Numerical behavior of NVIDIA tensor cores

Massimiliano Fasi [1], **Nicholas J Higham** [2], **Mantas Mikaitis** [Corresp., 2], **Srikara Pranesh** [2]

[1] School of Science and Technology, Orebro University, Orebro, Sweden

[2] Department of Mathematics, University of Manchester, Manchester, United Kingdom

Corresponding Author: Mantas Mikaitis
Email address: mantas.mikaitis@manchester.ac.uk

We explore the floating-point arithmetic implemented in the NVIDIA tensor cores, which are hardware accelerators for mixed-precision matrix multiplication available on the Volta, Turing, and Ampere microarchitectures. Using Volta V100, Turing T4 and Ampere A100 graphics cards, we determine what precision is used for the intermediate results, whether subnormal numbers are supported, what rounding mode is used, in which order the operations underlying 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 units. Knowing the answers to these questions is important if one wishes to: 1) accurately simulate NVIDIA tensor cores on conventional hardware; 2) understand the differences between results produced by code that utilizes tensor cores and code that uses only IEEE 754-compliant arithmetic operations; and 3) build custom hardware whose behavior matches that of NVIDIA tensor cores. As part of this work we provide a test suite that can be easily adapted to test newer versions of the NVIDIA tensor cores as well as similar accelerators from other vendors, as they become available. Moreover, we identify a non-monotonicity issue affecting floating point multi-operand adders if the intermediate results are not normalized after each step.

Numerical Behavior of NVIDIA Tensor Cores

Massimiliano Fasi[*,1]**, Nicholas J. Higham**[2]**, Mantas Mikaitis**[2]**, and Srikara Pranesh**[2]

[1]**School of Science and Technology, Örebro University, Örebro, Sweden**
[2]**Department of Mathematics, University of Manchester, Manchester, United Kingdom**

Corresponding author:

Mantas Mikaitis[2]

Email address: mantas.mikaitis@manchester.ac.uk

ABSTRACT

We explore the floating-point arithmetic implemented in the NVIDIA tensor cores, which are hardware accelerators for mixed-precision matrix multiplication available on the Volta, Turing, and Ampere microarchitectures. Using Volta V100, Turing T4 and Ampere A100 graphics cards, we determine what precision is used for the intermediate results, whether subnormal numbers are supported, what rounding mode is used, in which order the operations underlying 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 units. Knowing the answers to these questions is important if one wishes to: 1) accurately simulate NVIDIA tensor cores on conventional hardware; 2) understand the differences between results produced by code that utilizes tensor cores and code that uses only IEEE 754-compliant arithmetic operations; and 3) build custom hardware whose behavior matches that of NVIDIA tensor cores. As part of this work we provide a test suite that can be easily adapted to test newer versions of the NVIDIA tensor cores as well as similar accelerators from other vendors, as they become available. Moreover, we identify a non-monotonicity issue affecting floating-point multi-operand adders if the intermediate results are not normalized after each step.

1 INTRODUCTION

One hundred and twelve of the computers in the June 2020 TOP500 list[1] are equipped with NVIDIA graphics processing units (GPUs) based on the Volta, Turing, and Ampere microarchitectures. A prominent feature of these GPUs is the *tensor cores*, which are specialized hardware accelerators for performing a matrix multiply-accumulate operation. Here, in particular, we focus on the tensor cores available on the NVIDIA V100 (Volta microarchitecture), T4 (Turing architecture), and A100 (Ampere architecture)

---

*This work was carried out when this author was at the Department of Mathematics, University of Manchester, Manchester, M13 9PL, UK.

[1]https://www.top500.org/lists/top500/list/2020/06/

Table 1. Processing units or architectures equipped with mixed-precision matrix multiply-accumulate accelerators. The notation $m \times k \times n$ refers to the matrix multiply-accumulate operation in (1) where C and D are $m \times n$ matrices, and A and B have size $m \times k$ and $k \times n$, respectively.

Year of release	Device	Matrix dimensions	Input format	Output format	Reference
2016	Google TPU v2	$128 \times 128 \times 128$	bfloat16	binary32	(Google, 2020)
2017	Google TPU v3	$128 \times 128 \times 128$	bfloat16	binary32	(Google, 2020)
2017	NVIDIA V100	$4 \times 4 \times 4$	binary16	binary32	(NVIDIA, 2017)
2018	NVIDIA T4	$4 \times 4 \times 4$	binary16	binary32	(NVIDIA, 2018)
2019	Arm v8.6-A	$2 \times 4 \times 2$	bfloat16	binary32	(Arm Ltd., 2020)
2020	NVIDIA A100	$8 \times 8 \times 4$	bfloat16	binary32	(NVIDIA, 2020b)
		$8 \times 8 \times 4$	binary16	binary32	
		$4 \times 8 \times 4$	TensorFloat-32	binary32	
		$4 \times 2 \times 2$	binary64	binary64	

GPUs, which implement the operation

$$
\underbrace{D}_{\substack{\text{binary16 or} \\ \text{binary32}}} = \underbrace{C}_{\substack{\text{binary16 or} \\ \text{binary32}}} + \underbrace{A}_{\text{binary16}} \times \underbrace{B}_{\text{binary16}}, \tag{1}
$$

where A, B, C, and D are 4×4 matrices and "binaryxy" denotes the xy-bit format from the IEEE Standard 754 for floating-point arithmetic (IEEE, 2019). 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 newer A100 (Ampere microarchitecture) GPUs support a wider range of matrix dimensions, as well as an additional floating-point format, as shown in Table 1. The element d_{ij} in (1) 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 (NVIDIA, 2017, 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.

---

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

35   The official documentation (NVIDIA, 2020a) adds only a few more details:

36   *Element-wise multiplication of matrix A and B is performed with at least single*
37   *precision. When .ctype or .dtype is .f32, accumulation of the intermediate values is*
38   *performed with at least single precision. When both .ctype and .dtype are specified*
39   *as .f16, the accumulation is performed with at least half precision. The accumulation*
40   *order, rounding and handling of subnormal inputs is unspecified.*

41   From a numerical point of view, many essential aspects of tensor cores are not spec-
42   ified. This lack of key information makes it challenging to simulate tensor cores on
43   conventional IEEE 754-compliant systems and to build hardware that can reproduce
44   their behavior. Moreover, it can lead to unexpected differences between the results
45   computed on NVIDIA devices with tensor cores enabled or disabled.

