the NVIDIA Volta and Turing GPUs. For these we compare matrix multiplication and LU factorization with TC16 and TC32 forms of FMA, which differ in the precision used for the output of the tensor cores. Our experiments on an NVDIA V100 GPU confirm the predictions of the analysis that the TC32 variant is much more accurate than the TC16 one, while achieving almost the same performance.

Key words. fused multiply-add, tensor cores, floating-point arithmetic, rounding error analysis, NVIDIA GPU, matrix multiplication, LU factorization

AMS subject classifications. 65F05, 65G50

1. Introduction.

A new development in high performance computing is the emergence of hardware supporting low precision floating-point formats such as the 16-bit IEEE half precision format (fp16) and the 16-bit bfloat16[1] [14]. Examples of such hardware include the NVIDIA P100 and V100 GPUs, the AMD Radeon Instinct MI25 GPU, Google's Tensor Processing Units (TPUs), and the ARM NEON architecture [2]. Expected to join them in the near future are the Fujitsu A64FX ARM processor [6] (supporting fp16), IBM's next generation AI chips [7] (supporting an 8-bit floating-point format in addition to fp16), and Intel's upcoming Xeon Cooper Lake [18] and Nervana Neural Network processors [17].

These new computing units execute low precision arithmetic faster than single precision (fp32), typically by a factor 2. But in the NVIDIA V100 GPU, thanks to special computing units called tensor cores, fp16 arithmetic executes up to 8 times faster than fp32 arithmetic.

This faster low precision arithmetic can be exploited in numerical algorithms. In [8], [9], [10] it is shown how on an NVIDIA V100, fp16 arithmetic can be used

*Version of September 24, 2019. **Funding**: This work was supported by Engineering and Physical Sciences Research Council grant EP/P020720/1, The MathWorks, and the Royal Society. The opinions and views expressed in this publication are those of the authors, and not necessarily those of the funding bodies.

†School of Mathematics, The University of Manchester, Manchester M13 9PL, UK (pierre.blanchard00@gmail.com).

‡School of Mathematics, University of Manchester, Manchester, M13 9PL, UK (nick.higham@manchester.ac.uk).

§Innovative Computing Laboratory (ICL), University of Tennessee, Knoxville, USA (flopez@icl.utk.edu).

¶School of Mathematics, The University of Manchester, Manchester M13 9PL, UK (theo.mary@manchester.ac.uk).

‖School of Mathematics, The University of Manchester, Manchester M13 9PL, UK (srikara.pranesh@manchester.ac.uk).

[1] https://en.wikipedia.org/wiki/Bfloat16_floating-point_format

with mixed precision iterative refinement to solve a linear system $Ax = b$ up to 4 times faster and with 80 percent less energy usage than by an optimized double precision solver, with no loss of accuracy or stability. Moreover, the same iterative refinement approach running on Summit [16], [19], the machine with 4608 nodes with 6 NVIDIA V100 GPUs per node that leads the June 2019 TOP 500 list,[2] has achieved a performance of 445 petaflops [3].

The tensor cores in the NVIDIA Volta and Turing architectures are able to carry out the operation $D = C + AB$, where all matrices are 4×4, in only one clock cycle and with just one rounding error per element of the result [1]. Moreover, while they require the matrices A and B to be in the fp16 format, C and the result 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 can therefore be seen as a generalization of a fused multiply-add (FMA) unit to 4×4 matrices, and they are an instance of what we call a "block FMA."

Multiprecision computing units called matrix units (MXU) are available on Google TPUs [20]. They use bfloat16 rather than fp16 as the low precision format and they operate on 128×128 matrices. However, Google TPUs are not commercially available, and details of computation in MXUs has not been made publicly available.

Tensor cores are inherently mixed precision units. Existing rounding error analyses will be pessimistic when applied to computations with tensor cores, as they reflect neither the single rounding error per element nor the potential mixed precision nature of the computation.

In this work we define a mixed precision block FMA that includes NVIDIA tensor cores as a special case and should be general enough to include future units. We present algorithms for matrix multiplication and LU factorization with a block FMA and give detailed rounding error analyses of them. Our analysis provides more realistic error bounds than standard analysis and can be used to determine the optimal tradeoff between performance and accuracy. In particular, in the case of NVIDIA tensor cores our analysis and experiments show that performing the block FMA with C and D in fp32 rather than fp16 can significantly improve the accuracy and stability of the algorithms, while not hindering too much their performance.

We define a block FMA in section 2. In section 3 we show how to exploit it in matrix multiplication and give a rounding error analysis. We then test accuracy and performance on an NVIDIA V100 for several matrix multiplication variants. In section 4 we present an algorithm for LU factorization based on a block FMA and give a rounding error analysis for the factorization and the solution of $Ax = b$. We show that the analysis gives new insights into GMRES-based iterative refinement. Then we give numerical experiments on an NVIDIA V100 to illustrate the error analysis and test the performance of four LU factorization variants. Concluding remarks are given in section 5.

[2] https://www.top500.org/lists/2019/06/

We will denote by fl_{16} and fl_{32} the operations of rounding to the fp16 and fp32 formats, and note that $u_{16} = 2^{-11}$ and $u_{32} = 2^{-24}$ are the respective unit roundoffs. The absolute value of a matrix is defined componentwise: $|A| = (|a_{ij}|)$.

2. Block fused multiply-add.

2.1. General framework for mixed-precision block FMA. Let $A \in \mathbb{R}^{b_1 \times b}$, $B \in \mathbb{R}^{b \times b_2}$, and $C \in \mathbb{R}^{b_1 \times b_2}$. We assume that A and B are provided in a given precision u_{low}, whereas C can be provided either in precision u_{low} or in a higher precision u_{high}. A block FMA computes

$$\underbrace{D}_{u_{\mathrm{low}} \text{ or } u_{\mathrm{high}}} = \mathrm{fl}_{\mathrm{FMA}}\Big(\underbrace{C}_{u_{\mathrm{low}} \text{ or } u_{\mathrm{high}}} + \underbrace{A}_{u_{\mathrm{low}}} \underbrace{B}_{u_{\mathrm{low}}} \Big),$$

where $\mathrm{fl}_{\mathrm{FMA}} = \mathrm{fl}_{\mathrm{low}}$ or $\mathrm{fl}_{\mathrm{FMA}} = \mathrm{fl}_{\mathrm{high}}$, where $\mathrm{fl}_{\mathrm{low}}$ and $\mathrm{fl}_{\mathrm{high}}$ round their arguments to arithmetics with precisions u_{low} and u_{high}, respectively. Thus D is the correctly rounded matrix at precision u_{low} or u_{high}.

