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**

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 detailed 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, and they show that the accuracy boost is obtained with almost no performance loss.

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 format[1] [23]. Examples of such hardware include the NVIDIA P100 and V100 GPUs, the AMD Radeon Instinct MI25 GPU, Google's Tensor Processing Units (TPUs), the ARM NEON architecture [3], and the Fujitsu A64FX ARM processor [10]. Expected to join them in the near future are IBM's next generation AI chips [11] (supporting an 8-bit floating-point format in addition to fp16), and Intel's upcoming Xeon Cooper Lake [31] and Nervana Neural Network processors [28].

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 [12], [13], [14], [15] it is shown how on an NVIDIA V100, fp16 arithmetic can be

[*]Version of February 22, 2020. **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.

[†]Department of Mathematics, The University of Manchester, Manchester M13 9PL, UK (`pierre.blanchard00@gmail.com`).

[‡]Department 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`).

[¶]Sorbonne Université, CNRS, LIP6, F-75005 Paris, France (`theo.mary@lip6.fr`).

[‖]Department 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

Table 1.1: Parameters and year of announcement for block FMAs on current or future hardware. Source: [16].

		b_1	b	b_2	u_{low}	u_{high}
Google TPU v1	2016	256	256	256	bfloat16	fp32
Google TPU v2	2017	128	128	128	bfloat16	fp32
NVIDIA Volta	2017	4	4	4	fp16	fp32
Intel NNP-T	2018	32	32	32	bfloat16	fp32
Armv8-A	2019	2	4	2	bfloat16	fp32

used 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. Similar improvements over a single precision solver for complex systems are obtained in [1]. Moreover, the same iterative refinement approach running on Summit [27], [32], the machine with 4608 nodes with 6 NVIDIA V100 GPUs per node that leads the November 2019 TOP 500 list,[2] has achieved a performance of 550 petaflops [13].

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 in precision fp32 [2]. 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

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

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 [34]. They use bfloat16 rather than fp16 as the low precision format and they operate on square matrices of dimension 128 or 256. However, Google TPUs are not commercially available, and details of computation in MXUs has not been made publicly available. Other vendors, including Intel and Arm, have announced hardware that will use block FMA units; see Table 1.1.

Block FMAs are inherently mixed precision units. Existing rounding error analyses will be pessimistic when applied to computations with block FMAs, as they do not reflect the mixed precision nature of the computation.

In this work we define a mixed precision block FMA that includes the forms provided by the hardware in Table 1.1 as special cases 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

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

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 storing C and D in the block FMA 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.

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. With a standard abuse of notation we will write fl_p with an argument that is a matrix expression to denote that the expression is evaluated at some specified precision (possibly different from p) and then rounded to precision p. 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. A scalar FMA has the form $d = c + ab$ and is computed as the correctly rounded exact result, that is, with one rounding error rather than two [17, sect. 2.6]. Hardware implementing an FMA at the same speed as a single addition or multiplication first became available over twenty years ago.

We define a mixed precision block FMA to take as input matrices $A \in \mathbb{R}^{b_1\times b}$, $B \in \mathbb{R}^{b\times b_2}$, and $C \in \mathbb{R}^{b_1\times b_2}$, where A and B are provided in a given precision u_{low} and C is either in precision u_{low} or in a higher precision u_{high}. The block FMA computes

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

and returns D in precision u_{low} or u_{high}. A key point is that the output D can be taken in precision u_{high} and used as the input C to a subsequent FMA. By chaining FMAs together in this way, larger matrix products can be computed, as we will show in the next section. Table 1.1 gives the precisions and matrix dimensions for block FMAs that are currently available or have been announced.

There is a spectrum of ways in which D can be computed. With $\mathrm{fl}_{\mathrm{FMA}}$ denoting $\mathrm{fl}_{\mathrm{low}}$ or $\mathrm{fl}_{\mathrm{high}}$, two extreme possibilities are

$$D = \mathrm{fl}_{\mathrm{FMA}}(C + AB) \qquad \text{(rounded exact result)}, \tag{2.1}$$

$$D = \mathrm{fl}_{\mathrm{FMA}}\Big(\underbrace{C + AB}_{\text{precision } \overline{u}}\Big) \qquad (C + AB \text{ is computed at precision } \overline{u}). \tag{2.2}$$

In (2.1) every element of D is computed with just a single rounding error. In (2.2), the expression $C + AB$ is computed at a precision $\overline{u}$ equal to u_{low} or u_{high} and there are multiple rounding errors per element of D. The evaluation (2.1) is ideal from an accuracy perspective, but in general it will be expensive. The NVIDIA Volta and Turing architectures use (2.2) with $\overline{u} = u_{16}$ or $\overline{u} = u_{32}$ [2].

The evaluation (2.1) with all inputs and the output at precision u_{low} corresponds to the "long accumulator" proposed by Kulisch and Miranker [25], which has found use in interval arithmetic (see, for example, [24], [29]). Evaluating according to (2.1) is expensive, and although proposals have been made for implementing it in hardware (e.g., [5], [33]), it is not, to our knowledge, supported in commercial processors because of the hardware costs. However, manufacturers could implement something between (2.1) and (2.2) by using a little extra precision, perhaps the extended precisions defined in the IEEE standard [22] or the 80-bit registers on Intel processors. Indeed we note that NVIDIA's CUDA C++ Programming Guide [9] states that when C is in fp32 the computation is performed in *at least* single precision, allowing for the possibility of using extra precision in future implementations.

In order to capture the range of possible block FMA implementations in our analysis, we will use a model for the evaluation that includes a range of possibilities that has (2.1) and (2.2) as the extreme cases. Denote the precision of the output of the FMA by u_{FMA}, which is either u_{low} or u_{high}. We assume that $C + AB$ is computed at precision $\overline{u} \le u_{\text{FMA}}$ and then rounded to precision u_{FMA}; if $\overline{u} = u_{\text{FMA}}$ then the latter rounding is not needed.