46   We now briefly recall some key aspects of IEEE-compliant floating-point systems
47   and the definitions and main properties of *normalization* and *subnormal numbers*, which
48   are two important concepts in this work. A binary floating-point number x has the
49   form $(-1)^s \times m \times 2^{e-p}$, where s is the sign bit, p is the precision, $m \in [0, 2^p - 1]$ is the
50   integer significand, and $e \in [e_{\min}, e_{\max}]$, with $e_{\min} = 1 - e_{\max}$, is the integer exponent.
51   In order for x to have a unique representation, the number system is *normalized* so
52   that the most significant bit of m—the *implicit bit* in IEEE 754 parlance—is always set
53   to 1 if $|x| \geq 2^{e_{\min}}$. Therefore, all floating-point numbers with $m \geq 2^{p-1}$ are normalized.
54   Numbers below the smallest normalized number $2^{e_{\min}}$ in absolute value are called
55   *subnormal numbers*, and are such that $e = e_{\min}$ and $0 < m < 2^{p-1}$. Subnormal numbers
56   provide a means to represent values in the subnormal range, that is, between 0 and the
57   minimum normalized number. They have lower precision than normalized values (from
58   $p - 1$ bits to as low as 1 bit), and require special treatment to be implemented in software
59   and hardware. Therefore it is not uncommon for hardware manufacturer not to support
60   them, in order to avoid performance or chip area overheads.

61   In implementing floating-point operations the result of an operation must be normal-
62   ized (if possible) by shifting the significand left or right until it falls within the interval
63   $[2^{p-1}, 2^p - 1]$ and adjusting the exponent accordingly. More details can be found in
64   Muller et al. (2018, Sec. 7.3).

65   The IEEE 754 standard for floating-point arithmetic provides a somewhat relaxed
66   set of requirements for reduction operations such as dot product and multi-operand
67   addition (IEEE, 2019, Sec. 9.4): the order in which the partial sums should be evaluated
68   is not prescribed, and the use of a higher-precision internal format in allowed. In
69   particular, the standard does not specify: 1) whether this internal format should be
70   normalized, as it would be if the multi-operand addition were implemented using IEEE
71   754 elementary arithmetic operations, 2) which rounding mode should be used, and 3)
72   when the rounding should happen. These loose requirements can potentially cause the
73   results computed with a given multi-operand addition unit to be significantly different
74   from those obtained using other hardware implementations or a software implementation
75   based on IEEE 754-compliant elementary arithmetic operations.

76   With matrix multiplication being ubiquitous in artificial intelligence, accelerators for
77   mixed-precision matrix multiply-accumulate operations are becoming widely available,
78   as Table 1 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 1, for instance, use bfloat16, a 16-bit format that allocates 8 bits to the significand (including the implicit bit) and 8 bits to the exponent and does not support subnormal numbers (Intel Corporation, 2018, 2020). It is worth noting that Volta is the first NVIDIA microarchitecture supporting tensor cores—the older Pascal GPUs, such as the P100, which are not reported in Table 1, supported binary16 arithmetic but did not include tensor cores. The NVIDIA Ampere A100 GPUs introduce a new 19-bit format called TensorFloat-32, which was designed to have the same dynamic range as binary32 (8 bits) and the same precision as binary16 (11 fraction bits, including the implicit bit) (NVIDIA, 2020b). 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.

The situation is reminiscent of that before the widespread adoption of the IEEE 754-1985 standard, when different floating-point arithmetics had different properties. To address the issue, Kahan (1981) developed a program called paranoia that analyzes and diagnoses a floating-point arithmetic, and it was subsequently translated into C by Karpinski (1985). We follow a similar route: using idiosyncrasies of floating-point arithmetic, we design tests to better understand the numerical behavior of tensor cores, extending the testing approach recently introduced by Hickmann and Bradford (2019). Our aim is to explore the following questions.

- 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)?

- Is the result of each floating-point operation in (2) normalized, or do tensor cores only normalize the final sum? What rounding mode is used for the normalization?

The answers to these questions are of wide interest because these accelerators, despite being introduced to accelerate the training of deep neural networks (NVIDIA, 2017, p. 12), are increasingly being used in general-purpose scientific computing, where their fast low precision arithmetic can be exploited in mixed-precision algorithms (Abdelfattah et al., 2020), for example in iterative refinement for linear systems (Haidar et al., 2018a,b, 2020).

The results discussed here were produced by running our test suite, which is freely available on Github.[3] The file `tc_test_numerics.cu` can be compiled following the instructions in the `README.md` file, so as to produce an executable that will run all the tests we describe in the following sections and produce a report. We run the test suite on an NVIDIA Tesla V100 SXM2 16GB GPU (Volta microarchitecture), an NVIDIA Tesla T4 16GB GPU (Turing microarchitecture), and an NVIDIA Ampere A100-PCIA

---

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

120 40GB GPU. We used version 10.1 of the CUDA library on the machines equipped with
121 the Volta and Turing cards, and version 11.1 on the machines equipped with the A100
122 GPU, as GPUs based on the Ampere microarchitecture require at least version 11 of the
123 CUDA library in order to utilize the latest numerical features such as the TensorFloat-32
124 numerical type. Some graphic cards designed for intensive graphic processing workloads
125 such as video gaming, computer-aided design, or computer-generated imagery, also
126 include tensor cores: all the GPUs in the GeForce 20 and Quadro RTX series are based
127 on the Turing microarchitecture, and those in the recently announced GeForce 30 series
128 are based on the Ampere microarchitecture. We do not consider this wealth of different
129 graphic cards here, as our focus is on the NVIDIA hardware that is present in the
130 supercomputers in the TOP500 list and target scientific computing applications. We
131 stress, however, that the ideas we employ are very general and can be exploited to
132 understand the numerical features of any hardware accelerator based on operations of
133 the form (2).

134 Finally, we remark that the binary16 arithmetic implemented in the NVIDIA CUDA
135 cores is not fully IEEE 754 compliant, as round-to-nearest is the only rounding mode
136 implemented for elementary arithmetic operations (NVIDIA, 2020c)—we use this
137 observation in Section 3.1.1.

