2024 IEEE 24th International Symposium on Cluster, Cloud and Internet Computing (CCGrid) | 979-8-3503-9566-2/24/$31.00 ©2024 IEEE | DOI: 10.1109/CCGrid59990.2024.00014

FTTN: Feature-Targeted Testing for Numerical Properties of NVIDIA & AMD Matrix Accelerators

Xinyi Li*, Ang Li†, Bo Fang†, Katarzyna Swirydowicz†, Ignacio Laguna‡ and Ganesh Gopalakrishnan*
*Kahlert School of Computing, University of Utah
xin_yi.li@utah.edu, ganesh@cs.utah.edu
†Pacific Northwest National Laboratory
{ang.li, bo.fang, kasia.swirydowicz}@pnnl.gov
‡Lawrence Livermore National Lab Laboratory
ilaguna@llnl.gov

***Abstract*—NVIDIA Tensor Cores and AMD Matrix Cores (together called Matrix Accelerators) are of growing interest in high-performance computing and machine learning owing to their high performance. Unfortunately, some of their crucial numerical attributes pertaining to departures from full IEEE floating-point compatibility are not documented. This makes it impossible to reliably port codes across these differing accelerators. This paper contributes a collection of *Feature Targeted Tests for Numerical Properties* that that help determine these features across five floating-point formats, four rounding modes and additional that highlight the rounding behaviors and preservation of extra precision bits. To show the practical relevance of FTTN, we design a simple matrix-multiplication test designed with insights gathered from our feature-tests. We executed this very simple test on five platforms, producing different answers: V100, A100, and MI250X produced 0, MI100 produced 255.875, and Hopper H100 produced 191.875. Our matrix multiplication tests employ patterns found in iterative refinement-based algorithms, highlighting the need to check for significant result variability when porting code across GPUs.**

***Index Terms*—NVIDIA GPUs, Tensor Cores, AMD GPUs, Matrix Units, floating-point arithmetic, Correctness Portability**

I. Introduction

We are in an era of rising computing hardware heterogeneity where many new CPU and GPU components are introduced in rapid succession [1], and are fueling performance advances in HPC and ML: from drug discovery to climate simulations and beyond. Ensuring correctness has become a serious challenge given the sheer number of hardware units, the rapidity of their adoption, and how much scientific discovery nowadays depends on computational science. Specifically in the realm of GPU-based accelerators, programmers are interested in testing codes developed for NVIDIA GPUs on AMD GPUs that are becoming available: MI100 at first, MI250X used in Oakridge OLCF Frontier [2] and MI300 slated for the El Capitan Exascale machine [3]. While these GPUs are generally well-documented, when it comes to important details of their floating-point matrix accelerator units, one finds gaps that—as we show in this paper—can seriously affect numerical portability. We use the term "Matrix Accelerator" as a generic term to refer to what NVIDIA calls "Tensor Cores [4]" and AMD calls "Matrix Cores [5]." Matrix Accelerators are indispensable for achieving today's performance levels in ML (e.g., training can take at least 10 times longer without them [6]). Matrix accelerators are found attractive by HPC designers who see its 4× speedup with 80% less energy consumption [7] a real avenue toward much faster and energy-efficient codes [8]. Unfortunately, many crucial numerical attributes of available matrix accelerators are not documented; these include details pertaining to rounding modes, fused-multiply-addition (FMA), the number of extra precision bits provided, and the granularity of block fused-multiply-add operations. It is highly desirable to have *feature-targeted tests* that can reveal these feature differences between GPUs so that one can be sure whether floating-point code will reliably port across them. **Our primary contribution is the design and evaluation of these feature-targeted tests.** We also show how seemingly small differences can compound into major result-differences. To illustrate this point, recall that 32-bit floating-point format (FP32) helps preserve seven fractional digits of accuracy and 16-bit format (FP16, used internally by matrix accelerators) preserves three *fractional* digits. The feature-targeted tests described in this paper helped obtain following answers for the same matrix multiplication: 0 on the V100, A100, and MI250X GPUs; 255.875 on MI100; and 191.875 on Hopper H100. This error is *six orders of magnitude higher* than expected for FP16.

II. Background

Floating-point background: A floating-point number [9] $x = (s, e, m)$ consists of a single sign bit s, a mantissa (also called significand) m (of 23 bits) representing a value in the real interval $(0, 2)$ and an exponent e (of 8 bits, typically presented as a biased integer). Regard m and e as the intended (i.e., ignore the bias in e) real-numbered. Then the value of the floating-point number is fp_value$(x) = (-1)^s \cdot m \cdot 2^e$ (see Table I for other pertinent details). Aiming for a unique and convenient representation, the mantissa m remains in the range of $[1, 2)$ whenever $e > e_{min}$, and hence can be expressed as a fraction $1.(..23bits..)$ which is called the *normalized* representation. For cases where $e = e_{min}$, the mantissa falls within the open interval $(0, 1)$, and then represent *subnormal*

