Matrix Multiplication in Multiword Arithmetic: Error Analysis and Application to GPU Tensor Cores

Fasi, Massimiliano and Higham, Nicholas J. and Lopez, Florent and Mary, Theo and Mikaitis, Mantas

2022

MIMS EPrint: **2022.3**

ISSN 1749-9097

MATRIX MULTIPLICATION IN MULTIWORD ARITHMETIC: ERROR ANALYSIS AND APPLICATION TO GPU TENSOR CORES[*]

MASSIMILIANO FASI[†], NICHOLAS J. HIGHAM[‡], FLORENT LOPEZ[§], THEO MARY[¶], AND MANTAS MIKAITIS[‡]

Abstract. In multiword arithmetic, a matrix is represented as the unevaluated sum of two or more lower-precision matrices, and a matrix product is formed by multiplying the constituents in low precision. We investigate the use of multiword arithmetic for improving the performance–accuracy tradeoff of matrix multiplication with mixed precision block fused multiply–add (FMA) hardware, focusing especially on the tensor cores available on NVIDIA GPUs. Building on a general block FMA framework, we develop a comprehensive error analysis of multiword matrix multiplication. After confirming the theoretical error bounds experimentally by simulating low precision in software, we use the cuBLAS and CUTLASS libraries to implement a number of matrix multiplication algorithms using double-fp16 (double-binary16) arithmetic. When running the algorithms on NVIDIA V100 and A100 GPUs, we find that double-fp16 is not as accurate as fp32 (binary32) arithmetic despite satisfying the same worst-case error bound. Using probabilistic error analysis, we explain why this issue is likely to be caused by the rounding mode used by the NVIDIA tensor cores, and propose a parameterized blocked summation algorithm that alleviates the problem and significantly improves the performance–accuracy tradeoff.

Key words. matrix multiplication, numerical linear algebra, rounding error analysis, floating-point arithmetic, multiword arithmetic, reduced precision, mixed precision, GPUs, NVIDIA V100, NVIDIA A100, tensor cores, rounding modes, blocked summation, `FABsum`

AMS subject classifications. 65G50, 65Y04, 65Y10, 68M07

1. Introduction.

The NVIDIA tensor cores in the Volta microarchitecture [30] perform the mixed precision operation $D = AB + C$, where A and B are 4×4 fp16 matrices and C and D have same size but can have either fp16 or fp32 entries. Here, fp16 and fp32 denote the binary16 and binary32 formats, respectively, as defined in the last two revisions of the IEEE standard for floating-point arithmetic [22], [23]. GPUs based on the newer NVIDIA Ampere microarchitecture [8], [31] are equipped with updated versions of the tensor cores which support other floating-point formats: bfloat16 [24] (hereinafter bf16), TensorFloat-32 (hereinafter tf32), and binary64 (hereinafter fp64). The upcoming NVIDIA Hopper microarchitecture [32] adds yet more formats to the tensor cores (quarter precision): fp8-E5M2 (5 exponent and 2 significand bits) and fp8-E4M3 (4 exponent and 3 significand bits). Tensor cores provide a significant performance boost compared with standard floating-point units, and have been used with great success to accelerate numerical linear algebra algorithms [1], [5], [13], [14] [25]; see [20] for a survey of these algorithms. Other vendors also incorporate matrix arithmetic in their devices: for example, the accelerators in the AMD MI200

[*]Version of 12 July 2022.

Funding: This work was supported by Wenner-Gren Foundations grant UPD2019-0067 (Fasi); Engineering and Physical Sciences Research Council (EPSRC) grant EP/P020720/1 (Higham, Mikaitis); the Royal Society (Higham); and the Exascale Computing Project (17-SC-20-SC), a collaborative effort of the U.S. Department of Energy Office of Science and the National Nuclear Security Administration (Higham, Mikaitis).

[†]Department of Computer Science, Upper Mountjoy Campus, Stockton Road, Durham University, Durham, UK (massimiliano.fasi@durham.ac.uk).

[‡]Department of Mathematics, University of Manchester, Oxford Road, Manchester, UK (nick.higham@manchester.ac.uk, mantas.mikaitis@manchester.ac.uk).

[§]Livermore Software Technology, an ANSYS Company, Livermore, USA (florent.lopez@ansys.com).

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

series contain units that can perform vector and matrix operations faster than their scalar counterparts [2], [3], [4].

Tensor cores are instances of what we have called block fused multiply–add (FMA) units [6]. Block FMAs are attractive not only because of their high performance, but also because they are intrinsically mixed precision units: while their inputs A and B must be low precision matrices, the internal computations are performed in high precision, and the output can be accumulated in high precision if C is a high precision matrix. As a result, block FMAs significantly reduce the negative impact of the accumulation of rounding errors and can often provide more accurate results than standard low precision units [6]. Because of the need to convert A and B to low precision, however, computations with tensor cores still carry an error term of order the unit roundoff of the low precision (fp16 or bf16), which might be unacceptable in applications that require high accuracy.

The goal of this work is to investigate how multiword arithmetic [29, sec. 14.1] can allow us to extend the use of tensor cores to applications that cannot tolerate the loss of precision produced by the conversion of the input matrices A and B to fp16 or bf16. In multiword arithmetic, A and B are represented as the unevaluated sum of low precision matrices that, when added together, approximate the original A and B. It is important to point out that by using this technique we can extend the available precision, but not the range of representable floating-point values: the dynamic range of a multiword format remains that of the low precision format used to represent each individual word. Therefore, multiword arithmetic does not alleviate issues due to underflow and overflow and may in fact exacerbate them. Floating-point numbers that have exponent very close to the smallest exponent available in the low precision format may end up being less accurate than one would otherwise expect, as some of the less significant words may underflow to 0 and therefore lose all precision. Issues with overflow and underflow can be addressed by resorting to a data type with a wider dynamic range (such as bf16 instead of fp16) or to a suitable scaling strategy [21].

The best known example of multiword arithmetic is double-fp64 (commonly referred to as double-double) arithmetic, which achieves nearly binary128 (hereinafter fp128) precision by representing each number as the unevaluated sum of two fp64 numbers and by relying on error-free fp64 transformations for computation of arithmetic operations [29, sec. 14.1.1]. Double-fp64 arithmetic is thus an effective alternative to fp128 on hardware where fp64 is much faster than fp128.

The emergence of block FMA hardware supporting low precision matrix multiplication with high precision accumulators provides new perspectives into the potential of multiword arithmetic: reducing the precision of the input dramatically increases the throughput of these hardware units compared with the use of standard fp32 arithmetic (tensor cores are up to $8\times$ faster on Volta GPUs and up to $16\times$ faster on Ampere GPUs, for example). This suggests a simple strategy for accelerating the computation of the matrix product $C = AB$, where A and B are fp32 matrices: one can first approximate $A \approx A_1 + A_2$ and $B \approx B_1 + B_2$ as sums of fp16 matrices, and then compute C to nearly fp32 accuracy as $C \approx A_1B_1 + A_1B_2 + A_2B_1 + A_2B_2$, where each A_iB_j term is evaluated using a block FMA. Since there are only four terms, this approach can potentially be much faster than standard fp32 arithmetic. Moreover, double-fp16 addition and multiplication operations with error-free transformations are not required in this setting, because fp16 products are maintained as fp32 and additions in tensor cores are carried out in fp32 arithmetic. We refer to this approach as double-fp16 arithmetic.

The use of double-fp16 arithmetic with tensor cores was first proposed by Markidis et al. [26], who call this technique *precision refinement*. Sorna et al. [34] adopted a similar approach to accelerate the fast Fourier transform. Mukunoki and Ogita [27] investigated how to use multiword arithmetic to increase the accuracy of fp32 and fp64 matrix multiplication. Henry et al. [15] considered block FMA units modelled on future Intel hardware, and proposed the use of triple-bf16 arithmetic (which represents an fp32 value as the sum of three bf16 numbers). Mukunoki et al. [28] implemented a correctly rounded fp32/fp64 matrix multiplication algorithm using tensor cores on NVIDIA V100 GPUs. The authors note that on the V100 GPUs their method is slower than simply using the available fp64 units, but their goal is to enable the use of fp64 arithmetic on future GPUs where only lower precision will potentially be available. Pisha and Ligowski [33] similarly used a double-tf32 representation to compute Fourier transforms on A100 GPUs.

In section 2 we begin by developing a rigorous rounding error analysis of a general multiword matrix multiplication (`MMM`) algorithm based on the block FMA framework of Blanchard et al. [6]. In order to be as general as possible, our study considers the use of multiword arithmetic with an arbitrary number of words and with two parameterized precisions. The analysis provides a unified framework that encompasses all the approaches mentioned above and also includes some new cases. One goal of this work is to determine what level of accuracy can be expected from a given multiword arithmetic implemented using block FMAs. In section 3, we confirm the predictions of the analysis by means of simulations: we test an implementation of the `MMM` algorithm using emulated low precision and find that the experiments are in good agreement with the theoretical error bounds.

