Numerical Behavior of the NVIDIA Tensor Cores

Massimiliano Fasi, Nicholas J. Higham, Mantas Mikaitis, and Srikara Pranesh
Department of Mathematics, The University of Manchester, Manchester, UK
{massimiliano.fasi, nick.higham, mantas.mikaitis, srikara.pranesh}@manchester.ac.uk

***Abstract*—We explore the floating-point arithmetic used by the NVIDIA Volta tensor cores, which are hardware accelerators for mixed-precision matrix multiplication. We investigate what precision is used for intermediate results, whether subnormal numbers are supported, what rounding mode is used, in which order the operations in the dot products arising in the matrix multiplication are performed, and whether partial sums are normalized. These aspects are not documented by NVIDIA, and we gain insight by running carefully designed numerical experiments on these hardware accelerators. Knowing the answers to these questions is important if one wishes to: 1) build hardware that computes a matrix-matrix product matching the results of NVIDIA tensor cores; 2) achieve bit-reproducible results when designing on conventional hardware with IEEE 754 floating-point arithmetic code meant to run on NVIDIA tensor cores; and 3) understand the differences between results produced by code that utilizes tensor cores and code that uses only IEEE 754-compliant arithmetic operations. As an additional result, we point out a non-monotonicity issue that arises in floating-point multi-operand addition without the normalization of the intermediate results.**

***Index Terms*—NVIDIA V100 GPU, tensor core, dot product, matrix multiply-accumulate, rounding, floating-point arithmetic, half precision, binary16, fp16, IEEE 754 arithmetic**

I. Introduction

Ninety-four of the supercomputers in the November 2019 TOP500 list[1] are equipped with NVIDIA GPUs featuring Volta chips. A prominent feature of the Volta microarchitecture are the tensor cores, specialized hardware accelerators for performing the matrix multiply-accumulate operation

$$D = AB + C, \tag{1}$$

where A, B, C, and D are 4×4 matrices. The entries of A and B must be in binary16 format, whereas those of C and D can be either binary16 or binary32 floating-point numbers depending on the accumulation mode. The element d_{ij} can be seen as the sum of c_{ij} and the dot product between the ith row of A and the jth column of B, so that, for instance

$$d_{11} = a_{11}b_{11} + a_{12}b_{21} + a_{13}b_{31} + a_{14}b_{41} + c_{11}. \tag{2}$$

Unfortunately, NVIDIA provides very little information about the numerical features of these units, and many questions naturally arise. The white paper that describes the Volta microarchitecture [1, p. 15] states that[2]

Tensor Cores operate on FP16 input data with FP32 accumulation. The FP16 multiply results in a full precision product that is then accumulated using FP32 addition with the other intermediate products for a $4 \times 4 \times 4$ matrix multiply.

The official documentation [3] adds only a few more details:

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. The accumulation order, rounding and handling of subnormal inputs is unspecified.

From a numerical point of view, many essential aspects of tensor cores are not specified. This makes building hardware that can match the numerical results of tensor cores challenging, renders bit reproducibility between conventional IEEE 754-compliant systems and tensor cores hard to achieve, and can lead to unexpected differences between the results computed on NVIDIA devices with tensor cores enabled or disabled. The IEEE 754 standard provides a somewhat relaxed set of requirements for reduction operations such as multi-operand addition and dot product [2, Sec. 9.4], since it does not prescribe the order in which the partial sums should be evaluated and allows the use of a higher-precision internal format. In particular, the standard does not specify: 1) whether this internal format should be normalized, as it would be if the multi-operand addition were implemented using IEEE 754 elementary arithmetic operations, 2) which rounding mode should be used, and 3) when the rounding should happen. These loose requirements can potentially cause the results computed with a given multi-operand addition unit to be significantly different from those obtained using other hardware implementations or a software implementation based on IEEE 754-compliant elementary arithmetic operations.