979-8-3503-9566-2/24/$31.00 ©2024 IEEE
DOI 10.1109/CCGrid59990.2024.00014

TABLE I: Floating-Point Format Comparison (TF32 is a proprietary format and can be implementation-dependent). The more the mantissa bits, the lesser the ulp. Notice how BF16 sacrifices mantissae for higher dynamic range (larger exponent size)

Format	Sign Bit (S)	Exponent Bits (E)	Mantissa Bits (F)	Min. Exponent (e_{min})	Max. Exponent (e_{max})	ulp, i.e. ulp(1)
FP16	1	5	10	-14	15	2^{-10}
FP32	1	8	23	-126	127	2^{-23}
FP64	1	11	52	-1022	1023	2^{-52}
BF16	1	8	7	-63	63	2^{-7}
TF32	1	8	10	-126	127	2^{-10}

numbers.[1] We use ulp as an abbreviation for *units in the last place* and represents $fp\_value(x)$ when $s = 0, e = 0$ and only the LSB of m is set. The IEEE standard also details the specifications for 16-bit, 32-bit, and 64-bit floating-point numbers (FP16, FP32, and FP64 respectively). While Google introduced BF16 [10] in support of ML, NVIDIA unveiled TensorFloat32 [11] tailored for matrix multiplication.

Behavioral Portability Issues due to Subnormals: To motivate reasons for providing subnormal handling capability tests, consider two floating-point *normal* values a and b are close together but not individually equal to 0. Suppose we now have an expression $E_1 = c/(a - b)$ where E_1 is some expression and c is a normal number. If the result of $(a-b)$ is a subnormal number as per an infinite-precision calculation but the hardware does not provide support for subnormals, then the hardware turns the denominator into 0 causing a division-by-zero exception.[2] The problem posed by this situation is that some GPUs do not have hardware traps for exceptions [12]. Now if we have another expression E_2 similar to E_1 and we have E_1/E_2; then the resulting ∞/∞ results in a NaN ("not a number") exception—for which also GPUs lack adequate exception-trapping support in hardware.

Rounding Mechanisms in Floating-Point Arithmetic: The IEEE prescribes that rounding should emulate an intermediate result that is infinitely precise and possesses an unbounded range (this is called *correct rounding*). To realize this ideal, supplementary bits (guard or G, rounding or R, and sticky or S, collectively called "extra bits", see Table II) are incorporated in the design of IEEE-compliant hardware, with G, R, and S having lower significance (in that order) than the mantissa **least significant bit** (LSB). These bits are set when operation results are normalized via a right shift (see below for an illustration). To realize correct rounding [13], there are two requirements: (1) employ three extra precision bits, and (2) employ round-to nearest with ties to even.

Description of Rounding Here are the basic rounding steps: (1) *Align:* (If necessary), make the exponent of the two numbers to be added the same by right-shifting the number with the smaller exponent. (2) *Operate, normalize, set extra bits:* Perform the addition, and then normalize the result; specifically, if the result mantissa is 2 or more in value, bring it within $[1, 2)$ by right-shifting the mantissa, suitably adjusting the exponent. This right shift sets through and sets the extra bits. (3) *Round as per rules, normalize again if needed:* Consult Table II to round or truncate. **An Example:** Consider an FP scheme with one bit mantissa and suppose the result after calculation is positive 1.1100 in binary or 1.75 in decimal ($GRS = 100$ is attached at the end), and let $e = 0$. This cannot be represented using one mantissa bit, and so we must round. For RTN-TE[3] the mantissa LSB resulting in 10.0100. This needs normalization, and after that, the result is 1.0010 (and $e = 1$)—i.e., 2 in decimal. The answer for truncate is 1.

TABLE II: Rounding Rules: First locate the GRS bits. Then decide the result sign and the rounding mode desired. For all but truncate, add the specified bit to the mantissa least significant bit (LSB). For truncate, set LSB to this value.

The three extra bits GRS where $(x \vee y) = 1$	Result sign	New value for mantissa LSB (add this bit to m's LSB except for truncate it is assigned to LSB			
		Round up (tow. $+\infty$)	Round down (tow. $-\infty$)	RTN-TE	Round to zero (truncate)
0xy	+	1	0	0	0
	-	0	1	0	0
100	+	1	0	1	0
	-	0	1	1	0
1xy	+	1	0	1	0
	-	0	1	1	0

Fused Multiply-Add (FMA) Operation: In contemporary computational architectures, certain machines incorporate hardware components specifically designed to facilitate the Fused Multiply-Add (FMA) operation. As per the IEEE 754 standards, this operation computes $c + (a \cdot b)$ by ensuring two pivotal conditions: (1) computation is performed as though it has infinite precision and an unbounded range, and (2) rounding is applied only once, after the completion of both '*' and '+'. These are referred to as FMA conventions in this paper. Matrix multiplication, represented as $A \cdot B + C$, can be conceptualized as a series of blocked Multiply-Add Operations. This operation is natively supported by GPU architectures from both AMD and NVIDIA, as elaborated in Section II-A. Given this context, we hypothesize these matrix accelerators also adhere to these FMA conventions.

[1]Both $+0$ and -0 are supported, but neither is a subnormal number. However note that ulp and half of ulp are both normal numbers.

[2]Assuming that pertinent compiler flags are applied.