2 PREVIOUS WORK

139 From a hardware perspective, instruction-level details, register configuration, and mem-
140 ory layout of the tensor cores in the NVIDIA Volta (Jia et al., 2018b; Yan et al., 2020)
141 and Turing (Yan et al., 2020; Jia et al., 2018a) GPUs have been extensively described.
142 Another study by Basso et al. (2020) explores the reliability of tensor cores in terms of
143 rate of hardware errors in matrix multiplications. The main finding is that low-precision
144 operations and usage of tensor cores increase the amount of correct data produced by the
145 GPU, despite increasing the impact of numerical errors due to the use of lower-precision
146 data. In order to quantify the accuracy of tensor cores, Blanchard et al. (2020) provide
147 a rounding error analysis of what they call a block fused multiply-add (FMA), a gen-
148 eralization of the multiply-accumulate operation in (1) in which the matrix sizes, the
149 precisions of the arguments, and the internal precision of the accumulator are taken as
150 parameters.

151 Markidis et al. (2018) discuss various aspects of tensor cores and propose a tech-
152 nique, called precision refinement, to enhance the accuracy of mixed-precision matrix
153 multiplication. Improving the accuracy of tensor-core-based matrix multiplications was
154 further explored by Mukunoki et al. (2020).

155 None of the these sources, however, examines to what extent tensor cores conform to
156 the IEEE 754 standard or investigates how tensor cores compare with a matrix multiply-
157 accumulate operation based on dot products implemented in software. Hickmann
158 and Bradford (2019) explore some details of the numerical behavior of tensor cores
159 with the main goal of inferring the hardware-level design of these units. Our work
160 follows a similar approach and complements their findings by supplying further insights
161 into the subject. We show that the additions in (2) are performed starting from the
162 operand that is largest in magnitude, that at least 2 extra bits are used for carries, and
163 that (2) may be non-monotonic for certain sets of inputs. Furthermore, we consider the

Table 2. Summary of the subsections of Section 3.

Section	Devices used and description of tests performed on them
3.1	Tests performed on the NVIDIA V100 (Volta) GPU.
3.1.1	Support for subnormal numbers (on the inputs, outputs and the computation of subnormals from the normalized inputs).
3.1.2	Accuracy of the inner products performed as part of matrix multiply-accumulate (accuracy of scalar multiplies, accuracy of addition, and number of rounding errors introduced).
3.1.3	Tests for determining what rounding modes are used in the inner products and the final rounding.
3.1.4	Tests that explore the number of extra bits in the alignment step of floating-point addition inside the inner product (extra bits at the bottom of the internal significand).
3.1.4	A test to find out whether the normalization is performed only at the end or after each addition in the computation of the inner products.
3.1.4	A similar test for the normalization in subtraction.
3.1.4	Tests for determining the number of extra bits for carries in floating-point addition (extra bits at the top of the internal significand).
3.1.4	A test to check the monotonicity of the inner products.
3.2	All tests from Section 3.1 rerun on the NVIDIA T4 (Turing) GPU.
3.3	All tests from Section 3.1 rerun on the NVIDIA A100 (Ampere) GPU.

second generation of tensor cores, which equips the Turing T4 GPUs, and conclude that their internal accumulator has an extra bottom bit compared with the tensor cores on V100 GPUs. Finally, unlike Hickmann and Bradford (2019) we make our test suite freely available, in order to guarantee reproducibility and facilitate testing other matrix multiply-accumulate units, such as the third generation tensor cores in the latest NVIDIA A100 GPUs (NVIDIA, 2020b).

3 RESULTS

In this section we describe our testing methodology and give the results obtained on the two NVIDIA graphics cards we considered. Table 2 summarizes the tests we performed and indicates the subsections in which they are described. Our methodology is to find test cases that demonstrate specific numerical behaviours and, by making the (quite reasonable) assumption that the hardware has a consistent behavior across the input space, conclude that the discovered features should be true for all possible inputs.

3.1 NVIDIA Volta Tensor Cores

Tensor cores can be accessed using the cuBLAS library, or the native hardware assembly instructions `HMMA.884` (Jia et al., 2018b,a) and `HMMA.1688` (Yan et al., 2020; Jia et al., 2018a). 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

Table 3. Parameters of various floating-point formats: number of digits of precision including the implicit bit (p), boundary of the exponent range (e_{min} and e_{max}), machine epsilon (ε), and smallest positive representable normal (f_{min}) and subnormal (s_{min}) numbers. The formats from the IEEE 754 standard (IEEE, 2019) are binary16, binary32, and binary64.

	binary16	bfloat16	TensorFloat-32	binary32	binary64
p	11	8	11	24	53
e_{max}	15	127	127	127	1023
e_{min}	-14	-126	-126	-126	-1022
ε	2^{-10}	2^{-7}	2^{-10}	2^{-23}	2^{-52}
f_{min}	2^{-14}	2^{-126}	2^{-126}	2^{-126}	2^{-1022}
s_{min}	2^{-24}	2^{-133}	2^{-136}	2^{-149}	2^{-1074}

183   CUDA programming environment. In order to use only a single tensor core, we set
184   all but the top left 4×4 blocks to 0. We ensure that our experiments do use the tensor
185   cores by running our test suite with the NVIDIA profiler `nvprof`, which shows the
186   utilization levels of different hardware components on the GPU, and by observing that
187   the assembly code produced by the `nvcc` compiler contains `HMMA` instructions. At the
188   software level, tensor cores can be used in either binary16 or binary32 mode, which
189   defines the number format of D in (1).

190   The experiments in this section were run on an NVIDIA Tesla V100 SXM2 16GB
191   (Volta microarchitecture) GPU.

3.1.1 Support for subnormal numbers

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

195   Table 3 compares precision, exponent range, machine epsilon, and magnitude of
196   smallest representable subnormal and normal number for various floating-point number
197   formats, including those supported by the three generations of tensor cores. By looking
198   at the data in the table, we can make two important observations. First, conversion from
199   binary16 to binary32 does not result in subnormal numbers. Second, the product of two
200   binary16 numbers requires at most 22 bits for the significand, 6 bits for the exponent
201   and one for the sign, and thus can be represented exactly in binary32.

202   As for tensor cores, there are multiple questions regarding the support of subnormal
203   numbers.

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

207   2. Can tensor cores take binary32 subnormal numbers as inputs for C in (2) without
208   flushing them to zero, use them in computation, and return subnormal binary32
209   results?