With matrix multiplication being ubiquitous in artificial intelligence, accelerators for mixed-precision matrix multiply-

TABLE I
Processing units equipped with mixed-precision matrix multiply-accumulate accelerators.

	matrix dimensions	input	acc.	
('16) Google TPU v2	$128 \times 128 \times 128$	bfloat16	binary32	[4]
('17) Google TPU v3	$128 \times 128 \times 128$	bfloat16	binary32	[4]
('17) NVIDIA V100	$4 \times 4 \times 4$	binary16	binary32	[1]
('18) NVIDIA T4	$4 \times 4 \times 4$	binary16	binary32	[5]
('19) ARMv8.6-A arch.	$2 \times 4 \times 2$	bfloat16	binary32	[6]

[1] https://www.top500.org/lists/2019/11

[2] The IEEE 754 [2] formats binary16 and binary32 are sometimes referred to as fp16 (or FP16) and fp32 (or FP32), respectively.

accumulate operations are becoming widely available, as Table I shows. Hardware vendors often design these units focusing on performance rather than numerical reliability, and this may lead to the implementation of unconventional, non-IEEE-compliant arithmetics. Some of the hardware units in Table I, for instance, use bfloat16, a 16-bit format with an 8-bit significand (including the implicit bit) and an 8-bit exponent [7] which does not support subnormals. In order to better understand the differences between the results computed using different systems, it is necessary to develop techniques to probe the numerical features of these units. This is not much different from the situation with single and double precision before the wide adoption of the IEEE 754-1985 standard.

Using idiosyncrasies of floating-point arithmetic, we design tests to better understand the numerical behavior of tensor cores. Our aim is to clarify the following points.

- Are subnormal inputs supported or are they flushed to zero? Can tensor cores produce subnormal numbers?
- Are the multiplications in (2) exact and the additions performed in binary32 arithmetic, resulting in four rounding errors for each element of D? In what order are the four additions in (2) performed?
- What rounding mode is used in (2)?
- Where is floating-point normalization done in tensor cores and what rounding mode is used?

The results discussed here were produced by running our test suite[3] on an NVIDIA Tesla V100 SXM2 16GB (Volta microarchitecture) graphic card. We stress, however, that the ideas in Section III are very general, and can be exploited to understand the numerical features of any hardware accelerator based on operations of the form (2). Finally, it is worth noting that binary16 arithmetic in NVIDIA CUDA cores is not fully IEEE 754 compliant, as round-to-nearest is the only rounding mode implemented for elementary arithmetic operations [8].

II. Previous work

Despite being purpose-built to accelerate training of deep neural networks [1, p. 12] tensor cores have found applications in traditional high-performance scientific computing. Markidis et al. [9] discuss various aspects of tensor cores and propose a technique, called precision refinement, to enhance the accuracy of mixed-precision matrix multiplication. Tensor cores have also been used to accelerate the solution of linear systems using mixed-precision iterative refinement [10], [11].

From a hardware perspective, instruction-level details, register configuration, and memory layout of the tensor cores in the NVIDIA Volta [12], [13] and Turing [13], [14] GPUs have been extensively described. Another study [15] explores the reliability of tensor cores in terms of reducing the hardware error rate of the matrix multiplications. The main finding is that low-precision operations and usage of tensor cores increases the amount of correct data produced by the GPU. In order to quantify the accuracy of tensor cores, Blanchard et al. [16] provide a rounding error analysis of what they call a block FMA, a generalization of the multiply-accumulate operation in (1) in which the precisions of the arguments and the internal precision of the accumulator are taken as parameters.

None of the these sources, however, examines whether tensor cores conform to the IEEE 754 standard or investigates how tensor cores compare with a matrix multiply-accumulate operation based on dot products implemented in software. Our work complement the results therein and supplies details on the numerical behavior of these hardware accelerators that are now quickly gaining popularity.

III. Experiments

Tensor cores can be accessed using the cuBLAS library, or the native hardware assembly instructions `HMMA.884` [12], [14] and `HMMA.1688` [13], [14]. In our experiments, we opted for the warp-level C++ function `wmma::mma_sync()`, which performs a $16 \times 16 \times 16$ matrix multiply-accumulate operation. This is the lowest level interface to access the tensor cores in the NVIDIA CUDA programming environment. In order to use only a single tensor core, we set all but the top left 4×4 blocks to 0. We ensure that our experiments do use the tensor cores by running our test suite with the NVIDIA profiler `nvprof`.

A. Support for subnormal numbers

We start by investigating the support for subnormal numbers, as this knowledge will dictate what range of input values we are allowed to use in further tests.

TABLE II
Properties of binary16 and binary32 floating-point formats.