[3]RTN-TE stands for "Round to Nearest, Ties to Even," which is a rounding method commonly used in floating-point arithmetic.

A. Matrix Acceleration

NVIDIA and AMD have developed specialized compute units. NVIDIA's Tensor Cores and AMD's Matrix Cores are designed to optimize matrix operations, enhancing computational speed and efficiency [8]. We use the neutral term **matrix accelerator** when referring to either. Matrix multiplication, represented by the equation $D = A \cdot B + C$, is a foundational primitive in Linear Algebra (it is a BLAS level 3 operation). Equation 1 for all i, j in the allowed range of matrix indices $1 \ldots Size$ governs the behavior of matrix accelerators:

$$d_{ij} = a_{i1} * b_{j1} + a_{i2}b_{2j} + ... + a_{in}b_{nj} + c_{ij}. \quad (1)$$

Block FMA: Existing public documentation on matrix accelerators [7, 8] indicates that they employ the so-called *block FMA* where the calculation in Equation 1 is achieved in parallel, essentially suffering rounding error comparable to doing one scalar FMA. In other words, if one unrolls Equation 1 into a serial loop and determines the rounding error, then it is clear that each add-multiply step can incur a half ulp error in RTN-TE, thus making the worst-case error grow with $Size$. This is avoided in block-FMA (we assume block-FMA's presence).

Coding Matrix Acceleration: There are mainly two ways to invoke matrix accelerators: (1) *Via High-Level APIs:* One can make use of high-level C++ APIs such as `nvcuda::wmma` for NVIDIA and `nrocmwmma:wmma` for AMD. These APIs provide a structured and relatively user-friendly interface to interact with Tensor and Matrix cores, respectively. (2) *Intermediate-Level Assembly Manipulation:* For those delving deeper into the architecture, direct interaction with matrix accelerators unit is feasible through specific instruction sets. Within the NVIDIA platform, this is achieved by the PTX instruction set `wmma` operations while AMD offers compiler intrinsic instructions such as `__builtin_amdgcn_mfma_`. In our work, we employed the high-level API directly, abiding by all the requirements for its invocation such as meeting dimensionality restrictions published by manufacturers. To double-check that matrix accelerator units will be active during operation, we check the underlying code. For NVIDIA, the presence of `HMMA/DMMA` in the SASS code indicates that the Tensor Cores will be activated. For AMD, spotting `MFMA` in the LLVM intermediate representation indicates their use. [4] Similarly, AMD describes the role of `MFMA` in their official documentation [15]. These documents also mention the conditions to be met before these units are activated.

III. Numerical Behaviors of Matrix Accelerators

The overall goal of a matrix accelerator is to efficiently support the calculations in producing the D matrix where $D = AB + C$, with A, B and C also being matrices. Since all D entries are calculated in an identical manner, it suffices to focus on how one particular entry, namely d_{11} is calculated: $d_{11} = a_{11}b_{11} + a_{12}b_{21} + ... + a_{1n}b_{n1} + c_{11}$. *An important contribution we make* is to orchestrate these tests according to the order in the flow-chart in Figure 2 so that some cases are eliminated or concluded early.[5]

T_si_no: "*subnormal in; normal out;*" i.e., if a computation unit is fed subnormal inputs, can it handle it at the input (without zeroing it), and produce a normal output?

T_ni_so: "*normal in; subnormal out;*" i.e., if a computation unit is provided normal inputs, and the computation resulting in a subnormal, then can this subnormal be output (or will it get zeroed)?

T_sa: "subnormal accumulation ok;" i.e., if a set of subnormals are being added, is the accumulation successful (or is the output getting zeroed)?

T_1_bit: "*at least one extra bit;*" i.e., is there at least one extra precision bit in the computation unit?

T_rnd_dir: "*rounding direction;*" i.e., determine the rounding direction based on the test outcome: possible test outcomes are to say whether to zero (truncate), down (to $-\infty$), RTN-TE, or up (to $+\infty$) are happening.

T_3_bits_fin_rnd: "*three extra bits are provided, final rounding;*" i.e., tests that locate if three extra precision bits are provided. It also determines the final rounding direction followed.

T_prod: "*product rounding direction;*" similar to T_rnd_dir except for the product terms during block FMA.

T_blk_fma_width: "*block FMA width;*" i.e., what is the unit-width for the block FMA operations?

T_pres_extra_acc: "*preservation of extra bit during accumulation;*" i.e., are the extra bits preserved during the accumulation of a block FMA unit.

T_acc_order: "*accumulation order control;*" i.e., can we determine the accumulation order being followed by the block FMA during its accumulation stage?

A. Details of Each Test

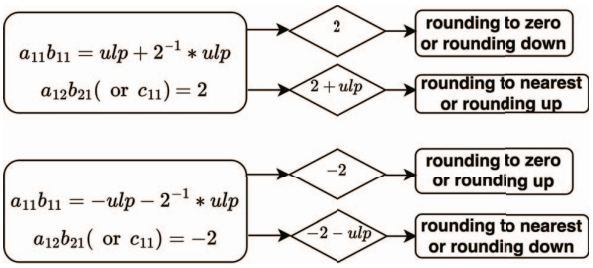

Fig. 1: The logic for test T_rnd_dir are presented here. By setting the $a_{11}b_{11}$ product as well as the $a_{12}b_{21}$ product (alternatively the c_{11} value) to the indicated value, the execution is carried out (all other inputs not mentioned are set to 0). Then by examining the d_{11} output, we can decide which case we fall into with respect to the rounding being used. A similar logic also underlies the T_{prod} test.