For $b = b_1 = b_2$ we will refer to the block FMA as a $b \times b$ FMA. For $b = 1$ and just one precision, $u_{\mathrm{low}} = u_{\mathrm{high}}$, a block FMA is the usual scalar FMA found on a number of processors dating back to the 1990s.

2.2. GPU tensor cores. The tensor cores in the NVIDIA Volta and Turing architectures perform a $b \times b$ FMA for $b = 4$. As noted in section 1, while they require A and B to be fp16 matrices (that is, $u_{\mathrm{low}} = u_{16}$), C and D can be either fp16 or fp32 (that is, $u_{\mathrm{high}} = u_{32}$). The FMA can return its outputs at precision fp16 or fp32. Therefore there are four cases:

$$\widehat{D} = \begin{cases} \mathrm{fl}_{16}(C^{(16)} + AB), & \text{case 1, TC16,} \\ \mathrm{fl}_{16}(C^{(32)} + AB), & \text{case 2, TC16,} \\ \mathrm{fl}_{32}(C^{(32)} + AB), & \text{case 3, TC32,} \\ \mathrm{fl}_{32}(C^{(16)} + AB), & \text{case 4, TC32,} \end{cases} \tag{2.1}$$

where $C^{(16)}$ and $C^{(32)}$ denote an fp16 matrix and an fp32 matrix, respectively. The computed $\widehat{D}$ satisfies

$$|\widehat{D} - D| \le u_{\mathrm{FMA}}|D|, \tag{2.2}$$

where $u_{\mathrm{FMA}} = u_{16}$ in cases 1 and 2 and $u_{\mathrm{FMA}} = u_{32}$ in cases 3 and 4. For error analysis purposes it is irrelevant whether C is an fp16 or fp32 matrix, in the sense that it does not affect the constant u_{FMA} in the error bound (2.2). It is therefore useful to denote cases 1 and 2 as TC16 and cases 3 and 4 as TC32, as in (2.1).

3. Matrix multiplication with block FMA.

In this section we describe an algorithm to exploit a block FMA in matrix multiplication. We perform the rounding error analysis of this algorithm and compare our error bounds with the results of numerical experiments using tensor cores on an NVIDIA V100.

3.1. Description of the algorithm. Consider matrices $A \in \mathbb{R}^{m \times n}$ and $B \in \mathbb{R}^{n \times t}$ partitioned into $b_1 \times b$ and $b \times b_2$ blocks, respectively, where for simplicity we assume that $p = m/b_1$, $q = n/b$, and $r = t/b_2$ are integers. We describe in Algorithm 3.1 how to multiply A and B in a way that can exploit a block FMA.

Note that the algorithm does not assume that A and B are given in precision u_{low}. In the context of mixed precision LU factorization [9], for example, they may be given

Algorithm 3.1 Let $A = (A_{ij}) \in \mathbb{R}^{m\times n}$ and $B = (B_{ij}) \in \mathbb{R}^{n\times t}$ be partitioned into $b_1 \times b$ and $b \times b_2$ blocks, 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. The output C is in precision u_{high} or u_{low} depending on the type of FMA.

```
1: Convert A and B to precision u_low if necessary.
2: for i = 1: p do
3:     for j = 1: r do
4:         C_ij = 0
5:         for ℓ = 1: q do
6:             Compute C_ij = C_ij + A_iℓ B_ℓj using a block FMA.
7:         end for
8:     end for
9: end for
```

in u_{high}. We use the term "convert" on line 1 rather than "round" because in practice it might be necessary to do some kind of scaling to avoid overflow or underflow; see [13] for such considerations. However, in our analysis we will assume that underflow and overflow do not occur.

3.2. Rounding error analysis. We now give a rounding error analysis for Algorithm 3.1. Our strategy is to start with an existing analysis for matrix multiplication and observe that many of the rounding errors are zero because of the block FMA.

We will use the standard model of floating-point arithmetic [11, sec. 2.2]

$$\mathrm{fl}(a \mathrm{\ op\ } b) = (a \mathrm{\ op\ } b)(1+\delta), \quad |\delta| \le u, \quad \mathrm{op} \in \{+,-,\times,/\} \tag{3.1}$$

and the alternative form

$$\mathrm{fl}(a \mathrm{\ op\ } b) = \frac{a \mathrm{\ op\ } b}{1+\delta}, \quad |\delta| \le u, \quad \mathrm{op} \in \{+,-,\times,/\}. \tag{3.2}$$

Suppose we compute the inner product $s_n = x^T y$ of $x, y \in \mathbb{R}^n$ by left to right evaluation of $x_1y_1 + \cdots + x_ny_n$. The computed value $\widehat{s}_n$ satisfies [11, eq. (3.2)]

$$\widehat{s}_n = x_1y_1(1+\delta)^n + x_2y_2(1+\delta)^n + x_3y_3(1+\delta)^{n-1} + \cdots + x_ny_n(1+\delta)^2, \tag{3.3}$$

where each occurrence of $1+\delta$ denotes a possibly different $1+\delta_i$ with $|\delta_i| \le u$. Assuming that $nu < 1$, this leads to the error bound [11, eq. (3.5)]

$$|s_n - \widehat{s}_n| \le \gamma_n |x|^T |y|, \tag{3.4}$$

where

$$\gamma_n = \frac{nu}{1-nu}. \tag{3.5}$$

Thinking of x as a row a_i^T of A and y a column b_j of B, and writing $s_n = a_i^T b_j$ as

$$s_n = (x_1y_1 + \cdots + x_by_b) + (x_{b+1}y_{b+1} + \cdots + x_{2b}y_{2b}) + \cdots + (x_{n-b+1}y_{n-b+1} + \cdots + x_ny_n),$$

we see that the block FMA in Algorithm 3.1 ensures that each term in parentheses is computed exactly and added to the previous partial sum exactly before a final

rounding. Therefore by (3.3) we have

$$\begin{aligned}\widehat{s}_n = (x_1y_1 + \cdots + x_by_b)(1+\delta)^q + (x_{b+1}y_{b+1} + \cdots + x_{2b}y_{2b})(1+\delta)^{q-1} + \cdots \\ + (x_{n-b+1}y_{n-b+1} + \cdots + x_ny_n)(1+\delta),\end{aligned} \tag{3.6}$$

where $q = n/b$ and $|\delta_i| \le u_{\mathrm{FMA}}$. Instead of a product of up to n $1+\delta$ terms, as in (3.3), we have up to q such terms, which means that (3.4) holds with n replaced by q. Hence

$$|s_n - \widehat{s}_n| \le \gamma_q^{\mathrm{FMA}} |a_i|^T |b_j|, \tag{3.7}$$

where

$$\gamma_q^{\mathrm{FMA}} = \frac{q u_{\mathrm{FMA}}}{1 - q u_{\mathrm{FMA}}}. \tag{3.8}$$

If the input matrices A and B are given in precision u_{low} then we can directly apply the above analysis to obtain the following bound for Algorithm 3.1.