Property	binary32	binary16
p (bits in the significand)	24	11
ε	2^{-23}	2^{-10}
$e_{\max}$	127	15
$e_{\min}$	-126	-14
Min. positive normal	2^{-126}	2^{-14}
Min. positive subnormal	$2^{-126} \times 2^{-23}$	$2^{-14} \times 2^{-10}$

Table II shows the precision p (number of bits in the significand, including the implicit bit), the machine epsilon $\varepsilon = 2^{1-p}$, the exponent range $[e_{\min}, e_{\max}]$, and the minimum normal and subnormal values for the two floating-point formats supported by tensor cores. Based on the table we highlight two points. First, conversion from binary16 to binary32 does not result in subnormal numbers. Second, the product of two binary16 numbers requires at most 22 bits for the significand, 6 bits for the exponent and one for the sign, and thus can be represented exactly in binary32 format. In terms of tensor cores, there are multiple questions regarding the support of subnormal numbers.

1) Can tensor cores take binary16 subnormal numbers as inputs for A and B in (2) without flushing them to zero, use them in computation, and return binary16 or binary32 normal or subnormal answers?

[3] Available at https://github.com/mfasi/tensor-cores-numerical-behavior.

2) Can tensor cores take binary32 subnormal numbers as inputs for C in (2) without flushing them to zero, use them in computation, and return subnormal binary32 answers?
3) Can tensor cores compute subnormal numbers from normal numbers and return them?

The first question can easily be answered by setting in (2) $a_{11} = 2^{-24}$, $b_{11} = 2^2$ (arbitrarily chosen), and the other elements to zero. Tensor cores return the subnormal result $a_{11}b_{11} = 2^{-22}$ in both binary16 and binary32 mode.

To clarify the second point, a similar idea can be used: setting c_{11} to the smallest positive binary32 subnormal 2^{-149} and A and B to zero matrix, yields $d_{11} = 2^{-149}$, which tells us that the subnormal c_{11} is not altered by the dot product in (2). We note, however, that the need for binary32 subnormals is questionable. The absolute value of the smallest nonzero value that can be produced from the multiplications of two binary16 numbers is 2^{-48}, thus c_{11} would simply be rounded off if it were a binary32 subnormal: in binary32 arithmetic with round-to-nearest $2^{-48} + x > 2^{-48}$ only if $x > 2^{-48} \times 2^{-24} = 2^{-72}$, which is normal in binary32 format.

For the third point, we can obtain subnormal numbers as outputs in a few ways. For instance, we can set a_{11} to 2^{-14}, the smallest normal number in binary16, and b_{11} to 2^{-1}, and confirm that tensor cores return the binary16 subnormal 2^{-15} both in binary16 and binary32 modes. Another possibility is to set $a_{11} = 2^{-14}$, $b_{11} = 1$, $c_{11} = -2^{-15}$, which produces the subnormal binary16 number $d_{11} = 2^{-15}$. As mentioned above, it is not possible to obtain subnormal binary32 number from binary16 inputs in (2). In summary, these experiments demonstrate that there is full support for subnormal inputs in tensor cores.

One might wonder whether tensor cores natively support subnormals or some degree of software interaction is present. The NVIDIA profiler confirms that the experiments discussed in this section make use of the tensor cores, but we implemented an additional test to further reinforce the evidence that subnormals are supported in hardware. In Section III-C we show that tensor cores use round-toward-zero. We can use the fact that CUDA cores provide only round-to-nearest for binary16 computations to show that subnormals are in fact manipulated with tensor cores. The test is to set a_{11} and a_{12} to 1, b_{11} to the binary16 subnormal $2^{-23} + 2^{-24}$, b_{21} to 2 and the other elements of A and B to 0. Since the addition in (2) is done in binary32 arithmetic, the smallest value that can be exactly added to $b_{21} = 2$ is 2^{-22}. Since $b_{11} = 2^{-23} + 2^{-24}$ is below this value and is $3/4$ of 2^{-22}, it will be either lost (round-toward-zero) or rounded up due to round-to-nearest if the summation is done in software to support subnormals. We found that the sum returned 2, which means b_{11} was rounded down—further indication that subnormals are supported in tensor cores.

B. Accuracy of the dot products

Our second goal is to test the accuracy of the dot product (2) with tensor cores. The first step is to check that the multiplications of binary16 values are computed exactly, which implies that the products must be kept in some wider intermediate format and accumulated without being rounded back to binary16. Specifically we want to test that $a_{11}b_{11}$ and the other products are exact. This can be achieved by ensuring that the four multiplications produce floating-point numbers that are not representable in binary16 and checking that these are preserved and returned as binary32 entries of D.

