HAWKEYE: REPRODUCING GPU-LEVEL NON-DETERMINISM

Erez Badash [*1] **Dan Boneh** [*2] **Ilan Komargodski** [*1] **Megha Srivastava** [*2]

ABSTRACT

We present Hawkeye, a system for analyzing and reproducing GPU-level arithmetic operations. Using our framework, anyone can re-execute on a CPU the exact matrix multiplication operations underlying a machine learning model training or inference workflow that was executed on an NVIDIA GPU, without any precision loss. This is in stark contrast to prior approaches to verifiable machine learning, which either introduce significant computation overhead to the original model owner, or suffer from non-robustness and quality degradation. The main technical contribution of Hawkeye is a systematic sequence of carefully crafted tests that study rounding direction, subnormal number handling, and order of (non-associative) accumulation during matrix multiplication on NVIDIA's Tensor Cores. We test and evaluate our framework on multiple NVIDIA GPU architectures (Ampere, Hopper, and Lovelace) and precision types (FP16, BFP16, FP8). In all test cases, Hawkeye enables perfect reproduction of matrix multiplication on a CPU, paving the way for efficient and trustworthy third-party auditing of ML model training and inference.

1 INTRODUCTION

Modern machine learning (ML), including both model training and inference, has become increasingly compute-intensive, driven by the rapid growth in number of model parameters, dataset size, and architectural complexity (Kaplan et al., 2020; Hoffmann et al., 2022). This is especially true with large language models, and the training and inference of state-of-the-art models now often requires massive compute clusters and careful orchestration of distributed systems (Shoeybi et al., 2019; Rajbhandari et al., 2020). To meet these growing demands, a new class of training and inference platforms has emerged, offering ML-as-a-service to offload the heavy computational burden from end users. For instance, cloud-based solutions like AWS SageMaker (Services, 2025), Google Vertex AI (Cloud, 2025), and Azure ML (Microsoft, 2024) provide full-stack infrastructure for model training, tuning, and deployment, while companies such as Replicate (Replicate, 2025) and TogetherAI (AI, 2025) provide API endpoints to support clients who lack the resources to train a model themselves.

However, these services require clients to trust them to train or run inference correctly, without introducing any backdoors, shortcuts (e.g. training for a reduced number of steps), or other runtime modifications to the provided models. How can we guarantee that the service provider executed the task as specified, and hold them accountable? This task, referred to as **verifiable ML**, seeks to design ML training and inference platforms with the additional guarantee that all tasks are executed correctly (Peng et al., 2025; Sun et al., 2024; Ganescu and Passerat-Palmbach, 2024; Lycklama et al., 2024; Srivastava et al., 2024; So et al., 2024).

There has been a long history in the cryptography and security literature on verifiable computation, where the typical setting considers a powerful server that performs a heavy computation for a computationally weak verifier (Shamir, 1992; Kilian, 1992; Goldwasser et al., 2008; Gennaro et al., 2013; Parno et al., 2013; Groth, 2016; Maller et al., 2019; Gabizon et al., 2019; Chiesa et al., 2020a; Ben-Sasson et al., 2018a; 2019; Chiesa et al., 2020b; Setty, 2020; Wüst et al., 2018; Xie et al., 2019; Kate et al., 2010; Ben-Sasson et al., 2018b; Bünz et al., 2018). These techniques guarantee that the prover executes the prescribed computation, without requiring the client to fully re-execute it. However, a prerequisite of these works is that the computation performed by the server is deterministic, which is not true for modern ML workflows (Hutson, 2018).

Typical ML training and inference tasks heavily depend on software and hardware implementations, which oftentimes are not fully known (see Section 2 for details). A primary source of non-determinism is in the accumulation process within specialized hardware units, such as the widely used NVIDIA Tensor Cores.[1] Non-determinism arises from many

[*]Equal contribution [1]Pearl Research Labs [2]Department of Computer Science, Stanford University. Correspondence to: Erez Badash <erez.badash@pearlresearch.ai>.