T_si_no, T_ni_so, and T_sa: The objective here is to discern whether the matrix unit can handle subnormal numbers, both

[4] NVIDIA's official documentation highlights the roles of `HMMA` and `DMMA` operations in Tensor Core operations [14]

[5] **It is important** to reiterate that the features discovered by these tests are largely undocumented or hard to find.

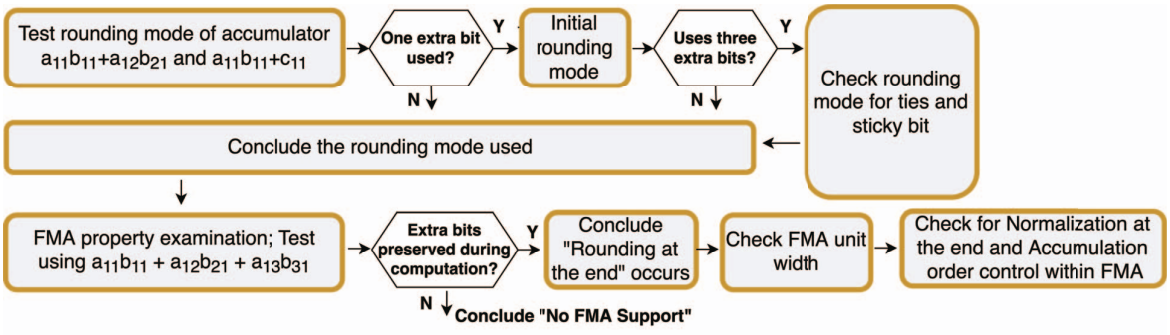

Fig. 2: Testing workflow that sharpens each later test based on the previous ones.

as input and output. The tests assign specific values to the right-hand side of the equation for d_{11}, and look for IEEE-compatible subnormal support.

T_si_no: Initialize a_{11} with an arbitrary subnormal number while ensuring that the product $a_{11}b_{11}$ yields a normal number; set all other input words in the d_{11} equation to 0. Now check whether d_{11} equals $a_{11}b_{11}$; if so, the check passes; else, d_{11} is expected to emerge as zero, when the check fails.

T_ni_so: Initialize a_{11} and b_{11} with arbitrary normal numbers while ensuring that the product $a_{11}b_{11}$ is a subnormal number. Now examine whether d_{11} is a subnormal value.

T_sa: Assign an arbitrary subnormal to c_{11} while keeping all other inputs at zero.The test observes whether d_{11} is this subnormal ("pass") or 0 ("fail").

T_1_bit, T_rnd_dir, and T_3_bits_fin_rnd: These tests follow the logic in Figure 1 for the first two tests, and Table III for the RTN-TE and truncate cases.

T_1_bit: We check the result of the operation $1-(2^{-1}\times ulp)$ to check if at least one extra bit is provided. Aligning $2^{-1} \times ulp$ to 1 in its binary representation necessitates a shift amount equivalent to the mantissa bit length plus one. Consequently, if the resultant value remains $1 - (2^{-1} \times ulp)$, it implies the existence of an extra bit in computations.

T_rnd_dir: Upon confirming the presence of an extra bit, the accumulator's rounding behavior is assessed (Figure 1) .

T_3_bits_fin_rnd: We finally proceed to determine if three extra bits are present, and also determine the final rounding modes supported. Its logic and implementation are now discussed. The key aspect of our testing approach was the alignment of three bits during the accumulation process. The goal was to ascertain the effects of preserving one, two, or all three extra bits on the mantissa component. This nuanced behavior was attained via subtraction operations (see Table III).

T_pres_extra_acc: The fact that the extra bits are retained during block-FMA accumulation can be confirmed using the expression $1 + 2^{-1} \cdot ulp + 2^{-1} \cdot ulp + 2^{-1} \cdot ulp$ (by making the accumulation of a block-FMA perform this calculation). If intermediate accumulation steps maintain these extra bits, the ultimate result will be $1 + ulp$; else it will emerge as 1.

T_acc_ord: The significance of accumulation order control primarily arises in scenarios employing rounding to zero with the preservation of just one extra bit. Contrarily, when three extra bits are enabled (which facilitates a sticky bit with the rounding-to-nearest mode), the results remain consistent irrespective of the accumulation order. If only one extra bit is maintained, we must test to ascertain the accumulation order. For this, one can test *all permutations* of the terms in the equation $1+2^{-2}\cdot ulp+2^{-2}\cdot ulp+2^{-2}\cdot ulp+2^{-2}\cdot ulp$. Given that only one extra bit is retained in the rounding to zero case, the precision associated with the terms $2^{-2}\cdot ulp+2^{-2}\cdot ulp+2^{-2}\cdot ulp+2^{-2}\cdot ulp$ tends to be lost, **save for when it is computed first**. That is, if we allow all the "small values" to add up first, then even with one extra bit, we will get the answer $1 + ulp$. In other words, ff we can *externally control the order of reduction* by assigning these terms to specific positions within an FMA unit, there exists a output yields $1 + ulp$ and hence the reduction order is under user control.