In order to demonstrate this, we set the first row of A and the first column of B to $1 - 2^{-11}$ and $c_{11} = 0$. Each of the partial products evaluates exactly to $(1 - 2^{-11}) \times (1 - 2^{-11}) = 1 - 2^{-10} + 2^{-22}$ which, if the products were stored in binary16 precision, would be rounded to $1 - 2^{-10}$ or $1 - 2^{-11}$, depending on the rounding mode. As tensor cores produce the exact binary32 answer $d_{11} = 4 \times (1 - 2^{-10} + 2^{-22})$, we conclude that partial products are held exactly.

Another question is whether the precision of the 5-term addition in (2) changes in any way when binary16 is used for C and D in (1). The test is to set $a_{11} = b_{11} = a_{12} = 1 - 2^{-11}$, $b_{21} = 2^{-11}$, and the remaining elements to 0. In this test, the first product is as before $a_{11}b_{11} = 1 - 2^{-10} + 2^{-22}$ and requires precision higher than binary16 to be represented, whereas the second evaluates to $a_{12}b_{21} = 2^{-11} - 2^{-22}$. The sum of these two products is $a_{11}b_{11} + a_{12}b_{21} = 1 - 2^{-10} + 2^{-11}$, which is representable in binary16 but could not be calculated exactly by binary16 accumulator, since the first product requires higher precision. Indeed we found that tensor cores output the exact value, confirming that the partial products are still held exactly even when C and D are in binary16 format.

A third question concerns the number of rounding errors in the 5-term addition of the partial products. The dot product in (2) contains 4 additions: 3 additions sum the exact partial products and a fourth adds the binary32 argument c_{11}. Our expectation is that the additions are done in binary32 rather than exactly, as indicated by [1], [3]. In order to confirm this, we can set the first row of A to 1, thereby reducing (2) to

$$d_{11} = b_{11} + b_{21} + b_{31} + b_{41} + c_{11}, \tag{3}$$

and then run 5 different cases with one of the addends in (3) set to 1 and the rest set to 2^{-24}. In this test, an exact addition would return $1 + 2^{-22}$, whereas inexact arithmetic would cause 4 round-off errors when adding 2^{-24} to 1, causing the number 1 to be returned. All permutations return $d_{11} = 1$, leading to the following conclusions.

- In the worst case each element of D includes four rounding errors, which conforms to the block FMA model used in [16, Sec 2.1].
- The partial products in (2) are not accumulated in a fixed order, but always starting from the value of largest magnitude. This sorting is necessary in order to know which arguments require to be shifted right in the significand alignment step of a standard floating-point addition algorithm [17, Sec. 7.3], [18], and is most likely done in hardware. This is in line with the literature on hardware dot products [19], [20], [21], [22], where

either sorting or a search for the maximum exponent is performed. Furthermore, this experiment demonstrates that none of the additions is performed in parallel with the first addition to the largest magnitude value: if evaluated before being shifted right, any other sum would return $2^{-24} + 2^{-24} = 2^{-23}$, a value that then could be added exactly to the total sum.

In summary, each entry of D in (1) can have up to four rounding errors and the 5-term additions to compute each element are performed starting from the largest summand in absolute value.

C. Rounding modes in tensor core computations

If binary32 mode is used, only the four additions in (2) can be subject to rounding errors. The IEEE 754 standard defines four rounding modes [2, Sec. 4.3], round-to-nearest, round-toward-zero, round-down, and round-up. In this section we use the notation defined in [17, Sec. 2.2.1] and denote the corresponding rounding operators by RN, RZ, RD, and RU, respectively.

As round-to-nearest is the default rounding direction in the IEEE 754 standard, we start by testing whether this is the rounding mode used by tensor cores. This can be verified by setting any two partial products to values that would be rounded up only if round-to-nearest or round-up were used. If the result has not been rounded up, then we will know that round-to-nearest (or round-up) is not implemented and that it instead is either round-toward-zero or round-down (Figure 1a). One such test is to set in (3) $b_{11} = 2$, $b_{21} = 2^{-23} + 2^{-24}$, and the remaining entries in the first column of B to 0. Note that in binary32 arithmetic $\mathrm{RN}(2+x) > 2$ if $x > 2 \times 2^{-24} = 2^{-23}$, whereas the smallest positive y such that $\mathrm{RZ}(2 + y) > 2$ is $2 \times 2^{-23} = 2^{-22}$. The choice $b_{21} = \frac{3}{4} \times 2^{-22}$ is such that $x < b_{21} < y$, thus $\mathrm{RN}(b_{11}+b_{21}) = \mathrm{RU}(b_{11}+b_{21}) = 2+2^{-22}$ while $\mathrm{RZ}(b_{11} + b_{21}) = \mathrm{RD}(b_{11} + b_{21}) = 2$. Running this experiment on tensor cores returns $c_{11} = 2$, suggesting that either round-to-zero or round-down is used for the additions in (2).