210   3. Can tensor cores compute subnormal numbers from normal numbers and return
211   them?

212   The first question can easily be addressed by considering (2) and setting $a_{11} = 2^{-24}$,
213   $b_{11} = 2^2$ (arbitrarily chosen), and the remaining elements to zero. The tensor cores
214   return the subnormal result $a_{11}b_{11} = 2^{-22}$ in both binary16 and binary32 mode, thereby
215   answering the first question in the affirmative.

216   An analogous idea can be used to clarify the second point: setting c_{11} to the smallest
217   positive binary32 subnormal 2^{-149} and all the elements of A and B to zero yields
218   $d_{11} = 2^{-149}$, which confirms that the subnormal number c_{11} is not altered by the dot
219   product in (2). We note, however, that whether support for binary32 subnormals is
220   needed is questionable. The absolute value of the smallest nonzero value that can be
221   produced from the multiplication of two binary16 numbers is 2^{-48}, thus c_{11} would
222   not contribute to the sum if it were a binary32 subnormal: in binary32 arithmetic with
223   round-to-nearest, one has that $2^{-48} + x > 2^{-48}$ only if $x > 2^{-48} \cdot 2^{-24} = 2^{-72}$, which is
224   normal in binary32.

225   For the third question, we can obtain subnormal numbers as outputs in several ways.
226   For instance, we can set a_{11} to 2^{-14}, the smallest normal number in binary16, and b_{11} to
227   2^{-1}, and confirm that tensor cores return the binary16 subnormal 2^{-15} in both binary16
228   and binary32 modes. Another possibility is to set $a_{11} = 2^{-14}$, $b_{11} = 1$, and $c_{11} = -2^{-15}$,
229   which produces the subnormal binary16 number $d_{11} = 2^{-15}$. As mentioned above, it
230   is not possible to obtain subnormal binary32 number from binary16 inputs in (2). In
231   summary, these experiments demonstrate that there is full support for subnormal inputs
232   in tensor cores.

233   One might wonder whether tensor cores natively support subnormals or some degree
234   of software interaction is present. The NVIDIA profiler confirms that the experiments
235   discussed in this section make use of the tensor cores, but we implemented an additional
236   test to further reinforce the evidence that subnormals are supported in hardware. In
237   Section 3.1.3 we show that tensor cores use round-towards-zero. We can use the fact
238   that CUDA cores provide only round-to-nearest for binary16 computations to show that
239   subnormals are in fact manipulated with tensor cores. In order to do so, we set a_{11} and
240   a_{12} to 1, b_{11} to the binary16 subnormal $2^{-23} + 2^{-24}$, b_{21} to 2 and the other elements of
241   A and B to 0. Since the addition in (2) is done in binary32 arithmetic, the smallest value
242   that can be exactly added to $b_{21} = 2$ is 2^{-22}. In this case, $b_{11} = 2^{-23} + 2^{-24} = \frac{3}{4} \cdot 2^{-22}$
243   will either be round down to 0, if round-towards-zero is being used, or rounded up
244   to 2^{-22}, if the summation is carried out using the CUDA cores, which support only
245   round-to-nearest. This computation returned the value 2, meaning that b_{11} was rounded
246   down—further indicating that subnormals are natively supported by tensor cores.

3.1.2 Accuracy of the dot products

248   Our second goal is to test the accuracy of the dot product (2) with the tensor cores. The
249   first step is to check that the products of two binary16 values are computed exactly,
250   which implies that the products must be kept in some wider intermediate format and
251   accumulated without being rounded back to binary16. Specifically we want to test that
252   $a_{1i}b_{i1}$, for $i = 1, \ldots, 4$, is computed exactly. This can be achieved by ensuring that
253   the four multiplications produce floating-point numbers that are not representable in
254   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} to 0. The exact value of each partial products is

$$(1 - 2^{-11}) \cdot (1 - 2^{-11}) = 1 - 2^{-10} + 2^{-22},$$

which, depending on the rounding mode, would be rounded to either $1 - 2^{-10}$ or $1 - 2^{-11}$, if the products were stored in binary16. As tensor cores produce the exact binary32 answer $d_{11} = 4 \cdot (1 - 2^{-10} + 2^{-22})$, we conclude that partial products are held exactly.

Another question is whether the precision of the 5-operand addition in (2) changes in any way when binary16 is used to store the elements of the matrices 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 $a_{11}b_{11} = 1 - 2^{-10} + 2^{-22}$ 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 computed exactly by a binary16 accumulator, since storing 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-operand adder that accumulates the partial products. The dot product in (2) contains four additions, three to sum up the exact partial products and a fourth to add the binary32 argument c_{11}. Our expectation is that the additions are done in binary32 rather than exactly, as indicated by NVIDIA (2017, 2020a). 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 binary32 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 by Blanchard et al. (2020, Sec. 2.1).

- The partial products in (2) are not accumulated in a fixed order, but always starting from the largest value in 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 (Muller et al., 2018, Sec. 7.3), (Tenca, 2009), and is most likely done in hardware. This is in line with the literature on hardware dot products (Kim and Kim, 2009; Tao et al., 2013; Sohn and Swartzlander, 2016; Kaul et al., 2019), where either sorting or a search for the maximum exponent is performed. Furthermore, this experiment demonstrates that none of the additions are performed before aligning the significands relative to the largest exponent: if evaluated before the arguments are shifted right relative to the largest magnitude arguments' exponent (by having multiple alignment stages), any other sum would compute $2^{-24} + 2^{-24} = 2^{-23}$, a value that then could be added exactly to the total sum as the least significand bits would not be lost in the alignment.
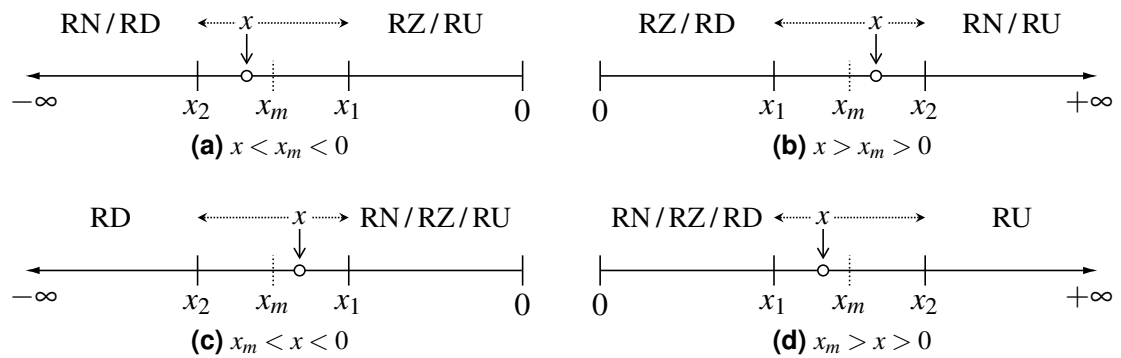
Figure 1. Demonstration of the possible IEEE 754 rounding modes for different positions of the exact value x; x_1 and x_2 are the two floating-point numbers closest to x, and x_m is the half-way point between them. The dotted arrows surrounding x show the direction in which various rounding modes would round it.

