An SMT Formalization of Mixed-Precision Matrix Multiplication

Modeling Three Generations of Tensor Cores

Benjamin Valpey$^{1[0000-0002-2245-3022]}$ ✉, Xinyi Li$^{2[0009-0005-7276-7715]}$, Sreepathi Pai$^{1[0000-0002-3691-7238]}$, and Ganesh Gopalakrishnan$^{2[0000-0002-4161-9278]}$

^{1} University of Rochester, Rochester, NY 14627, USA
`{bvalpey,sree}@cs.rochester.edu`
^{2} University of Utah, Salt Lake City, UT 84112, USA {xin_yi.li@utah.edu, ganesh@cs.utah.edu}

Abstract. Many recent computational accelerators provide non-standard (e.g., reduced precision) arithmetic operations to enhance performance for floating-point matrix multiplication. Unfortunately, the properties of these accelerators are not widely understood and lack sufficient descriptions of their behavior. This makes it difficult for tool builders beyond the original vendor to target or simulate the hardware correctly, or for algorithm designers to be confident in their code. To address these gaps, prior studies have probed the behavior of these units with manually crafted tests. Such tests are cumbersome to design, and adapting them as the accelerators evolve requires repeated manual effort.
We present a formal model for the tensor cores of NVIDIA's Volta, Turing, and Ampere GPUs. We identify specific properties—rounding mode, precision, and accumulation order—that drive these cores' behavior. We formalize these properties and then use the formalization to automatically generate discriminating inputs that illustrate differences among machines. Our results confirm many of the findings of previous tensor core studies, but also identify subtle disagreements. In particular, NVIDIA's machines do not, as previously reported, use round-to-zero for accumulation, and their 5-term accumulator requires 3 extra carry-out bits for full accuracy. Using our formal model, we analyze two existing algorithms that use half-precision tensor cores to accelerate single-precision multiplication with error correction. Our analysis reveals that the newer algorithm, designed to be more accurate than the first, is actually less accurate for certain inputs.

Keywords: GEMM · Tensor Cores · Test Generation and Linear Algebra · Decision Procedures · Floating Point and Error Analysis · GPU

1 Introduction

As applications strive for greater performance in today's post Moore's Law era, hardware designers have turned to specialized hardware units and non-standard

ISA extensions to satisfy this need. In 2016, Google announced its Tensor Processing Unit, a hardware unit specializing in matrix multiplication [27]. The next year, NVIDIA announced its new Volta architecture would feature Tensor Cores [39] which have since evolved with each new architecture. Today, all major CPU and GPU vendors incorporate ISA extensions for matrix multiplication that operate on reduced-precision floating point formats. Notably, the functionality of these extensions is not standardized yet, and is poorly documented as well. While non-standard designs might be successfully employed within closed-source vendor libraries and for low-precision AI applications, they can prove to be difficult and error-prone for others who seek to build innovative linear algebra methods that are precision-sensitive [20]. Tensor cores have demonstrated numerical inconsistency across architectures that has an impact on the portability of algorithms. For example, a tensor core implementation of Fast-Fourier transform [32] saw its mean relative error drop by as much as 34%[3] when moving from Volta's tensor cores to Ampere's. This shifting behavior, coupled with the lack of a specification, presents a challenge for safety-critical applications that are sensitive to an implementation's numerical behavior. Furthermore, without a behavioral specification, efforts such as Goodloe et al. [19] that verify numerical programs cannot be utilized. Thankfully, work done by researchers to understand tensor core functionality [16, 23, 33] has led to novel algorithms that can use these cores to speed up even single-precision computations for HPC [37, 44]. As new cores are released, though, these same efforts must be repeated.

In this paper, we provide a formal description for the tensor cores across three generations of graphics cards. These formal descriptions can not only provide accurate and reliable component-level specifications enabling automated reasoning about their functionality but also facilitate the the creation of novel, hitherto unimagined, uses, while also improving debuggability, correctness checking, and security analyses.

Our models of tensor cores support two key properties: i) they are executable, ii) they can be used in automated reasoning. Our models are also parametric, enabling them to be quickly adapted for new architectures.

Our novel contributions include:

- A formalization of the numerical properties of mixed-precision block FMA, collected from prior literature, that can be used to identify the properties of different matrix multiplication units
- A formal, executable model of the matrix multiplication units across three generations of GPUs - Volta, Turing, and Ampere
- A revision of a mischaracterization in prior work that had concluded the rounding mode of tensor cores is round-to-zero. The actual rounding behavior, truncation, is subtly different.
- An analysis of two error-correcting matrix-multiplication algorithms that shows, due to the properties of the tensor cores, the method which trades

[3] An accuracy test that accompanies the implementation [31] reports an error of $1.5e^{-2}$ on Volta and $9.92e^{-3}$ on Ampere.

speed for accuracy can actually produce less accurate results than its faster counterpart.

The functional and performance aspects of tensor core behavior have received significant scrutiny through testing [50, 16, 54]. Nevertheless, our SMT formalization unearths subtle discrepancies between test-based reverse engineered descriptions and the actual hardware.

In addition to enabling program analyses, formal models enable the construction of hardware simulators. These simulators in turn allow developers to evaluate the numerical behavior of their algorithms on multiple different architectures, all from the same machine and without requiring access to many different and potentially expensive devices.

The rest of this paper is structured as follows. Section 2 provides a list of closely related work. Section 3 provides background on PTX and SASS, two instruction sets pertinent to NVIDIA GPUs and necessary to understand Tensor Cores. It also details the HMMA instruction that carries out the matrix operation $D = A \times B + C$. Then, in section 4, we formalize the numerical properties of tensor cores, compare our findings with previous works, and describe our resulting formal model. In section 6, we study two methods used to perform single-precision multiplications using the half-precision tensor cores discussed in Ootomo and Yokota [44]. We then analyze these algorithms, using SMT to try to prove that the error of Ootomo and Yokota's method is always better than Markidis et al. [37].

2 Related Work

We survey closely related work on floating-point formalization and testing-based specification discovery, followed by some non-floating-point formalization efforts.

An SMT theory for floating point reasoning was proposed by Rümmer and Wahl [47], which also included formalizations for rounding modes. However, SMT-based floating point reasoning has historically been found to have poor scalability [11, 48], but has been successfully used for error analysis [49]. Leeser et al. [30] demonstrated success in using SMT for floating-point reasoning, albeit using the theory of Reals. Brain et al. [9] redefined the floating point theory, substantially improving SMT's capabilities. Darulova et al. [10] used SMT to statically analyze floating point programs, for instance to compare roundoff errors between fixed-point and floating-point arithmetic. Floating point capabilities have similarly been implemented in other theorem provers, such as Coq [7] which has also been used for error analysis [1]. Each of these formalizations follows the IEEE standard [25] and hence do not contain support for the non-standard accumulator which our work provides. Titolo et al. [51] present an abstract interpretation framework for floating-point program roundoff error analysis.