We can discriminate between these two rounding modes by repeating the same experiment on the negative semiaxis (Figure 1b), by changing the sign of the nonzero elements in B. This experiment produces $c_{11} = -2$, and assuming that the rounding mode does not depend on the input, we conclude that the additions in (2) are performed in round-toward-zero. We note that this rounding mode is known to be the cheapest option to implement [23, Sec. 6.1] and is usually chosen for that reason.

In binary16 mode, the result computed in the 5-term adder's internal accumulator's format has to be rounded to binary16. To test the rounding mode of this operation, we set $a_{11} = a_{12} = 2^{-24}$, $b_{11} = 2^{-1}$, $b_{21} = 2^{-2}$, and the rest of elements of A and B as well as c_{11} to 0. The exact result of the dot product in this case is $2^{-25} + 2^{-26}$, which is not representable in binary16, and therefore will cause rounding error in the result. Note that $2^{-25}+2^{-26} = \frac{3}{4}\times 2^{-24}$, therefore $\mathrm{RN}(2^{-25}+2^{-26}) = 2^{-24}$ while $\mathrm{RZ}(2^{-25}+2^{-26}) = \mathrm{RD}(2^{-25}+2^{-26}) =$ 0. The fact that tensor cores return 2^{-24} confirms that round-to-nearest is used, thereby suggesting that this conversion is performed in software rather than hardware.

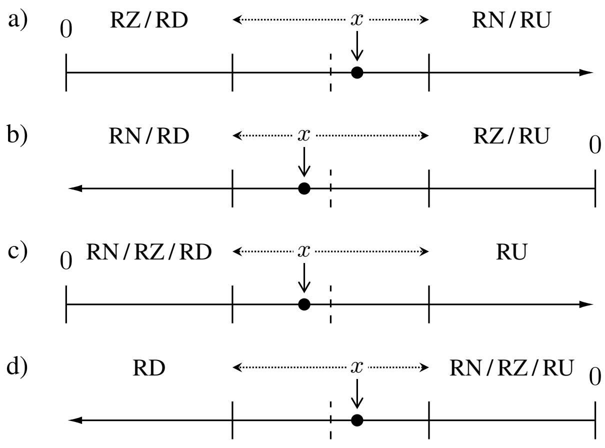

Fig. 1. Demonstration of the possible IEEE 754 rounding modes for different positions of the exact value x. The solid markers illustrate two adjacent floating-point values and the dashed marker the half-way point between them. The dashed arrows near x show in which direction various rounding modes would round it.

D. Features of the accumulator

We now discuss our tests that disclose various features of the internal accumulator of the 5-term adder calculating (2). Note that the quotes from NVIDIA that we have provided in Section I indicate that the internal accumulation is done in binary32 format—however, here we show that the format has more precision and that no normalization of partial sums is performed.

1) Extra bits in the significand alignment: In order to compute the sum of two floating-point values, the floating-point adder matches the exponents of the two summands by shifting the significand of the number that has the smaller exponent to the right. In general this operation causes loss of information, as the least significant bits of the shifted significand are typically discarded, but it is customary to retain a few of these digits to guard the computation against numerical cancellation and to obtain faithful rounding in round-to-nearest, round-up, and round-down. As tensor cores use truncation in the additions, we know that they do not require any such *guard digits* for rounding, and we can easily show that in fact they do not implement guard digits at all. If in (3) we set $b_{11} = 1$ and $c_{11} = -1 + 2^{-24}$, tensor cores return $d_{11} = 2^{-23}$, which represents a relative error of $(2^{-23} - 2^{-24})/2^{-24} = 1$.

2) Normalization in addition: When two floating-point values are added, the significand of the result may become larger than 2, in which case a normalization step is required (shift the significand right by one place and increase the exponent by one) [17, Sec. 7.3]. On an IEEE 754-compliant system, the result of each partial sum in (2) would be normalized, as floating-point adders always normalize the result in order to produce a normalized floating-point number. But tensor cores are not IEEE-compliant, and it is natural to ask whether the each partial result in (2) is normalized or only the final answer is. We can verify this by adding values chosen so to produce different results with and without normalization of the partial sums. In (3) we set $c_{11} = 1 - 2^{-24}$ and the elements of the first column of B to 2^{-24}.