Proceedings of the 9th MLSys Conference, Bellevue, WA, USA, 2026. Copyright 2026 by the author(s).

[1]Tensor Cores are specialized hardware units that perform highly optimized matrix multiplications.

unspecified details including rounding strategy, subnormal numbers, and the order of accumulation during matrix multiplication operations (as floating point arithmetic is not associative) (Shanmugavelu et al., 2024). Thus, the same model provided the same input and random seed on different GPUs could result with different inference outputs (see (Srivastava et al., 2024) for an example).

Motivation. Suppose an auditor wishes to *re-execute* a specific computation on a CPU in order to verify correctness. For example, an auditor may wish to verify that a service provider hosting a language model is indeed providing the correct next-token output during inference, with respect to a particular sampling method. In an idealized world of purely deterministic computation, such a re-execution would either match the outputs exactly, or reveal discrepancies attributable to incorrect execution (e.g., a service provider wishing to cut costs by serving a smaller language model). However, in practice, running the same computation on different hardware, or even on the same hardware with different low-level execution schedules, can produce different numerical results.

Consider the following concrete simplified FP16 vector-vector multiplication $A \cdot B^T$ (which can easily be generalized to matrix-matrix multiplication):

$$A = \begin{bmatrix} 65504 & 1 & -65504 & 1 \end{bmatrix},$$
$$B = \begin{bmatrix} 1 & 0.001 & 1 & 0.001 \end{bmatrix}.$$

On Tensor Cores of NVIDIA's L40S GPU (Ada Lovelace architecture), the result is 0, whereas on NVIDIA's A100 GPU (Ampere architecture), the result is 0.0020. Such small discrepancies can accumulate over the course of an entire repeated inference or training run, resulting in different model weights and downstream predictive behavior. Additional examples on the effect of non-associativity on the stability of computations, or downstream performance of models, can be found in (Shanmugavelu et al., 2024; Srivastava et al., 2024).

As a result, simply comparing outputs between the original execution and an auditor's re-execution is insufficient for verification: even a perfectly correct computation may not result in a bitwise match. To *provably* avoid ambiguity in ML computations, one systematic approach is to disable non-deterministic hardware features. This guarantees reproducibility, but often incurs significant slowdowns (Arun et al., 2025). An alternative approach (Srivastava et al., 2024) for verifiable training proposes rounding intermediate computations to eliminate the accumulation of errors, but results in high storage cost for the model provider who stores rounding decisions.

Other works (e.g., (Ong et al., 2025)) take a heuristic approach, asserting that the numerical discrepancies between different hardware backends in typical ML workloads are "small enough" to be ignored. While such heuristics may hold in many practical scenarios, they lack formal guarantees, and it has not been shown whether they remain valid in adversarial or edge-case settings. For instance, a malicious service provider might deliberately exploit non-determinism to introduce imperceptible but harmful changes. Therefore, developing a principled framework for *verifiable and reproducible ML* that is robust to hardware-induced non-determinism remains a fundamental and urgent research challenge.

Our work. Our main contribution is designing Hawkeye, a platform for **accurately reproducing matrix multiplication**, the fundamental operation behind training and inference, across hardware types without modifying the original GPU kernels. Our only requirement is knowing which GPU was used for the execution. We show that, despite architectural disparities, it is possible to replicate the exact bit operations of NVIDIA's GPU Tensor Cores on CPUs. Such a capability unlocks new verification workflows where an offline reference CPU run serves as an *oracle* against which arbitrary GPU executions can be checked at essentially no overhead. We test our framework across different GPU architectures and datatypes, showing a 100% success rate in replicating large (*4096 x 4096*) matrix multiplication. Source code for Hawkeye is provided at `https://github.com/badasherez/gpu-simulator`.