In section 4 we implement our `MMM` algorithm on NVIDIA V100 and A100 GPUs with tensor cores using the cuBLAS library and make a surprising discovery: we observe that multiword matrix multiplication is significantly less accurate than matrix multiplication performed using only fp32 arithmetic, although the two algorithms have the same theoretical worst-case error bound. This is especially true for matrices with elements of nonzero mean: this may explain why this issue was not observed in previous work, which mostly focused on matrices with entries drawn from a distribution with zero mean, such as the uniform distribution over the interval $[-1, 1]$. To understand this behavior, we make use of recent results in probabilistic error analysis [18], [19], and show that the `MMM` algorithm implemented on GPUs yields an error that is on average very close to its worst-case bound, unlike the standard fp32 algorithm which benefits from the statistical distribution of rounding errors. We relate this difference to the fact that inside the block FMA computation the current tensor cores use a rounding mode other than round-to-nearest [11], [16]. We cure this numerical issue by reducing the worst-case error bound of the `MMM` algorithm. We achieve this by using mixed precision blocked summation, an instance of the recently proposed `FABsum` algorithm [7]. We develop a high performance implementation of `FABsum` based on the CUTLASS library, and show that this new algorithm can achieve a much better performance–accuracy tradeoff than the cuBLAS-based algorithm.

Finally, we introduce some notation. A hat indicates a quantity computed in floating-point arithmetic. We denote by u the unit roundoff of a given floating-point arithmetic, and refer to that arithmetic as being of precision u.

2. Error analysis of multiword matrix multiplication with block FMAs.

In this section we develop a rigorous error analysis that applies to previously proposed multiword algorithms for matrix multiplication and suggests new variants of interest. We use the general block FMA framework of Blanchard et al. [6], in which a block FMA unit uses two arithmetics of precisions u_{low} and u_{high}. For two matrices $A \in \mathbb{R}^{m\times n}$ and $B \in \mathbb{R}^{n\times q}$ given in precision u_{low}, we can use [6, Alg. 3.1] to evaluate $C = AB$ using a block FMA so that the computed $\widehat{C}$ satisfies [6, Thm. 3.1]

$$\widehat{C} = C + \mathit{\Delta} C, \qquad |\mathit{\Delta} C| \le \gamma_n^{\text{high}} |A||B|, \tag{2.1}$$

where $\gamma_n^{\text{high}} = nu_{\text{high}}/(1 - nu_{\text{high}})$ for $nu_{\text{high}} < 1$. (This assumes we have taken $\bar{u} = u_{\text{high}}$ in the analysis of [6], which corresponds to what is called there the "TC32 variant" in the case of the tensor cores.)

Let fl_{low} denote the operator that rounds to precision u_{low}. For any $x \in \mathbb{R}$, we have that $x_1 = \mathrm{fl}_{\text{low}}(x) = x(1+\delta_1)$, where $|\delta_1| \le u_{\text{low}}$, and by using the fact that $x_2 = \mathrm{fl}_{\text{low}}(x - x_1) = -x\delta_1(1+\delta_2)$, where $|\delta_2| \le u_{\text{low}}$, we obtain

$$x_1 + x_2 = x - x\delta_1\delta_2 = x(1+\delta), \quad |\delta| \le u_{\text{low}}^2.$$

We can apply this idea recursively and elementwise to A and B by computing, for $i, j = 1, \dots, p$, the matrices

$$A_i = \mathrm{fl}_{\text{low}}\left(A - \sum_{k=1}^{i-1} A_k\right), \quad B_j = \mathrm{fl}_{\text{low}}\left(B - \sum_{k=1}^{j-1} B_k\right). \tag{2.2}$$

Barring underflow in the conversion to low precision, we get

$$A = \sum_{i=1}^{p} A_i + \mathit{\Delta} A, \qquad |\mathit{\Delta} A| \le u_{\text{low}}^p |A|, \tag{2.3}$$

$$B = \sum_{j=1}^{p} B_j + \mathit{\Delta} B, \qquad |\mathit{\Delta} B| \le u_{\text{low}}^p |B|. \tag{2.4}$$

We note in passing that for large enough p, terms $\mathit{\Delta} A$ and $\mathit{\Delta} B$ in the multiword decompositions (2.3) and (2.4) can be equal to zero. Then the product $C = AB$ is given by

$$C = \sum_{i=1}^{p}\sum_{j=1}^{p} A_i B_j + A\mathit{\Delta} B + \mathit{\Delta} A B - \mathit{\Delta} A \mathit{\Delta} B.$$

If the p^2 products $G_{ij} = A_i B_j$ are computed with a block FMA, by (2.1) the computed $\widehat{G}_{ij}$ satisfy

$$\widehat{G}_{ij} = G_{ij} + \mathit{\Delta} G_{ij}, \quad |\mathit{\Delta} G_{ij}| \le \gamma_n^{\text{high}} |A_i||B_j|.$$

If the $\widehat{G}_{ij}$ are accumulated in precision u_{high}, then the computed $\widehat{C}$ satisfies

$$\begin{aligned}
\widehat{C} &= \sum_{i=1}^{p}\sum_{j=1}^{p} \widehat{G}_{ij} \circ (1 + \Theta_{ij}), \qquad |\Theta_{ij}| \le \gamma_{p^2-1}^{\text{high}} \\
&= \sum_{i=1}^{p}\sum_{j=1}^{p} (G_{ij} + \mathit{\Delta} G_{ij}) \circ (1 + \Theta_{ij}) \\
&= \sum_{i=1}^{p}\sum_{j=1}^{p} (A_i B_j + \mathit{\Delta} G_{ij}) \circ (1 + \Theta_{ij}) \\
&= AB - A\mathit{\Delta} B - \mathit{\Delta} A B + \mathit{\Delta} A \mathit{\Delta} B + \sum_{i=1}^{p}\sum_{j=1}^{p} (A_i B_j) \circ \Theta_{ij} + \mathit{\Delta} G_{ij} \circ (1 + \Theta_{ij}),
\end{aligned}$$

where $\circ$ denotes the Hadamard (elementwise) product. Overall we have

$$\widehat{C} = AB + E, \quad |E| \le \big(2u_{\mathrm{low}}^{p} + u_{\mathrm{low}}^{2p}\big)|A||B| + \gamma_{n+p^2-1}^{\mathrm{high}} \sum_{i=1}^{p}\sum_{j=1}^{p}|A_i||B_j|, \tag{2.5}$$

where we have used the fact that $\gamma_n^{\mathrm{high}} + \gamma_{p^2-1}^{\mathrm{high}} + \gamma_n^{\mathrm{high}}\gamma_{p^2-1}^{\mathrm{high}} \le \gamma_{n+p^2-1}^{\mathrm{high}}$ [17, Lem. 3.3]. Note that we cannot directly replace $\sum_{i=1}^{p}\sum_{j=1}^{p}|A_i||B_j|$ by $|A||B|$, because a given entry does not necessarily have the same sign in all A_i (or B_j) terms.

Clearly, for practical choices of u_{low} and u_{high} a small value of p is sufficient to make the two terms in the bound (2.5) of similar size. For fp16 ($u_{\mathrm{low}} = 2^{-11}$) and fp32 ($u_{\mathrm{high}} = 2^{-24}$), for example, setting $p = 2$ will suffice: in this case $u_{\mathrm{low}}^2 = 4u_{\mathrm{high}}$, and taking larger values of p would not improve significantly the bound (2.5), as the term $\gamma_{n+p^2-1}^{\mathrm{high}}$ would then dominate. For bf16 ($u_{\mathrm{low}} = 2^{-8}$) and fp32, the case $p = 3$ is also of interest.

Importantly, not all p^2 products A_iB_j need be computed. This is because, as a result of the construction (2.2), the magnitude of the elements of A_i and B_j rapidly decreases as i and j increase. More precisely, we have

$$\begin{aligned} |A_i| &\le u_{\mathrm{low}}^{i-1}(1+u_{\mathrm{low}})|A|, \quad i = 1,\dots,p,\\ |B_j| &\le u_{\mathrm{low}}^{j-1}(1+u_{\mathrm{low}})|B|, \quad j = 1,\dots,p, \end{aligned}$$

and thus

$$|A_i||B_j| \le u_{\mathrm{low}}^{i+j-2}(1+u_{\mathrm{low}})^2|A||B|. \tag{2.6}$$

Therefore ignoring any product A_iB_j such that $i+j > p+1$ only introduces an error of order u_{low}^{p} or higher, which has no significant impact on the bound (2.5). Indeed, by only computing the products A_iB_j such that $i+j \le p+1$, we obtain $\widehat{C} = AB+E$ with the modified bound

$$\begin{aligned} |E| &\le \left[2u_{\mathrm{low}}^{p} + u_{\mathrm{low}}^{2p} + \left(\gamma_{n+p^2-1}^{\mathrm{high}}\Big(1 + \sum_{k=1}^{p-1}u_{\mathrm{low}}^{k}\Big) + \sum_{i=1}^{p-1}(p-i)u_{\mathrm{low}}^{p+i-1}\right)(1+u_{\mathrm{low}})^2\right]|A||B| \\ &\le \Big((p+1)u_{\mathrm{low}}^{p} + \gamma_{n+p^2-1}^{\mathrm{high}}\Big)|A||B| + O(u_{\mathrm{high}}u_{\mathrm{low}} + u_{\mathrm{low}}^{p+1}). \end{aligned} \tag{2.7}$$

With respect to (2.5), we have only increased the constant in front of the term u_{low}^{p} from 2 to $p+1$, but we have reduced the number of matrix products to be evaluated from p^2 to $p(p+1)/2$. In practice, with fp16 and fp32 ($p = 2$), we only need three products, which is less than the four used by Markidis et al. [26], and with bf16 and fp32 ($p = 3$) we can reduce the number of products from nine to six, as already suggested by Henry, Tang, and Heinecke [15].