We will use the standard model of floating-point arithmetic [17, sec. 2.2]

$$\text{fl}(a \text{ op } b) = (a \text{ op } b)(1+\delta), \quad |\delta| \le u, \quad \text{op} \in \{+,-,\times,/\} \tag{2.3}$$

and the alternative form

$$\text{fl}(a \text{ op } b) = \frac{a \text{ op } b}{1+\delta}, \quad |\delta| \le u, \quad \text{op} \in \{+,-,\times,/\}. \tag{2.4}$$

We will need the quantity

$$\gamma_k = \frac{ku}{1-ku}.$$

Throughout the paper we assume that $ku < 1$ for the relevant integers k and precisions u. An accent on γ denotes that u carries that accent and a superscript on γ denotes that u also carries that superscript as a subscript; thus below we will use $\overline{\gamma}_k = k\overline{u}/(1-k\overline{u})$ and $\widetilde{\gamma}_k^{\text{FMA}} = k\widetilde{u}_{\text{FMA}}/(1-k\widetilde{u}_{\text{FMA}})$.

By standard analysis for matrix multiplication [17, sect. 3.5], the matrix $\widetilde{D}$ computed in precision $\overline{u}$ satisfies

$$|\widetilde{D} - D| \le \overline{\gamma}_{b+1}(|C| + |A||B|).$$

Using (2.4), the final rounding to precision u_{FMA} gives $\widehat{D}$ satisfying

$$(1+\delta_{ij})\widehat{d}_{ij} = \widetilde{d}_{ij} \quad |\delta_{ij}| \le u_{\text{FMA}}, \quad i = 1:b_1, \ j = 1:b_2. \tag{2.5}$$

Hence

$$|\widehat{D} - D| \le |\widehat{D} - \widetilde{D}| + |\widetilde{D} - D| \le u_{\text{FMA}}|\widehat{D}| + \overline{\gamma}_{b+1}(|C| + |A||B|).$$

Our model is therefore

$$|\widehat{D} - D| \le u_{\text{FMA}}|\widehat{D}| + \overline{\gamma}_{b+1}(|C| + |A||B|). \tag{2.6}$$

Setting $\overline{u} = 0$ corresponds to (2.1) (to within $O(u_{\text{FMA}}^2)$ because of the presence of $\widehat{D}$ instead of D on the right-hand side), while $\overline{u} = u_{\text{high}}$ corresponds to (2.2). While the model (2.6) is useful, it is not so convenient when chaining FMAs, so in the next section we will directly analyze a chain of FMAs as it arises in Algorithm 3.1 for matrix multiplication.

For $b = b_1 = b_2$ we will refer to the block FMA as a $b \times b$ FMA.

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, $u_{\text{low}} = u_{16}$), C and D can be either fp16 or fp32 (that is, $u_{\text{high}} = u_{32}$). We will consider two cases:

$$\widehat{D} = \begin{cases} \mathrm{fl}_{16}(C^{(16)} + AB), & \text{case 1: TC16}, \\ \mathrm{fl}_{32}(C^{(32)} + AB), & \text{case 2: TC32}, \end{cases} \tag{2.7}$$

where $C^{(16)}$ and $C^{(32)}$ denote an fp16 matrix and an fp32 matrix, respectively. We note that the NVIDIA CUDA C++ Programming Guide [9] states that

> Element-wise multiplication of matrix A and B is performed with at least single precision. When .ctype or .dtype is .f32, accumulation of the intermediate values is performed with at least single precision. When both .ctype and .dtype are specified as .f16, the accumulation is performed with at least half precision.

We therefore have $\overline{u} = u_{16}$ for TC16 and $\overline{u} = u_{32}$ for TC32. (For TC16, the multiplications are actually performed in precision u_{32}, but taking this into account makes the analysis more complicated and barely improves the resulting bounds). The computed $\widehat{D}$ satisfies (2.6) with $u_{\text{FMA}} = u_{16}$ for TC16 and $u_{\text{FMA}} = u_{32}$ for TC32.

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.

Algorithm 3.1 Let $A \in \mathbb{R}^{m\times n}$ and $B \in \mathbb{R}^{n\times t}$ be partitioned into $b_1 \times b$ blocks A_{ij} and $b \times b_2$ blocks B_{ij}, respectively, where $p = m/b_1$, $q = n/b$, and $r = t/b_2$ are assumed to be integers. This algorithm performs the matrix multiplication $C = AB$ using a block FMA. The output C is in precision u_{FMA}, which is either u_{high} or u_{low}.

```
1:  $\widetilde{A} \leftarrow \mathrm{fl}_{\text{low}}(A)$ and $\widetilde{B} \leftarrow \mathrm{fl}_{\text{low}}(B)$ (if necessary).
2:  for $i = 1\colon p$ do
3:      for $j = 1\colon r$ do
4:          $C_{ij} = 0$
5:          for $\ell = 1\colon q$ do
6:              Compute $C_{ij} = C_{ij} + \widetilde{A}_{i\ell}\widetilde{B}_{\ell j}$ using a block FMA with output at precision $u_{\text{FMA}}$.
7:          end for
8:          If $u > u_{\text{FMA}}$ then round $C_{ij}$ to precision $u$.
9:      end for
10: end for
```

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 [13], [14], for example, they may be given 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 [21] 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. We recall from standard error analysis [17, chap. 3] that the computed value of an expression $w = p_0 + p_1q_1 + \cdots + p_nq_n$ evaluated in precision u from left to right can be expressed as

$$\widehat{w} = p_0(1+\theta_n) + p_1q_1(1+\theta_{n+1}) + p_2q_2(1+\theta'_n) + p_3q_3(1+\theta_{n-1}) + \cdots + p_nq_n(1+\theta_2),$$

where $|\theta_k| \le \gamma_k$.