2 The Challenge: Reproducing NNs

At a very high level, the workflow of both training or running inference with a neural network (NN) consists primarily of a sequence of matrix multiplication (MatMul) operations interleaved with nonlinear element-wise functions, such as activation functions.

GPU architecture. Single-Instruction Multiple-Thread (SIMT) cores, often referred to as CUDA cores, serve as versatile computing units capable of executing a broad spectrum of instructions, such as integer arithmetic, floating-point calculations, and load/store operations. These cores process scalar or vector instructions over individual or grouped data elements. In contrast, Tensor Cores (NVIDIA, 2022; 2020) are purpose-built hardware units optimized for high-throughput matrix multiplication. For example, the Tensor Cores on NVIDIA's A100 and H100 GPUs deliver at least $10\times$ improved performance for particular calculations over SIMT cores. Importantly, Tensor Cores operate on a coarser granularity: a single `mma` instruction can multiply two 16×16 matrices in one instruction.

Sources of non-determinism. Two main sources of non-determinism arise within this workflow due to the nature of

floating-point arithmetic:

- **Software-based non-determinism.** Floating-point operations performed in different orders across software implementations can yield varying numerical results, despite mathematical equivalence. This issue can be resolved by explicitly enforcing a consistent operation ordering, such as different algorithms for the convolution operation or batching (Heumos et al., 2023; PyTorch Contributors, 2026; Thinking Machines Lab, 2025).
- **Hardware-based non-determinism** Different hardware architectures can produce divergent floating-point results for the same computations due to hardware design choices. Libraries such as (pyxis-roc, 2021) accurately reproduce GPU element-wise functions on CPUs, but more complex operations such as matrix multiplication remain unresolved.

Problem statement. Given a specific GPU architecture employing Tensor Cores for matrix multiplication, we aim to replicate these Tensor Core computations precisely on CPU architectures, achieving bit-exact equivalence for every individual output. In particular, we seek to emulate the Tensor Core operation that updates an FP32 accumulator tile C with the product of two input tiles A and B of dimensions 16×16 (in FP16, BF16, or FP8 precision), following

$$C_{t+1} = C_t + A_t B_t.$$

Achieving bit-exact equivalence is challenging because Tensor Core computations are implemented through hardware-specific mixed-precision pipelines whose numerical behavior is only partially exposed through the programming model. While individual floating-point operations follow IEEE 754 semantics, Tensor Cores use implementation-specific accumulation structures and internal intermediate representations. As a result, the effective accumulation order and rounding points differ from those produced by a straightforward CPU implementation, making faithful software emulation of Tensor Core arithmetic non-trivial.

3 Floating-Point Preliminaries

Floating-point (FP) numbers provide a method to represent real numbers using finite precision, characterized by three components: a sign bit (s), exponent bits (e), and mantissa (fractional) bits (m). According to the IEEE 754 standard (Kahan, 1996), a FP number x is represented as:

$$x = (-1)^{\mathsf{sign}} \cdot 2^{\mathsf{exponent}-\mathsf{bias}} \cdot \mathsf{mantissa}$$
$$\mathsf{bias} = 2^{len(\mathsf{exponent})-1} - 1.$$

Additionally, the mantissa includes an implicit leading bit, typically 1 for normalized numbers and 0 for subnormal numbers.

Floating-point data types differ in their allocation of bits, and with Hawkeye we focus on the following types:

- **FP32 (Single precision)**: 1 sign bit, 8 exponent bits, and 23 mantissa bits. An FP32 number can represent values in the range of approximately $\pm 3.4 \times 10^{38}$.
- **FP16 (Half precision)**: 1 sign bit, 5 exponent bits, and 10 mantissa bits. An FP16 number can represent numbers in the range $\pm 65,504$.
- **BF16 (Brain floating-point)**: 1 sign bit, 8 exponent bits, and 7 mantissa bits; exponent bias = 127. A BF16 number shares the same exponent range as FP32 ($\pm 3.4 \times 10^{38}$), prioritizing dynamic range over precision.
- **FP8 (E4M3 format)**: 1 sign bit, 4 exponent bits, and 3 mantissa bits; exponent bias = 7. An FP8 number can represent values roughly in the range ± 448.