It is possible to further reduce the number of products (such as using two products for $p = 2$ as attempted in [26]), but our analysis tells us that such a choice is unlikely to be advantageous. Indeed, ignoring any product A_iB_j such that $i + j \le p + 1$ would introduce an error of order u_{low}^{p-1}, thus the resulting algorithm would not be significantly more accurate than one using $p-1$ rather than p splits.

We summarize the proposed approach in Algorithm 2.1 and its rounding error analysis in Theorem 2.1.

THEOREM 2.1. *Let $A \in \mathbb{R}^{m\times n}$ and $B \in \mathbb{R}^{n\times q}$ and let $C = AB$ be computed by Algorithm 2.1. The computed $\widehat{C}$ satisfies*

$$\widehat{C} = AB + E, \quad |E| \le \Big((p+1)u_{\mathrm{low}}^{p} + \gamma_{n+p^2-1}^{\mathrm{high}}\Big)|A||B| + O(u_{\mathrm{high}}u_{\mathrm{low}} + u_{\mathrm{low}}^{p+1}). \tag{2.8}$$

Algorithm 2.1: Multiword matrix multiplication (`MMM`), using a mixed precision block FMA with precisions u_{low} and u_{high}. On line 8 an algorithm that satisfies (2.1), such as [6, Alg 3.1] for example, should be used.

```
Input  : Two matrices $A \in \mathbb{R}^{m\times n}$ and $B \in \mathbb{R}^{n\times q}$ and number of splits $p$.
Output: The matrix $C = AB$ computed in $p$-word arithmetic.
1 for $i \leftarrow 1$ to $p$ do
2     $A_i \leftarrow \mathrm{fl}_{\text{low}}(A - \sum_{k=1}^{i-1} A_k)$
3     $B_i \leftarrow \mathrm{fl}_{\text{low}}(B - \sum_{k=1}^{i-1} B_k)$
4 $C \leftarrow 0$
5 for $i \leftarrow 1$ to $p$ do
6     for $j \leftarrow 1$ to $p$ do
7         if $i + j \le p + 1$ then
8             Compute $C_{ij} \leftarrow A_i B_j$ with a block FMA.
9             $C \leftarrow C + C_{ij}$
```

TABLE 2.1. *Dominant term in the error bound (2.8) for u_{high} corresponding to fp32 and various choices of u_{low} and p.*

u_{high}	u_{low}	Split	Name	Bound
2^{-24} (fp32)	2^{-11} (fp16)	$p = 1$	fp16	$2 \times 2^{-11} + n \times 2^{-24}$
		$p = 2$	double-fp16	$n \times 2^{-24}$
	2^{-8} (bf16)	$p = 1$	bf16	$2 \times 2^{-8} + n \times 2^{-24}$
		$p = 2$	double-bf16	$3 \times 2^{-16} + n \times 2^{-24}$
		$p = 3$	triple-bf16	$n \times 2^{-24}$

As mentioned, not only does the analysis above encompass previously proposed algorithms, but it also suggests new variants that might be of interest. For example, we may use a binary split ($p = 2$) with bf16 and fp32 which requires three products rather than six (when $p = 3$) and delivers an accuracy of order 2^{-16} rather than 2^{-24}. We summarize in Table 2.1 the dominant term in the error bound (2.8) for several choices of u_{low} and p.

3. Implementation and experiments with simulated arithmetic.

In order to confirm the theoretical error bound derived in the previous section, we implemented Algorithm 2.1 with simulated mixed precision fp16-fp32 block FMA arithmetic using the `CPFloat` package [12]. The simulations in this section and in section 4.3 were compiled using version 11.3.0 of the GNU Compiler Collection on a machine equipped with an AMD Ryzen 7 PRO 5850U CPU.

We focus on double-fp16 arithmetic, that is, we set $p = 2$, $u_{\text{low}} = 2^{-11}$, and $u_{\text{high}} = 2^{-24}$. For comparison, we also compute the matrix product in fp32 and fp64 arithmetics in hardware by using the `Eigen` C++ library.[1] For fp64 arithmetic, we use the default `Eigen` matrix multiplication implementation, for all other arithmetics we use the blocked FMA algorithm [6, Alg. 3.1] with a block FMA of dimension 1.

To measure the accuracy of the computed result $\widehat{C}$, we compute the maximum

[1] https://eigen.tuxfamily.org/

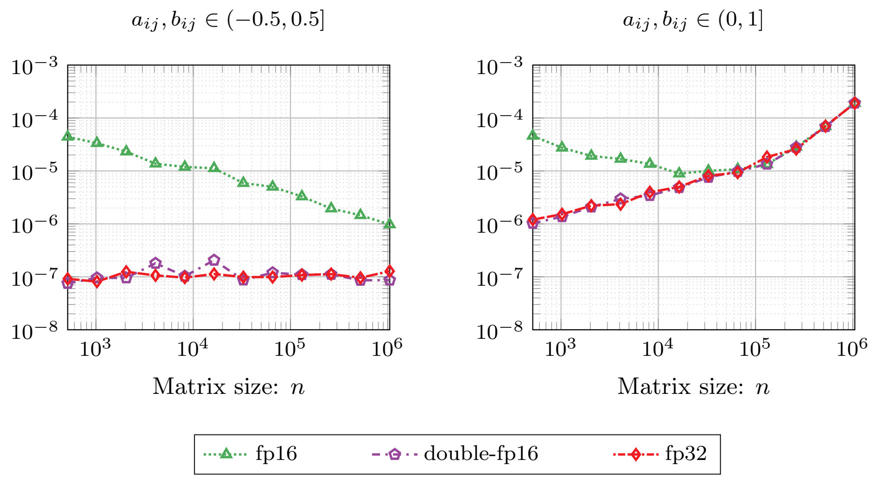

FIG. 3.1. *Componentwise relative error of algorithms for computing the product AB. The methods and arithmetics used are discussed in section 3. The double-fp16 matrices $A \in \mathbb{R}^{16\times n}$ and $B \in \mathbb{R}^{n\times 16}$ have entries sampled uniformly at random from the interval at the top.*

componentwise relative error

$$\max_{i,j} \frac{|C - \widehat{C}|_{ij}}{(|A||B|)_{ij}}, \tag{3.1}$$

where C is a reference solution computed using fp64 arithmetic. We will use the same error metric in section 4 when running the experiments on GPUs.

Figure 3.1 compares the error of double-fp16 arithmetic with that of standard fp16 and fp32 arithmetics. We will repeat the same experiments on GPUs in section 4, and the matrix dimensions used here reflect the memory limits of the NVIDIA GPUs used there. We consider the multiplication of two matrices $A \in \mathbb{R}^{m\times n}$ and $B \in \mathbb{R}^{n\times q}$, with $m = q = 16$, and we vary n between 2^9 and 2^{20}.

The results show that double-fp16 arithmetic can substantially improve accuracy compared with fp16 arithmetic. As expected, the benefit is reduced as n increases, since the error term of order $n \times 2^{-24}$ becomes more significant and can eventually outgrow the term 2×2^{-11} (see Table 2.1). This is especially true when the data is drawn from the interval $(0, 1]$, in which case fp16 arithmetic becomes as accurate as double-fp16 and fp32 arithmetic for $n = 6 \times 10^4$. If the entries of the input matrices are drawn from the uniform distribution over $(-0.5, 0.5]$, the data has zero mean, which leads to an error not growing with n and even decreasing with n in some cases, which is explained by probabilistic error analysis [19]; as a result, double-fp16 arithmetic remains at least one order of magnitude more accurate than fp16 arithmetic for $n \leq 10^6$. In our experiments, we also tested a variant of double-fp16 which does not drop the A_2B_2 term and computes four fp16 products instead of three. The results are not shown in Figure 3.1 as the curve for this variant is indistinguishable from that of the algorithm that computes only the first three products. This confirms that A_2B_2 can be neglected without any impact on the accuracy, as predicted by our analysis.