Let x denote a row of A and y a column of B and write $s_n = x^Ty$ as

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

Algorithm 3.1 evaluates s_n by setting $s_0 = 0$, forming

$$s_i = s_{i-1} + x_{(i-1)b+1}y_{(i-1)b+1} + \cdots + x_{ib}y_{ib}, \quad i = 1:q, \tag{3.1}$$

where the right-hand side is evaluated in the block FMA at precision $\overline{u}$ and we assume that the evaluation is from left to right, then rounding the result back to precision u_{FMA}, or to precision u if $u > u_{\mathrm{FMA}}$. Since the block FMA produces output only at precision u_{low} or u_{high}, two roundings will be needed if $u > u_{\mathrm{FMA}} > \overline{u}$: first to precision u_{FMA} and then to precision u. We will include only a single rounding in our analysis, for simplicity; this has a negligible affect on the final error bound (see [30] for analysis of the effects of double rounding). The computed $\widehat{s}_i$ satisfies

$$\widehat{s}_i = \big[\widehat{s}_{i-1}\big(1+\theta_b^{(i)}\big) + x_{(i-1)b+1}y_{(i-1)b+1}\big(1+\theta_{b+1}^{(i)}\big) + \cdots + x_{ib}y_{ib}\big(1+\theta_2^{(i)}\big)\big](1+\delta_i) \tag{3.2}$$

where $|\theta_j| \le \overline{\gamma}_j$ and $|\delta_j| \le \widetilde{u}_{\mathrm{FMA}}$, where

$$\widetilde{u}_{\mathrm{FMA}} = \begin{cases} u, & u > u_{\mathrm{FMA}}, \\ 0, & \overline{u} \ge u_{\mathrm{FMA}}, \\ u_{\mathrm{FMA}}, & \overline{u} < u_{\mathrm{FMA}}. \end{cases} \tag{3.3}$$

Here, the first condition is not mutually exclusive with the second and third, so the equality should be read as taking the first choice (u) if the first condition is satisfied. For $i = 1$, since $s_0 = 0$, (3.2) holds with $b+1$ replaced by b in the θ term multiplying x_1y_1. Overall, we have

$$\begin{aligned}\widehat{s}_n &= \Big(x_1y_1\big(1+\theta_b^{(1)}\big) + \cdots + x_by_b\big(1+\theta_2^{(1)}\big)\Big)\prod_{i=2}^{q}\big(1+\theta_b^{(i)}\big)\prod_{i=1}^{q}(1+\delta_i) \\ &\qquad + \cdots + \Big(x_{n-b+1}y_{n-b+1}\big(1+\theta_{b+1}^{(q)}\big) + \cdots + x_ny_n\big(1+\theta_2^{(q)}\big)\Big)(1+\delta_q) \\ &= \sum_{i=1}^{n} x_iy_i(1+\alpha_i)(1+\beta_i),\end{aligned}$$

where, by [17, Lems. 3.1, 3.3] and using $qb = n$, $|\alpha_i| \le \widetilde{\gamma}_q^{\mathrm{FMA}}$ and $|\beta_i| \le \overline{\gamma}_n$. Hence

$$|s_n - \widehat{s}_n| \le \big(\widetilde{\gamma}_q^{\mathrm{FMA}} + \overline{\gamma}_n + \widetilde{\gamma}_q^{\mathrm{FMA}}\overline{\gamma}_n\big)|x|^T|y|, \tag{3.4}$$

Table 3.1: First order part of constant in (3.5) in particular cases. Here, we assume that A and B are given in precision u_{low} and that $u \le u_{\text{FMA}}$.

u_{FMA}	$\overline{u}$	$\widetilde{u}_{\text{FMA}}$	Bound
u_{low}	u_{low}	0	nu_{low}
u_{low}	u_{high}	u_{low}	$qu_{\text{low}} + nu_{\text{high}}$
u_{low}	0	u_{low}	qu_{low}
u_{high}	u_{low}	0	nu_{low}
u_{high}	u_{high}	0	nu_{high}
u_{high}	0	u_{high}	qu_{high}

We note that if (3.1) is evaluated from right to left—which amounts to blocked summation [4]—then s_1 (for example) participates in q rather than n additions and so $\overline{\gamma}_n$ can be replaced by $\overline{\gamma}_{q+b-1}$ in (3.4). It is not hard to see that (3.4) is valid for all orders of evaluation of (3.1).

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 $q = n/b$. The computed $\widehat{C}$ satisfies*

$$|C - \widehat{C}| \le (\widetilde{\gamma}_q^{\text{FMA}} + \overline{\gamma}_n + \widetilde{\gamma}_q^{\text{FMA}}\overline{\gamma}_n)|A||B|. \tag{3.5}$$

Table 3.1 shows the first order part of the constant in the bound (3.5) in several particular cases. Let us compare the constants in the table with the constant nu_{low} in the bound for a standard evaluation of C at precision u_{low}. The case $u_{\text{FMA}} = \overline{u} = u_{\text{low}}$ has the same constant, so the block FMA offers no accuracy benefits in this case. But when $u_{\text{FMA}} = \overline{u} = u_{\text{high}}$ the constant is a factor $u_{\text{high}}/u_{\text{low}}$ smaller than for the standard evaluation, while for $u_{\text{FMA}} = u_{\text{low}}$ and $\overline{u} = u_{\text{high}}$ the constant is a factor b smaller.

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

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

Applying (3.5) to the computation of $C = \widetilde{A}\widetilde{B}$ yields

$$\widehat{C} = \widetilde{A}\widetilde{B} + \Delta C, \qquad |\Delta C| \le (\widetilde{\gamma}_q^{\text{FMA}} + \overline{\gamma}_n + \widetilde{\gamma}_q^{\text{FMA}}\overline{\gamma}_n)|\widetilde{A}||\widetilde{B}|,$$

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 $q = n/b$. The computed $\widehat{C}$ satisfies*

$$|C - \widehat{C}| \le \big(2u_{\text{low}} + u_{\text{low}}^2 + (\widetilde{\gamma}_q^{\text{FMA}} + \overline{\gamma}_n + \widetilde{\gamma}_q^{\text{FMA}}\overline{\gamma}_n)(1 + u_{\text{low}})^2\big)|A||B|. \tag{3.6}$$

Table 3.2: Dominant terms in the error constant multiplying $|A||B|$ for standard matrix multiplication and for the block FMA-based Algorithm 3.1, from (3.6). Here, $q = n/b$, $u \le u_{\mathrm{FMA}}$ is assumed, and A and B are not assumed to be given in precision u_{low}.