Finally, there exists exceptional bit patterns for floating point representations, including subnormal numbers (values closer to zero), infinity (all exponent bits are ones and the mantissa is zero), and NaN (all exponent bits are ones and the mantissa is non-zero).

3.1 Multiplication and Addition of FPs

The IEEE 754 standard defines how floating-point numbers are represented and manipulated in binary systems. It includes rules for operations like addition and multiplication to ensure consistency across different computing platforms.

We describe the algorithms for floating-point multiplication and addition in Algorithm 1 and Algorithm 2, respectively. For clarity, the algorithms below omit special cases (e.g., NaN, infinity, subnormal numbers, overflow, and underflow), but present the core steps for normalized FP numbers.

Algorithm 1 IEEE 754 Floating-Point Multiplication

Require: Floating-point inputs (s_a, e_a, m_a) and (s_b, e_b, m_b)
Ensure: Resulting floating-point value $(s_{\text{result}}, e_{\text{result}}, m_{\text{result}})$
1: $s_{\text{result}} \leftarrow s_a \oplus s_b$
2: $e_{\text{result}} \leftarrow e_a + e_b - \text{bias}$
3: $m_{\text{result}} \leftarrow (1 + m_a) \times (1 + m_b)$
4: **if** $m_{\text{result}} \geq 2$ **then**
5: $\quad m_{\text{result}} \leftarrow m_{\text{result}} >> 1$
6: $\quad e_{\text{result}} \leftarrow e_{\text{result}} + 1$
7: **end if**

8: Round m_{result} to the nearest representable number in the target FP data type, using tie-breaking towards even.

Algorithm 2 IEEE 754 Floating-Point Addition

Require: Floating-point inputs (s_a, e_a, m_a) and (s_b, e_b, m_b)
Ensure: Resulting floating-point value $(s_{\text{result}}, e_{\text{result}}, m_{\text{result}})$
1: $e_{\max} \leftarrow \max(e_a, e_b)$
2: Align mantissas:

$$m'_a \leftarrow (1+m_a)\times 2^{(e_a-e_{\max})}, \quad m'_b \leftarrow (1+m_b)\times 2^{(e_b-e_{\max})}$$

3: Perform addition or subtraction based on signs:

$$m_{\text{result}} \leftarrow (-1)^{s_a} m'_a + (-1)^{s_b} m'_b$$

4: $s_{\text{result}} \leftarrow \text{sign}(m_{\text{result}})$
5: $m_{\text{result}} \leftarrow |m_{\text{result}}|$
6: $e_{\text{result}} \leftarrow e_{\max}$
7: Normalize m_{result}:

- If $m_{\text{result}} \geq 2$, shift m_{result} right until $1 \leq m_{\text{result}} < 2$, incrementing e_{result} by the shift amount.
- If $m_{\text{result}} < 1$, shift m_{result} left until $1 \leq m_{\text{result}} < 2$, decrementing e_{result} by the shift amount.

8: Remove implicit leading bit from m_{result} (retain fractional part only)
9: Round m_{result} to the nearest representable number in the target FP data type, using tie-breaking towards even.

Non-associativity. Floating-point arithmetic is sensitive to the order of operations due to rounding. Although IEEE 754 specifies the exact result of each individual floating-point operation, different evaluation orders introduce different intermediate rounding steps, which can lead to different final results. For example, consider three FP16 values $a = 65504$, $b = -65504$, and $c = 1$. Evaluating $(a + b) + c$ yields 1, since $a + b = 0$ exactly and $0 + 1 = 1$. However, evaluating $a + (b + c)$ yields 0: the sum $b + c = -65503$ cannot be represented in FP16 and rounds to -65504, after which $a + (-65504) = 0$.