TABLE 4.1. *Maximum theoretical throughput (in Tflop/s) of various arithmetics in the NVIDIA V100 [30] and A100 [31] GPUs. These figures are based on the "GPU boost clock" [30], [31], which is* 1530 MHz *on the V100 and* 1410 MHz *on the A100. The suffix "-tc" refers to the figures for mixed precision matrix–matrix multiplication with tensor cores enabled.*

GPU	fp64	fp64-tc	fp32	tf32-tc	bf16	bf16-tc	fp16	fp16-tc
V100	7.8		15.7				31.4	125.0
A100	9.7	19.5	19.5	156.0	39.0	312.0	78.0	312.0

4. Experiments on NVIDIA V100 and A100 GPUs.

In this section we evaluate the accuracy and performance of various GPU implementations of Algorithm 2.1. The codes target the NVIDIA V100 and A100 GPUs, and the tensor cores are used for fp16 and double-fp16 but not for fp32 arithmetic.

In our first experiments with the cuBLAS library (section 4.2), we find that double-fp16 arithmetic is not as accurate as fp32 arithmetic, although the two possess an almost identical error bound. In section 4.3, we identify the cause of the issue as related to the rounding mode used by the tensor cores. We propose a cure for this problem in section 4.4.

4.1. Properties of the NVIDIA V100 and A100 GPUs.

The NVIDIA V100 and A100 GPUs provide a wide range of different arithmetics with varying levels of performance. The third-generation tensor cores that equip the Ampere cards provide more levels of precision than the tensor cores available on the Volta chips [8]. Table 4.1 compares the performance of these two GPUs for different arithmetics. The throughput is expressed in floating-point operations per second (flop/s), and one Tflop/s corresponds to 10^{12} flop/s.

Note that different arithmetics have different numbers of functional units on GPUs and this has an impact on throughput—for example, an NVIDIA A100 GPU has 3,456 fp64 cores and 6,912 fp32 cores [31], therefore fp32 arithmetic is expected to have at least $2\times$ higher throughput than fp64. We say "at least" because the actual figure could be larger in practice, for example if a single fp32 elementary arithmetic operation requires fewer cycles than a single fp64 one to complete, or if the fp32 and fp64 cores run at different frequencies. In fact, on A100 GPUs the throughput of fp32 arithmetic is exactly twice that of fp64 (Table 4.1), therefore most likely both arithmetics have the same latency, or the declared performance numbers assume hazard-free pipelining of instructions (once the pipeline is full, a floating-point unit completes a new FMA instruction at every cycle).

4.2. Experiments with cuBLAS.

In this section we evaluate the performance and accuracy of double-fp16 arithmetic using the cuBLAS library. Our implementation of Algorithm 2.1 uses the `cublasGemmEx` routine for computing matrix–matrix products, as described in Algorithm 4.1. We use version 11.6.124 of the CUDA library and the NVIDIA Tesla V100-SXM2 16GiB and NVIDIA A100-SXM 80GiB GPUs. The function `cublasGemmEx` allows the programmer to choose between 24 (if tensor cores are disabled) or 16 (if they are enabled) default algorithms, or a heuristic approach that selects the best algorithm according to undisclosed criteria. The latter option has been used for the experiments below since on the A100 GPU and newer it is the default and only option.

4.2.1. Performance.

Figure 4.1 plots the maximum observed throughput for the computation of the product of two $n \times n$ matrices of increasing size on the V100

Algorithm 4.1: `MMM` algorithm (Algorithm 2.1) using cuBLAS.

```
Input  : Two fp32 matrices $A \in \mathbb{R}^{m\times n}$ and $B \in \mathbb{R}^{n\times q}$.
Output: The fp32 matrix $C \approx AB$ computed in double-fp16 arithmetic.
1 $C \leftarrow 0$
2 $A_1 \leftarrow \mathrm{fl}_{16}(A)$
3 $A_2 \leftarrow \mathrm{fl}_{16}(A - A_1)$
4 $B_1 \leftarrow \mathrm{fl}_{16}(B)$
5 $B_2 \leftarrow \mathrm{fl}_{16}(B - B_1)$
6 Compute $C \leftarrow C + A_2B_1$ using cublasGemmEx.
7 Compute $C \leftarrow C + A_1B_2$ using cublasGemmEx.
8 Compute $C \leftarrow C + A_1B_1$ using cublasGemmEx.
```

and A100 GPUs. As is common when comparing the performance of algorithms that execute a different number of floating-point operations (flops), we choose as performance metric the "effective Tflop/s" rate, which is computed by dividing the number of executed flops, in our case $2n^3$, by the runtime of each algorithm:

$$\text{Effective Tflop/s} = \frac{2n^3}{t_{\text{avg}}} \times 10^{-12}, \tag{4.1}$$

with $t_{\text{avg}} = t_s/R$, where t_s is the total runtime of the experiment in seconds and R is the number of times each run is repeated. We found $R = 10$ to give sufficiently consistent measurements for the experiments in this section. The total runtime t_s includes only the computation of the matrix–matrix product; for large enough problems, any other tasks, such as splitting the high precision input matrices into low precision ones, have negligible performance overhead. Note that the definition of throughput in (4.1) corresponds to the usual Tflop/s for fp32 and fp16, but not for double-fp16, which performs $p(p+1)n^3$ (or twice as many, if all products are computed) rather than $2n^3$ flops.

In Figure 4.1, fp32 arithmetic attains a maximum throughput of 14 Tflop/s on the V100 and 19 Tflop/s on the A100, whereas fp16 arithmetic can achieve the much higher rates of 91 Tflop/s and 158 Tflop/s on the V100 and A100, respectively. These figures are relatively close to the corresponding theoretical peak performance of each arithmetic, as reported in Table 4.1. While we are not always able to reach the peak performance, our measurements are consistent with those of other independent studies [26], [35]. Turning now to Algorithm 4.1, which implements double-fp16 arithmetic, we achieve a maximum performance of 30 effective Tflop/s on the V100 and of 76 effective Tflop/s on the A100. Compared with fp16 precision, the double-fp16 approach is thus about 3× slower on both the V100 and the A100. This is expected, as both methods rely on the tensor core, but double-fp16 performs three times as many flops. More importantly, double-fp16 arithmetic is up to 2.2× faster than fp32 arithmetic on the V100, and up to 7.3× faster on the A100. The speedup is generally higher on the A100 compared with the V100; this is expected, as the performance ratio between fp16 arithmetic on the tensor cores and fp32 arithmetic is also different between the two cards.

4.2.2. Accuracy. The performance results above are very positive: we find double-fp16 arithmetic to be much faster than fp32 arithmetic, while possessing an almost identical error bound of order $n \times 2^{-24}$ (see Table 2.1). We now seek to confirm

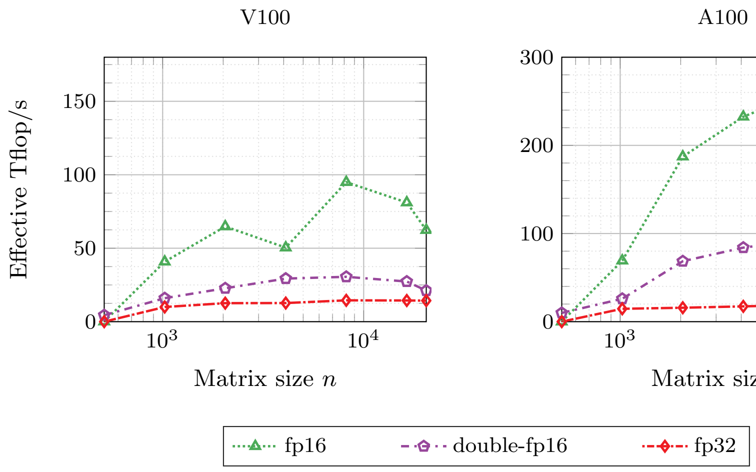

FIG. 4.1. *Throughput of GPU implementations of algorithms for computing the product AB, where $A, B \in \mathbb{R}^{n\times n}$. The methods and arithmetics used are discussed in section 4.2.1.*

experimentally whether double-fp16 arithmetic can indeed deliver the same accuracy as fp32. We will see that this is not always the case.

In the following experiments we consider two matrices $A \in \mathbb{R}^{m\times n}$ and $B \in \mathbb{R}^{n\times q}$, and we set the outer dimensions m and k to 16 while we vary n. By doing so, we can measure the accuracy for larger values of n without hitting the memory limit of a single GPU. We generate random matrices A and B with entries drawn from the uniform distribution over the intervals $(0, 1]$ or $(-0.5, 0.5]$. We then split A and B into four fp16 matrices such that $\mathrm{fl}_{64}(A) = \mathrm{fl}_{32}(A) = A_1 + A_2$ and $\mathrm{fl}_{64}(B) = \mathrm{fl}_{32}(B) = B_1 + B_2$.