THEOREM 3.1. *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_{low} be evaluated by Algorithm 3.1, where $p = m/b_1$, $q = n/b$, and $r = t/b_2$ are assumed to be integers. The computed $\widehat{C}$ satisfies*

$$|C - \widehat{C}| \le \gamma_q^{\mathrm{FMA}} |A||B|. \tag{3.9}$$

If A and B are not given in precision u_{low} then we must account for the initial conversion to precision u_{low}. Assuming rounding with no underflow or overflow, we have

$$\begin{aligned}\widetilde{A} &= \mathrm{fl}_{\mathrm{low}}(A) = A + \Delta A, \quad |\Delta A| \le u_{\mathrm{low}}|A|, \\ \widetilde{B} &= \mathrm{fl}_{\mathrm{low}}(B) = B + \Delta B, \quad |\Delta B| \le u_{\mathrm{low}}|B|.\end{aligned}$$

Applying (3.7) to the computation of $C = \widetilde{A}\widetilde{B}$ yields

$$\widehat{C} = \widetilde{A}\widetilde{B} + \Delta C, \qquad |\Delta C| \le \gamma_q^{\mathrm{FMA}} |\widetilde{A}||\widetilde{B}|,$$

so

$$\widehat{C} = AB + \Delta AB + A\Delta B + \Delta A\Delta B + \Delta C =: AB + E.$$

Bounding E yields the following result.

THEOREM 3.2. *Let the product $C = AB$ of $A \in \mathbb{R}^{m\times n}$ and $B \in \mathbb{R}^{n\times t}$ (not necessarily given in precision u_{low}) be evaluated by Algorithm 3.1, where $p = m/b_1$, $q = n/b$, and $r = t/b_2$ are assumed to be integers. The computed $\widehat{C}$ satisfies*

$$|C - \widehat{C}| \le \left(2u_{\mathrm{low}} + u_{\mathrm{low}}^2 + \gamma_q^{\mathrm{FMA}}(1 + u_{\mathrm{low}})^2\right)|A||B|. \tag{3.10}$$

The bounds (3.9) and (3.10) should be compared with [11, eq. (3.13)]

$$|C - \widehat{C}| \le \gamma_n |A||B| \tag{3.11}$$

for a standard matrix product that does not use a block FMA.

We gather in Table 3.1 the dominant terms in the error bounds for four matrix multiplication implementations, with A and B given in precisions u_{high} or u_{low}: standard multiplication in precision u_{high} or u_{low} (error bound (3.11)) and multiplication

Table 3.1: Dominant terms in the error constant multiplying $|A||B|$ for standard matrix multiplication and for Algorithm 3.1, from (3.9) and (3.10). Here, $q = n/b$.

Precision of A and B	Standard in precision u_{low}	Block FMA $u_{\mathrm{FMA}} = u_{\mathrm{low}}$	Block FMA $u_{\mathrm{FMA}} = u_{\mathrm{high}}$	Standard in precision u_{high}
u_{low}	nu_{low}	qu_{low}	qu_{high}	nu_{high}
u_{high}	$(n+2)u_{\mathrm{low}}$	$(q+2)u_{\mathrm{low}}$	$2u_{\mathrm{low}} + qu_{\mathrm{high}}$	nu_{high}

Table 3.2: Specialization of the bounds in Table 3.1 for NVIDIA tensor cores, for which $b = 4$ and where $u_{\mathrm{low}} = u_{16}$ (TC16) or $u_{\mathrm{high}} = u_{32}$ (TC32).

Precision of A and B	Standard fp16	Tensor core TC16	Tensor core TC32	Standard fp32
u_{16}	nu_{16}	$nu_{16}/4$	$nu_{32}/4$	nu_{32}
u_{32}	$(n+2)u_{16}$	$(n/4+2)u_{16}$	$2u_{16} + nu_{32}/4$	nu_{32}

with a block FMA with $u_{\mathrm{FMA}} = u_{\mathrm{high}}$ or $u_{\mathrm{FMA}} = u_{\mathrm{low}}$ (error bounds (3.9) and (3.10), respectively). For the sake of readability, in the bounds in Table 3.1 we have assumed that $u_{\mathrm{high}} \ll u_{\mathrm{low}}$.

Compared with the standard multiplication in precision u_{low}, the block FMA multiplication with $u_{\mathrm{FMA}} = u_{\mathrm{low}}$ has a smaller error bound by a factor approximately b. Importantly, the bound in (3.9) is about a factor $u_{\mathrm{low}}/u_{\mathrm{high}}$ smaller when the block FMA is returned at the higher precision rather than the lower precision. Moreover, when $u_{\mathrm{FMA}} = u_{\mathrm{high}}$ the bound (3.10) does not grow with n as long as $n < 2bu_{\mathrm{low}}/u_{\mathrm{high}}$ and, for larger n, the bound remains roughly a factor b smaller than the bound for standard multiplication in precision u_{high}.

The bounds above are worst-case and so may be pessimistic, especially for large dimensions (see the numerical results in the next subsection). To derive more realistic bounds, we can use a probabilistic model of the rounding errors [12]. A probabilistic analogue of Theorem 3.2 directly follows from [12, Thm. 3.1]. It contains a modified version of (3.10) with γ_q^{FMA} replaced by a relaxed constant $\gamma_q^{\mathrm{FMA}}(\lambda)$ proportional to $\lambda q^{1/2} u_{\mathrm{high}}$, and it holds with a probability at least a given quantity that is very close to 1 for λ of order 10 or so. In particular, this means that the block FMA bound with $u_{\mathrm{FMA}} = u_{\mathrm{high}}$ may only start growing with n for much larger n than the worst-case bound suggests (perhaps for n larger than $4b(u_{\mathrm{low}}/u_{\mathrm{high}})^2$).