Evaluation method			Bound
Standard in precision u_{low}			$(n+2)u_{\mathrm{low}}$
Block FMA	$u_{\mathrm{FMA}} = u_{\mathrm{low}}$	$\overline{u} = u_{\mathrm{low}}$	$(n+2)u_{\mathrm{low}}$
Block FMA	$u_{\mathrm{FMA}} = u_{\mathrm{low}}$	$\overline{u} = u_{\mathrm{high}}$	$(q+2)u_{\mathrm{low}} + nu_{\mathrm{high}}$
Block FMA	$u_{\mathrm{FMA}} = u_{\mathrm{low}}$	$\overline{u} = 0$	$(q+2)u_{\mathrm{low}}$
Block FMA	$u_{\mathrm{FMA}} = u_{\mathrm{high}}$	$\overline{u} = u_{\mathrm{low}}$	$(n+2)u_{\mathrm{low}}$
Block FMA	$u_{\mathrm{FMA}} = u_{\mathrm{high}}$	$\overline{u} = u_{\mathrm{high}}$	$2u_{\mathrm{low}} + nu_{\mathrm{high}}$
Block FMA	$u_{\mathrm{FMA}} = u_{\mathrm{high}}$	$\overline{u} = 0$	$2u_{\mathrm{low}} + qu_{\mathrm{high}}$
Standard in precision u_{high}			nu_{high}

The bound (3.6) should be compared with [17, eq. (3.13)]

$$|C - \widehat{C}| \le \gamma_n |A||B| \tag{3.7}$$

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

We gather in Table 3.2 the dominant terms in the error bounds (3.6) for five block FMA implementations, with A and B given in precision u_{high}, and compare them with the bounds for standard multiplication in precision u_{high} or u_{low} (error bound (3.7)). For the sake of readability, in Table 3.2 we have assumed that $u_{\mathrm{high}} \ll u_{\mathrm{low}}$.

Let us now compare these five different variants, using Table 3.2. Compared with the standard multiplication in precision u_{low}, the mixed-precision block FMA multiplication achieves smaller error bounds in all cases except for $u_{\mathrm{FMA}} = \overline{u} = u_{\mathrm{low}}$. Let us first consider $\overline{u} = u_{\mathrm{high}}$. With $u_{\mathrm{FMA}} = u_{\mathrm{low}}$, the bound is reduced by a factor approximately b, while with $u_{\mathrm{FMA}} = u_{\mathrm{high}}$, the factor of improvement is even larger and equal to $\min(n/2, u_{\mathrm{low}}/u_{\mathrm{high}})$. Indeed, the bound (3.6) does not grow with n to first order (that is, as long as $n \le 2u_{\mathrm{low}}/u_{\mathrm{high}}$) and, for larger n, the bound becomes equivalent to the bound for standard multiplication in precision u_{high}. With $\overline{u} = 0$, the bounds are even smaller: for $u_{\mathrm{FMA}} = u_{\mathrm{low}}$ the improvement is negligible since it amounts to removal of the nu_{high} term, while for $u_{\mathrm{FMA}} = u_{\mathrm{high}}$ and $nu_{\mathrm{high}} \gg 2u_{\mathrm{low}}$ the bound is reduced by a factor approximately b.

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 [18]. A probabilistic analogue of Theorem 3.2 directly follows from [18, Thm. 3.1]. It contains a modified version of (3.6) with $\widetilde{\gamma}_q^{\mathrm{FMA}}$ and $\overline{\gamma}_n$ replaced by relaxed constants $\widetilde{\gamma}_q^{\mathrm{FMA}}(\lambda)$ and $\overline{\gamma}_n(\lambda)$ proportional to $\lambda q^{1/2}\widetilde{u}_{\mathrm{FMA}}$ and $\lambda n^{1/2}\overline{u}$, respectively. This relaxed bound 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 $u_{\mathrm{FMA}} = u_{\mathrm{high}} = \overline{u}$ may only start growing with n for much larger n than the worst-case bound suggests (perhaps for n larger than $4(u_{\mathrm{low}}/u_{\mathrm{high}})^2$).

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 Theorem 3.1 and Theorem 3.2 (or Table 3.2). We gather the resulting bounds in Table 3.3, where we consider standard half and

Table 3.3: Specialization of the bounds in Theorem 3.1 and Theorem 3.2 (or Table 3.2) for NVIDIA tensor cores, for which $b = 4$, $u_{\text{high}} = u_{32}$, and $u_{\text{low}} = u_{16}$, and where $u_{\text{FMA}} = \overline{u} = u_{16}$ (TC16) or $u_{\text{FMA}} = \overline{u} = u_{32}$ (TC32).

Precision of A and B	Standard fp16	Tensor core, TC16 $u_{\text{FMA}} = \overline{u} = u_{16}$	Tensor core, TC32 $u_{\text{FMA}} = \overline{u} = u_{32}$	Standard fp32
u_{16}	nu_{16}	nu_{16}	nu_{32}	nu_{32}
u_{32}	$(n+2)u_{16}$	$(n+2)u_{16}$	$2u_{16} + nu_{32}$	nu_{32}

single precision multiplication (fp16 and fp32), and tensor core multiplication in either TC16 ($u_{\text{FMA}} = \overline{u} = u_{16}$) or TC32 ($u_{\text{FMA}} = \overline{u} = u_{32}$) mode. We see clearly that the TC32 mode achieves a much smaller bound than the TC16 mode, by removing the constant n multiplying the u_{16} term and relegating it to the u_{32} term.

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 componentwise error $\max_{i,j} |\widehat{C} - C|_{ij}/(|A||B|)_{ij}$ 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. 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 the 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 $n \approx 10^6$. Interestingly, TC16 achieves a noticeably smaller error than fp16, even though they both have the same error bound. We suspect this might be because the matrix multiplication algorithm of cuBLAS implements blocked summation which, as mentioned previously, achieves a bound a factor $b = 4$ smaller. More importantly, the errors for TC32 and fp32 are much smaller than those for fp16 and TC16, by a factor 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 of the error with n. Finally, we mention that the fact that the error *decreases* with n in Figure 3.2 is related to the entries of the matrices being distributed uniformly in $[-1, 1]$ and thus having zero mean, as explained in the recent probabilistic analysis for random data [19].