In Figure 4.2 we plot the normwise and componentwise relative errors obtained by computing the product $C = AB$ in fp32, fp16, and double-fp16 arithmetics on both V100 and A100 GPUs. For matrices with entries sampled from the interval $(-0.5, 0.5]$, double-fp16 arithmetic is often significantly more accurate than fp16 arithmetic, but not always as accurate as fp32, especially on the A100. All three arithmetics provide results that are much more accurate than what the worst-case error bounds would suggest; this is due not only to statistical effects in the accumulation of rounding errors, but also to the fact that the matrix entries have zero mean [19]. The results for matrices with entries in $(0, 1]$ are much worse. We observe a severe accumulation of rounding errors that leads double-fp16 arithmetic to be no more accurate than fp16 arithmetic for large values of n, and much less accurate than fp32 arithmetic. There is no breach in the theory: the worst-case error bounds in Table 2.1 are not violated, but the error of double-fp16 arithmetic attains its worst-case bound, growing linearly with n, whereas the error of fp32 arithmetic maintains a slower error growth well below the bound. To our knowledge, this is the first time that this behaviour of the tensor cores for data in $[0, 1)$ is reported in the literature. This may be explained by the fact that previous work on multiword arithmetic with tensor cores [15], [26], [34] mostly focused on matrices with zero mean entries.

In the next section, we investigate the causes of this behavior and show that it is related to the rounding mode used by the NVIDIA tensor cores.

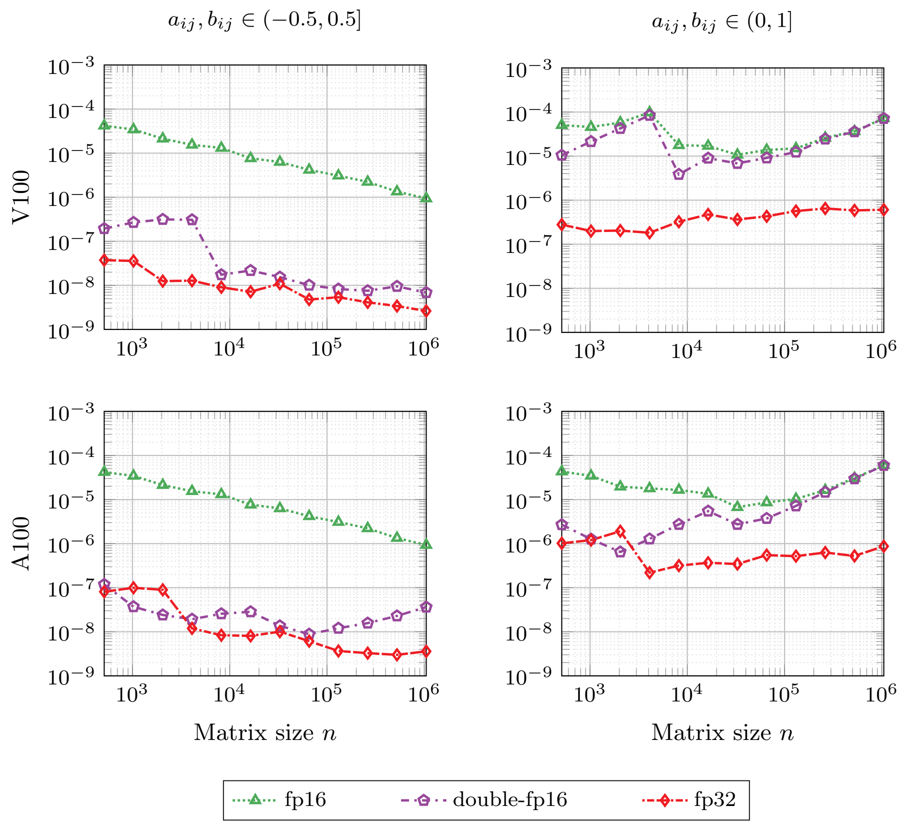

FIG. 4.2. *Componentwise relative error of GPU implementations of algorithms for computing the product AB. The methods and arithmetics used are discussed in section 4.2.2. The double-fp16 matrices $A \in \mathbb{R}^{16\times n}$ and $B \in \mathbb{R}^{n\times 16}$ have entries sampled uniformly at random from the interval at the top.*

4.3. Effects of round-toward-zero in NVIDIA tensor cores. One of the main numerical features that we have identified when studying the numerical behavior of the tensor cores [11] is that these units compute dot products using round-toward-zero (RZ) rather than the more common round-to-nearest (RN). We now explain why this difference may be the cause of the issue described in the previous section.

By using a probabilistic rounding error analysis one can typically replace, in worst-case error bounds such as (2.5), (2.7), or (2.8), the constants depending on the problem dimension (n in this case) by their square root [18]. Hence, probabilistic analogues of the bounds in Table 2.1 can be obtained by replacing n with $\sqrt{n}$. However, probabilistic error analysis is based on a model that assumes that rounding errors are random variables of zero mean. Whether this assumption holds or not may well depend on the rounding mode used to carry out the computation: for stochastic rounding, for example, the model is always valid [9], [10]. RN does not guarantee it, but the assumption has been observed to hold in many cases in practice, thus standard floating-point arithmetic with RN usually benefits from the reduced $\sqrt{n}$ error growth. With RZ, on the other hand, the assumption does not hold if the data have

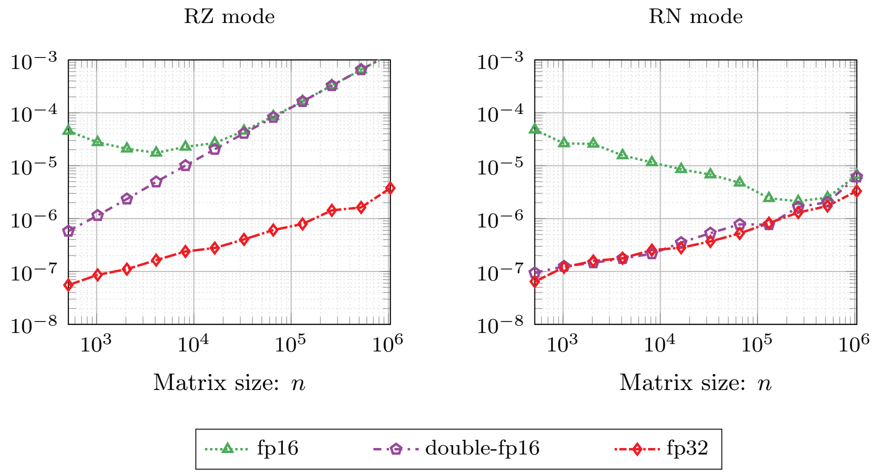

Fig. 4.3. *Componentwise relative error of algorithms for computing the product AB using simulated block FMAs. The methods and arithmetics used are discussed in section 4.3. Two rounding modes are used: round-toward-zero* (*left*) *and round-to-nearest* (*right*). *The double-fp16 matrices* $A \in \mathbb{R}^{16\times n}$ *and* $B \in \mathbb{R}^{n\times 16}$ *have entries sampled uniformly at random from the interval* $(0, 1]$.

nonzero mean: since the sign of the partial sums in the dot products remains constant throughout the computation, RZ always rounds in the same direction, and the rounding errors all have the same sign. For zero-mean data, such as those sampled from the uniform distribution over the interval $(-0.5, 0.5]$, the issue can still occur if the partial sums computed in the evaluation of the dot product remain of the same sign for many consecutive additions. However, since the data is uniformly distributed around zero, positive and negative rounding errors are equally likely, and we can expect to benefit, at least partially, from statistical error cancellation.

We now seek to confirm experimentally that the rounding mode of tensor cores is the cause behind the underwhelming results observed in Figure 4.2. To do so, we rely on a software emulator, implemented using the `CPFloat` library [12], which aims to behave as closely as possible to the NVIDIA tensor cores [11]. With this tool, we can switch the rounding mode from RZ to RN and assess the impact of this change on the final accuracy delivered by the algorithms. The CUDA function `cublasGemmEx` implements many matrix multiplication algorithms and selects the one to use for a given input heuristically at runtime. Details on these algorithms are not publicly available, thus we had to roughly match the results of the simulation to those obtained when running on the tensor cores. The official NVIDIA documentation,[2] indicates that the algorithms can be based on a blocked summation approach, which means that different blocks of the dot product are computed independently first and then combined together. In our simulation, we split the dot products into 16 blocks, that is, we use a variable block size equal to $n/16$ for all arithmetics. Each product of two blocks is computed with the emulated V100 tensor cores, that is, using a block FMA matrix multiply [6, Alg. 3.1] of dimension 4.