We now apply these bounds to NVIDIA tensor cores. This amounts to taking $b = 4$, $u_{\mathrm{high}} = u_{32}$, and $u_{\mathrm{low}} = u_{16}$ in the bounds of Table 3.1. We gather the resulting bounds in Table 3.2, where we consider standard half and single precision multiplication (fp16 and fp32), and tensor core multiplication in either TC16 ($u_{\mathrm{FMA}} = u_{16}$) or TC32 ($u_{\mathrm{FMA}} = u_{32}$) mode. We see clearly that the tensor cores bring two benefits: TC16 reduces the constant n multiplying u_{16} to $n/4$, while TC32 remove n entirely from the u_{16} term and relegates it to the u_{32} term. Perhaps surprisingly, for $n \gtrsim 8u_{16}/3u_{32} = 21845$ the analysis suggests that TC32 may be more accurate than fp32.

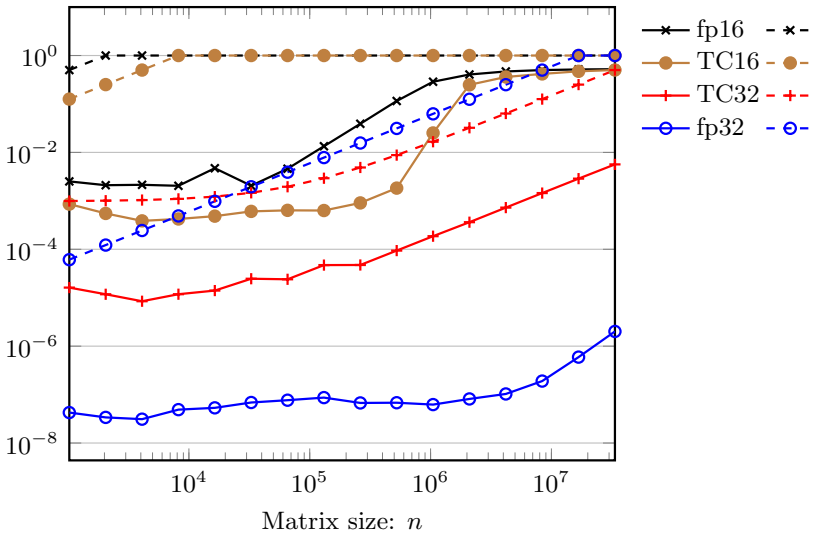

Fig. 3.1: Forward errors for four matrix multiplication variants on NVIDIA V100 GPU (fp16, TC16, TC32, and fp32) where matrix entries are sampled uniformly in $[0, 10^{-3}]$. Solid lines represent errors while dashed lines represent error bounds.

3.3. Numerical experiments with tensor core matrix multiplication. We now present numerical experiments to investigate whether the error bounds correctly predict the relative accuracy of the different methods and to assess the performance of the methods. We use the implementation of the four matrix multiplication variants provided in the cuBLAS library v10.1. The matrices A and B are random, with entries sampled uniformly from $[0, 10^{-3}]$ or $[-1, 1]$. We set the upper bound to 10^{-3} in the former case to avoid overflow in the multiplication for large sizes. Matrices are generated in single precision in order to compare the accuracy of the fp16, TC16, and TC32 computations with those for fp32. As a result, for the first three methods matrices A and B will be converted to half precision prior to multiplication. To assess the sharpness of the bounds we consider $A \in \mathbb{R}^{m\times n}$ and $B \in \mathbb{R}^{n\times t}$ with n varying from 1024 to approximately 2×10^6, and in order that the matrices fit on a single GPU we take $m = t = 8$. Figures 3.1 and 3.2 plot the forward error measure $\|\widehat{C} - C\|_F/(\|A\|_F\|B\|_F)$ for increasing values of n, where C is an approximation of the exact result obtained with a standard double precision multiplication computed with cuBLAS. This error measure is not the true relative forward error but is what the analysis bounds when norms are taken in the bounds. Bounds associated with each variant are represented in dashed lines. Once a bound reaches the value 1, we set it to 1 as it provides no useful information.

As expected almost all errors are relatively far from their worst-case bounds. Moreover, the errors in Figure 3.2 are generally much smaller than their counterparts in Figure 3.1. For matrices with positive entries (Figure 3.1) the errors for fp16 and TC16 range from 10^{-3} to 1 and exceed 0.1 for $n \approx 10^6$. By contrast, the errors for TC32 and fp32 are much smaller: between 10^{-5} and 10^{-2} for TC32 and between 10^{-7} and 10^{-5} for fp32. We observe that the error bound for TC32 is rather loose (a constant two orders of magnitude gap) but still insightful, as it captures the growth

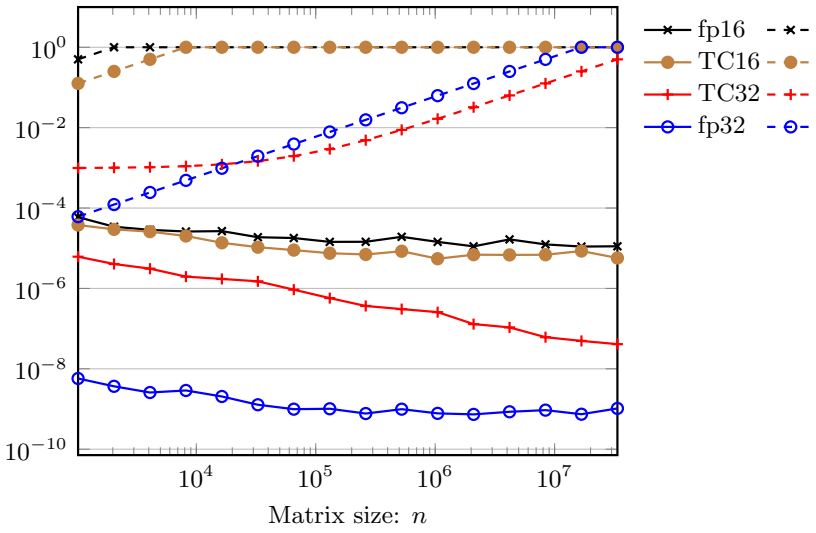

Fig. 3.2: Forward errors for four matrix multiplication variants on NVIDIA V100 GPU (fp16, TC16, TC32, and fp32) where matrix entries are sampled uniformly in $[-1, 1]$. Solid lines represent errors while dashed lines represent error bounds.

of the error with n. On the other hand, error growth is not observed before $n = 10^6$ with fp32. Consequently, the possibility suggested by the analysis that at $n \approx 2 \times 10^4$ TC32 could become more accurate than fp32 is not realized with this dataset. A probabilistic bound, as mentioned in the previous section, would predict crossover at the square of this value of n, which is beyond our range of n.