In order to give a baseline for the performance of each matrix multiplication

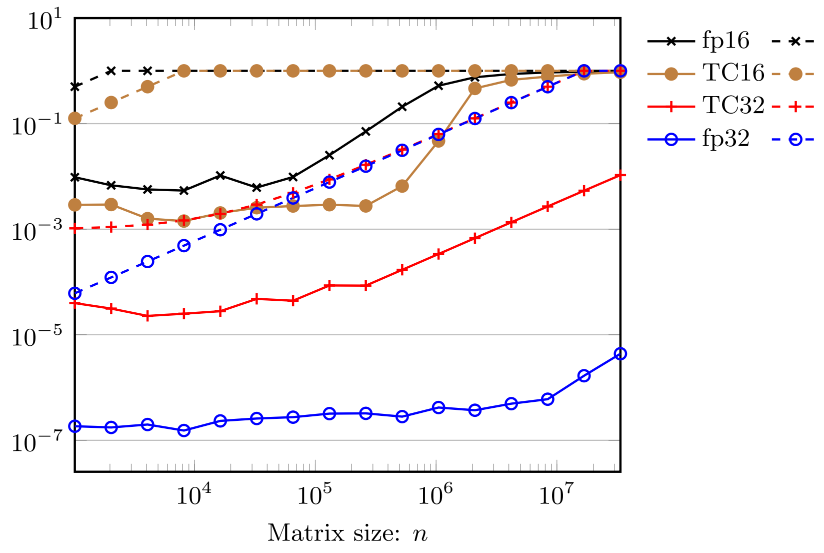

Fig. 3.1: Componentwise errors for four matrix multiplication variants on NVIDIA V100 GPU (fp16, TC16, TC32, and fp32) where matrix entries are sampled uniformly in $[0, 10^{-3}]$. Solid lines represent errors while dashed lines represent error bounds.

method we run additional simulations for square matrices (with uniform entries). Figure 3.3 shows the maximum flop rate out of five runs (in TFlops) 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 (about 90% of the theoretical performance, namely 112.7 TFlops) for $n = 8000$. Our performance results are in good agreement with other existing benchmarks, e.g., [26].

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.

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* $B = A - \sum_{j=1}^{q} X_j Y_j$, *where* $A, B, X_j, Y_j \in \mathbb{R}^{b \times b}$ *are given*

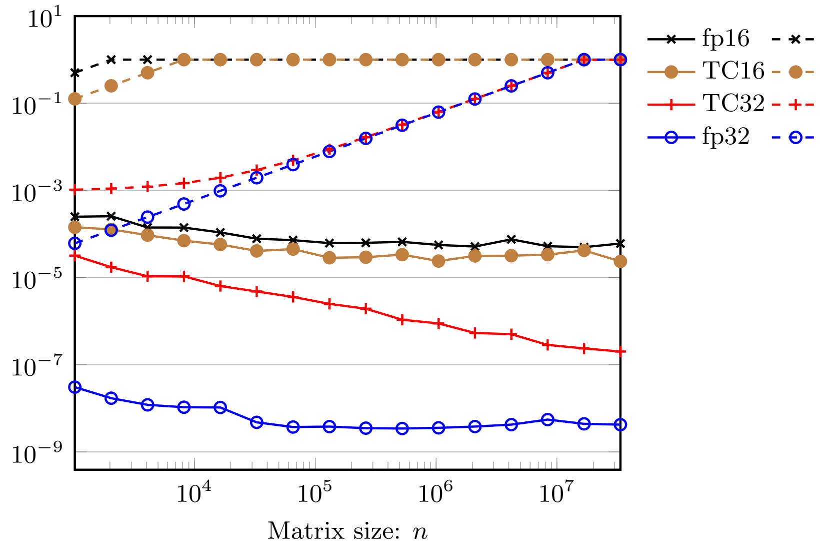

Fig. 3.2: Componentwise errors for four matrix multiplication variants on NVIDIA V100 GPU (fp16, TC16, TC32, and fp32) where matrix entries are sampled uniformly in $[-1, 1]$. Solid lines represent errors while dashed lines represent error bounds.