In Figure 4.3 we show the errors obtained by RZ and RN in the simulation. Since emulating the behavior of the tensor cores has a rather negative impact on

[2]https://docs.nvidia.com/cuda/archive/11.6.1/cublas/index.html#gemm-algorithms

Algorithm 4.2: `FABsum-v1`: compute intra-block sums with tensor cores and inter-block sums in standard fp32 arithmetic.

Input : Two fp16 matrices $A \in \mathbb{R}^{m\times n}$ and $B \in \mathbb{R}^{n\times q}$ partitioned into $b_1 \times b$ blocks A_{ik} and $b \times b_2$ blocks B_{kj}.
Output: The fp32 matrix $C = AB$.

```
1 Initialize $C$ to the zero matrix stored in fp32.
2 for $i \leftarrow 1$ to $m/b_1$ do
3     for $j \leftarrow 1$ to $q/b_2$ do
4         for $k \leftarrow 1$ to $n/b$ do
5             Compute $D \leftarrow A_{ik}B_{kj}$ using the tensor cores (fp16 arithmetic).
6             Compute $C_{ij} \leftarrow C_{ij} + D$ using fp32 arithmetic.
```

performance, for this experiment we set the outer dimensions of the matrix factors to the smallest size that is necessary to fill the simulated tensor cores, and we consider the product of two matrices $A \in \mathbb{R}^{m\times n}$ and $B \in \mathbb{R}^{n\times q}$ with $m = q = 16$. The results for RZ in Figure 4.3 (left plot) should be compared with those from the actual tensor cores in the right panel of Figure 4.2: these are relatively similar, which indicates that the behavior of our simulator is not dissimilar from that of the algorithm chosen by `cublasGemmEx`. Importantly, this experiment demonstrates that switching from RZ to RN does fix the issue observed previously, as double-fp16 and fp32 now deliver comparable accuracy even for data sampled from the interval $(0, 1]$.

4.4. Using `FABsum` to reduce the accumulation of errors caused by RZ. In this last section, we seek a cure for the accumulation of errors caused by the rounding behavior of the NVIDIA tensor cores. As it is not possible to change the rounding mode these hardware units use (as far as we are aware), we propose the use of a more accurate summation algorithm within the dot product that underlies the computation of each element of the matrix–matrix product. Specifically, we consider the use of the `FABsum` (fast and accurate blocked summation) algorithm [7]. Like standard blocked summation algorithms, `FABsum` splits the summands into blocks of size b. Unlike other techniques, however, `FABsum` uses two different summation algorithms: the sums of the elements within each block are computed with a fast algorithm, and the partial sums are then accumulated using a more accurate algorithm in higher precision.

Here we investigate two variants of `FABsum`, described in Algorithms 4.2 and 4.3. Both instances use fp16 arithmetic with tensor cores to compute the intra-block dot products, whilst the inter-block sums are computed with standard floating-point arithmetic: Algorithm 4.2 (`FABsum-v1`) uses fp32 arithmetic, whereas Algorithm 4.3 (`FABsum-v2`) uses fp64 arithmetic. `FABsum-v1` has a double advantage over computing the entire dot product with tensor cores. First, the use of blocked summation reduces the worst-case error bound from $n \times 2^{-24}$ to $(b + n/b) \times 2^{-24}$. Second, since the inter-block dot products are computed with standard arithmetic and thus using RN, we can expect the bound to hold with the constant n/b replaced with its square root. `FABsum-v2` further reduces the worst-case error bound to $b \times 2^{-24} + n/b \times 2^{-53}$, and can therefore be more accurate than `FABsum-v1` for large n.

Algorithm 4.3: `FABsum-v2`: compute intra-block sums with tensor cores and inter-block sums in standard fp64 arithmetic.

```
Input  : Two fp16 matrices $A \in \mathbb{R}^{m\times n}$ and $B \in \mathbb{R}^{n\times q}$ partitioned into $b_1 \times b$
         blocks $A_{ik}$ and $b \times b_2$ blocks $B_{kj}$.
Output: The fp32 matrix $C = AB$.
1 Initialize $C$ to the zero matrix stored in fp64.
2 for $i \leftarrow 1$ to $m/b_1$ do
3     for $j \leftarrow 1$ to $q/b_2$ do
4         for $k \leftarrow 1$ to $n/b$ do
5             Compute $D \leftarrow A_{ik}B_{kj}$ using the tensor cores (fp16 arithmetic).
6             Compute $C_{ij} \leftarrow C_{ij} + D$ using fp64 arithmetic.
7 $C \leftarrow \mathrm{fl}_{32}(C)$
```

To implement `FABsum` we rely on the CUTLASS library.[3] The library provides an efficient routine for matrix–matrix multiplication that exploits blocked summation (this operation is called "SplitK" in the library) and allows the user to freely choose the block size b. For `FABsum-v2`, we have modified the routine to perform the inter-block accumulation in fp64 arithmetic, which yields an implementation of `FABsum` that is both accurate and efficient. Nevertheless, compared with the `cublasGemmEx` baseline, the use of a more accurate summation algorithm carries a relatively significant performance penalty, especially for smaller values of the block size b. Therefore, our aim will be to use `FABsum` to find a better tradeoff between the fast but inaccurate cuBLAS-based double-fp16 arithmetic and the accurate but slower fp32 arithmetic.

The naive approach to improve the accuracy of the cuBLAS-based implementation would be to replace every call to `cublasGemmEx` in Algorithm 4.1 with a call to `FABsum`. In light of the error analysis in section 2, and specifically of (2.6), this is not necessary: the entries of $|A_1||B_2|$ and $|A_2||B_1|$ are of order $2^{-11}|A||B|$, thus the error in computing the products A_1B_2 and A_2B_1 is bounded by $n \times 2^{-11} \times 2^{-24}$. Assuming that with standard fp32 arithmetic the accuracy follows the probabilistic error bound $\sqrt{n}\times 2^{-24}$, the error introduced by the A_1B_2 and A_2B_1 products can only become significant for n larger than $2^{22} \approx 4 \times 10^6$. As a result, for matrices of order less than about four million, applying `FABsum` only to the first order product A_1B_1 should be sufficient. The resulting method is given in Algorithm 4.4.

In Figure 4.4, we assess the performance and accuracy of Algorithm 4.4 for both versions of `FABsum` and different choices of the block size b. We compare these implementations against the cuBLAS-based codes in both double-fp16 and fp32 arithmetics. As expected, `FABsum-v2` is more expensive to use than `FABsum-v1`, but preserves a high accuracy even when n is very large (compare the solid and dashed blue lines in the figure). Also as expected a smaller block size b reduces both error and throughput (compare different marker types in the figure). Interestingly, given a desired level of performance–accuracy tradeoff, the best variant to choose depends on the problem size n. For example, let us compare `FABsum-v1` with $b = 128$ (solid light blue line with square markers) and `FABsum-v2` with $b = 256$ (dashed darker blue line with asterisk markers). These two variants have a very similar performance, but different accuracy: the latter is less accurate for small values of n because of the larger value

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

Algorithm 4.4: `MMM` algorithm (Algorithm 2.1) using `FABsum` for the first order product.

```
Input  : Two fp32 matrices $A \in \mathbb{R}^{m\times n}$ and $B \in \mathbb{R}^{n\times q}$.
Output: The fp32 matrix $C \approx AB$ computed using double-fp16 arithmetic.
1 $C \leftarrow 0$
2 $A_1 \leftarrow \mathrm{fl}_{16}(A)$
3 $A_2 \leftarrow \mathrm{fl}_{16}(A - A_1)$
4 $B_1 \leftarrow \mathrm{fl}_{16}(B)$
5 $B_2 \leftarrow \mathrm{fl}_{16}(B - B_1)$
6 Compute $C \leftarrow C + A_2B_1$ with cublasGemmEx.
7 Compute $C \leftarrow C + A_1B_2$ with cublasGemmEx.
8 Compute $C \leftarrow C + A_1B_1$ with Algorithm 4.2 or Algorithm 4.3 (FABsum).
```

of b, but becomes more accurate once n is large enough to make the n/b term in the error dominate. In fact, the two lines cross in the figure at $n \approx 3 \times 10^5$ for the V100 and at $n \approx 10^5$ for the A100.

Comparing against the cuBLAS-based implementations now, we see that the `FABsum`-based algorithms achieve a flexible and significantly improved tradeoff between performance and accuracy. With a moderately large block size and the `FABsum-v1` version, Algorithm 4.4 can be as fast as Algorithm 4.1 whilst remaining significantly more accurate, especially for large n. Alternatively, using the `FABsum-v2` version with a smaller block size, Algorithm 4.4 can match the accuracy of fp32 arithmetic while remaining significantly faster.