In order to give a baseline for the performance of each matrix multiplication method we run additional simulations for square matrices. Figure 3.3 shows the maximum flop rate out of five runs (in TFlops/s) for each multiplication method and each matrix size. We see that although the TC16 variant performs slightly better than the TC32 variant for matrix sizes up to $n = 8000$, the two variants have similar asymptotic performance for $n > 8000$. In that range of sizes the flop rates associated with tensor core-enabled multiplication are about 3.5× larger than fp16 multiplication (3.3 to 3.6) and about 7× larger than single precision multiplication (6.8 to 7.3), both executed on CUDA cores. The flop rate of TC16 multiplication reaches a maximum of 101.2 TFlops/s (about 90% of the theoretical performance, namely 112.7TFlops/s) for $n = 8000$. Our performance results are in good agreement with other existing benchmarks, e.g., [15].

4. Solution of linear systems with block FMA. Now we consider the solution of linear systems $Ax = b$ by LU factorization, where A is a dense $n \times n$ matrix. Since LU factorization can be formulated to exploit matrix multiplication it can benefit from using a block FMA.

Algorithm 4.1 computes an LU factorization using a block FMA. The algorithm employs three precisions: the working precision u and the precisions u_{low} and u_{high} used by the block FMA employed within the call to Algorithm 3.1 on line 9. We assume that A is given in precision u and that precision u is used on lines 2 and 4. Other versions of the algorithm can be defined by varying these precisions.

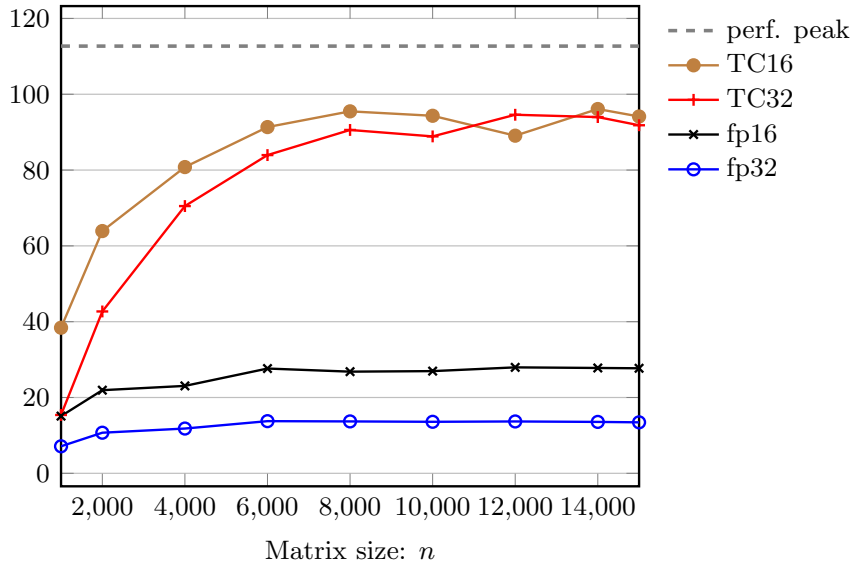

Fig. 3.3: Performance results (in TFlops/s) for matrix multiplication variants on NVIDIA V100 GPU (fp16, TC16, TC32 and fp32) with square matrices.

Algorithm 4.1 Let $A = (A_{ij}) \in \mathbb{R}^{n\times n}$ be given in precision u and partitioned into $b \times b$ blocks, where $q = n/b$ is assumed to be an integer. This algorithm performs the right-looking LU factorization $A = LU$ (with L and U partitioned into $b \times b$ blocks) exploiting a $b \times b$ FMA.

1: **for** $k = 1\colon q$ **do**
2: Factorize $L_{kk}U_{kk} = A_{kk}$.
3: **for** $i = k+1\colon q$ **do**
4: Solve $L_{ik}U_{kk} = A_{ik}$ and $L_{kk}U_{ki} = A_{ki}$ for L_{ik} and U_{ki}.
5: **end for**
6: **for** $i = k+1\colon q$ **do**
7: **for** $j = k+1\colon q$ **do**
8: $\widetilde{L}_{ik} \leftarrow \mathrm{fl}_{\mathrm{low}}(L_{ik})$ and $\widetilde{U}_{ki} \leftarrow \mathrm{fl}_{\mathrm{low}}(U_{ki})$.
9: $A_{ij} \leftarrow A_{ij} - \widetilde{L}_{ik}\widetilde{U}_{kj}$ using Algorithm 3.1.
10: **end for**
11: **end for**
12: **end for**

4.1. Rounding error analysis. We now perform a rounding error analysis of Algorithm 4.1. and its use to solve linear systems $Ax = b$. We begin with a simple application of Theorem 3.1.

COROLLARY 4.1. *Let $B = A - \sum_{i=1}^{q} X_i Y_i$, where $A, B, X_i, Y_i \in \mathbb{R}^{b\times b}$ are given in precision u_{high}, be computed with a $b \times b$ FMA. The computed $\widehat{B}$ satisfies*

$$|\widehat{B} - B| \le \gamma_q^{\mathrm{FMA}}\left(|A| + \sum_{k=1}^{q} |X_k||Y_k|\right).$$

Proof. Write $B = A - [X_1\ X_2\ \dots\ X_q][Y_1^T\ Y_2^T\ \dots\ Y_q^T]^T$, and apply Theorem 3.1, noting that A partakes in the first block FMA and so has no effect on the constant in the bound. ∎