Using tests to identify the implementation peculiarities of floating point units dates as far back as Karpinski [28]. In the case of GPU tensor cores, there has been considerable interest in understanding their functional as well as performance characteristics. Sun et al. [50] studied the tensor core implementations across various NVIDIA architectures. While they primarily focused on the throughput and latency, they briefly investigated the numerical behavior of tensor cores by studying the relative error for different floating point formats. Blanchard et al. [4] devised a framework to perform an error analysis of block fused multiply-add units. Their method incorporates the supported precision of the unit in its formulation, allowing it to support future units that may offer a different precision. Hickmann and Bradford [23] and Fasi et al. [16] studied tensor cores by using carefully constructed experiments to determine the hardware's behavior such as its rounding mode, precision, and support for subnormals. Xinyi et al. [33] employed similar techniques while further exploring the block-FMA feature and additional bits for tensor cores and AMD's matrix cores. Yan et al. [54] also studied the instruction-level details of the tensor cores, providing insights into how the matrix operation is performed on Turing, showing how the threads in a warp cooperate to compute the mma operation.

Formal descriptions of architectural components have been used to detect subtle correctness and security properties unrelated to floating-point arithmetic. The Check tools (TriCheck [53], CoatCheck [35], CCICheck [36], PipeCheck [34]), focus on memory consistency models and highlight the pitfalls resulting from under-specified ISA details. The CheckMate tool [52] uses model checking to automatically create exploits for cache side channels. Manual formalization of specifications is costly and this has led to work that seeks to automate the creation of formal ISA semantics. SAIL [2] and K [12] have been explicitly built for ISA specifications. For x86, Godefroid and Taly [17] leveraged SMT to find input examples, while Heule et al. [22] explored stratified synthesis. Using program synthesis has been explored to automatically formalize hardware specifications for memory consistency models [24, 38].

3 PTX and SASS Background

The NVIDIA GPUs we use in this work are commonly programmed in the CUDA programming language, a C++ dialect that supports explicit data parallelism and the ability to specify which functions run on the CPU and which run on the GPU. To use the tensor cores, CUDA provides library functions that are internally implemented using inline assembly in the *virtual* PTX instruction set architecture (ISA) [42]. PTX is a GPU independent ISA which resembles a compiler intermediate representation with features such as types, infinite registers, scoping, and so on. The physical ISA, commonly referred to as SASS [40], resembles a more traditional machine ISA and, unlike PTX, changes across GPU architectures. PTX is compiled to SASS using an architecture-specific assembler called `ptxas`. PTX provides forward compatibility with newer GPU architectures. If the GPU driver does not find the SASS for the current architecture in the executable, it will recompile the PTX in the executable at runtime to the architecture-appropriate SASS. While NVIDIA provides a PTX specification, it does not provide any information about SASS, prompting reverse engineering efforts [50, 54, 14, 21].

3.1 Tensor Cores and the `HMMA` Instruction

`HMMA` is the primary SASS instruction that interacts with the tensor cores [26]. Programmers usually use CUDA's Warp Matrix or `wmma` Functions [41, §7.24] to use the tensor cores. However, the cores can also be accessed directly using inline assembly by using the PTX `mma.m8n8k4` instruction. Examining with `cuobjdump` [40] the disassembly of SASS programs that use either of these methods confirms variants of the `HMMA` SASS instruction are used.

Across different architectures, the behavior of the tensor cores and its corresponding `HMMA` instruction changes. On Volta and Turing, the tensor cores are invoked via the `HMMA.884` instruction, while the Ampere architecture replaced this with `HMMA.16816`. Both instructions multiply two matrices, A and B, and add a third matrix, C, though the `884` variant operates on 4×4 matrices, while the `16816` variant operates on 8×8 matrices. In fact, this change highlights another portability concern: the `mma.m8n8k4` PTX instruction that previously invoked tensor cores no longer does so on the Ampere architecture. Instead, it produces a sequence of Fused Multiply-Add (`FMA`) instructions that use the device's slower floating point cores whose numerical properties are quite different from tensor cores.

For both the `884` and `16816` variants, the `HMMA` operation consists of three steps: 1) multiplying matrix A and B, 2) accumulating the products of A, B along with matrix C, and 3) rounding the final result. Each element in the resulting $N \times N$ matrix D is computed via the following equation:

$$D_{i,j} = A_{i,1} \cdot B_{1,j} + A_{i+1,1} \cdot B_{1,j+1} + \ldots + A_{N,1} \cdot B_{1,N} \tag{1}$$

Unlike most GPU instructions, where each thread's calculations are independent of other threads, the `HMMA` instruction requires all threads within the warp to cooperate to compute the result, and only one matrix multiplication is performed per warp per instruction. Prior work by Yan et al. [54], Fang et al. [14] has described how matrix elements are mapped to each participating threads' registers. The `HMMA` instruction supports both F16 and F32 types for elements of C and D which can be individually specified. A and B are always F16. Since Volta, tensor cores introduced have support for more formats: INT4 (4-bit integers) and INT8 (8-bit integers) in Turing, followed by Ampere's support for double-precision (FP64) and a custom format, Tensor Float 32 (TF32).

This work focuses on the FP16 and FP32 formats that are supported on all tensor cores, though the properties we establish can be adapted to study tensor core behavior for different formats. In the FP16 format, corresponding to IEEE754's binary16 format, 16 bits are used to represent the number. From most significant to least significant: the first bit represents the sign, the next 5 bits encode the exponent (with a bias of 15), and the remaining 10 bits encode the mantissa. Similarly, the FP32 format, corresponding to IEEE-754's binary32 format, 32 bits are used: the first bit encodes the sign, the next 8 encode the exponent (with a bias of 127), and the remaining 23 bits encode the mantissa.

4 Tensor Core Semantics

Although Equation 1 appears to be a sufficient description of how HMMA behaves, floating point cognoscenti will immediately inquire about the following details which are needed to build a sufficiently detailed formal model:

1. Are the multiplications and additions exact? What rounding mode is then used?
2. Since standard floating point addition is not associative, how does the computation differ when terms are rearranged?
3. Are the intermediate sums normalized, or only the final result?

NVIDIA detailed the architecture of their tensor cores in their whitepaper describing the Volta GPUs [39], though is missing this level of detail:

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

The PTX documentation is also unhelpful, stating that "The accumulation order, rounding, and handling of subnormal inputs is unspecified." [43, §9.7.13.3.5]. Previously, Fasi et al. [16] answered some of these questions for the Volta architecture using carefully reasoned empirical tests. Our goal, in contrast, is to provide a complete description of tensor core behavior as a formal model to not only answer such questions but also allow reasoning about other properties.

To establish a precise semantics for the HMMA instruction and its variants, we focus on the computation for a single element in the output matrix – we fix a single row of matrix A, a single column of matrix B, and the corresponding element in C. Here, we just use the first row and column, making the computation equivalent to equation 1 when $i = j = 1$. While this assumes that the same computation is done for each row in the result matrix, it is easy to verify this is the case by repeating the hardware evaluation for different elements in the final matrix.

The operations in equation 1 may be implemented in myriad ways impacting the final result. Constructing a model requires determining which choices were made in hardware. Doing this unavoidably requires some manual effort to model the hardware design space and inform the possible implementations that need to be evaluated. To construct our model, we scoured the literature for implementations of matrix multiplication and dot products. The specification that accumulation is done in FP32 enables us to eliminate several possibilities, namely techniques like Bohlender and Kulisch [5] for exact accumulation and the fixed-point approaches such as described by Boldo et al. [6]. Ultimately, our tensor core model is primarily built upon the existing work of Fasi et al. [16] and Hickmann and Bradford [23]. We revise their findings and offer models for Turing an Ampere with a framework that can quickly adapt to new architectures.