Algorithm 4.1 Let $A \in \mathbb{R}^{n\times n}$ be given in precision u and partitioned into $b \times b$ blocks A_{ij}, 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
```

in precision u_{high}, *be computed with a* $b \times b$ *FMA. The computed* $\widehat{B}$ *satisfies*

$$|\widehat{B} - B| \le (\widetilde{\gamma}_q^{\mathrm{FMA}} + \overline{\gamma}_{n+1} + \widetilde{\gamma}_q^{\mathrm{FMA}}\overline{\gamma}_{n+1})\Bigl(|A| + \sum_{j=1}^{q}|X_j||Y_j|\Bigr). \tag{4.1}$$

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, where the $+1$ subscript on $\overline{\gamma}$ comes from the initial subtraction with A. ∎

We also need the following lemma for the next theorem.

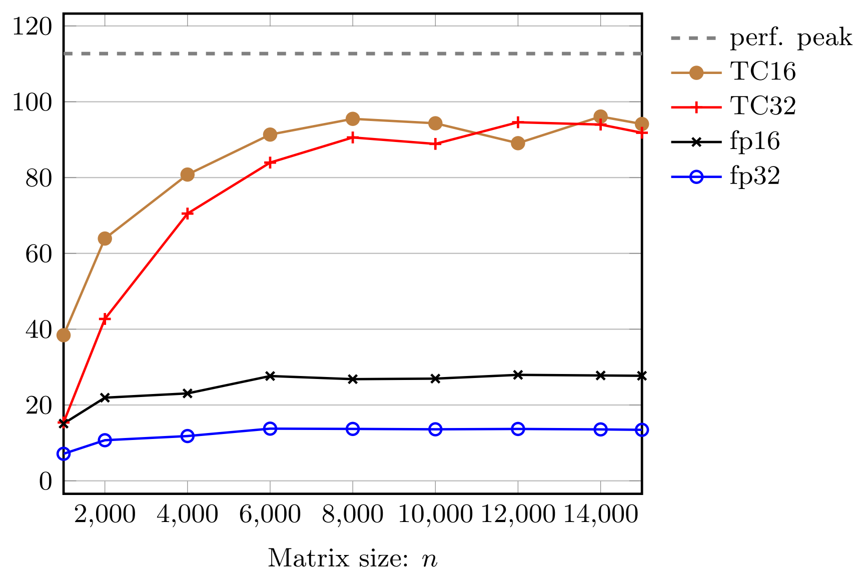

Fig. 3.3: Performance results (in TFlops) for matrix multiplication variants on NVIDIA V100 GPU (fp16, TC16, TC32 and fp32) with square matrices.

Lemma 4.2. *If the LU factorization of $A \in \mathbb{R}^{b\times b}$ runs to completion then the computed LU factors $\widehat{L}$ and $\widehat{U}$ satisfy*

$$\widehat{L}\widehat{U} = A + \Delta A, \quad |\Delta A| \le \gamma_b |\widehat{L}||\widehat{U}|. \tag{4.2}$$

Moreover, the computed solution $\widehat{X}$ to the multiple right-hand side system $TX = B$, where $T \in \mathbb{R}^{b\times b}$ is nonsingular and triangular, and $B \in \mathbb{R}^{b\times c}$, satisfies

$$T\widehat{X} = B + \Delta B, \quad |\Delta B| \le \gamma_b |T||\widehat{X}|. \tag{4.3}$$

Proof. See [17, Thm. 9.3] and [17, Thm. 8.5]. ∎

Theorem 4.3. *Let $A \in \mathbb{R}^{n\times n}$ be partitioned in $b \times b$ blocks, with $q = n/b$. 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*

$$\begin{aligned}|\Delta A| \le &\Big(2u_{\text{low}} + u_{\text{low}}^2 + \max(\widetilde{\gamma}_{q-1}^{\text{FMA}} + \overline{\gamma}_{n-b+1} + \widetilde{\gamma}_{q-1}^{\text{FMA}}\overline{\gamma}_{n-b+1}, \gamma_b)(1+u_{\text{low}})^2\Big)\\ &\times (|A| + |\widehat{L}||\widehat{U}|).\end{aligned} \tag{4.4}$$

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, since $k \le q$,

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

By (4.3) we have

$$|\widehat{L}_{ik}\widehat{U}_{kk} - \widehat{R}_{ik}| \le \gamma_b|\widehat{L}_{ik}||\widehat{U}_{kk}|. \tag{4.5}$$

Combining these two inequalities gives

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

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

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

where

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

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 $i > k$,

$$|A_{ik} - \sum_{j=1}^{k}\widehat{L}_{ij}\widehat{U}_{jk}| \le \Big(2u_{\mathrm{low}} + u_{\mathrm{low}}^2 + \max(\widetilde{\gamma}_{q-1}^{\mathrm{FMA}} + \overline{\gamma}_{n-b+1} + \widetilde{\gamma}_{q-1}^{\mathrm{FMA}}\overline{\gamma}_{n-b+1}, \gamma_b) \times (1+u_{\mathrm{low}})^2\Big) \times \Big(|A_{ij}| + \sum_{j=1}^{k}|\widehat{L}_{ij}||\widehat{U}_{jk}|\Big). \tag{4.6}$$

For $i = k$, L_{kk} is determined with U_{kk} on line 2 of Algorithm 4.1, and by (4.2) we have $|\widehat{L}_{kk}\widehat{U}_{kk} - \widehat{R}_{kk}| \le \gamma_b|\widehat{L}_{kk}||\widehat{U}_{kk}|$. Therefore (4.5) holds for $i = k$, too, and hence so does (4.6). In a similar way, the inequality (4.6) can be shown to hold for $i < k$. ◻

THEOREM 4.4. *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*

$$|\Delta A| \le \Big(2u_{\mathrm{low}} + u_{\mathrm{low}}^2 + \max(\widetilde{\gamma}_{q-1}^{\mathrm{FMA}} + \overline{\gamma}_{n-b+1} + \widetilde{\gamma}_{q-1}^{\mathrm{FMA}}\overline{\gamma}_{n-b+1}, \gamma_b)(1+u_{\mathrm{low}})^2 + 2\gamma_n + \gamma_n^2\Big) \times (|A| + |\widehat{L}||\widehat{U}|). \tag{4.7}$$

Proof. The result is obtained by combining Theorem 4.3 with the error analysis for the solution of triangular systems [17, Thm. 8.5] and is analogous to the proof of [17, Thm. 9.4]. ◻

Table 4.1: Dominant terms in the error constant $c(n, u_{16}, u_{32})$ in the backward error bound $|\Delta A| \le c(n, u_{16}, u_{32})(|A| + |\widehat{L}||\widehat{U}|)$ in (4.4) and (4.7) for LU factorization and the solution of $Ax = b$ using NVIDIA tensor cores. We have taken $u = u_{16}$ for fp16 and TC16, and $u = u_{32}$ for fp32 and TC32.

	fp16	TC16	TC32	fp32
LU (Thm. 4.3)	nu_{16}	$(n-1)u_{16}$	$2u_{16} + (n-3)u_{32}$	nu_{32}
$Ax = b$ (Thm. 4.4)	$3nu_{16}$	$(3n-1)u_{16}$	$2u_{16} + (3n-3)u_{32}$	$3nu_{32}$

We gather in Table 4.1 the dominant terms in the error bound for LU factorization and the solution of linear systems using NVIDIA tensor cores, for which $b = 4$, $\overline{u} = u_{\text{high}} = u_{32}$, and $u_{\text{low}} = u_{16}$. We distinguish the same four variants of matrix multiplication as in Table 3.3. In the fp16 and fp32 cases, we naturally take the working precision to be $u = u_{16}$ and $u = u_{32}$, respectively, and the bounds are the standard ones [17, Thms. 9.3, 9.4]. In the TC16 case, both $u = u_{16}$ and $u = u_{32}$ are possible, but since the FMA uses $u_{\text{FMA}} = \overline{u} = u_{16}$, 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 $u = u_{\text{FMA}} = \overline{u} = u_{32}$. We have $\widetilde{u}_{\text{FMA}} = 0$ in (3.3) in both cases.