Algorithm 1: Test Minimum Unit for FMA property preservation. The idea is to assign a moving position the $2^{-1}ulp$ value and when that position goes beyond the width of the block FMA, we get a 1 output. That index is the FMA block width.

Data: Matrices a, b, c, and d. $a's$ row's length K.
Initialize all values in a, b, c to 0
$c_{11} \leftarrow 1$.
$a_{11} \times b_{11} \leftarrow 2^{-1} \times$ ulp
for $i \leftarrow 1$ **to** K **do**
  if $i > 1$ **then**
    $a_{1(i-1)} \times b_{(i-1)1} \leftarrow 0$
  $(a_{1i} \times b_{i1}) \leftarrow (2^{-1} \times$ ulp$)$
  Call **wmma**(a, b, c, d)
  if $d_{11} = 1$. **then**
    break
if *index* $< K$ **then**
  $min\_preserve\_uint \leftarrow index$
else
  $min\_preserve\_uint$ is larger than K

T_blk_fma_width: To determine the *block size* of the block FMA unit, we execute a test loop given in Figure 1. Within a single FMA unit, precision remains intact throughout computation, and rounding occurs only at the concluding bit position. The key idea realized by this test is to load-up a 1-bit at a pair of moving positions denoted by a_{1i} and b_{i1} such that $a_{1i} \times b\_i1$ is ensured to be half a ulp ($2^{-1} \times ulp$). We use the aforementioned equation and shift the last term, $2^{-1} \cdot ulp$, across the matrix multiplication positions as illustrated. A loss of precision at Line 9 (the half ulp vanishes) signals that the initialization occurred in a detached FMA unit where it suffers a rounding precision loss; the "end of a FMA unit" in effect gets detected when the final value is 1.

T_prod: Tests for the rounding mode of the product are only performed for FP64 and FP32 inputs. Assuming each multiplier has an n-bit mantissa, their products can occupy only $2n$ bits. Taking this into consideration, the input formats such as FP16, TF32, and BF16—which respectively possess 10, 10, and 7-bit mantissa bits—do not experience precision loss when operating within an FP32 environment where a 23-bit mantissa is used. To check product rounding, we can employ the same methodology used for rounding mode assessment during accumulation (Figure 1). Specifically, for the product $a_{11} \cdot b_{11}$, if we set one term to be $1+2 \cdot ulp+ulp$ and the other as $1+2^{-2}$, the exact result is $1+2^{-2}+3 \cdot ulp+2^{-1} \cdot ulp+2^{-2} \cdot ulp$, with a `110` suffix to the end of the mantissa bit. Similarly we can incorporate negative-number test scenarios (Figure 1) and referencing the rounding tests depicted there, we can deduce the rounding mode used in the multiplication operation.

IV. Feature Test Results (Table IV)

Subnormal Supports All the GPUs tested support subnormal numbers for inputs and outputs, with the exception of FP16 and BF16 formats of MI250X which does not. It is important to note that the absence of subnormal support could lead to the risk of generating exceptions such as division by zero (§II).

Extra Bits for Computation AMD GPUs consistently use three extra bits for precise rounding. In contrast, NVIDIA GPUs have evolved across generations: the V100 does not include any extra bits, the A100 includes one, and the H100 includes at least two extra bits[6]. For FP64 inputs, all GPUs incorporate an additional three bits.

Rounding Modes The chosen rounding mode is consistent across NVIDIA and AMD GPUs, with all models adhering to the chosen mode consistently across generations.

FMA Feature NVIDIA's V100 has an FMA unit width of 4, and the A100 expands this to 8, as documented. The H100's FMA unit width is suggested to be at least 16, a detail not officially confirmed. For TF32 inputs on NVIDIA GPUs, the FMA unit width is 4, which suits the 19-bit size of TF32. The AMD MI100 maintains FMA features with different widths for FP16 and BF16 inputs, but the MI250X lacks this feature. While FMA units can enhance accuracy, they may complicate the porting of CPU algorithms which do not typically support blocked FMA operations.

Rounding Mode for Outputting FP16 and BF16 We have examined the rounding mode used when GPUs output FP16 and BF16. All GPU models use the RTN-TE rounding mode. We hypothesize that the conversion to lower precision is performed after the computation at full precision.

Rounding Mode for Product For products involving FP32/FP64 inputs, all GPUs utilize the RTN-TE mode, demonstrating consistency in following IEEE floating-point arithmetic standards.