We conclude by investigating the use of `FABsum` in fp16 arithmetic. As we rely on `FABsum` to improve the accuracy of double-fp16 arithmetic, for fairness we should check what we obtain by applying the same approach directly to the faster fp16 arithmetic. For very large n, we can expect the error term corresponding to the accumulation within the inner products to exceed the errors caused by the conversion to fp16. Figure 4.5 shows the performance and accuracy of `FABsum-v2` with fp16 arithmetic for several block sizes, and compares it to that of `FABsum-v2` in double-fp16 arithmetic with fixed block size $b = 256$. Only results on A100 are shown, those on V100 being similar. For the smallest block size ($b = 128$), fp16 is indeed as accurate as double-fp16 with $b = 256$ for $n \approx 10^6$, but in this case double-fp16 is faster because of the larger block size. For $b = 256$ or larger, fp16 becomes faster than double-fp16, but does not reach the same level of accuracy. We can conjecture that this will eventually happen for larger values of n. The conclusion is that the use of double-fp16 arithmetic is of interest for a wide range of matrix dimensions.

5. Conclusion.

The emergence of hardware with accelerators for low precision arithmetics has generated renewed interest in multiword arithmetic as a way to obtain fp32 accuracy using fp16 arithmetic. We have proposed a general class of multiword matrix multiplication algorithms based on the block FMA framework [6] and have carried out their error analysis.

We have implemented our algorithms and run them on NVIDIA GPUs equipped with tensor cores. We have identified some cases where double-fp16 arithmetic is, unexpectedly, unable to achieve full fp32 accuracy. With the help of probabilistic rounding error analysis we have showed that a possible cause is the fact that these devices use round-toward-zero rather than round-to-nearest [11], [16]. To support

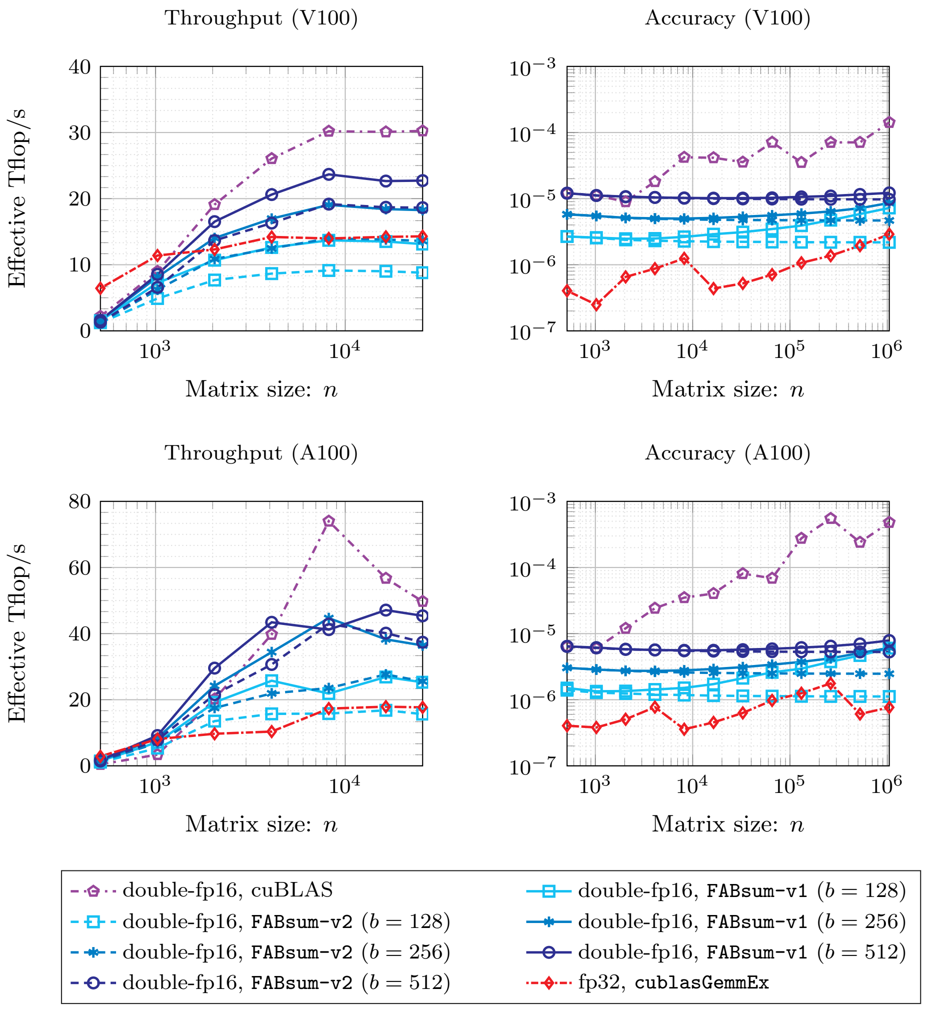

FIG. 4.4. *Throughput (left) and componentwise relative error (right) of GPU implementations of algorithms for computing the product AB. Implementations of Algorithm 4.1 (MMM using cuBLAS) and of Algorithm 4.4 (MMM using `FABsum`), using either Algorithm 4.2 (`FABsum-v1`) or Algorithm 4.3 (`FABsum-v2`), are benchmarked against the `cublasGemmEx` using fp32 arithmetic. In the left panel $A, B \in \mathbb{R}^{n\times n}$, whereas in the right panel the double-fp16 matrices $A \in \mathbb{R}^{512\times n}$ and $B \in \mathbb{R}^{n\times 512}$ have entries sampled uniformly at random from the interval* $(0, 1]$.

our conclusion, we have developed a simulator of the tensor cores and have showed that switching between the two rounding modes has indeed the expected impact on accuracy. Finally, we have explained how to alleviate the issue by using blocked summation algorithms based on `FABsum` [7], which allow for a flexible tradeoff between accuracy and performance when using block FMA hardware such as the tensor cores.

With this improvement, we have obtained multiword matrix multiplication algorithms that can achieve fp32 accuracy and have a throughput significantly higher than that of standard fp32 arithmetic. We expect that multiword arithmetic algorithms will become increasingly attractive as more hardware devices with limited support for

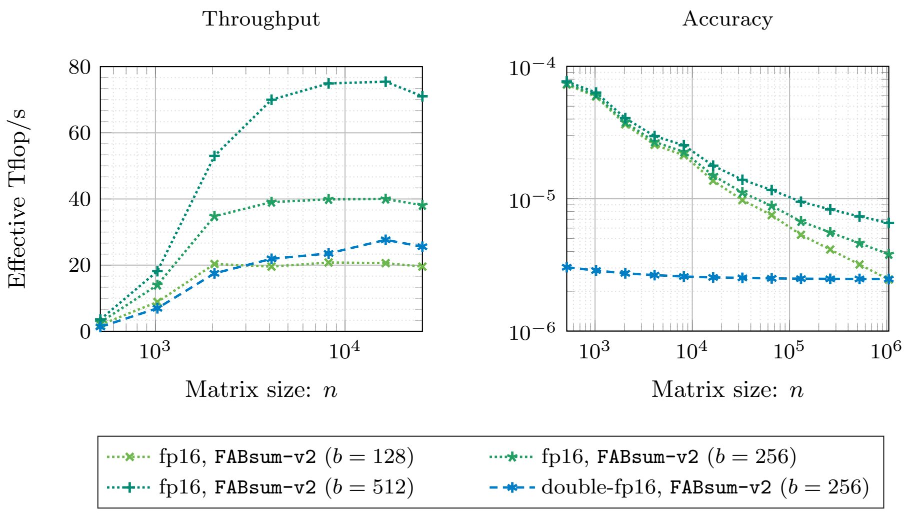

FIG. 4.5. *Throughput (left) and componentwise relative error (right) of GPU (A100) implementations of algorithms for computing the product AB. Several implementations of Algorithm* 4.3 *(`FABsum-v2`) in fp16 arithmetic are compared with a selected version of `FABsum` from Figure* 4.4 *(`FABsum-v2` with $b = 256$). In the left plot we take $A, B \in \mathbb{R}^{n\times n}$, whereas in the right plot the double-fp16 matrices $A \in \mathbb{R}^{512\times n}$ and $B \in \mathbb{R}^{n\times 512}$ have entries sampled uniformly at random from the interval* $(0, 1]$. *The objective of this experiment is to check that double-fp16 arithmetic remains of interest even when `FABsum` is used.*

high precision arithmetic appear.

Acknowledgements. We are grateful to Srikara Pranesh for early discussions on the matrix multiplication algorithm in multiword arithmetic with tensor cores. We thank the Innovative Computing Laboratory at the University of Tennessee, Knoxville, US for providing access to the NVIDIA A100 graphics cards, and the University of Manchester for providing access to the NVIDIA V100 and A100 graphic cards through the Computational Shared Facility.