4 Overview of Hawkeye

The goal of Hawkeye is to reproduce the bit-exact result of matrix multiplication with Tensor Cores on CPUs, in order to enable verifiable and reproducible machine learning. To this end, we developed a systematic testing methodology aimed at understanding and reproducing the internal numerical behavior of GPU Tensor Cores.

Tensor Core Characterization Methodology. To characterize the numerical behavior of Tensor Core matrix multiply-accumulate operations, we implement custom CUDA kernels that directly invoke the hardware MMA instructions via inline PTX assembly. For BF16 and FP16 on NVIDIA GPUs, the PTX `wmma.mma_sync` instruction compiles to a single HMMA SASS instruction. Using these kernels, we execute a series of targeted tests designed to isolate individual properties of the internal accumulation pipeline. These targeted tests are performed over the multiplication of three tiles, A, B, and C (the accumulator), of dimensions 16 x 16 each, which we select to test certain properties. Finally, we then encode these characterization results into a software simulator that reproduces the discovered accumulation semantics.

Target Tests Hawkeye consists of a suite of targeted tests, each designed to characterize specific aspects of matrix multiply-accumulate (`mma`) operations on NVIDIA Tensor Cores. The same set of tests allows us to recover the exact sequence of operations across different GPU architectures and data types. While we expect the framework to generalize to additional architectures and precisions, our study focuses on three GPU architectures (Ampere, Hopper, and Ada Lovelace) and three precision formats (FP16, BF16, and FP8).

- **Summation Dependency and Order Test**: This test determines the precise sequence in which summations are executed during the accumulation phase.
- **Internal Precision Detection Test**: This test identifies the exact internal numeric precision employed during each individual summation step within the Tensor Core computations.
- **Rounding Mode Detection Test**: This test detects and characterizes the rounding mode applied for each arithmetic operation, enabling accurate replication of GPU rounding behavior.
- **Normalization Stage Detection Test**: This test recovers at which specific stages the intermediate computational states are normalized into standard floating-point representations.
- **Subnormal Behavior Detection Test**: This test characterizes the handling and representation of subnormal floating-point numbers throughout the Tensor Core computation pipeline.

By combining the results of these tests, we can reconstruct a computational model that accurately reproduces the Tensor Core arithmetic pipeline, ensuring bit-for-bit equivalence across every matrix multiplication output. In the following

sections, we walk through the test outputs for reproducing multiplications of FP16 on Ampere (Section 5), FP16 on Hopper (Section 6), and BFP16 on Ampere (Section 7). We found that Lovelace followed the same architecture structures as Ampere, and present empirical results confirming successful bit-wise replication over hundreds of thousands of randomly generated 16×16 tiles in Section 8.

5 REPRODUCING FP16 WORKFLOW ON AMPERE

We now go through in detail how Hawkeye can enable replication of FP16 matrix multiplication on Ampere Tensor Cores. However, before applying our sequence of targeted tests, we will first verify that that a single multiplication of two FP16 numbers does retain full precision.

5.1 Full-Precision FP16 Products Verification

We first want to understand whether multiplication of two FP16 numbers retains full precision in the context of a GPU-style dot product. To examine this behavior, We set $A_{i,1} = B_{1,j} = 2^{11} - 1$, which, results in a product that exceeds the dynamic range of FP16. This value produces a product that requires the widest mantissa—up to 21 bits—among all possible multiplications of two finite FP16 values, exceeding the precision directly representable in the FP16 format.

All other elements of tiles A and B are initialized to zero, and the accumulator $C_{i,j}$ is also set to zero. This configuration ensures that the dot-product reduces to a single multiplication:

$$D_{i,j} = C_{i,j} + A_{i,1} \cdot B_{1,j} = (2^{11} - 1)^2$$

Note that the product $(2^{11} - 1)^2 = 4190209$ exceeds the range of FP16, and would therefore result in infinity if stored as FP16 during accumulation. However, after applying the Algorithm 3 Test, we see that the result is exact and no precision is lost. This is consistent with the known details of NVIDIA Tensor Cores (Appleyard et al., 2017).