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

3.1.3 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 for elementary arithmetic operations (IEEE, 2019, Sec. 4.3): round-to-nearest with even-on-ties, round-towards-zero, round-towards-minus-infinity, and round-towards-plus-infinity. In this section we use the notation defined by Muller et al. (2018, Sec. 2.2.1) and denote these four rounding operators by RN, RZ, RD, and RU, respectively.

As round-to-nearest is the default rounding mode in the IEEE 754 standard, we start by testing whether this is the rounding mode used by the tensor cores. This can be verified by setting any two partial products to values such that one of them would be rounded up only if round-to-nearest or round-towards-plus-infinity were used. If the result is rounded down we can conclude that the tensor cores implement either round-towards-zero or round-towards-minus-infinity, but neither round-to-nearest nor round-towards-plus-infinity (Figure 1b). 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 $RN(2+x) > 2$ if $x > 2 \cdot 2^{-24} = 2^{-23}$, whereas the smallest positive y such that $RZ(2+y) > 2$ is $2 \cdot 2^{-23} = 2^{-22}$. The choice $b_{21} = \frac{3}{4} \cdot 2^{-22}$ is such that $x < b_{21} < y$, thus $RN(b_{11} + b_{21}) = RU(b_{11} + b_{21}) = 2 + 2^{-22}$ while $RZ(b_{11} + b_{21}) = RD(b_{11} + b_{21}) = 2$. Running this experiment on tensor cores returns $c_{11} = 2$, suggesting that either round-towards-zero or round-towards-minus-infinity 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 1a), 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-towards-zero. We note that this rounding mode is known to be the cheapest option to implement (Santoro et al., 1989, Sec. 6.1) and is usually chosen for that reason.

317   When tensor cores are used in binary16 mode, the result computed in the format of
318 the internal accumulator of the 5-operand adder has to be rounded to binary16 before
319 being returned. In order to test the rounding mode used by this operation, we set
320 $a_{11} = a_{12} = 2^{-24}$, $b_{11} = 2^{-1}$, $b_{21} = 2^{-2}$, and the rest of elements of A and B as well
321 as c_{11} to 0. The exact result of the dot product in this case is $2^{-25} + 2^{-26}$, which is
322 not representable in binary16, and therefore will cause rounding errors in the result.
323 Note that $2^{-25} + 2^{-26} = \frac{3}{4} \cdot 2^{-24}$, therefore $\text{RN}(2^{-25} + 2^{-26}) = \text{RU}(2^{-25} + 2^{-26}) = 2^{-24}$
324 while $\text{RZ}(2^{-25} + 2^{-26}) = \text{RD}(2^{-25} + 2^{-26}) = 0$. The fact that tensor cores return 2^{-24}
325 confirms that round-towards-zero is not used, thereby suggesting that this conversion is
326 performed in software rather than inside the tensor cores, which use round-towards-zero
327 as we have determined above.
328   Figures 1c and 1d show how round-towards-minus-infinity and round-towards-plus-
329 infinity could alternatively be determined by choosing x such that $|x| < |x_m|$.

3.1.4 Features of the accumulator

331 We now discuss some tests that allowed to determine various features of the internal
332 accumulator of the 5-operand adder calculating (2). The quotes from NVIDIA that we
333 provided in Section 1 indicate that the internal accumulation is done in binary32 format;
334 here we show that the internal format used by the accumulator has higher precision and
335 that the partial sums are not normalized.

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

Normalization in addition   When two floating-point values are added, the significand
348 of the result may become larger than 2 (with $m > 2^p - 1$, where m is an integer significand
349 as in our definitions in Section 1), in which case a normalization step (shifting the
350 significand right by one place and increase the exponent by one) is required (Muller
351 et al., 2018, Sec. 7.3). In an IEEE 754-compliant arithmetic, the result of each partial sum
352 in (2) would be normalized, as floating-point adders always normalize the result in order
353 to produce a normalized floating-point number. But tensor cores compute the whole
354 expression (2) in hardware rather than by means of IEEE 754 elementary arithmetic
355 operations, and it is natural to ask whether each partial result in (2) is normalized or only
356 the final answer is. We can verify this by adding values chosen so to produce different
357 results with and without normalization of the partial sums. In (3) we set $c_{11} = 1 - 2^{-24}$
358 and the elements of the first column of B to 2^{-24}.
359   Recalling that the values are accumulated on the summand of largest magnitude,
360 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 them to s separately would have no effect with round-towards-zero. If the partial results were not normalized, on the other hand, the sum of c_{11} and 2^{-24} would be held with one extra bit, and the remaining addends could be added to 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 similar choice was made for example in the hardware accelerator for performing vector inner product described by Kim and Kim (2009).

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 a normal number. 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 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. We can conclude that the result of the subtraction is not normalized, as long as we assume that the summands in (3) are accumulated on the largest in magnitude in a fixed order.

Extra bits for carry out   Another question concerns the number of 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 (Ercegovac and Lang, 2004, Sec. 3.1), 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 internal accumulator of the 5-operand adder 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}. We consider all four possible permutations of these values in the first column of B, as we assume that the addends apart from the largest in magnitude are not sorted. The main idea is to show that if 1 is added to c_{11} three times, then the last two bits of c_{11} are not dropped as they would be if the accumulation were performed using IEEE 754 floating-point arithmetic, as the significand of $c_{11} + 3 > 4$ would have to be shifted by two places to the right in order to be normalized. Then, when 2^{-23} is added at the end, the carry propagates into the third bit from the bottom and therefore is not lost in the final normalization step. If there are 2 extra bits, then all the four possible orderings of the first column of B will return the exact result $4 + 2^{-21}$. Running these tests on tensor cores, we found that all four combinations returned the exact result, thereby proving that the significand of the internal accumulator has at least 2 extra bits

406 for carries.