V. FTTN to Highlight Porting Dangers

As a demonstration of the practical importance of FTTN [16], consider mixed-precision GMRES (Generalized Minimal Residual Method) iterative refinement algorithms [17, 18] that are of significant interest in efficient linear system solving methods. In this application, the (so called) trailing matrix update pattern $A_i = A_i - P_i T_i$ arises. This pattern also arises in cuSolvers[7]. In general, a computation of the type $D = C - A \cdot B$ **is part of a standard BLAS** (Basic Linear Algebra Subprograms) level 3 family. To test the behavior of this pattern, we consider computing D which equals $C - AB$ where we fill matrices A, B and C of sizes $2^{13} \times 2^{13}$ as follows (these fill values were manually derived as detailed in [19]).

$$
\begin{aligned}
D_{ij} &= -(A_{i0} \cdot B_{0j} + \sum_{j\%2=1} A_{ij} \cdot B_{ij} + \sum_{j\%2=0}^{j\neq 0} A_{ij} \cdot B_{ij}) + C_{ij} \\
&= -(2^{10} \cdot 2^{10} - \sum_{2^{12}} 2^{-2} \cdot 2^{-3} - \sum_{2^{12}-1} 2^{-3} \cdot 2^{-3}) + 2^{20} \\
&= 2^7 + 2^6 - 2^{-6} \approx 191.99218
\end{aligned}
$$

Here, for matrix C, $C_{ij} = 2^{20}$ for all i and j. In matrix A, $A_{i0} = 2^{10}$, $A_{ij} = 2^{-2}$ for odd j, and $A_{ij} = 2^{-3}$ for even j (except $j = 0$). For matrix B, $B_{0j} = 2^{10}$, with other B_{ij} values set at 2^{-3}. Note that the terms $2^{-2} \cdot 2^{-3}$ and $2^{-3} \cdot 2^{-3}$ require bit shifts (25 bits or 26 bits) to align $2^{20} = 2^{10} \cdot 2^{10}$. Thus there would be precision loss for FP32 computation unit. This loss may vary depending on the number of extra bits preserved and the length of the FMA operation. **This variation is what produces the sharp result-difference that we observed.** Specifically, we observed these (rather highly different) D_{ij} values computed using a simple GEMM implementation on different GPUs *for the very same A, B, and C matrix inputs*: 0 on NVIDIA A100, V100, AMD MI250 and CPU; 255.875 on AMD MI100; and 191.875 on NVIDIA H100. *This degree of result-difference is very likely to prove unacceptable to those who port code across GPUs.*

Acknowledgments This material is based upon work supported by the U.S. Department of Energy, Office of Science, Office of Advanced Scientific Computing Research, ComPort: Rigorous Testing Methods to Safeguard Software Porting, under Award Number 78284. The Pacific Northwest National Laboratory is operated by Battelle for the U.S. Department

[6]Due to our limited access to the H100, we can only test for more than 2 extra bits. We did not conduct further FMA unit width tests for the same reason. We can, however, easily expand our tests to include three extra bits.

[7]https://docs.nvidia.com/cuda/cusolver/index.html#cusolverirsrefinement-t

TABLE III: Methodology for testing extra bits with different preliminary rounding modes: truncation and rounding to the nearest (RTN). Here, `Value` is used to denote the exact floating-point number **post-rounding**. `FP Rep` shows the binary representation of the actual floating-point value. The format within this cell includes the sign on the first row, the exponent value on the second row, and the binary representation of the mantissa on the third row. Numbers appearing after the bracket in the mantissa row represent additional bits that are not part of the standard floating-point representation.

		$a_{11}b_{11}$	$a_{12}b_{21}$	$a_{11}b_{11}$ alignment	$a_{11}b_{11}$+$a_{12}b_{21}$		
					extra 1 bit	extra 2 bit	extra 3 bit
test truncate	Value	$-(2^{-3}+2ulp)$	$1+2^{-2}$	N.A.	$1+2^{-3}$	$1+2^{-3}-ulp$	$1+2^{-3}-ulp$
	FP Rep	(-) (-3) (1.000...010)	(+) (0) (1.010...000)	(-) (0) (0.0010...000)010	(+) (0) (1.0010...000)0	(+) (0) (1.0001...111)11	(+) (0) (1.0001...111)110
test RTN	Value	$-(2^{-3}+2^2ulp$ $+ulp)$	$1+2^2$	N.A.	$1+2^{-3}-ulp$	$1+2^{-3}-ulp$	$1+2^{-3}$
	FP Rep	(-) (-3) (1.000...101)	(+) (0) (1.010...000)	(-) (-0) (0.0010...000)101	(+) (0) (1.0001...111)1	(+) (0) (1.0001...111)10	(+) (0) (1.0010...000)011

TABLE IV: Result summary: Here, "FMA unit size" is the number of words considered before rounding and normalization are performed ("Block FMA Size"). Note that V100 only support FP16. Further, FP64 is not supported in MI100. Last column: "Case 1" is for add/accumulate, and "Case 2" refers to the product test T_prod. ✓is yes, and ✗is no.