REFERENCES

[1] A. Abdelfattah, H. Anzt, E. G. Boman, E. Carson, T. Cojean, J. Dongarra, A. Fox, M. Gates, N. J. Higham, X. S. Li, J. Loe, P. Luszczek, S. Pranesh, S. Rajamanickam, T. Ribizel, B. F. Smith, K. Swirydowicz, S. Thomas, S. Tomov, Y. M. Tsai, and U. M. Yang, *A survey of numerical linear algebra methods utilizing mixed-precision arithmetic*, Int. J. High Perform. Comput. Appl., 35 (2021), pp. 344–369, https://doi.org/10.1177/10943420211003313.

[2] *"AMD Instinct MI200" Instruction Set Architecture. Reference Guide*, Advanced Micro Devices, Inc., Nov. 2021, https://developer.amd.com/wp-content/resources/CDNA2_Shader_ISA_18November2021.pdf.

[3] *AMD Instinct™ MI200 Series Accelerator*, Advanced Micro Devices, Inc., Nov. 2021, https://www.amd.com/system/files/documents/amd-instinct-mi200-datasheet.pdf.

[4] *Introducing AMD CDNA™ 2 Architecture*, Advanced Micro Devices, Inc., Nov. 2021, https://www.amd.com/system/files/documents/amd-cdna2-white-paper.pdf.

[5] H. Anzt, Y. M. Tsai, A. Abdelfattah, T. Cojean, and J. Dongarra, *Evaluating the performance of NVIDIA's A100 Ampere GPU for sparse and batched computations*, in Proceedings of the 2020 IEEE/ACM Performance Modeling, Benchmarking and Simulation of High Performance Computer Systems, GA, USA, Jan. 2021, pp. 26–38, https://doi.org/

10.1109/PMBS51919.2020.00009.

[6] P. Blanchard, N. J. Higham, F. Lopez, T. Mary, and S. Pranesh, *Mixed precision block fused multiply-add: Error analysis and application to GPU tensor cores*, SIAM J. Sci. Comput., 42 (2020), pp. C124–C141, https://doi.org/10.1137/19M1289546.

[7] P. Blanchard, N. J. Higham, and T. Mary, *A class of fast and accurate summation algorithms*, SIAM J. Sci. Comput., 42 (2020), pp. A1541–A1557, https://doi.org/10.1137/19M1257780.

[8] J. Choquette, E. Lee, R. Krashinsky, V. Balan, and B. Khailany, *3.2 the A100 datacenter GPU and Ampere architecture*, in Proceedings of the 2021 IEEE International Solid-State Circuits Conference, San Francisco, CA, USA, Mar. 2021, pp. 48–50, https://doi.org/10.1109/ISSCC42613.2021.9365803.

[9] M. P. Connolly, N. J. Higham, and T. Mary, *Stochastic rounding and its probabilistic backward error analysis*, SIAM J. Sci. Comput., 43 (2021), pp. A566–A585, https://doi.org/10.1137/20m1334796.

[10] M. Croci, M. Fasi, N. J. Higham, T. Mary, and M. Mikaitis, *Stochastic rounding: Implementation, error analysis and applications*, Roy. Soc. Open Sci., 9 (2022), pp. 1–25, https://doi.org/10.1098/rsos.211631.

[11] M. Fasi, N. J. Higham, M. Mikaitis, and S. Pranesh, *Numerical behavior of NVIDIA tensor cores*, PeerJ Comput. Sci., 7 (2021), pp. e330(1–19), https://doi.org/10.7717/peerj-cs.330.

[12] M. Fasi and M. Mikaitis, *CPFloat: A C library for emulating low-precision arithmetic*, MIMS EPrint 2020.22, Manchester Institute for Mathematical Sciences, The University of Manchester, UK, Oct. 2020, http://eprints.maths.manchester.ac.uk/2785/. Updated May 2022.

[13] A. Haidar, H. Bayraktar, S. Tomov, J. Dongarra, and N. J. Higham, *Mixed-precision iterative refinement using tensor cores on GPUs to accelerate solution of linear systems*, Proc. Roy. Soc. London A, 476 (2020), p. 20200110, https://doi.org/10.1098/rspa.2020.0110.

[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, SC18 (Dallas, TX), Piscataway, NJ, USA, 2018, IEEE, pp. 47:1–47:11, https://doi.org/10.1109/SC.2018.00050.

[15] G. Henry, P. T. P. Tang, and A. Heinecke, *Leveraging the bfloat16 artificial intelligence datatype for higher-precision computations*, in Proceedings of the 26th IEEE Symposium on Computer Arithmetic, Kyoto, Japan, Oct. 2019, pp. 69–76, https://doi.org/10.1109/ARITH.2019.00019.

[16] B. Hickmann and D. Bradford, *Experimental analysis of matrix multiplication functional units*, in Proceedings of the 26th IEEE Symposium on Computer Arithmetic, Kyoto, Japan, Oct. 2019, pp. 116–119, https://doi.org/10.1109/ARITH.2019.00031.

[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*, SIAM J. Sci. Comput., 42 (2020), pp. A3427–A3446, https://doi.org/10.1137/20M1314355.

[20] N. J. Higham and T. Mary, *Mixed precision algorithms in numerical linear algebra*, Acta Numerica, 31 (2022), pp. 347–414, https://doi.org/10.1017/s0962492922000022.

[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 754-2008 (revision of IEEE Std 754-1985)*, IEEE, Piscataway, NJ, USA, Aug. 2008, https://doi.org/10.1109/IEEESTD.2008.4610935.

[23] *IEEE Standard for Floating-Point Arithmetic, IEEE Std 754-2019 (revision of IEEE Std 754-2008)*, IEEE, Piscataway, NJ, USA, 2019, https://doi.org/10.1109/IEEESTD.2008.4610935.

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

[25] F. Lopez and T. Mary, *Mixed precision LU factorization on GPU tensor cores: Reducing data movement and memory footprint*, Sept. 2020, http://eprints.maths.manchester.ac.uk/

2782/. Submitted to *ACM Trans. Math. Softw.*

[26] S. Markidis, S. W. D. Chien, E. Laure, I. B. Peng, and J. S. Vetter, *NVIDIA tensor core programmability, performance & precision*, in Proceedings of the 32nd IEEE International Parallel and Distributed Processing Symposium Workshops, Vancouver, BC, Canada, Aug. 2018, pp. 522–531, https://doi.org/10.1109/IPDPSW.2018.00091.

[27] D. Mukunoki and T. Ogita, *Performance and energy consumption of accurate and mixed-precision linear algebra kernels on GPUs*, J. Comput. Appl. Math., 372 (2020), https://doi.org/10.1016/j.cam.2019.112701.

[28] D. Mukunoki, K. Ozaki, T. Ogita, and T. Imamura, *DGEMM using tensor cores, and its accurate and reproducible versions*, in High Performance Computing, P. Sadayappan, B. L. Chamberlain, G. Juckeland, and H. Ltaief, eds., Cham, June 2020, Springer International Publishing, pp. 230–248.

[29] J.-M. Muller, N. Brunie, F. de Dinechin, C.-P. Jeannerod, M. Joldes, V. Lefèvre, G. Melquiond, N. Revol, and S. Torres, *Handbook of Floating-Point Arithmetic*, Birkhäuser, Cham, Switzerland, second ed., 2018, https://doi.org/10.1007/978-3-319-76526-6.

[30] *NVIDIA Tesla V100 GPU Architecture*, NVIDIA, 2017, https://images.nvidia.com/content/volta-architecture/pdf/volta-architecture-whitepaper.pdf. NVIDIA whitepaper WP-08608-001_v1.1.

[31] *NVIDIA A100 Tensor Core GPU architecture*, NVIDIA, 2020, https://www.nvidia.com/content/dam/en-zz/Solutions/Data-Center/nvidia-ampere-architecture-whitepaper.pdf. NVIDIA whitepaper v1.0.

[32] *NVIDIA H100 Tensor Core GPU architecture*, NVIDIA, 2022. NVIDIA whitepaper v1.2. Online: https://nvdam.widen.net/content/tdwwiwotwr/original/gtc22-whitepaper-hopper.pdf.

[33] L. Pisha and Ł. Ligowski, *Accelerating non-power-of-2 size Fourier transforms with GPU tensor cores*, in Proceedings of the 2021 IEEE International Parallel and Distributed Processing Symposium, Portland, OR, USA, May 2021, pp. 507–516, https://doi.org/10.1109/IPDPS49936.2021.00059.

[34] A. Sorna, X. Cheng, E. D'Azevedo, K. Won, and S. Tomov, *Optimizing the fast Fourier transform using mixed precision on tensor core hardware*, in Proceedings of the 25th IEEE International Conference on High Performance Computing Workshops, Bengaluru, India, 2018, pp. 3–7, https://doi.org/10.1109/hipcw.2018.8634417.

[35] D. Yan, W. Wang, and X. Chu, *Demystifying tensor cores to optimize half-precision matrix multiply*, in Proceedings of the 2020 IEEE International Parallel and Distributed Processing Symposium, New Orleans, LA, USA, July 2020, pp. 634–643, https://doi.org/10.1109/IPDPS47924.2020.00071.