Like previous work, we use tests to discriminate between the different possibilities. However, unlike previous work, these discriminating tests are *automatically generated* using an automated theorem prover (e.g., cvc5 [3]). Essentially, we write formulae to capture the possible ways in which the implementation of the tensor core unit may behave *and ask the automated theorem prover to find input values that would yield different outputs based on the design choices under investigation.* Our framework consists of these formalizations, encoded in SMT, and produces inputs that are then provided to the hardware to probe their behavior. The queries are fully parametric, allowing them to be easily adjusted to probe different implementation possibilities.

Using SMT solvers to generate inputs this way is routine [45, 29], but we are not aware of prior work that utilizes them to elicit the latent numeric behavior of nonstandard floating point operations that are performed by GPU tensor cores. This approach is particularly well suited for the ill-documented tensor cores whose behavior continues to evolve with each new generation. New tests can easily be automatically generated as the underlying architecture changes. For instance, the tensor cores in Volta and Turing multiplied 4×4 matrices. Ampere shifted to 8×8 matrices, requiring new tests to explore its behavior.

Additionally, compared to prior work that has investigated these properties without using SMT, we contribute knowledge about corner-cases involving rounding-modes and the number of carry bits required due to lack of normalization.[4]

4.1 Precision

To demonstrate our approach, we begin by testing one of the claims made in the whitepaper: that the multiplication between the elements of A and B are performed in single-precision. We use SMT to identify values that are representable in FP16 but whose product is not. Then, we provide these values to the hardware and determine whether the correct result is reported.

Listing 1.1 demonstrates how we identify discriminating inputs for our tests using cvc5's Python API [3]. Lines 6 and 7 express the multiplication in half-precision and single-precision respectively, with the for loop iterating over each rounding mode. Line 8 asserts that the two results differ. Each of these lines adds a constraint on the values of a and b that must be met in order for the model to be satisfiable. In line 9, we ask the solver to find values for a and b which satisfy each of these conditions. A `sat` response indicates the solver has found such values. In line 10, `get_model` obtains these values from the solver, producing the values shown in Table 1. Once the solver has proved that the

[4] Recent growth in power of SMT-solvers to deal with floating-point queries was essential for this to be practical [8]; see Table 4 and surrounding discussions.

```
from cvc5.pythonic import *  # can also import Z3
s = Solver()
a = FreshConst(Float16())
b = FreshConst(Float16())
for rm in {RNE(), RTZ(), RTN(), RTP()}:
    FP16_result = fpToFP(RTZ(), fpMul(rm, a, b), Float32())
    FP32_result = fpMul(rm, fpToFP(RTZ(), a, Float32()), fpToFP(RTZ(), b,
     Float32())
    s.add(Not(FP16_result == FP32_result))
if s.check() == sat:
    m = s.get_model()
    # record values for a and b
    print(m.eval(a), m.eval(b))
else:
    print("Unsat/unknown")
```

Listing 1.1. The python script showing how to use cvc5's (or Z3's) python api to identify a pair of half-precision values whose product is not exact when the multiplication is performed in half-precision.

assertion holds, we can then extract the model in order to obtain inputs which can test if the tensor cores indeed perform the multiplication in single-precision.

Table 1. Values showing the multiplications are performed with full precision

a	$1.2587890625 \cdot 2^{-15}$
b	$1.3681640625 \cdot 2^{-1}$
Exact Result	$1.4162635803222656 \cdot 2^{-17}$
Result (half)	$1.1767578125 \cdot 2^{-15}$
Hardware Result	$1.4162635803222656 \cdot 2^{-17}$

Fasi et al. [16] presumably used manual analysis in order to identify inputs that could be used to test the numerical behavior of the tensor cores such as its rounding mode and support for subnormals. Here, we show that an automated theorem prover can be used to avoid this manual effort. In addition, our method found two discrepancies in the work by Fasi et al., which we elaborate on in detail in the following sections.

Table 2. A description of the notation and terms used in the properties

Symbol	**Meaning**
$a \cdot b$	Denotes multiplication of a and b in single-precision.
$a \oplus_{rm} b$	Denotes addition of a and b in single-precision. *rm*, when specified, denotes the rounding mode, defaulting to round-to-zero.
ToFP32(a)	Converts the FP16 input argument to its single-precision representation
ToFP16(rm, a)	Converts the FP32 input argument to half-precision, rounded with *rm*

Precision of Accumulation Tensor cores allow for the source and destination to hold values in either half precision or single precision. The whitepaper states that the accumulation is performed in single-precision. The PTX documentation states that for half precision inputs and outputs, the accumulation is performed with "at least half precision" [43, §9.7.13.4.14]. Here, we determine the actual precision used during accumulation for half-precision inputs and outputs by finding input values that satisfy the following:

Property 1 $a \cdot b \oplus c \cdot d \neq \mathit{ToFP32}(a) \cdot \mathit{ToFP32}(b) \oplus \mathit{ToFP32}(c) \cdot \mathit{ToFP32}(d)$ *Where* $a, b, c, d, a \cdot b \oplus c \cdot d \in \mathit{FP16}$, $a \cdot b, c \cdot d \notin \mathit{FP16}$

This states that certain FP16 inputs can yield products that cannot be represented in FP16, but whose sum can be. Our tests using inputs generated from the SMT solver confirm that the hardware performs accumulation in single-precision for half-precision inputs and outputs.

4.2 Rounding

Previously, Fasi et al. [16] determined that the intermediate results of the accumulation were rounded using round-to-zero, while Hickmann and Bradford [23] suggested that the results are truncated. Fasi et al. also determined that no additional bits were used for rounding. However, properly rounding RTZ requires additional guard bits. In a standard floating point addition algorithm, the terms are aligned so that they have the same exponent. This requires shifting the mantissa of the term with the smaller magnitude to the right in what is called the significand alignment step. IEEE-754 [25] round-to-zero requires that the result be equal to the largest magnitude no larger than the exact result.

Table 3. Round-to-zero demonstration

Input a	**Input c**	**Properly-rounded RTZ result**	**Tensor Core Result**
2^1	-2^{-40}	$2 - 2^{-23}$	2^1 ✗

Consider the example in table 3. If a and c are 2^1 and -2^{-40} (and b is one), then the result in round-to-zero mode should return the value $2 - 2^{-23}$. However, because the significance alignment step requires shifting the 2^{-40} term 41 bits to the right, all of its bits would be lost and the result would be 2^1.

To properly handle this, floating point adders often make use of a "sticky bit" that tracks whether any bits were lost during alignment [18, §2.1.4]. However, a sticky bit does not work when aligning more than 2 terms, as the lost bits might have had different magnitudes across terms. Instead, accurate RTZ rounding requires preserving the bits that would be lost during the alignment, which Fasi et al. concluded were not present in Volta.

Our tests for the rounding mode on hardware generate inputs that discriminate between each pair of rounding modes.[5] Evaluating the resulting inputs, we

[5] Available online at `https://pyxis-roc.github.io/tensor_core_semantics/`

```
HMMA.884.F16.F16.STEP0 R12, R32.ROW, R2.COL, R12 ;
HMMA.884.F16.F16.STEP1 R14, R32.ROW, R2.COL, R14 ;
ST.E.SYS [R4], R12 ; /* ... */
```

Listing 1.2. The SASS disassembly (Volta) for half-precision mma. Only the first store is shown.

find that tensor cores do not adhere to any of the IEEE-754 rounding modes, *including* RTZ rounding, contrary to the findings in Fasi et al. [16]. Instead, they truncate, ignoring all of the lower bits when computing the final result. On the Volta tensor cores, adding the aforementioned example, 2^1 and -2^{-40}, results in 2^1, which would be the result in round-to-nearest. While one may conclude that the rounding mode depends on the inputs, there is in fact a simpler explanation: during the significand alignment step, mantissa bits that were discarded during the shift are lost. This means that performing an effective subtraction with numbers having an exponent difference greater than the number of bits in the mantissa (23 in this case) is the same as adding zero. As mentioned before, this behavior violates the guarantees of round-to-zero which mandates the rounded result cannot be greater than the true result. This difference only manifests in effective subtraction, a subtlety that explains the mischaracterization while exposing the fragility of informal tests.