THEOREM 4.2. *Let $A \in \mathbb{R}^{n\times n}$ be partitioned in $b \times b$ blocks, with $q = n/b$ an integer. If Algorithm 4.1 runs to completion then the computed LU factors $\widehat{L}$ and $\widehat{U}$ satisfy $A + \Delta A = \widehat{L}\widehat{U}$, where*

$$|\Delta A| \le \Big(2u_{\mathrm{low}} + u_{\mathrm{low}}^2 + \max(\gamma_{q-1}^{\mathrm{FMA}}, \gamma_b)(1+u_{\mathrm{low}})^2\Big)(|A| + |\widehat{L}||\widehat{U}|).$$

Proof. The (i,k) block of the L factor is computed by solving

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

where $\widetilde{L}$ and $\widetilde{U}$ denote the computed factors that have been converted to precision u_{low} (line 8 of Algorithm 4.1) and satisfy $\widetilde{L}_{ij} = \widehat{L}_{ij} + E_{ij}$ and $\widetilde{U}_{jk} = \widehat{U}_{jk} + F_{jk}$, where $|E_{ij}| \le u_{\mathrm{low}}|\widehat{L}_{ij}|$ and $|F_{jk}| \le u_{\mathrm{low}}|\widehat{U}_{jk}|$. By Corollary 4.1, the computed $\widehat{R}_{ik}$ satisfies

$$|R_{ik} - \widehat{R}_{ik}| \le \gamma_{q-1}^{\mathrm{FMA}}\Big(|A_{ik}| + \sum_{j=1}^{k-1} |\widetilde{L}_{ij}||\widetilde{U}_{jk}|\Big)$$

and by [11, Thm. 8.5] we have

$$|\widehat{L}_{ik}\widehat{U}_{kk} - \widehat{R}_{ik}| \le \gamma_b |\widehat{L}_{ik}||\widehat{U}_{kk}|. \tag{4.1}$$

Combining these two inequalities gives

$$\Big|A_{ik} - \sum_{j=1}^{k-1} \widetilde{L}_{ij}\widetilde{U}_{jk} - \widehat{L}_{ik}\widehat{U}_{kk}\Big| \le \max(\gamma_{q-1}^{\mathrm{FMA}}, \gamma_b)\Big(|A_{ik}| + \sum_{j=1}^{k-1} |\widetilde{L}_{ij}||\widetilde{U}_{jk}| + |\widehat{L}_{ik}||\widehat{U}_{kk}|\Big).$$

Replacing $\widetilde{L}_{ij}$ by $\widehat{L}_{ij} + E_{ij}$ and $\widetilde{U}_{jk}$ by $\widehat{U}_{jk} + F_{jk}$, we obtain

$$\Big|A_{ik} - \sum_{j=1}^{k} \widehat{L}_{ij}\widehat{U}_{jk} - G\Big| \le \max(\gamma_{q-1}^{\mathrm{FMA}}, \gamma_b)(1+u_{\mathrm{low}})^2\Big(|A_{ik}| + \sum_{j=1}^{k} |\widehat{L}_{ij}||\widehat{U}_{jk}|\Big),$$

where

$$G = \sum_{j=1}^{k-1} \big(E_{ij}\widehat{U}_{jk} + \widehat{L}_{ij}F_{jk} + E_{ij}F_{jk}\big)$$

and thus $|G| \le (2u_{\mathrm{low}} + u_{\mathrm{low}}^2)\sum_{j=1}^{k-1} |\widehat{L}_{ij}||\widehat{U}_{jk}|$. We conclude that for $i > k$,

$$\begin{aligned} |A_{ik} - \sum_{j=1}^{k} \widehat{L}_{ij}\widehat{U}_{jk}| \le{}& \Big(2u_{\mathrm{low}} + u_{\mathrm{low}}^2 + \max(\gamma_{q-1}^{\mathrm{FMA}}, \gamma_b)(1+u_{\mathrm{low}})^2\Big) \\ &\times \Big(|A_{ij}| + \sum_{j=1}^{k} |\widehat{L}_{ij}||\widehat{U}_{jk}|\Big). \end{aligned} \tag{4.2}$$

For $i = k$, L_{kk} is determined with U_{kk} on line 2 of Algorithm 4.1, and by [11, Thm. 9.3] we have $|\widehat{L}_{kk}\widehat{U}_{kk} - \widehat{R}_{kk}| \le \gamma_b |\widehat{L}_{kk}||\widehat{U}_{kk}|$. Therefore (4.1) holds for $i = k$, too, and hence so does (4.2). In a similar way, the inequality (4.2) can be shown to hold for $i > k$. ∎

Table 4.1: Dominant terms in the error constant $c(n, u_{16}, u_{32})$ in the backward error bound $|\Delta A| \le c(n, u_{16}, u_{32})(|A| + |\widetilde{L}||\widetilde{U}|)$ in (4.3) for the solution of $Ax = b$ using tensor cores. We have taken $u = u_{16}$ for fp16 and TC16, and $u = u_{32}$ for fp32 and TC32.

	fp16	TC16	TC32	fp32
$c(n, u_{16}, u_{32})$	$(3n+1)u_{16}$	$(9n/4+1)u_{16}$	$2u_{16} + (9n/4-1)u_{32}$	$3nu_{32}$

THEOREM 4.3. *Let $A \in \mathbb{R}^{n\times n}$ be partitioned in $b \times b$ blocks, with $q = n/b$ an integer. If Algorithm 4.1 produces computed LU factors $\widehat{L}$ and $\widehat{U}$ and substitution yields a computed solution $\widehat{x}$ to $Ax = b$ then $(A + \Delta A)\widehat{x} = b$, where*

$$|\Delta A| \le \big(2u_{\text{low}} + u_{\text{low}}^2 + \max(\gamma_{q-1}^{\text{FMA}}, \gamma_b)(1 + u_{\text{low}})^2 + 2\gamma_n + \gamma_n^2\big)(|A| + |\widehat{L}||\widehat{U}|). \tag{4.3}$$

Proof. The result is obtained by combining Theorem 4.2 with the error analysis for the solution of triangular systems [11, Thm. 8.5] and is analogous to the proof of [11, Thm. 9.4]. ∎

We gather in Table 4.1 the dominant terms in the error bound for the solution of linear systems using NVIDIA tensor cores. We distinguish the same four variants of matrix multiplication as in Table 3.2. In the fp16 and fp32 cases, we naturally take the working precision to be $u = u_{16}$ and $u = u_{32}$, respectively. In the TC16 case, both $u = u_{16}$ and $u = u_{32}$ are possible, but since the FMA uses u_{16} precision, we might as well take the working precision to be $u = u_{16}$. Finally, in the TC32 case, in order to preserve the accuracy benefit of using an FMA and avoid the error growing with n to first order, we must take $u = u_{32}$.