Inputs	GPU	Subnormal inputs handled?	Subnormal outputs handled?	Extra bit present? How many?	Rounding mode exhibited	FMA unit width	Order within one FMA unit is controllable?	Rounding mode for: 1. outputting FP16/BF16 (only for FP16/BF16 inputs) 2. product (only for FP32/FP64 inputs)
FP16	V100	✓	✓	0	truncate	N.A.	N.A.	RTN-TE
	A100	✓	✓	1	truncate	8	✗	RTN-TE
	H100	✓	✓	≥ 2	truncate	≥ 16	✗	RTN-TE
	MI100	✓	✓	3	RTN-TE	4	✗	RTN-TE
	MI250X	✗	✗	3	RTN-TE	1	N.A.	RTN-TE
BF16	A100	✓	✓	1	truncate	8	✗	N.A.**
	H100	✓	✓	≥ 2	truncate	≥ 16	✗	RTN-TE
	MI100	✓	✓	3	RTN-TE	2	✗	RTN-TE
	MI250X	✗	✗	3	RTN-TE	1	N.A.	RTN-TE
TF32(NVIDIA) FP32(AMD)	A100	✓	✓	1	RTN-TE	4	✗	N.A.
	H100	✓	✓	≥ 2	truncate	4	✗	N.A.
	MI100	✓	✓	3	RTN-TE	1	N.A.	RTN-TE
	MI250X	✓	✓	3	RTN-TE	1	N.A.	RTN-TE
FP64	A100	✓	✓	3	RTN-TE	1	✗	RTN-TE
	H100	✓	✓	3	RTN-TE	1	✗	RTN-TE
	MI250X	✓	✓	3	RTN-TE	1	N.A.	RTN-TE

* Since the V100 does not preserve extra bits, its FMA functionality cannot be evaluated. ** A100 doesn't support BF16 output.
The results can also be applied to GPUs with the same architecture (e.g. A100 to RTX 30 series), as they use the same generation of tensor cores.

of Energy under Contract DE-AC05-76RL01830. This work was performed under the auspices of the U.S. Department of Energy by Lawrence Livermore National Laboratory under Contract DE-AC52-07NA27344 (LLNL-CONF-861088). It is also based on NSF CISE Awards 2217154, 2124100 and 1956106, and DOE DE-SC0022252. This research used resources supported by U.S. DOE Office of Science, Office of Advanced Scientific Computing Research, under award 66150: "CENATE - Center for Advanced Architecture Evaluation". This research used resources of the Oak Ridge Leadership Computing Facility, which is a DOE Office of Science User Facility supported under Contract DE-AC05-00OR22725. This research used resources of the National Energy Research Scientific Computing Center (NERSC), a U.S. Department of Energy Office of Science User Facility located at Lawrence Berkeley National Laboratory, operated under Contract No. DE-AC02-05CH11231.

References

[1] Ganesh Gopalakrishnan, Ignacio Laguna, Ang Li, Pavel Panchekha, Cindy Rubio-González, and Zachary Tatlock. Guarding numerics amidst rising heterogeneity. In *2021 IEEE/ACM 5th International Workshop on Software Correctness for HPC Applications*, pages 9–15, 2021.

[2] https://www.olcf.ornl.gov/olcf-resources/compute-systems/frontier/.

[3] https://www.llnl.gov/article/49131/llnl-scientists-eagerly-anticipate-el-capitans-potential-impact.

[4] NVIDIA. NVIDIA A100 Tensor Core GPU architecture. https://images.nvidia.com/aem-dam/en-zz/Solutions/data-center/nvidia-ampere-architecture-whitepaper.pdf, 2020.

[5] AMD. Amd cdna architecture. https://www.amd.com/system/files/documents/amd-cdna-whitepaper.pdf, 2020.

[6] Deepak Narayanan, Mohammad Shoeybi, Jared Casper, Patrick LeGresley, Mostofa Patwary, Vijay Anand Korthikanti, Dmitri Vainbrand, Prethvi Kashinkunti, Julie Bernauer, Bryan Catanzaro, Amar Phanishayee, and Matei Zaharia. Efficient large-scale language model training on gpu clusters using megatron-lm, 2021.

[7] Pierre Blanchard, Nicholas J Higham, Florent Lopez, Théo Mary, and Srikara Pranesh. Mixed Precision Block Fused Multiply-Add: Error Analysis and Application to GPU Tensor Cores. *SIAM Journal on Scientific Computing*, 2020.

[8] Jack Dongarra, Laura Grigori, and Nicholas Higham. Numerical algorithms for high-performance computational science. *Phil. Trans. R. Soc. A.3782019006620190066*, 2020. http://doi.org/10.1098/rsta.2019.0066.

[9] Jean-Michel Muller, Nicolas Brunie, Florent de Dinechin, Claude-Pierre Jeannerod, Mioara Joldes, Vincent Lefvre, Guillaume Melquiond, Nathalie Revol, and Serge Torres. *Handbook of Floating-Point Arithmetic*. Birkhäuser Basel, 2nd edition, 2018.

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

[11] https://blogs.nvidia.com/blog/2020/05/14/tensorfloat-32-precision-format/.

[12] https://docs.nvidia.com/cuda/floating-point/index.html.

[13] Nicholas J Higham. *Accuracy and stability of numerical algorithms*. SIAM, 2002.