Algorithm 3 FP16×FP16 Overflow and Precision Behavior

1: Initialize tiles A, B, and C to all zeros
2: Set $A_{0,1} \leftarrow 2^{11} - 1 = 2047$
3: $B_{1,0} \leftarrow 2^{11} - 1 = 2047$
4: Compute in the Tensor cores:

$$D \leftarrow C + A \cdot B$$

5: Verify $D_{0,0} = (2^{11} - 1)^2$ to ensure full precision product.

Conclusion: A single product of two FP16 elements is maintained with full precision (FP32).

5.2 Summation Dependency and Order

We now proceed with the first test in Hawkeye: reverse-engineering the specific order of partial sums of products, which impacts the final bit-level result. This requires identifying computationally independent subgroups within the summation, which involves an exhaustive search across all possible subsets of the 16 products. For each subset, we perform a call to Algorithm 4 to check for *computational neutrality*. A subgroup is defined as computationally neutral if, when its elements are engineered to sum to zero, they have no impact on the final bit-level result of the total accumulation.

By collecting these neutral subgroups, we can reconstruct the nested structure of the calculation. This nesting shows how smaller groups of operations are bundled inside larger ones. For example, if a small set of products is found to be neutral and is also part of a larger neutral group, it indicates that the hardware first combines the smaller set into a partial sum before adding it to the rest of the elements. This hierarchy allows us to trace the branches of the hardware's summation logic, as well as determine whether the accumulator enters the pipeline at the beginning or the end of the process.

Algorithm 4 Test for a Computationally Neutral Subgroup

1: S is a set of product indices $\{k_1, k_2, ..., k_m\}$ from $\{0, ..., 16\}$.
2: **Constants:**
3: $V_{large} \leftarrow 2^{20}$
4: $V_{small} \leftarrow 2^{-20}$
5:
6: **Test 1: Cancellation Scenario**
7: $C_{0,0} \leftarrow V_{large}$ if $0 \in S$ else V_{small}
8: **for** $k \in \{1, ..., 16\}$ **do**
9:   **if** $k \notin S$ **then**
10:     $A_{0,k}, B_{k,0} \leftarrow \sqrt{V_{small}}, \sqrt{V_{small}}$
11:   **else**
12:     $A_{0,k}, B_{k,0} \leftarrow \sqrt{V_{large}}, \sqrt{V_{large}}$
13:   **end if**
14: **end for**
15: Multiply half of the elements in S by -1 in order to get zero sum
16: $D \leftarrow$ Compute in Tensor cores$(C + AB)$
17: $R_{cancel} \leftarrow D_{0,0}$
18:
19: **Test 2: Zeroed Subgroup Scenario (Baseline)**
20: $C_{0,0} \leftarrow 0$ if $0 \in S$ else V_{small}
21: **for** $k \in \{1, ..., 16\}$ **do**
22:   **if** $k \notin S$ **then**
23:     $A_{0,k}, B_{k,0} \leftarrow \sqrt{V_{small}}, \sqrt{V_{small}}$
24:   **else**
25:     $A_{0,k}, B_{k,0} \leftarrow 0, 0$

26: **end if**
27: **end for**
28: $D \leftarrow$ Compute in Tensor Cores$(C + AB)$
29: $R_{zero} \leftarrow D_{0,0}$
30: **return** BitwiseCompare(R_{cancel}, R_{zero})

Conclusion: Our experimental results show that the group containing the initial accumulator and the first eight products is the only non-singleton subgroup that is computationally neutral. We can conclude that the hardware implements the specific two-stage accumulation structure depicted in Figure 1. Crucially, there is no evidence of dynamic sorting or reordering within the summation mechanism, indicating a straightforward and deterministic accumulation strategy implemented in the hardware.