Recalling that the values are accumulated on the summand of largest magnitude, we start by examining what would happen if each partial result were normalized. The exact value of the partial sum $s := c_{11} + 2^{-24}$ is 1, and the normalization step would shift the significand by one bit to the right. At this point the three remaining addends would be smaller than the least significant bit of the partial sum, thus adding it to s would have no effect in round-to-zero. If the partial results were not normalized, on the other hand, the partial sum of $c_{11} + 2^{-24}$ would be held with one extra bit and therefore the rest of the addends could be added into it. Running this test on tensor cores shows that only the final result of the dot product is normalized. This has probably been done in order to simplify the implementation, a choice that has also been made for example in the hardware accelerator for performing vector inner product described in [21].

3) Normalization in subtraction: As the products in (2) can be positive as well as negative, some of the partial sums can in fact be subtractions. The significand of the difference of two-floating point numbers may be smaller than 1, in which case the result has to be normalized by shifting the significand left and decreasing the exponents accordingly until the result becomes normalized. We can show that tensor cores do not perform this kind of normalization as follows. If in (3) we set $c_{11} = -1 + 2^{-24}$, and two of the elements of the first column of B to 1 and -2^{-24}, we will have that d_{11} evaluates to 0 if the partial sums are normalized. Instead, running this experiment on tensor cores yields $d_{11} = 2^{-23}$, which can be explained as follows. When the sum is evaluated as $(1 + c_{11}) - 2^{-24}$, then the lack of guard digit implies that $1 + c_{11}$ evaluates to 2^{-23}, and if the partial results were normalized the tensor cores would return $2^{23} - 2^{24}$, which can be represented exactly in binary32 format's precision. If, on the other hand, the sum were evaluated as $(-2^{-24} + 1) + c_{11}$, the first sum would return 1 due to the lack of guard digit, and the lack of normalization would not have any effect in this case. Therefore we can assume that the result of the subtraction is not normalized only if we assume that the summands in (3) are accumulated on the largest in magnitude in a fixed order.

4) Extra bits for carry out: Another question concerns extra bits required due to lack of normalization. If only the final result is normalized, then accumulating k addends requires $\lceil \log_2 k \rceil$ bits for the carry-out bits, and the hardware for accumulating the 5 values in (2) would internally require $\lceil \log_2 5 \rceil = 3$ extra carry-out bits at the top of the significand. We can prove that the 5-term adder's accumulator in tensor cores has at least two extra bits as follows. In (3) we take $c_{11} = 1 + 2^{-22} + 2^{-23}$, which sets the two least significant bits of the significand to 1, and assign to the first column of B a permutation of the values 1, 1, 1, and 2^{-23}. All four possible permutations of assigning the elements of the first column of B should be run, assuming that the addends apart from the largest in magnitude are not sorted. The main idea is to show that when 1 is accumulated three times into c_{11} the last two bits of c_{11} are not dropped as they would be in IEEE 754 floating-point arithmetic, due to the sum becoming larger than 4 and requiring to be shifted right by two positions. Then, when 2^{-23} is added into it at the end, the carry propagates into the third bottom bit and therefore is not lost in the final normalization step. If there are 2 extra bits, then all 4 combinations of ordering the first column of B would return the precise results of $4 + 2^{-21}$. Running these tests on tensor cores, we found that all four combinations returned the exact result of this test case, therefore proving that there are at least 2 extra bits in the internal accumulator's significand.

It worth to note at this point that if 1) there is no normalization, 2) the additions in (2) start with the largest value in magnitude, and 3) all of the significands of the addends are shifted right relative to the exponent of the largest value in magnitude, then the order of the rest of the addends will not impact the final result.

In the test case above, replacing one of the 2^{-23} by 1, we can also confirm using the methods developed in Section III-C. that the rounding mode in the final normalization step (internal accumulator conversion to binary32 answer) is round-toward-zero.

5) Monotonicity of dot product: The observation in section III-D2 raises one final question regarding the monotonicity of the sums in (2). The accumulation is monotonic if in floating-point arithmetic the sum $x_1 + \cdots + x_n$ is no larger than $y_1 + \cdots + y_n$ when $x_i \leq y_i$ for all $1 \leq i \leq n$.