Tensor cores support outputting a FP16 result, requiring the FP32 accumulation to be rounded. We encode property 2 and generate tests that identify the rounding mode used in this case.

Property 2 $\mathit{ToFP16}\big(rm_1,\ \mathit{ToFP32}(a)\ \cdot\ \mathit{ToFP32}(b)\big)\ \neq$ $\mathit{ToFP16}\big(rm_2,\ \mathit{ToFP32}(a)\ \cdot\ \mathit{ToFP32}(b)\big)\qquad \textit{Where } a, b \in \mathit{FP16},\ rm_1 \neq rm_2$

In this experiment, we find a pair of values in FP16 whose product, when rounded to FP16, differs for different rounding modes. We limit ourselves to a single term, setting the rest of the values to 0 so as to avoid behavior that may be attributed to the rounding mode that is used for computing the partial sums. From the experiments, we conclude that the final rounding is performed in round-to-nearest. While the behavior is consistent with the findings in Fasi et al., there it was concluded that this rounding was done in software. Our experiments reveal that this is not the case. This is further evidenced upon examining the disassembly (listing 1.2) which shows no intervening instructions before the result is stored to memory.[6] This indicates that the rounding to FP16 is in fact handled by the tensor cores.

4.3 Accumulation Order & Normalization

IEEE-754 addition is not associative due to the rounding and normalization that occurs after each operation. Fasi et al. [16] determined that the accumulation for

[6] This disassembly is consistent across multiple different compiler versions — we tested nvcc versions 11.3 through 12.4

Volta is performed into the element with the largest magnitude, and that there is no normalization of intermediate sums. We evaluate property 3 and determine that the result does not depend on the order of the terms.

Property 3 $(a_1b_1 \;\oplus_{rtz}\; a_2b_2) \;\oplus_{rtz}\; a_3 \cdot b_3 \neq a_1 \cdot b_1 \;\oplus_{rtz}\; (a_2 \cdot b_2 \oplus_{rtz} a_3 \cdot b_3)$

Testing for normalization requires an implementation of a multi-term floating-point accumulator that could accumulate without normalizing intermediate sums. However, the floating-point operations provided by SMTLIB are IEEE-754 compliant, which means that the intermediate sums will always be normalized. To overcome this, we developed our own floating-point accumulator using bitvector operations. Our implementation takes into account each of the previous discoveries regarding tensor cores.

Property 4 $(a_1 \cdot b_1 \;\oplus\; a_2 \cdot b_2) \;\oplus\; c \neq$ *no-normalize-sum*$(a_1 \cdot b_1,\; a_2 \cdot b_2,\; c)$*. Where* $a_1, a_2, b_1, b_2, c > 0$ *and no-normalize-sum(x, y, z) sums x, y, and z without normalizing the intermediate results.*

To accumulate without normalization, the terms are first aligned to the maximal exponent before being accumulated. Terms are aligned by right shifting their mantissas according to the difference between their exponent and the maximal exponent. An implementation may choose whether or not to track some or all of the bits that were shifted out. We find that all shifted bits are discarded on Volta, while one is preserved on Turing and Ampere. Evaluating the inputs provided by the theorem prover for property 4 confirms that the tensor cores do not normalize intermediate sums.

Number of Carry-Out Bits Required Due to Lack of Normalization As Fasi et al. [16] noted, for accumulation of N-terms, $\lceil \log_2(N) \rceil$ extra carry-out bits are needed, meaning 5-terms require 3 bits of carry out. However, they were only able to find examples that required 2 carry-out bits. Finding inputs that show that 3 extra carry-out bits are needed (and are used by the actual tensor cores) in order to perform 5-term accumulation is a perfect task for SMT solvers. As our implementation is fully parametric, we can easily model the use of differing numbers of carry bits and then use SMT to find values where they differ. It takes cvc5 just over a minute to find inputs proving that Volta and Turing require 3 bits for carry-out, which we confirm by finding the hardware computes the correct result.

When we evaluate the values produced by the automated theorem prover, we find that the hardware reports the value that would be the result if 3 bits were indeed used. Additionally, we also use SMT to prove that no more than 3 bits are needed by proving that accumulation with 4 extra bits is equivalent to accumulation with 3 extra bits. We thus improve upon the findings of Fasi et al. [16] and find that the 5-term accumulator uses 3 extra bits for carry out. Ampere's 9-term accumulator should likewise require 4 extra bits, though attempts to find confirming inputs timed out after 6 hours.

4.4 Subnormals and Exceptional Values

Both Fasi et al. [16] and Hickmann and Bradford [23] examined support for subnormals, though the former found that subnormal FP32 inputs were supported while the latter found otherwise. To remedy this, we conduct our own experiments similar to Fasi et al., though once again using SMT, to determine the handling of subnormal inputs and outputs in both FP16 and FP32. The queries to evaluate this are straightforward: subnormal FP16 inputs can be tested by prompting SMT for a pair of subnormal FP16 inputs whose product is normal in FP16, support for subnormal outputs can likewise be tested by prompting SMT for a pair of normal inputs whose product is subnormal. Testing for subnormal outputs in FP32 cannot be done without also testing for subnormal inputs, though the test is straightforward: simply set C to a value that is subnormal in FP32. Our findings on the three generations of tensor cores agree with Fasi et al.: tensor cores fully support subnormal numbers for both inputs (of all matrices A, B, and C) and outputs (Matrix D); there is no automatic flush-to-zero.

Hickmann and Bradford [23] also examined how tensor cores handle exceptional values like NaN and infinity. Our findings align with theirs: tensor cores follow IEEE-754 standard for computations that should result in NaNs and infinities. Additionally, we probed the hardware to determine the bit patterns used for NaN outputs. For example, we exhaustively tested multiplying each NaN pattern with 1.0 and multiplying infinities of different signs, to determine support for NaN payloads. In every case, all NaN outputs yield a canonical NaN with the same bit pattern: a sign bit of 0, all exponent bits set, and the most significant bit of the mantissa set. We also determined that tensor cores do not output -0: all 0 outputs will be positive.