[14] Nvidia. Cuda binary utilities. https://docs.nvidia.com/cuda/pdf/CUDA_Binary_Utilities.pdf, 2023. Accessed: 2023-12-17.

[15] AMD. "amd instinct mi100" instruction set architecture reference guide. https://www.amd.com/content/dam/amd/en/documents/instinct-tech-docs/instruction-set-architectures/instinct-mi100-cdna1-shader-instruction-set-architecture.pdf, 2020. Accessed: 2023-12-17.

[16] Xinyi Li, Ang Li, Ignacio Laguna, and Ganesh Gopalakrishnan. https://github.com/LLNL/FTTN.git.

[17] Azzam Haidar, Stanimire Tomov, Jack Dongarra, and Nicholas J Higham. Harnessing gpu tensor cores for fast fp16 arithmetic to speed up mixed-precision iterative refinement solvers. In *SC18: International Conference for High Performance Computing, Networking, Storage and Analysis*, pages 603–613. IEEE, 2018.

[18] Azzam Haidar, Harun Bayraktar, Stanimire Tomov, Jack Dongarra, and Nicholas J Higham. Mixed-precision iterative refinement using tensor cores on gpus to accelerate solution of linear systems. *Proceedings of the Royal Society A*, 476(2243):20200110, 2020.

[19] Xinyi Li, Ang Li, Bo Fang, Katarzyna Swirydowicz, Ignacio Laguna, and Ganesh Gopalakrishnan. Fttn: Feature-targeted testing for numerical properties of nvidia & amd matrix accelerators, 2024.

APPENDIX: ARTIFACT FOR FTTN

FTTN comprises a suite of tests designed to studying the floating-point behaviors of matrix accelerators, such as tensor cores and matrix cores, on NVIDIA and AMD GPUs. The suite focuses on evaluating three principal aspects:

- The handling of subnormal numbers.
- The rounding methodology, including the utilization of extra bits.
- The characteristics of Fused Multiply-Add (FMA) unit, i.e. whether the rounding and normalization happened once at the end within an FMA unit.

The specifics of these tests are thoroughly explained in Section 3 of the paper.

This appendix[8] provides detailed instructions for deploying and executing this testing suite.

A. Prerequisites

We outline the hardware and software configurations on which our test suite has been successfully evaluated.

Hardware Requirements.

The suite has been tested on a range of GPUs from NVIDIA and AMD as listed below:

- NVIDIA GPUs: H100, A100, V100, RTX 3060
- AMD GPUs: MI100, MI250X

Software Requirements.

The following software environments are required to run the test suite:

- CUDA Toolkit: Versions 11.x or 12.x are supported.
- ROCm Platform: Version 5.7.0 or 5.3.0 should be installed.
- C++ Compiler: A compiler that supports the C++17 standard is necessary.

B. Organization and Experiment

The test suite's directory structure is depicted in Figure 3. For each data type, there is a corresponding `<datatype>-in.cpp` file containing all the relevant tests. It should be noted that the TF32 input format is only for to NVIDIA GPUs, and FP32 input is not available in NVIDIA GPUs.

Executing All Tests.

Following a simple two-step process, you can excute all tests for one machine.

1) Modify the `NVCC`, `CUDA_ARCH`[9], and `HIPCC` variables to match the compilers and architecture on your machine.
2) Execute the command `make amd` for AMD GPUs or `make nvidia` for NVIDIA GPUs.

This will compile and execute all tests for the chosen machine type, with results being saved in the respective text files.

[8]The link for our artifact is https://zenodo.org/record/10673370, we will also publish our tool at https://github.com/LLNL/FTTN.git

[9]You can find the compute capability for your GPU in these tables https://developer.nvidia.com/cuda-gpus#compute. In our experiment, use `sm_80` for A100, and `sm_70` for V100 and `sm_86` for RTX 3060.

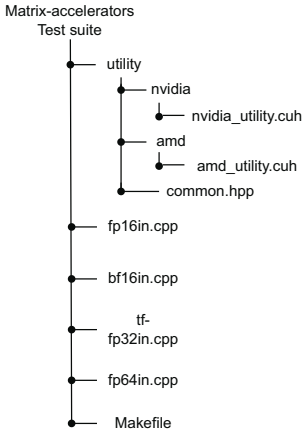

Fig. 3: Directory organization of our test suites.

Executing a Single Test.

To run tests for a single data type, use the command `make <cppfilename>-<GPU_Type>`. For example, to run all tests for the FP16 input data type on an NVIDIA GPU, the command would be `make fp16_NVIDIA`.

Run with Sbatch Script.

The tests can also be run using `.sbatch` script. Ensure you load the relevant modules (C++ compiler, CUDA for NVIDIA GPUs, or ROCm for AMD GPUs) before executing the `make` command as outlined previously.

C. Results

The outcomes of the tests will be recorded in a text file named according to the pattern `<filename>_result<GPU_type>.txt`; for instance, the results for FP16 test on NVIDIA GPU would be in `fp16_resultNVIDIA.txt`. At the end of the file, there's a summary that compiles the results into a format corresponding to a row in Table 3 of the paper. Before this summary, the results of each individual test are printed, providing an in-depth review of the specific test outcomes.