We can show that the lack of normalization causes the dot product in tensor cores and most likely any other similar architectures without the normalization of partial sums [19], [20], [21], [22] to behave non-monotonically by setting in (3) all the elements in the first column of B to 2^{-24} and then c_{11} to $1 - 2^{-24}$ and 1 in turn. When $c_{11} = 1 - 2^{-24}$, the difference in exponents guarantees that the values in B are large enough to be added to c_{11}. This causes the result to become larger than 1, requiring a normalization that returns $1 - 2^{-24} + 3 \times 2^{-24} = 1 + 2^{-23}$. On the other hand, when $c_{11} = 1$, none of the summands in (3) is large enough be added to c_{11}, as the elements in the first column of B are all zeroed out during the significand alignment step of each addition. This happens because the exponent of 1 is larger than that of $1 - 2^{-24}$. In summary, the following two calculations demonstrate that tensor cores can produce non-monotonic behaviour:

$$\begin{aligned} d_{11} &= c_{11} + 2^{-24} + 2^{-24} + 2^{-24} + 2^{-24} = 1 + 2^{-23}, \\ &\quad \text{when } c_{11} = 1 - 2^{-24}, \\ d_{11} &= c_{11} + 2^{-24} + 2^{-24} + 2^{-24} + 2^{-24} = 1, \\ &\quad \text{when } c_{11} = 1. \end{aligned}$$

IV. Conclusion

In summary, our experiments indicate that the tensor cores in the NVIDIA V100 architecture have the following numerical features.

- Subnormal numbers in binary16 and binary32 are supported.
- The binary16 products in (2) are computed exactly, and the results kept in full precision and not rounded to binary16 after the multiplication.
- The five summands in (2) are accumulated starting with the largest in absolute value.
- The additions in (2) are performed using binary32 arithmetic with round-toward-zero.
- Only the final result of (2) is normalized; the partial sums are not, but the accumulator uses two extra bits for carries.
- The dot products in a tensor core are non-monotonic: in some cases, increasing the magnitude of the inputs to (2) reduces the magnitude of the final result when the summands in (2) are all nonnegative or all nonpositive.

Some of these findings confirm and clarify what was reported by NVIDIA and some had not previously been reported.

The test suite that we have developed as part of this work can be used to test various properties of the floating-point arithmetic of future versions of tensor cores as well as similar accelerators. We aim to keep extending our test suite by adding new test cases for standard non-mixed-precision binary32 or binary64 dot product or matrix multiply units, as well as for integer arithmetic. The new NVIDIA Turing tensor cores, for instance, added support for 4- and 8-bit integer modes [5], and rounding issues become relevant when these are used to implement fixed-point arithmetic.

V. Acknowledgements

The authors thank the University of Manchester for providing access to the NVIDIA V100 graphic cards through the Computational Shared Facility. MM was supported by an EPSRC Doctoral Prize Fellowship. NJH and SP were supported by Engineering and Physical Sciences Research Council grant EP/P020720/1. NJH and MF were supported by the Royal Society.

References

[1] NVIDIA, "NVIDIA Tesla V100 GPU architecture," p. 58, 2017, NVIDIA whitepaper WP-08608-001_v1.1. [Online]. Available: https://images.nvidia.com/content/volta-architecture/pdf/volta-architecture-whitepaper.pdf

[2] *IEEE Standard for Floating-Point Arithmetic, IEEE Std 754-2019 (revision of IEEE Std 754-2008)*. Piscataway, NJ, USA: Institute of Electrical and Electronics Engineers, 2019. [Online]. Available: https://doi.org/10.1109/IEEESTD.2008.4610935

[3] NVIDIA, "Multiply-and-Accumulate Instruction: mma," accessed: April 15, 2020. [Online]. Available: https://docs.nvidia.com/cuda/parallel-thread-execution/index.html#warp-level-matrix-instructions-mma

[4] Google, "System architecture," accessed: April 15, 2020. [Online]. Available: https://cloud.google.com/tpu/docs/system-architecture

[5] NVIDIA, "NVIDIA Turing GPU architecture," p. 58, 2018, NVIDIA whitepaper WP-09183-001_v01. [Online]. Available: https://www.nvidia.com/content/dam/en-zz/Solutions/design-visualization/technologies/turing-architecture/NVIDIA-Turing-Architecture-Whitepaper.pdf

[6] D. R. Lutz, "ARM floating point 2019: Latency, area, power," in *Proc. 26th IEEE Symposium on Computer Arithmetic*, 2019, pp. 97–98. [Online]. Available: https://doi.org/10.1109/ARITH.2019.00025