Algorithm 1 Dot-Product on Volta, Turing, and Ampere. (A: ...), (T:), and (TA: ...) correspond to Ampere, Turing, or both Ampere and Turing, respectively

```
Inputs: Two four(eight)-term FP16 vectors $a$ and $b$ and one FP16/FP32 scalar $c$
Output: One FP16/FP32 scalar
1: Pairwise-multiply $a$ and $b$, computing their exact results in FP32, to get $ab$
2: Collect $ab$ together with the FP32 representation of $c$ and find the largest exponent
2b: If any number is NaN, or there are infinities of different signs, then return NaN
2c: (TA: Add one bit of padding to the least significant bit of each term's mantissa)
3: Right shift each term's 24(TA: 25)-bit mantissa, discarding all excess bits
4: Add three(four) bits of padding to the most significant bit of each term's mantissa
5: Take the now 27(T: 28)(A: 29)-bit mantissas and accumulate them in any order
6: Normalize the result to FP32, shifting the mantissa and adjusting the exponent
7: Discard the upper 3(4) bits (and the lowest one bit)
8: Return the result in the requested precision, using round-to-nearest for FP16
```

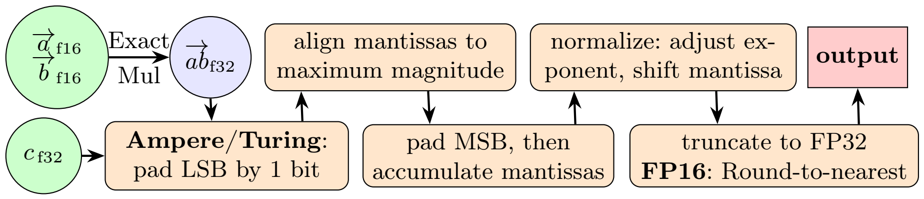
$\vec{a}_{f16}$
$\vec{b}_{f16}$
Exact Mul
$\vec{ab}_{f32}$
c_{f32}
Ampere/Turing: pad LSB by 1 bit
align mantissas to maximum magnitude
pad MSB, then accumulate mantissas
normalize: adjust exponent, shift mantissa
truncate to FP32 **FP16**: Round-to-nearest
output

Fig. 1. Behavior of Tensor Cores

Table 4. Timings for the queries sent to the solvers. In the last row, - means that the solvers were unable to find a result given a 6 hour timeout

Query	**Z3**	**cvc5**
Exact Multiplication	0.55s	0.027s
Exact Addition in FP16	27.59s	0.12s
Rounding of Final Result	0.52s	0.008s
Rounding of Accumulator	0.34s	0.017s
Accumulation Order	9.87s	0.25s
Normalization	9.63s	0.25s
Subnormal FP16 Inputs	0.117s	0.016s
Subnormal FP16 Outputs	0.682s	0.024s
3 Carry Bits	410.25s	72.793s
4 Carry Bits (Ampere)	-	-

5 Model and Results

Figure 1 outlines the behavior of the tensor cores' dot product step from equation 1, which is described in more detail in algorithm 1.[7] The inputs, (shown in green in the figure) are two FP16 vectors containing four (eight on Ampere) scalars, and an FP16 or FP32 scalar. Note that for the FP16 a and b values, the multiplication of their 11-bit mantissas can always be represented in FP32's 24 bit mantissa, meaning that no rounding is required and the results are always exact. Our findings show that the Volta and Turing cores differ in that Turing uses an extra lower bit during significand alignment. Ampere tensor cores differ in that the multiplication is now performed for 8×8 matrices. Additionally, while Volta and Turing discard all shifted bits during significand alignment, on Ampere, one shifted bit is preserved.

Table 4 demonstrates how the recent advancements in SMT's floating point capabilities have dramatically improved SMT's floating-point capabilities, resulting in impressive query times that in many cases respond in sub-second time. As expected, the more inputs that are required to prove the property, the longer it takes for the query. Queries for exact multiplication, addition and rounding each

[7] The full implementation, including the SMT models, is available online at `https://github.com/pyxis-roc/tensor_core_semantics`

require just two inputs [8], normalization and accumulation order require 3, and 3 carry bits requires 9 inputs. Proving Ampere's 9-term accumulator requires 4 carry bits requires finding 17 total inputs, a task which the current solvers were ultimately unable to finish. Each of the queries are tested on a machine running an AMD EPYC 7502P 32-core processor with 256GB RAM at 1.5GHz and given a 6 hour timeout. For hardware tests, we used a Titan V GPU for Volta, an RTX 2080Ti for Turing, and an RTX A6000 for Ampere. Kernels were compiled with CUDA version 11.8. The timing results in Table 4 show the query time, in seconds, using Z3 version 4.8.9 [13] and cvc5 version 1.0.2 [3].

6 Ootomo and Yokota Case Study

While tensor cores provide high-performance, their FP16 inputs mean they have low precision. Ingenious methods have therefore been developed to take advantage of tensor cores' performance without sacrificing precision. One such method was proposed by Markidis et al. [37], which introduced a residual matrix to record the loss of mantissa (the difference between FP32 and FP16 inputs) which can then be used to recover precision. A similar technique was used by Fasi et al. [15]. For a matrix product $A \cdot B$, the residual matrices R_A and R_B are calculated as the difference between the single-precision and half-precision representations of A and B as $R_A = A_{f32} - A_{f16}$ and $R_B = B_{f32} - B_{f16}$ respectively. The final recovered result is calculated using

$$A_{f32} \cdot B_{f32} = R_A R_B + A_{f16} R_B + R_A B_{f16} + A_{f16} B_{f16}$$

To further reduce the error, Ootomo and Yokota [44] (which we abbreviate to O-Y) improved Markidis et al. method by incorporating rounding to nearest in the accumulator by performing accumulation outside of the tensor core unit. Additionally, they implemented scaling when computing the residual matrix. The updated procedure is as follows:

$$\begin{aligned} R_A &= (A_{f32} - A_{f16}) \cdot 2^{11} \qquad R_B = (B_{f32} - B_{f16}) \cdot 2^{11} \\ A_{f32} B_{f32} &= \frac{R_A R_B}{2^{22}} + \frac{A_{f16} R_B + R_A B_{f16}}{2^{11}} + A_{f16} B_{f16} \\ D &= RN(A_{f32} B_{f32} + C) \end{aligned}$$

where 2^{11} is the scaling factor and RN denotes rounding to nearest.

The O-Y method is meant to improve the error correction of Markidis et al.'s at the cost of extra computation. Using our models from Section 4, we implement both error correction methods described by the paper in SMT. We then attempt to prove that the absolute accuracy for one of the final elements of the matrix, when using [44] method, can never be worse than [37]. To do this, we ask an SMT solver to prove the following formula:

[8] or four if querying for FP16 values. When possible, we asked the solver to first find FP32 values that demonstrated the property, and then asked for FP16 values whose product results in that value

Property 5 $\exists$ *inputs s.t.* $|Markidis(inputs) - actual(inputs)| < |Ootomo(inputs) - actual(inputs)|$

Where $Markidis$ and $Ootomo$ correspond to the result of one element in the final matrix computed using equation (6) and equation (24) from Ootomo and Yokota [44], respectively; *actual* corresponds to the result obtained by performing the dot product in double precision. We also restrict the exponent ranges for the inputs to 2^{-15} and 2^{14} as was done for O-Y's Type 1 experiments.

Table 5 shows the values for which O-Y's method has a higher error than Markidis et al.'s. For this single query, it takes cvc5 less than 5 minutes to find values for which the error using O-Y's method can be worse. This is not to say that O-Y's method is worse overall, but rather proves that it is not more accurate for every input. Nor does this contradict their empirical results showing that their method was more accurate in general. In Ootomo and Yokota [44], it was noted that one of the main contributors to the error was due to the round-to-zero mode of tensor cores. This means that for many cases, performing the accumulation outside of tensor cores can *improve* the accuracy of the final result. However, as we showed in section 4, tensor cores do not normalize the intermediate sums. This can improve the final accuracy by keeping parts of the mantissa that would have been lost during the normalization step thus making O-Y's method worse on some inputs.