407   This technique cannot be used to incontrovertibly show that the five-operand adder
408 has a third extra bit for carries. On the one hand, all inputs to the multi-operand adder
409 must have the most significant bit of the fraction (the implicit bit) set to 1, in order to
410 produce carry that requires all three extra bits, on the other, the result of one of the four
411 products of the form $a_{1k}b_{k1}$ in (2) must have the least significant bit of the fraction set
412 to 1. As the product of two binary16 numbers can have at most 22 significant digits, no
413 combination of input can produce a partial product with the required characteristics.

414   It is possible, however, to show the presence of the third bit if we assume that the
415 alignment of the significand is always performed so that the most significant bit of the
416 largest summand in the five-operand adder occupies the left-most position. Since we
417 know that there are two extra bits and no normalization, we can show that there is also
418 a third bit by showing that there is no overflow when each of the four additions in (3)
419 causes carry out.

420   We can set, for example, $c_{11} = 1 + 2^{-1} + 2^{-2} + 2^{-3}$, $b_{11} = 1$, $b_{21} = 1 + 2^{-1}$,
421 $b_{31} = 1 + 2^{-1} + 2^{-2}$, and $b_{41} = 1 + 2^{-1} + 2^{-2} + 2^{-3}$ and observe that the tensor core
422 returns $d_{11} = 8$. If only two extra bits were present, on the other hand, overflow would
423 occur and the adder would incorrectly return $d_{11} = 0$. In summary, we can conclude that
424 the internal significand of the tensor cores is most likely 27 bits wide in Volta cards and
425 28 bits wide in Turing and Ampere cards (see Sections 3.2 and 3.3).

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

431   In the test case above, by replacing one of the 2^{-23} by 1 we can also confirm, using
432 the methods developed in Section 3.1.3, that the rounding mode in the final normalization
433 step (internal accumulator conversion to binary32 answer) is round-towards-zero.

434 **Monotonicity of dot product**   The observation in section 3.1.4 raises one final question
435 regarding the monotonicity of the sums in (2). The accumulation is monotonic if in
436 floating-point arithmetic the sum $x_1 + \cdots + x_n$ is no larger than $y_1 + \cdots + y_n$ if $x_i \le y_i$ for
437 all $1 \le i \le n$.

We can show that the lack of normalization causes the dot product in tensor cores—
and most likely in any other similar architectures in which partial sums are not normal-
ized (Kim and Kim, 2009; Tao et al., 2013; Sohn and Swartzlander, 2016; Kaul et al.,
2019)—to behave non-monotonically. Let us consider (3) and set 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 $1 - 2^{-24} + 3 \cdot 2^{-24} = 1 + 2^{-23}$. On the other hand, when $c_{11} = 1$, none of the
summands in (3) is large enough to 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 we have

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

These two sets of inputs demonstrate that tensor cores can produce non-monotonic behavior.

3.2 NVIDIA Turing tensor cores

NVIDIA Turing T4 GPUs are equipped with the second generation of tensor cores, which adds an integer matrix multiply-accumulate operation. It is not documented whether the binary16/binary32 tensor core arithmetic in Turing chips differs from that in Volta cards, therefore it is of interest to run the test suite we designed on one of the Turing cards.

We ran all the above experiments on an NVIDIA Tesla T4 16GB (Turing microarchitecture) GPU, and noticed that some of the results were different from those obtained on a V100 GPU. We found that this is due to the presence of an additional extra bit of precision at the bottom of the significand of the internal accumulator of the 5-operand adder. This has an impact over several of the tests above: the operation $1 + 2^{-24} + 2^{-24}$, for example, can now be performed exactly because of the presence of the extra bit. The results obtained on the V100 GPU can be replicated by means of a suitable change of the constants that are chosen depending on the number of extra bits in the accumulator. For instance, in the test for the order of operations in Section 3.1.2, the constant 2^{-24} should be replaced by 2^{-25}, which is the value of the next bit to the right. Using this approach, we found that all the conclusions we drew about the tensor cores in V100 GPUs remain valid for the second version of tensor cores in the T4 GPUs, with the only exception of the extra bit at the bottom of the internal storage of the 5-operand adder.

If we denote the fixed-point format with I integer bits and F fraction bits by $\{I.F\}$, the significand of the internal format of the 5-operand adder of a V100 GPU has format $\{3.23\}$ (or $\{4.23\}$ if 3 extra bits for carries are present as discussed in Section 3.1.4), whereas that of a T4 GPU has format $\{3.24\}$ (or $\{4.24\}$). The final normalization and rounding produce a number whose significand has format $\{1.23\}$, which is the format of the significand of a binary32 floating-point number.

3.3 NVIDIA Ampere tensor cores

As shown in Table 1, the Ampere microarchitecture offers four different variants of tensor core operations (input/output format): binary16/binary32, bfloat16/binary32, TensorFloat-32/binary32, and binary64/binary64. In this section we summarize, for each of these four configurations, the results obtained by running our test suite on the A100 GPUs. We refer to the 4-element dot product in (2), even though some of the tensor core operations available on A100 GPUs use input vectors of dimension two or eight, as can be seen in Table 1. In our tests we only use the first few elements of each input vector, so (2) is relevant even when dealing with operations that would require some of the input vectors to have eight elements, as we tacitly assume that all the remaining entries are set to 0.

In binary16/binary32 mode, we found no differences between the Ampere and Turing tensor cores, as all the tests in our suite produce the same results on the two GPUs.

478 Therefore we moved to the other three modes, which are new as they were introduced
479 with the NVIDIA Ampere architecture.