Overall, these bounds lead to the same conclusions as in the matrix multiplication case: the TC16 bound is smaller than the fp16 one by about a factor $b/3 = 4/3$, while the TC32 variant leads to a much smaller bound, which only starts growing with n when $n \gtrsim \frac{8}{9}u_{16}/u_{32} \approx 7.3 \times 10^3$ (at which point it is slightly smaller than the fp32 bound).

4.2. Application to GMRES-based iterative refinement. Our analysis is applicable to the work in Haidar et al. [9], in which an implementation of Algorithm 4.1 on an NVIDIA V100 was used with single precision as the working precision and fp16 or TC32 for the matrix multiplications. The resulting LU factorization was used as a preconditioner in GMRES-based iterative refinement [4], [5]. In the experiments reported in [9], the total number of GMRES iterations (a good measure of the cost of refinement) for TC32 was at most half that for fp16 (with a significant increase in performance too). This is what would be expected from Table 4.1, where the error constant for TC32 is a factor ranging from 2.9×10^3 to 3.5×10^4 smaller than that for fp16 for the matrix sizes $n \in [2000, 34000]$ used in those experiments.

4.3. Numerical experiments with tensor core LU factorization. We now present experiments testing the accuracy and performance of the LU factorization computed by Algorithm 4.1 for solving $Ax = b$ on an NVIDIA V100 GPU. Our implementation does not use pivoting in the LU factorization and it performs all the operations (factor, solve, and update) solely on the GPU. We use our own CUDA kernels for the factor and solve operations and use the `cublasGemmEx` routine from

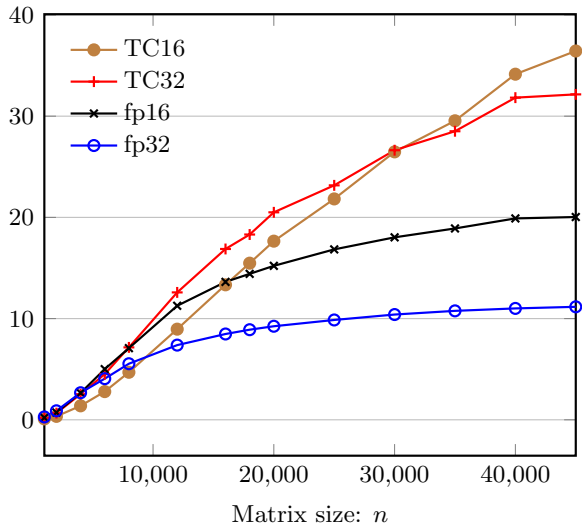

Fig. 4.1: Performance in Tflop/s of the LU factorization computed by Algorithm 4.1 on an NVIDIA V100 GPU for the four variants fp16, TC16, TC32, and fp32.

the cuBLAS library, which is the same as the routine tested in Section 3 for the matrix multiplication, for performing the update operation. In the following experiments, we use fp32 as working precision and compare the four variants fp16, TC16, TC32, and fp32 listed in Table 4.1.

The test matrices are randomly generated as $Q_1 D Q_2$, where Q_1 and Q_2 are random orthogonal matrices and D is diagonal, with $d_{ii} = 10^{-c(i-1)/(n-1)}$. The resulting matrix has singular values lying between 1 and 10^{-c} and thus a condition number equal to 10^c. In our experiments we set $c = 3$.

In Figure 4.1 we show the performance for the fp16, TC16, TC32, and fp32 variants for square matrices with n ranging between 1000 and 45000. Note that we do not include the times for the forward and backward substitution in these results because the cost of the factorization largely dominates the total cost for solving the linear system. In this figure we see that the fp32 variant asymptotically reaches 10 TFlop/s and that, as expected, the fp16 variant achieves twice the performance of that variant with around 20 TFlop/s. Note that the cuSOLVER library, as part of the CUDA toolkit, provides a single-precision LU factorization routine called `cusolverDnSgetrf` corresponding to our fp32 variant. For the sake of clarity, we do not include experimental results for the `cusolverDnSgetrf` routine but we observed that our implementation achieves similar performance to this routine. The TC16 and TC32 variants achieve much higher asymptotic performance, respectively around 36 and 32 TFlop/s, due to the use of the tensor cores. Although the TC16 and TC32 variants show similar performance behavior, the TC16 variant is slightly less efficient on the smaller matrices. On the largest matrix, though, the TC16 variants offer slightly better performance than TC32 which is consistent with the performance results obtained with the matrix multiply operation shown in Figure 3.3.

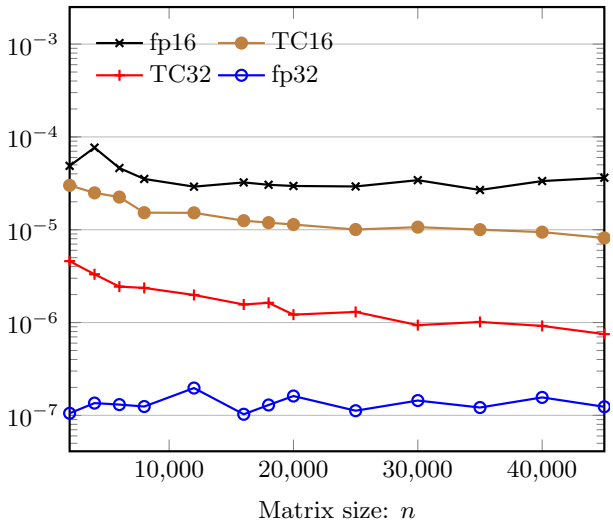

Fig. 4.2: Componentwise backward error for the solution of $Ax = b$ using an LU factorization on an NVIDIA V100 GPU for the four variants fp16, TC16, TC32, and fp32.

A comparison of the componentwise backward errors

$$\max_i \frac{|A\widehat{x} - b|_i}{((|A| + |\widehat{L}||\widehat{U}|)|x|)_i}$$

is given in Figure 4.2. The TC16 variant gives a smaller backward error than the fp16 one. This is due to the use of a block FMA operation in the tensor cores and is consistent with the error bounds shown in Table 4.1. The TC32 variant gives a backward error between one and two orders of magnitude smaller than the fp16 and TC16 variants. The fp32 variant gives the smallest backward error but it is up to 3 times slower than the TC32 variant, as shown by Figure 4.1.