Table 5. Inputs which show the the error of Ootomo and Yokota [44] **(O-Y)** can be greater than Markidis et al. [37] **(M)**

a	$1.0009765625 \cdot 2^{-8}$	$1.326171875 \cdot 2^{-14}$	2^{-12}	2^{-12}
b	$1.998046875 \cdot 2^{-7}$	$1.4443359375 \cdot 2^{-7}$	2^{-12}	2^{-12}
c	2^{-24}			
True	$1 + 2^{-23}$			
M	$1 + 2^{-23}$			
O-Y	1.0			

The values produced by our experiment in Table 5 follows the same pattern. To illustrate precisely why the error occurs, we walk through the example below.

1. $\boldsymbol{a}$ and $\boldsymbol{b}$ are multiplied; largest exponent of all terms is -1.
2. Each term is shifted to align their exponent to -1
3. $a_1 b_1$ and $a_2 b_2$ are accumulated, resulting in exactly $1 - 2^{-24}$. *This term is not normalized.*
4. 2^{-24} ($a_3 b_3$) is added to the previous term, resulting in exactly 1.0, represented internally as $2^{-1} \cdot 2^{1}$
5. 2^{-24} ($a_4 b_4$) is added to the previous term resulting in $1.0 + 2^{-24}$, represented as $2^{-1} \cdot 2^{1} + 2^{-23}$
6. At this point, O-Y's method diverges from Markidis et al.
   (a) 0.0 is added to $1.0 + 2^{-24}$
   (b) $1.0 + 2^{-24}$ is normalized, yielding 1.0, as the lowest bit is lost in the shift.

   (c) 1.0 is accumulated with 2^{-24} outside tensor cores. The result is 1.0.
7. In Markidis et al., c is accumulated *inside* the tensor cores
   (a) 2^{-24} (c) is added to $1.0+2^{-24}$, resulting in $1.0+2^{-23}$, represented internally as 2^{-1} for the exponent with $2^{1}+2^{-22}$ in the mantissa.
   (b) The term is normalized, yielding $1.0+2^{-23}$

This experiment demonstrates precisely how the lack of normalization inside tensor cores can lead to a result with less error. Fasi et al. [16, §D-2] also demonstrated how normalization contributes to error with an experiment in which the value $1-2^{-24}$ is accumulated with four values, each 2^{-24}. When partial sums are normalized, the accumulation between $1-2^{-24}$ and 2^{-24} would result in the value of 1. After being normalized, the exponent difference between the accumulated term and the remaining terms would cause the remaining additions to have no effect in round-to-zero, as their sums would be shifted out. Instead, when the intermediate sum is not normalized, none of the bits from the 2^{-24} terms are lost and the final error is only 2^{-24}.

7 Conclusions and Future Work

Using SMT, we formalized the properties of tensor cores and modeled their behavior across three generations. We showed how the in-progress specification and an automated theorem prover could be used together to resolve contradictory observations obtained using solely test-based methods. While most of our findings align with those of Fasi et al. [16], our model provided evidence that the rounding mode used for accumulation was simply truncation and Volta and Turing's 5-term accumulator used three extra carry-out bits. Once the model was built, we used it and an automated theorem prover to investigate two algorithms that utilize tensor cores and examine claims about their relative accuracies, demonstrating its usefulness of our model to algorithm designers.

The framework we established is fully parametric and future work can reuse it to study the properties of tensor cores as they evolve across generations. Preliminary experiments on Hopper GPUs (to which we lacked sufficient access for thorough study), for instance, indicate that even more bits may be preserved during significand alignment. Our model can also be adjusted to study different floating-point formats such as NVIDIA's 8-bit exponent, 10-bit mantissa TF32 format, or the two FP8 formats supported on Hopper. Given that future HPC hardware will likely be supported by non-standard hardware developed primarily for AI (including especially Tensor Cores) [46], formalizations such as ours can play a central role in supporting reliable scientific computing in the future. We plan to develop formal support to analyze such algorithms using techniques presented in this paper.

Acknowledgments.

We acknowledge the anonymous reviewers whose comments improved this work. This work is supported in part by NSF Awards 2403379, 2346394, 2217154, and 2124100.

Any opinions, findings and conclusions or recommendations expressed in this material are those of the author(s) and do not necessarily reflect the views of the National Science Foundation.

Disclosure of Interests.

The authors have no competing interests to declare that are relevant to the content of this article.

Bibliography

[1] Appel, A., Kellison, A.: VCFloat2: Floating-Point Error Analysis in Coq. In: Proceedings of the 13th ACM SIGPLAN International Conference on Certified Programs and Proofs, pp. 14–29 (2024), https://doi.org/10.1145/3636501.3636953

[2] Armstrong, A., Bauereiss, T., Campbell, B., Reid, A., Gray, K.E., Norton, R.M., Mundkur, P., Wassell, M., French, J., Pulte, C., Flur, S., Stark, I., Krishnaswami, N., Sewell, P.: ISA semantics for ARMv8-a, RISC-v, and CHERI-MIPS. Proceedings of the ACM on Programming Languages **3**(POPL), 1–31 (jan 2019), ISSN 2475-1421, 2475-1421, https://doi.org/10.1145/3290384

[3] Barbosa, H., Barrett, C.W., Brain, M., Kremer, G., Lachnitt, H., Mann, M., Mohamed, A., Mohamed, M., Niemetz, A., Nötzli, A., Ozdemir, A., Preiner, M., Reynolds, A., Sheng, Y., Tinelli, C., Zohar, Y.: cvc5: A Versatile and Industrial-Strength SMT Solver. In: Fisman, D., Rosu, G. (eds.) Tools and Algorithms for the Construction and Analysis of Systems - 28th International Conference, TACAS 2022, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2022, Munich, Germany, April 2-7, 2022, Proceedings, Part I, Lecture Notes in Computer Science, vol. 13243, pp. 415–442, Springer (2022), https://doi.org/10.1007/978-3-030-99524-9_24

[4] Blanchard, P., Higham, N.J., Lopez, F., Mary, T., Pranesh, S.: Mixed Precision Block Fused Multiply-Add: Error Analysis and Application to GPU Tensor Cores. SIAM Journal on Scientific Computing **42**(3), C124–C141 (2020), https://doi.org/10.1137/19M1289546

[5] Bohlender, G., Kulisch, U.: Comments on Fast and Exact Accumulation of Products. In: Jónasson, K. (ed.) Applied Parallel and Scientific Computing, pp. 148–156, Springer Berlin Heidelberg, Berlin, Heidelberg (2012), ISBN 978-3-642-28145-7, https://doi.org/10.1007/978-3-642-28145-7_15

[6] Boldo, S., Gallois-Wong, D., Hilaire, T.: A Correctly-Rounded Fixed-Point-Arithmetic Dot-Product Algorithm. In: ARITH 2020 - IEEE 27th Symposium on Computer Arithmetic, pp. 9–16, IEEE, Portland, United States (Jun 2020), https://doi.org/10.1109/ARITH48897.2020.00011

[7] Boldo, S., Melquiond, G.: Flocq: A Unified Library for Proving Floating-Point Algorithms in Coq. In: 2011 IEEE 20th Symposium on Computer Arithmetic, pp. 243–252 (2011), https://doi.org/10.1109/ARITH.2011.40