[7] Intel Corporation, "BFLOAT16—hardware numerics definition," November 2018, white paper. Document number 338302-001US. [Online]. Available: https://software.intel.com/en-us/download/bfloat16-hardware-numerics-definition

[8] NVIDIA, "NVIDIA CUDA Math API," accessed: April 15, 2020. [Online]. Available: https://docs.nvidia.com/cuda/cuda-math-api/index.html

[9] S. Markidis, S. W. D. Chien, E. Laure, I. B. Peng, and J. S. Vetter, "NVIDIA tensor core programmability, performance & precision," in *Proc. 32rd IEEE International Parallel and Distributed Processing Symposium Workshops*, 2018, pp. 522–531. [Online]. Available: https://doi.org/10.1109/IPDPSW.2018.00091

[10] 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 *Proc. International Conf. for High Performance Computing, Networking, Storage, and Analysis*. Piscataway, NJ, USA: IEEE Press, 2018, pp. 47:1–47:11.

[11] 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*, ser. Lect. Notes Comput. Sci., Y. Shi, H. Fu, Y. Tian, V. V. Krzhizhanovskaya, M. H. Lees, J. Dongarra, and P. M. A. Sloot, Eds., vol. 10860, 2018, pp. 586–600. [Online]. Available: https://doi.org/10.1007/978-3-319-93698-7_45

[12] Z. Jia, M. Maggioni, B. Staiger, and D. P. Scarpazza, "Dissecting the NVIDIA Volta GPU architecture via microbenchmarking," arXiv:1804.06826 [cs.DC], p. 66, 2018. [Online]. Available: https://arxiv.org/abs/1804.06826

[13] D. Yan, W. Wang, and X. Chu, "Demystifying tensor cores to optimize half-precision matrix multiply," 2020, to appear in the *Proc. 34th IEEE International Parallel and Distributed Processing Symposium*. [Online]. Available: http://home.cse.ust.hk/~weiwa/papers/yan-ipdps20.pdf

[14] Z. Jia, M. Maggioni, J. Smith, and D. P. Scarpazza, "Dissecting the NVidia Turing T4 GPU via microbenchmarking," arXiv:1903.07486 [cs.DC], p. 65, 2018. [Online]. Available: https://arxiv.org/abs/arXiv:1903.07486

[15] P. M. Basso, F. F. Dos Santos, and P. Rech, "Impact of tensor cores and mixed-precision on the reliability of matrix multiplication in GPUs," *IEEE Trans. Nucl. Sci.*, Mar. 2020, early access accepted paper. [Online]. Available: https://doi.org/10.1109/TNS.2020.2977583

[16] 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," Manchester Institute for Mathematical Sciences, The University of Manchester, UK, MIMS EPrint 2019.18, Sep. 2019, revised February 2020. To appear in SIAM J. Sci. Comput. [Online]. Available: http://eprints.maths.manchester.ac.uk/2749/

[17] 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*, 2nd ed. Cham, Switzerland: Birkhäuser, 2018. [Online]. Available: https://doi.org/10.1007/978-3-319-76526-6

[18] A. F. Tenca, "Multi-operand floating-point addition," in *Proc. 19th IEEE Symposium on Computer Arithmetic*, 2009, pp. 161–168. [Online]. Available: https://doi.org/10.1109/ARITH.2009.27

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

[20] H. Kaul, M. Anders, S. Mathew, S. Kim, and R. Krishnamurthy, "Optimized fused floating-point many-term dot-product hardware for machine learning accelerators," in *Proc. 26th IEEE Symposium on Computer Arithmetic*, 2019, pp. 84–87. [Online]. Available: https://doi.org/10.1109/ARITH.2019.00021

[21] D. Kim and L. Kim, "A floating-point unit for 4D vector inner product with reduced latency," *IEEE Trans. Comput.*, vol. 58, no. 7, pp. 890–901, Jul. 2009. [Online]. Available: https://doi.org/10.1109/TC.2008.210

[22] J. Sohn and E. E. Swartzlander, "A fused floating-point four-term dot product unit," *IEEE Trans. Circuits Syst. I, Reg. Papers*, vol. 63, no. 3, pp. 370–378, Mar. 2016. [Online]. Available: https://doi.org/10.1109/TCSI.2016.2525042

[23] M. R. Santoro, G. Bewick, and M. A. Horowitz, "Rounding algorithms for IEEE multipliers," in *Proc. 9th IEEE Symposium on Computer Arithmetic*, 1989, pp. 176–183. [Online]. Available: https://doi.org/10.1109/ARITH.1989.72824