Overall, these bounds lead to the same conclusions as in the matrix multiplication case: the TC16 bound is almost identical to the fp16 one and exhbits linear growth with n, while the TC32 variant leads to a much smaller bound, which only starts growing with n when $n \gtrsim 2u_{16}/u_{32} = 16384$ (LU factorization) or when $n \gtrsim \frac{2}{3}u_{16}/u_{32} \approx 5461$ (linear system), at which point it is almost equivalent to the fp32 bound.

Our analysis is applicable to the work in Haidar et al. [14], 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 [6], [7]. In the experiments reported in [14], 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 LU factorization error constant for TC32 is a factor ranging from approximately 900 to 5500 smaller than that for fp16 for the matrix sizes $n \in [2000, 34000]$ used in those experiments.

4.2. Numerical experiments with tensor core LU factorization.

4.2. 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 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 from the Haar distribution and D is diagonal, with $d_{ii} =$

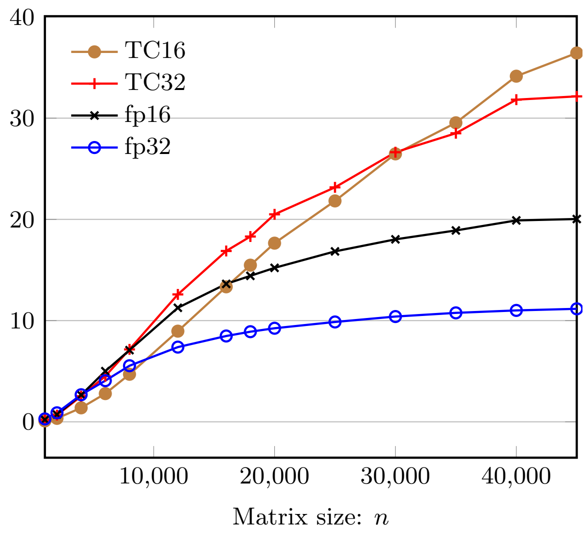

Fig. 4.1: Performance in Tflop of the LU factorization computed by Algorithm 4.1 on an NVIDIA V100 GPU for the four variants fp16, TC16, TC32, and fp32.

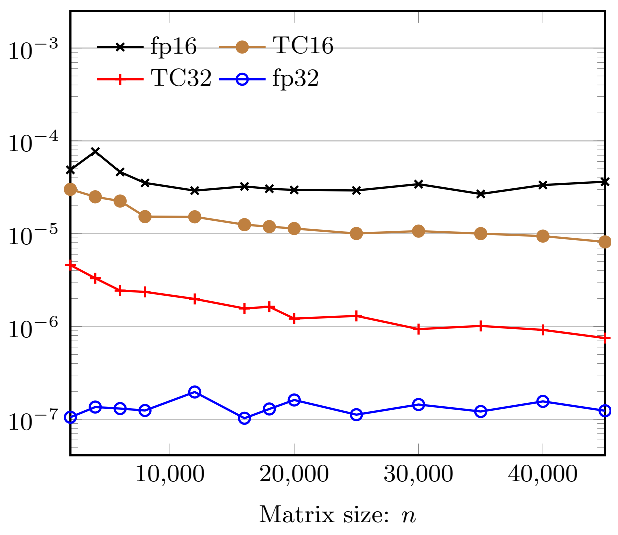

Fig. 4.2: Componentwise backward error for the solution of $Ax = b$ using an LU factorization on an NVIDIA V100 GPU for the four variants fp16, TC16, TC32, and fp32.

$10^{-c(i-1)/(n-1)}$. 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 TFlops and that, as expected, the fp16 variant achieves twice the performance of that variant with around 20 TFlops. 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 TFlops, 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.

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. Just as for matrix multiplication (section 3.3), the TC16 variant gives a smaller backward error than the fp16 one, even though their bounds are identical: this might again be explained by the use of blocked summation within the cuBLAS implementation of matrix multiplication. The TC32 variant gives a backward error between one and two orders of magnitude smaller than the fp16 and TC16 variants, as could be expected from the bounds shown in Table 4.1. 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} (where $u_{\text{high}} < u_{\text{low}}$), and returns D in precision u_{FMA} (equal to u_{high} or u_{low}), carrying out the computation at precision $\overline{u}$.

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 the choice of each precision parameter.

If $u_{\text{FMA}} = u_{\text{low}}$ and $\overline{u} = u_{\text{high}}$, then a $b \times b$ block FMA leads to error bounds a factor b smaller than those for conventional algorithms in precision u_{low}. More significantly, by storing C and D in precision u_{high} (that is, $u_{\text{FMA}} = u_{\text{high}}$), 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 with a much weaker dependence on n, 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 ($u_{\text{FMA}} = \overline{u} = u_{16}$) and TC32 ($u_{\text{FMA}} = \overline{u} = u_{32}$), which differ in the precision used for the output of the tensor cores. Our analysis predicts the TC32 variant to be much more accurate than the TC16 one. Our numerical experiments confirm this prediction and show that the accuracy boost is achieved with almost no performance loss.

Our analysis can be applied to other matrix factorizations that are able to exploit a block FMA, such as mixed precision QR factorization (used in [8]) and mixed precision Cholesky factorization (used in [20]). The analysis is sufficiently general that it should be applicable to future block FMAs, including those in Table 1.1.

REFERENCES

[1] A. Abdelfattah, S. Tomov, and J. Dongarra, *Towards half-precision computation for complex matrices: A case study for mixed-precision solvers on GPUs*, in 2019 IEEE/ACM 10th Workshop on Latest Advances in Scalable Algorithms for Large-Scale Systems (ScalA), 2019, pp. 17–24, https://doi.org/10.1109/ScalA49573.2019.00008.

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

[3] *ARM Architecture Reference Manual. ARMv8, for ARMv8-A Architecture Profile*, ARM Limited, Cambridge, UK, 2018, https://developer.arm.com/docs/ddi0487/latest. Version dated 31 October 2018. Original release dated 30 April 2013.

[4] P. Blanchard, N. J. Higham, and T. Mary, *A class of fast and accurate summation algorithms*, MIMS EPrint 2019.6, Manchester Institute for Mathematical Sciences, The University of Manchester, UK, Apr. 2019, http://eprints.maths.manchester.ac.uk/2729/. Revised September 2019.

[5] P. R. Capello and W. L. Miranker, *Systolic super summation*, IEEE Trans. Comput., 37 (1988), pp. 657–677, https://doi.org/10.1109/12.2205.

[6] E. Carson and N. 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 (2017), pp. A2834–A2856, https://doi.org/10.1137/17M1122918.

[7] E. Carson and N. J. Higham, *Accelerating the solution of linear systems by iterative refinement in three precisions*, SIAM J. Sci. Comput., 40 (2018), pp. A817–A847, https://doi.org/10.1137/17M1140819.