480 We started by adapting our test suite to the bfloat16/binary32 mode. The main
481 difference in this configuration is that the subnormals in bfloat16 are a subset of the
482 subnormals in binary32, whereas all binary16 subnormals were normalized values in bi-
483 nary32. For this reason, in binary16/binary32 mode, we could only observe that binary32
484 subnormals in the input are preserved in the output. In this mode, however, binary32
485 subnormal outputs may in principle be produced during the computation from normal
486 inputs, and we can verify that this is indeed the case with the following experiment. If
487 in (2) we set $a_{11} = 2^{-126}$, $b_{11} = 2^{-1}$ (normalized bfloat16 values),and the remaining
488 coefficients to zero, the tensor cores correctly produce the subnormal binary32 result
489 $d_{11} = 2^{-127}$. To summarize, this precision configuration presents the same numerical
490 behavior as the binary16/binary32 configuration in the T4 GPU tensor cores, and an
491 additional test for subnormals confirmed full support for bfloat16 subnormal input and
492 binary32 subnormal output.

493 Next we looked at the A100 with tensor cores configured in TensorFloat-32/binary32
494 mode. TensorFloat-32 can represent more subnormal values than bfloat16 due to extra 3
495 bits in the significand (Table 3), and this was taken into account in our tests. The rest of
496 the test suite was run with the same input used for the binary16/binary32 configuration,
497 since the significands are of the same width, and produced the same results. Our
498 conclusion is that the TensorFloat-32/binary32 mode exhibits the same features as the
499 binary16/binary32 configuration in the T4 GPU.

500 The last configuration of the Ampere tensor cores is binary64/binary64. We found
501 that the numerical behavior of this mode differs significantly from that of the other three
502 configurations that are available on these graphic cards. In the multi-operand addition
503 in (2), in particular, we observed that round-to-nearest with ties to even is used, and that
504 the normalization is performed after each addition rather than only once at the end of the
505 computation. As the result of each addition is normalized, no extra bits for carries are
506 required, and the monotonicity issue discussed in Section 3.1.4 does not occur for this
507 rounding mode. We also note that since accumulation is performed in binary64, there
508 would be no advantage in holding the binary64 products exactly. Moreover, the additions
509 in (2) are not always performed starting from the largest addend, as was the case with
510 the other modes. In summary, our results suggest that in binary64/binary64 mode the
511 Ampere tensor cores have the same numerical features that a matrix multiply-accumulate
512 operation implemented using IEEE 754-compliant arithmetic operations would have.

4 CONCLUSIONS

514 In summary, our experiments indicate that the tensor cores in the NVIDIA V100 ar-
515 chitecture have the following numerical features. The first two of these (except the
516 rounding mode for the second one) are stated in the NVIDIA documentation and are
517 confirmed by our experiments, while the rest are not documented by NVIDIA and have
518 been revealed by our tests.

519 1. The binary16 products in (2) are computed exactly, and the results are kept in full
520 precision and not rounded to binary16 after the multiplication.

2. The additions in (2) are performed using binary32 arithmetic with round-towards-zero.

3. Subnormal numbers in binary16 and binary32 are supported.

4. The five summands in (2) are accumulated starting with the largest in absolute value.

5. Only the final result of (2) is normalized; the partial sums are not, and the accumulator has an internal significand that is 27 bits wide to accommodate for carries of the 5-term addition.

6. The dot products in tensor cores 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,

The same properties were found in the second generation tensor cores which equip the NVIDIA T4 GPUs, the main difference being one extra bit of precision in the significand of the internal accumulator of the binary32 5-operand adder. In the third generation of tensor cores, available on the NVIDIA A100 GPUs, all features were the same as in the T4, with the exception of the binary64/binary64 tensor cores, which we found are implemented to behave as if implemented in IEEE 754-compliant arithmetic.

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 NVIDIA Turing and Ampere tensor cores, for instance, added support for 4- and 8-bit integer modes (NVIDIA, 2018, 2020b), and rounding issues become relevant when these are used to implement fixed-point arithmetic.

5 ACKNOWLEDGEMENTS

The authors thank the University of Manchester for providing access to the NVIDIA V100 graphic cards through the Computational Shared Facility, the IT services of the university for setting up access to the NVIDIA T4 graphic cards through the Amazon Web Services, and the Innovative Computing Laboratory at the University of Tennessee, Knoxville, US for providing access to the NVIDIA A100 graphics cards.

REFERENCES

Abdelfattah, A., Anzt, H., Boman, E. G., Carson, E., Cojean, T., Dongarra, J., Gates, M., Grützmacher, T., Higham, N. J., Li, S., Lindquist, N., Liu, Y., Loe, J., Luszczek, P., Nayak, P., Pranesh, S., Rajamanickam, S., Ribizel, T., Smith, B., Swirydowicz, K., Thomas, S., Tomov, S., Tsai, Y. M., Yamazaki, I., and Yang, U. M. (2020). A survey of numerical methods utilizing mixed precision arithmetic. ArXiv:2007.06674.

Arm Ltd. (2020). Arm architecture reference manual. Technical Report ARM DDI 0487F.b (ID040120).

Basso, P. M., dos Santos, F. F., and Rech, P. (2020). Impact of tensor cores and mixed precision on the reliability of matrix multiplication in GPUs. *IEEE Transactions on Nuclear Science*, 67(7):1560–1565.

Blanchard, P., Higham, N. J., Lopez, F., Mary, T., and Pranesh, S. (2020). Mixed precision block fused multiply-add: Error analysis and application to GPU tensor cores. *SIAM Journal on Scientific Computing*, 42(3):C124–C141.

Ercegovac, M. D. and Lang, T. (2004). *Digital Arithmetic*. Morgan Kauffmann, San Francisco, CA, USA.

Google (2020). System architecture. Available at `https://cloud.google.com/tpu/docs/system-architecture` (accessed 15 April 2020).

Haidar, A., Abdelfattah, A., Zounon, M., Wu, P., Pranesh, S., Tomov, S., and Dongarra, J. (2018a). The design of fast and energy-efficient linear solvers: On the potential of half-precision arithmetic and iterative refinement techniques. In Shi, Y., Fu, H., Tian, Y., Krzhizhanovskaya, V. V., Lees, M. H., Dongarra, J., and Sloot, P. M. A., editors, *Computational Science—ICCS 2018*, volume 10860 of *Lecture Notes in Computer Science*, pages 586–600.