[8] Brain, M., Schanda, F., Sun, Y.: Building Better Bit-Blasting for Floating-Point Problems. In: Vojnar, T., Zhang, L. (eds.) Tools and Algorithms for the Construction and Analysis of Systems, pp. 79–98, Lecture Notes in Computer Science, Springer International Publishing, Cham (2019), ISBN 978-3-030-17462-0, https://doi.org/10.1007/978-3-030-17462-0_5

[9] Brain, M., Tinelli, C., Ruemmer, P., Wahl, T.: An Automatable Formal Semantics for IEEE-754 Floating-Point Arithmetic. In: 2015 IEEE 22nd Symposium on Computer Arithmetic, pp. 160–167 (2015), https://doi.org/10.1109/ARITH.2015.26

[10] Darulova, E., Izycheva, A., Nasir, F., Ritter, F., Becker, H., Bastian, R.: Daisy-Framework for Analysis and Optimization of Numerical Programs (Tool Paper). In: Tools and Algorithms for the Construction and Analysis of Systems: 24th International Conference, TACAS 2018, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2018, Thessaloniki, Greece, April 14-20, 2018, Proceedings, Part I 24, pp. 270–287, Springer (2018), https://doi.org/10.1007/978-3-319-89960-2_15

[11] Darulova, E., Kuncak, V.: Sound Compilation of Reals. In: Proceedings of the 41st ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, pp. 235–248 (2014), https://doi.org/10.1145/2535838.2535874

[12] Dasgupta, S., Park, D., Kasampalis, T., Adve, V.S., Roşu, G.: A Complete Formal Semantics of x86-64 User-level Instruction Set Architecture. In: Proceedings of the 40th ACM SIGPLAN Conference on Programming Language Design and Implementation, pp. 1133–1148, PLDI 2019, ACM, New York, NY, USA (2019), ISBN 978-1-4503-6712-7, https://doi.org/10.1145/3314221.3314601

[13] De Moura, L., Bjørner, N.: Z3: An Efficient SMT Solver. In: Proceedings of the Theory and practice of software, 14th international conference on Tools and algorithms for the construction and analysis of systems, pp. 337–340, TACAS'08/ETAPS'08, Springer-Verlag, Berlin, Heidelberg (mar 2008), ISBN 978-3-540-78799-0, https://doi.org/10.5555/1792734.1792766

[14] Fang, B., Hari, S.K.S., Tsai, T., Li, X., Gopalakrishnan, G., Laguna, I., Barker, K., Li, A.: Towards Precision-Aware Fault Tolerance Approaches for Mixed-Precision Applications. In: 2022 IEEE/ACM 12th Workshop on Fault Tolerance for HPC at eXtreme Scale (FTXS), pp. 47–52, IEEE (2022), https://doi.org/10.1109/FTXS56515.2022.00010

[15] Fasi, M., Higham, N.J., Lopez, F., Mary, T., Mikaitis, M.: Matrix Multiplication in Multiword Arithmetic: Error Analysis and Application to GPU Tensor Cores. SIAM Journal on Scientific Computing **45**(1), C1–C19 (2023), https://doi.org/10.1137/21M1465032

[16] Fasi, M., Higham, N.J., Mikaitis, M., Pranesh, S.: Numerical Behavior of NVIDIA Tensor Cores. PeerJ Computer Science **7**, e330 (2021), https://doi.org/10.7717/peerj-cs.330

[17] Godefroid, P., Taly, A.: Automated Synthesis of Symbolic Instruction Encodings from I/O Samples. SIGPLAN Not. **47**(6), 441–452 (jun 2012), ISSN 0362-1340, https://doi.org/10.1145/2345156.2254116

[18] Goldberg, D.: What Every Computer Scientist Should Know About Floating-Point Arithmetic. ACM Comput. Surv. **23**(1), 5–48 (mar 1991), ISSN 0360-0300, https://doi.org/10.1145/103162.103163

[19] Goodloe, A.E., Muñoz, C., Kirchner, F., Correnson, L.: Verification of Numerical Programs: From Real Numbers to Floating Point Numbers. In: Brat, G., Rungta, N., Venet, A. (eds.) NASA Formal Methods, p. 441–446, Springer Berlin Heidelberg, Berlin, Heidelberg (2013), ISBN 978-3-642-38088-4, https://doi.org/10.1007/978-3-642-38088-4_31

[20] Haidar, A., Bayraktar, H., Tomov, S., Dongarra, J., Higham, N.J.: 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), https://doi.org/10.1098/rspa.2020.0110

[21] Hayes, A.B., Hua, F., Huang, J., Chen, Y., Zhang, E.Z.: Decoding CUDA Binary. In: Proceedings of the 2019 IEEE/ACM International Symposium on Code Generation and Optimization, p. 229–241, CGO 2019, IEEE Press (2019), ISBN 9781728114361, https://doi.org/10.1109/CGO.2019.8661186

[22] Heule, S., Schkufza, E., Sharma, R., Aiken, A.: Stratified Synthesis: Automatically Learning the x86-64 Instruction Set. In: Proceedings of the 37th ACM SIGPLAN Conference on Programming Language Design and Implementation, pp. 237–250, PLDI '16, ACM, New York, NY, USA (2016), ISBN 978-1-4503-4261-2, https://doi.org/10.1145/2908080.2908121, event-place: Santa Barbara, CA, USA

[23] Hickmann, B.J., Bradford, D.: Experimental Analysis of Matrix Multiplication Functional Units. 2019 IEEE 26th Symposium on Computer Arithmetic (ARITH) pp. 116–119 (2019), https://doi.org/10.1109/ARITH.2019.00031

[24] Hsiao, Y., Mulligan, D.P., Nikoleris, N., Petri, G., Trippel, C.: Synthesizing Formal Models of Hardware from RTL for Efficient Verification of Memory Model Implementations. In: MICRO-54: 54th Annual IEEE/ACM International Symposium on Microarchitecture, p. 679–694, MICRO '21, Association for Computing Machinery, New York, NY, USA (2021), ISBN 9781450385572, https://doi.org/10.1145/3466752.3480087

[25] IEEE: IEEE Standard for Floating-Point Arithmetic. IEEE Std 754-2008 pp. 1–70 (2008), https://doi.org/10.1109/IEEESTD.2008.4610935

[26] Jia, Z., Maggioni, M., Staiger, B., Scarpazza, D.P.: Dissecting the NVIDIA Volta GPU Architecture via Microbenchmarking (2018), https://doi.org/10.48550/arXiv.1804.06826

[27] Jouppi, N.: Google Supercharges Machine Learning Tasks with TPU Custom Chip (may 2016), URL https://cloud.google.com/blog/products/ai-machine-learning/google-supercharges-machine-learning-tasks-with-custom-chip

[28] Karpinski, R.: Paranoia: A Floating-Point Benchmark. Byte Magazine **10**(2), 223–235 (Feb 1985), URL https://www.netlib.org/paranoia/

[29] Kim, B., Masuda, T., Shiraishi, S.: Test Specification and Generation for Connected and Autonomous Vehicle in Virtual Environments. ACM Transactions on Cyber-Physical Systems **4**(1), 1–26 (2019), https://doi.org/10.1007/978-3-642-20398-5_22

[30] Leeser, M., Mukherjee, S., Ramachandran, J., Wahl, T.: Make it Real: Effective Floating-Point Reasoning via Exact Arithmetic. In: 2014 Design, Automation & Test in Europe Conference & Exhibition (DATE), pp. 1–4, IEEE (2014), https://doi.org/10.7873/DATE.2014.130