[8] E. Carson, N. J. Higham, and S. Pranesh, *Three-precision GMRES-based iterative refinement for least squares problems*, MIMS EPrint 2020.5, Manchester Institute for Mathematical Sciences, The University of Manchester, UK, Feb. 2020, http://eprints.maths.manchester.ac.uk/2745/.

[9] *CUDA C++ programming guide*. https://docs.nvidia.com/cuda/cuda-c-programming-guide/index.html#wmma-type-sizes, Nov. 2019. Version 10.2.89.

[10] M. 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, Aug. 2018. Accessed November 22, 2018.

[11] M. 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/, Dec. 2018. Accessed January 8, 2019.

[12] A. Haidar, A. Abdelfattah, M. Zounon, P. Wu, S. Pranesh, S. Tomov, and J. 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, Y. Shi, H. Fu, Y. Tian, V. V. Krzhizhanovskaya, M. H. Lees, J. Dongarra, and P. M. A. Sloot, eds., Springer International Publishing, Cham, 2018, pp. 586–600, https://doi.org/10.1007/978-3-319-93698-7_45.

[13] A. Haidar, H. Bayraktar, S. Tomov, J. Dongarra, and N. J. Higham, *Mixed-precision solution of linear systems using accelerator-based computing*, tech. report, 2020. In preparation.

[14] A. Haidar, S. Tomov, J. Dongarra, and N. 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, IEEE Press, pp. 47:1–47:11,

https://doi.org/10.1109/SC.2018.00050.

[15] A. Haidar, P. Wu, S. Tomov, and J. 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), Nov. 2017, pp. 10:1–10:8, https://doi.org/10.1145/3148226.3148237.

[16] J. W. Hanlon, *New chips for machine intelligence*. https://jameswhanlon.com/new-chips-for-machine-intelligence.html, Oct. 2019. Accessed November 27, 2019.

[17] N. J. Higham, *Accuracy and Stability of Numerical Algorithms*, Society for Industrial and Applied Mathematics, Philadelphia, PA, USA, second ed., 2002, https://doi.org/10.1137/1.9780898718027.

[18] N. J. Higham and T. Mary, *A new approach to probabilistic rounding error analysis*, SIAM J. Sci. Comput., 41 (2019), pp. A2815–A2835, https://doi.org/10.1137/18M1226312.

[19] N. J. Higham and T. Mary, *Sharper probabilistic backward error analysis for basic linear algebra kernels with random data*, MIMS EPrint 2020.4, Manchester Institute for Mathematical Sciences, The University of Manchester, UK, Jan. 2020, http://eprints.maths.manchester.ac.uk/2743/.

[20] N. J. Higham and S. Pranesh, *Exploiting lower precision arithmetic in solving symmetric positive definite linear systems and least squares problems*, MIMS EPrint 2019.20, Manchester Institute for Mathematical Sciences, The University of Manchester, UK, Nov. 2019, http://eprints.maths.manchester.ac.uk/2736/.

[21] N. J. Higham, S. Pranesh, and M. Zounon, *Squeezing a matrix into half precision, with an application to solving linear systems*, SIAM J. Sci. Comput., 41 (2019), pp. A2536–A2551, https://doi.org/10.1137/18M1229511.

[22] *IEEE Standard for Floating-Point Arithmetic, IEEE Std* 754-2019 (*Revision of IEEE* 754-2008), IEEE Computer Society, New York, 2019, https://doi.org/10.1109/IEEESTD.2019.8766229.

[23] Intel Corporation, *BFLOAT16—hardware numerics definition*, Nov. 2018, https://software.intel.com/en-us/download/bfloat16-hardware-numerics-definition. White paper. Document number 338302-001US.

[24] W. Krämer and M. Zimmer, *Fast (parallel) dense linear system solvers in C-XSC using error free transformations and BLAS*, in Numerical Validation in Current Hardware Architectures, A. Cuyt, W. Krämer, W. Luther, and P. Markstein, eds., Berlin, Heidelberg, 2009, Springer Berlin Heidelberg, pp. 230–249, https://doi.org/10.1007/978-3-642-01591-5_15.

[25] U. W. Kulisch and W. L. Miranker, *The arithmetic of the digital computer: A new approach*, SIAM Rev., 28 (1986), pp. 1–40, https://doi.org/10.1137/1028001.

[26] S. Markidis, S. Wei Der Chien, E. Laure, I. B. Peng, and J. S. Vetter, *NVIDIA tensor core programmability, performance & precision*, in 2018 IEEE International Parallel and Distributed Processing Symposium Workshops (IPDPSW), May 2018, pp. 522–531, https://doi.org/10.1109/ipdpsw.2018.00091.

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

[28] N. 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.

[29] S. M. Rump, *Verification methods: Rigorous results using floating-point arithmetic*, Acta Numerica, 19 (2010), pp. 287–449, https://doi.org/10.1017/S096249291000005X.

[30] S. M. Rump, *IEEE754 precision-k base-β arithmetic inherited by precision-m base-β arithmetic for $k < m$*, ACM Trans. Math. Software, 43 (2016), pp. 20:1–20:15, https://doi.org/10.1145/2785965.

[31] A. 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, Apr. 2019. Accessed May 22, 2019.

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

[33] Y. Tao, G. Deyuan, F. Xiaoya, and J. Nurmi, *Correctly rounded architectures for floating-point multi-operand addition and dot-product computation*, in 2013 IEEE 24th International Conference on Application-Specific Systems, Architectures and Processors, June 2013, pp. 346–355, https://doi.org/10.1109/ASAP.2013.6567600.

[34] S. Wang and P. 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, Aug. 2019. Accessed September 14, 2019.