We conclude from these results that the TC32 variant offers the best performance versus accuracy tradeoff, as it exploits the performance capabilities of the tensor cores and has similar performance to TC16 variant, while giving much smaller backward errors than the fp16 and TC16 variants and backward errors only 1 to 1.5 orders of magnitude larger than for fp32.

5. Conclusion. We have considered a general mixed precision block FMA unit that carries out a mixed-precision fused multiply-add operation $D \leftarrow C + AB$ on $b \times b$ matrices. This block FMA generalizes the usual scalar FMA in two ways. First, it works on matrices (for $b > 1$) instead of scalars. Second, it takes A and B stored in precision u_{low} and C stored in precision u_{low} or u_{high}, computes D with a single rounding error per element, and returns D in precision u_{low} or u_{high}. Here, $u_{\text{high}} < u_{\text{low}}$.

We have proposed matrix multiplication and LU factorization algorithms that exploit such units and given detailed rounding error analyses of the algorithms, distinguishing several variants depending on which precisions are used to store each matrix.

If only one precision, $u_{\text{low}} = u_{\text{high}}$ is used, a $b \times b$ block FMA leads to error bounds a factor b smaller than those for conventional algorithms. More significantly, by storing C and D in precision u_{high}, the error bounds are reduced from $O(nu_{\text{low}})$ to $cu_{\text{low}} + O(nu_{\text{high}})$, where c is independent of the problem size n. Assuming $u_{\text{high}} \ll u_{\text{low}}$, we obtain bounds that are independent of n to first order, which suggests we can obtain more accurate results than for algorithms with only one precision, u_{low}.

We applied our analysis to the tensor core units available in the NVIDIA Volta and Turing GPUs, which are specific block FMA units with $b = 4$ and with fp16 and fp32 precisions. We compared two variants, TC16 and TC32, which differ in the precision used by the FMA to accumulate the operations. Our analysis predicts the TC32 variant to be much more accurate than the TC16 one; our numerical experiments confirm this prediction and moreover show that the accuracy boost is achieved with almost no performance loss.

REFERENCES

[1] Jeremy Appleyard and Scott Yokim. Programming tensor cores in CUDA 9. https://devblogs.nvidia.com/programming-tensor-cores-cuda-9/, October 2017. Accessed March 25, 2019.

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

[3] Ian Buck. World's fastest supercomputer triples its performance record. https://blogs.nvidia.com/blog/2019/06/17/hpc-ai-performance-record-summit/, June 2019. Accessed June 24, 2019.

[4] Erin Carson and Nicholas J. Higham. A new analysis of iterative refinement and its application to accurate solution of ill-conditioned sparse linear systems. *SIAM J. Sci. Comput.*, 39(6): A2834–A2856, 2017.

[5] Erin Carson and Nicholas J. Higham. Accelerating the solution of linear systems by iterative refinement in three precisions. *SIAM J. Sci. Comput.*, 40(2):A817–A847, 2018.

[6] Michael Feldman. Fujitsu reveals details of processor that will power Post-K supercomputer. https://www.top500.org/news/fujitsu-reveals-details-of-processor-that-will-power-post-k-supercomputer, August 2018. Accessed November 22, 2018.

[7] Michael Feldman. IBM takes aim at reduced precision for new generation of AI chips. https://www.top500.org/news/ibm-takes-aim-at-reduced-precision-for-new-generation-of-ai-chips/, December 2018. Accessed January 8, 2019.

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

[9] Azzam Haidar, Stanimire Tomov, Jack Dongarra, and Nicholas J. Higham. Harnessing GPU tensor cores for fast FP16 arithmetic to speed up mixed-precision iterative refinement solvers. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage, and Analysis*, SC '18 (Dallas, TX), Piscataway, NJ, USA, 2018, pages 47:1–47:11. IEEE Press.

[10] Azzam Haidar, Panruo Wu, Stanimire Tomov, and Jack Dongarra. Investigating half precision arithmetic to accelerate dense linear system solvers. In *Proceedings of the 8th Workshop on Latest Advances in Scalable Algorithms for Large-Scale Systems*, ScalA '17 (Denver, CO), November 2017, pages 10:1–10:8.

[11] Nicholas J. Higham. *Accuracy and Stability of Numerical Algorithms*. Second edition, Society for Industrial and Applied Mathematics, Philadelphia, PA, USA, 2002. xxx+680 pp. ISBN 0-89871-521-0.

[12] Nicholas J. Higham and Theo Mary. A new approach to probabilistic rounding error analysis. *SIAM J. Sci. Comput.*, 41(5):A2815–A2835, 2019.

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

[14] Intel Corporation. BFLOAT16—hardware numerics definition, November 2018. White paper. Document number 338302-001US.

[15] Stefano Markidis, Steven Wei Der Chien, Erwin Laure, Ivy Bo Peng, and Jeffrey S. Vetter. NVIDIA tensor core programmability, performance & precision. In *2018 IEEE International Parallel and Distributed Processing Symposium Workshops (IPDPSW)*, May 2018, pages 522–531.

[16] ORNL launches Summit supercomputer. https://www.ornl.gov/news/ornl-launches-summit-supercomputer, June 2018. Accessed June 30, 2018.

[17] Naveen Rao. Beyond the CPU or GPU: Why enterprise-scale artificial intelligence requires a more holistic approach. https://newsroom.intel.com/editorials/artificial-intelligence-requires-holistic-approach, May 2018. Accessed November 5, 2018.

[18] Anton Shilov. Intel architecture manual updates: bfloat16 for Cooper Lake Xeon scalable only? https://www.anandtech.com/show/14179/intel-manual-updates-bfloat16-for-cooper-lake-xeon-scalable-only, April 2019. Accessed May 22, 2019.

[19] Summit by the numbers. https://www.olcf.ornl.gov/wp-content/uploads/2018/06/Summit_bythenumbers_FIN.png, June 2018. Accessed June 30, 2018.

[20] Shibo Wang and Pankaj Kanwar. BFloat16: the secret to high performance on cloud TPUs. https://cloud.google.com/blog/products/ai-machine-learning/bfloat16-the-secret-to-high-performance-on-cloud-tpus, August 2019. Accessed September 14, 2019.