[31] Li, B.: tcfft. https://github.com/rox906/tcFFT (2024), accessed: 2024-12-17

[32] Li, B., Cheng, S., Lin, J.: tcfft: A fast half-precision fft library for nvidia tensor cores. In: 2021 IEEE International Conference on Cluster Computing (CLUSTER), pp. 1–11 (2021), https://doi.org/10.1109/Cluster48925.2021.00035

[33] Li, X., Li, A., Fang, B., Swirydowicz, K., Laguna, I., Gopalakrishnan, G.: FTTN: Feature-Targeted Testing for Numerical Properties of nvidia & AMD Matrix Accelerators. In: 2024 IEEE/ACM 24th International Symposium on Cluster, Cloud and Internet Computing (CCGrid), IEEE (2024), https://doi.org/10.1109/CCGrid59990.2024.00014

[34] Lustig, D., Pellauer, M., Martonosi, M.: PipeCheck: Specifying and Verifying Microarchitectural Enforcement of Memory Consistency Models. In: 2014 47th Annual IEEE/ACM International Symposium on Microarchitecture, pp. 635–646, IEEE, Cambridge (dec 2014), ISBN 978-1-4799-6998-2, https://doi.org/10.1109/MICRO.2014.38

[35] Lustig, D., Sethi, G., Martonosi, M., Bhattacharjee, A.: COATCheck: Verifying Memory Ordering at the Hardware-OS Interface. In: Proceedings of the Twenty-First International Conference on Architectural Support for Programming Languages and Operating Systems, pp. 233–247, ACM, Atlanta Georgia USA (mar 2016), ISBN 978-1-4503-4091-5, https://doi.org/10.1145/2872362.2872399

[36] Manerkar, Y.A., Lustig, D., Pellauer, M., Martonosi, M.: CCICheck: Using μhb Graphs to Verify the Coherence-Consistency Interface. In: Proceedings of the 48th International Symposium on Microarchitecture, pp. 26–37, ACM, Waikiki Hawaii (dec 2015), ISBN 978-1-4503-4034-2, https://doi.org/10.1145/2830772.2830782

[37] Markidis, S., Der Chien, S.W., Laure, E., Peng, I.B., Vetter, J.S.: Nvidia Tensor Core Programmability, Performance & Precision. In: 2018 IEEE International Parallel and Distributed Processing Symposium Workshops (IPDPSW), pp. 522–531, IEEE (2018), https://doi.org/10.1109/ipdpsw.2018.00091

[38] Norman, C., Godbole, A., Manerkar, Y.A.: PipeSynth: Automated Synthesis of Microarchitectural Axioms for Memory Consistency. In: Proceedings of the 28th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 3, p. 513–527, ASPLOS 2023, Association for Computing Machinery, New York, NY, USA (2023), ISBN 9781450399180, https://doi.org/10.1145/3582016.3582056

[39] NVIDIA: NVIDIA Tesla V100 GPU Architecture. whitepaper WP-08608-001_v1.1, NVIDIA Corporation (Aug 2017), URL

https://images.nvidia.com/content/volta-architecture/pdf/volta-architecture-whitepaper.pdf

[40] NVIDIA Corporation: cuda-binary-utilities (Mar 2024), URL https://docs.nvidia.com/cuda/archive/12.4.0/pdf/CUDA_Binary_Utilities.pdf

[41] NVIDIA Corporation: Cuda C++ Programming Guide (Mar 2024), URL https://docs.nvidia.com/cuda/archive/12.4.0/pdf/CUDA_C_Programming_Guide.pdf

[42] NVIDIA Corporation: Inline PTX Assembly in CUDA (Mar 2024), URL https://docs.nvidia.com/cuda/archive/12.4.0/pdf/Inline_PTX_Assembly.pdf

[43] NVIDIA Corporation: Parallel Thread Execution ISA Version 8.4 (Mar 2024), URL https://docs.nvidia.com/cuda/archive/12.4.0/pdf/ptx_isa_8.4.pdf

[44] Ootomo, H., Yokota, R.: Recovering Single Precision Accuracy from Tensor Cores While Surpassing the FP32 Theoretical Peak Performance. The International Journal of High Performance Computing Applications **36**(4), 475–491 (2022), https://doi.org/10.1177/10943420221090256

[45] Peleska, J., Vorobev, E., Lapschies, F.: Automated Test Case Generation with SMT-solving and Abstract Interpretation. In: NASA Formal Methods: Third International Symposium, NFM 2011, Pasadena, CA, USA, April 18-20, 2011. Proceedings 3, pp. 298–312, Springer (2011), https://doi.org/10.1007/978-3-642-20398-5_22

[46] Reed, D., Gannon, D., Dongarra, J.: HPC Forecast: Cloudy and Uncertain. Communications of the ACM **66**(2), 82–90 (2023)

[47] Rümmer, P., Wahl, T.: An SMT-LIB Theory of Binary Floating-Point Arithmetic. In: International Workshop on Satisfiability Modulo Theories (SMT), vol. 151 (2010)

[48] Schkufza, E., Sharma, R., Aiken, A.: Stochastic Optimization of Floating-Point Programs with Tunable Precision. ACM SIGPLAN Notices **49**(6), 53–64 (2014), https://doi.org/10.1145/2666356.2594302

[49] Solovyev, A., Baranowski, M.S., Briggs, I., Jacobsen, C., Rakamarić, Z., Gopalakrishnan, G.: Rigorous Estimation of Floating-Point Round-off Errors with Symbolic Taylor Expansions. ACM Transactions on Programming Languages and Systems (TOPLAS) **41**(1), 1–39 (2018), https://doi.org/10.1145/3230733

[50] Sun, W., Li, A., Geng, T., Stuijk, S., Corporaal, H.: Dissecting Tensor Cores via Microbenchmarks: Latency, Throughput and Numeric Behaviors. IEEE Transactions on Parallel and Distributed Systems **34**(1), 246–261 (2022), https://doi.org/10.1109/TPDS.2022.3217824

[51] Titolo, L., Feliú, M.A., Moscato, M., Muñoz, C.A.: An Abstract Interpretation Framework for the Round-Off Error Analysis of Floating-Point Programs. In: Dillig, I., Palsberg, J. (eds.) Verification, Model Checking, and Abstract Interpretation, pp. 516–537, Springer International Publishing, Cham (2018), ISBN 978-3-319-73721-8

[52] Trippel, C., Lustig, D., Martonosi, M.: CheckMate: Automated Synthesis of Hardware Exploits and Security Litmus Tests. In: 2018 51st Annual IEEE/ACM International Symposium on Microarchitecture (MICRO), pp. 947–960, IEEE, Fukuoka (oct 2018), ISBN 978-1-5386-6240-3, https://doi.org/10.1109/MICRO.2018.00081

[53] Trippel, C., Manerkar, Y.A., Lustig, D., Pellauer, M., Martonosi, M.: TriCheck: Memory Model Verification at the Trisection of Software, Hardware, and ISA. SIGARCH Comput. Archit. News **45**(1), 119–133 (apr 2017), ISSN 0163-5964, https://doi.org/10.1145/3093337.3037719

[54] Yan, D., Wang, W., Chu, X.: Demystifying Tensor Cores to Optimize Half-Precision Matrix Multiply. In: 2020 IEEE International Parallel and Distributed Processing Symposium (IPDPS), pp. 634–643 (2020), https://doi.org/10.1109/IPDPS47924.2020.00071