Haidar, A., Bayraktar, H., Tomov, S., Dongarra, J., and Higham, N. J. (2020). Mixed-precision solution of linear systems using accelerator-based computing. Technical Report ICL-UT-20-05, Innovative Computing Laboratory, University of Tennessee, Knoxville, TN, USA. To appear in Proceedings of the Royal Society A: Mathematical, Physical and Engineering Sciences.

Haidar, A., Tomov, S., Dongarra, J., and Higham, N. J. (2018b). 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), pages 47:1–47:11, Piscataway, NJ, USA. IEEE Press.

Hickmann, B. and Bradford, D. (2019). Experimental analysis of matrix multiplication functional units. In *Proceedings of the 26th IEEE Symposium on Computer Arithmetic*, pages 116–119, Kyoto, Japan.

IEEE (2019). *IEEE Standard for Floating-Point Arithmetic, IEEE Std 754-2019 (revision of IEEE Std 754-2008)*. Institute of Electrical and Electronics Engineers, Piscataway, NJ, USA.

Intel Corporation (2018). BFLOAT16—hardware numerics definition. Available at `https://software.intel.com/en-us/download/bfloat16-hardware-numerics-definition` (accessed 15 July 2020). White paper. Document number 338302-001US.

Intel Corporation (2020). Intel architecture instruction set extensions and future features programming reference. Available at `https://software.intel.com/sites/default/files/managed/c5/15/architecture-instruction-set-extensions-programming-reference.pdf` (accessed 15 July 2020).

Jia, Z., Maggioni, M., Smith, J., and Scarpazza, D. P. (2018a). Dissecting the NVidia Turing T4 GPU via microbenchmarking. arXiv:1903.07486 [cs.DC].

Jia, Z., Maggioni, M., Staiger, B., and Scarpazza, D. P. (2018b). Dissecting the NVIDIA Volta GPU architecture via microbenchmarking. arXiv:1804.06826 [cs.DC].

Kahan, W. M. (1981). Paranoia. Available at `http://www.netlib.org/`

606 paranoia/paranoia.b (accessed 15 July 2020).

607 Karpinski, R. (1985). Paranoia: A floating-point benchmark. *Byte Magazine*, 10(2):223–
608 235. Code available at http://www.netlib.org/paranoia/.

609 Kaul, H., Anders, M., Mathew, S., Kim, S., and Krishnamurthy, R. (2019). Optimized
610 fused floating-point many-term dot-product hardware for machine learning acceler-
611 ators. In *Proceedings of the 26th IEEE Symposium on Computer Arithmetic*, pages
612 84–87.

613 Kim, D. and Kim, L. (2009). A floating-point unit for 4D vector inner product with
614 reduced latency. *IEEE Transactions on Computers*, 58(7):890–901.

615 Markidis, S., Chien, S. W. D., Laure, E., Peng, I. B., and Vetter, J. S. (2018). NVIDIA
616 tensor core programmability, performance & precision. In *Proceedings of the 32rd*
617 *IEEE International Parallel and Distributed Processing Symposium Workshops*, pages
618 522–531.

619 Mukunoki, D., Ozaki, K., Ogita, T., and Imamura, T. (2020). DGEMM using tensor
620 cores, and its accurate and reproducible versions. In Sadayappan, P., Chamberlain,
621 B. L., Juckeland, G., and Ltaief, H., editors, *Proceedings of the International Con-*
622 *ference on High Performance Computing*, pages 230–248. Springer International
623 Publishing.

624 Muller, J.-M., Brunie, N., de Dinechin, F., Jeannerod, C.-P., Joldes, M., Lefèvre, V.,
625 Melquiond, G., Revol, N., and Torres, S. (2018). *Handbook of Floating-Point*
626 *Arithmetic*. Birkhäuser, Cham, Switzerland, 2nd edition.

627 NVIDIA (2017). NVIDIA Tesla V100 GPU architecture. Available at
628 https://images.nvidia.com/content/volta-architecture/
629 pdf/volta-architecture-whitepaper.pdf (accessed 15 July 2020).
630 NVIDIA whitepaper WP-08608-001_v1.1.

631 NVIDIA (2018). NVIDIA Turing GPU architecture. Available at https:
632 //www.nvidia.com/content/dam/en-zz/Solutions/design-
633 visualization/technologies/turing-architecture/NVIDIA-
634 Turing-Architecture-Whitepaper.pdf (accessed 15 July 2020).
635 NVIDIA whitepaper WP-09183-001_v01.

636 NVIDIA (2020a). Multiply-and-Accumulate Instruction: mma. Avail-
637 able at https://docs.nvidia.com/cuda/parallel-thread-
638 execution/index.html#warp-level-matrix-instructions-mma
639 (accessed 15 April 2020).

640 NVIDIA (2020b). NVIDIA A100 tensor core GPU architecture. Avail-
641 able at https://www.nvidia.com/content/dam/en-zz/Solutions/
642 Data-Center/nvidia-ampere-architecture-whitepaper.pdf (ac-
643 cessed 15 July 2020). NVIDIA whitepaper v1.0.

644 NVIDIA (2020c). NVIDIA CUDA Math API. Available at https://docs.nvidia.
645 com/cuda/cuda-math-api/index.html (accessed 15 April 2020).

646 Santoro, M. R., Bewick, G., and Horowitz, M. A. (1989). Rounding algorithms for IEEE
647 multipliers. In *Proceedings of the 9th IEEE Symposium on Computer Arithmetic*,
648 pages 176–183.

649 Sohn, J. and Swartzlander, E. E. (2016). A fused floating-point four-term dot product
650 unit. *IEEE Transactions on Circuits and Systems I: Regular Papers*, 63(3):370–378.

651 Tao, Y., Deyuan, G., Xiaoya, F., and Nurmi, J. (2013). Correctly rounded architectures

652   for floating-point multi-operand addition and dot-product computation. In *Proceedings*
653   *of the 24th International Conference on Application-Specific Systems, Architectures*
654   *and Processors*, pages 346–355.

655   Tenca, A. F. (2009). Multi-operand floating-point addition. In *Proceedings of the 19th*
656   *IEEE Symposium on Computer Arithmetic*, pages 161–168.

657   Yan, D., Wang, W., and Chu, X. (2020). Demystifying tensor cores to optimize half-
658   precision matrix multiply. In *2020 IEEE International Parallel and Distributed*
659   *Processing Symposium*, pages 634–643, New Orleans, LA, USA.