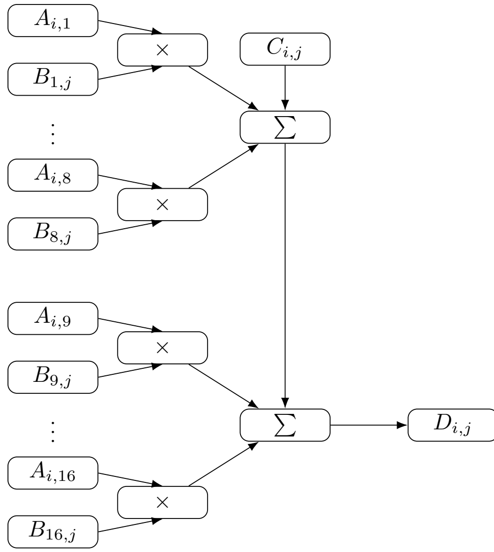

Figure 1. Computational graph of the two-stage tensor core accumulation in a pyramid structure. The initial accumulator $C_{i,j}$ and the first 8 products are summed into an intermediate result, which is then summed with the last 8 products.

5.3 Inferring GPU Internal Representation During Summation

We next design a test to infer the size of the internal significand (mantissa) used during summation. Specifically, our goal is to determine the smallest representable value that survives addition without being rounded, thereby revealing the effective precisionsof the internal accumulator.

In this test, formalized in Algorithm 5, we set $A_{i,1} = A_{i,2} = 1$, $B_{1,j} = 1$, and $B_{2,j} = -1$, so that not only are their combined contribution to the dot product equal to zero, but also so that any elements with significantly smaller exponents will be "swallowed" due to limited precision during addition. We then set a small, non-canceling, term by setting $A_{i,3} = 2^{\lceil -c/2 \rceil}$ and $B_{3,j} = 2^{\lfloor -c/2 \rfloor}$, which will result in the product $A_{i,3}B_{3,j} = 2^{-c}$. All other elements in A, B, C were set to zero. We then gradually increase c in order observe the smallest value of c such that $D_{i,j} = 2^{-c}$ is preserved in the result, without being rounded out.

Algorithm 5 Reproducing GPU Internal Representation

1: Initialize matrices A, B, C to zeros
2: Set $A_{i,1} \leftarrow 1, \quad B_{1,j} \leftarrow 1$
3: Set $A_{i,2} \leftarrow 1, \quad B_{2,j} \leftarrow -1$
4: **for** each integer $c \geq 1$ **do**
5: Set $A_{i,3} \leftarrow 2^{\lceil -c/2 \rceil}, \quad B_{3,j} \leftarrow 2^{\lfloor -c/2 \rfloor}$
6: Compute in the Tensor cores:

$$D \leftarrow C + A \cdot B$$

7: **if** $D_{i,j} \neq 2^{-c}$ **then**
8: Record $c - 1$ as the smallest exponent surviving without rounding
9: **break**
10: **end if**
11: **end for**

Conclusion: Running this experiment on NVIDIA Ampere Tensor Cores with FP16 datatype resulted in a smallest value of $c = 24$, indicating a 24-bit significand (including the implicit leading bit) consistent with FP32 accumulation.

5.4 Rounding Mode in Shift Operation During Summation

During floating-point addition, operands with different exponents must first be aligned by shifting the significand of the smaller-magnitude operand. This *shift operation* may discard low-order bits when the shift exceeds the available precision of the internal accumulator. The handling of these discarded bits depends on the rounding mode used during alignment. In this section, we determine the rounding mode applied by Tensor Cores during this internal shift step.

Algorithm 6 Recovering Rounding Mode

1: Initialize matrices A, B, C to zeros
2: Set $A_{i,1}, A_{i,2} \leftarrow 1$, $B_{1,j} \leftarrow 1$, $B_{2,j} \leftarrow -1$
3: **Test 1:** Set $A_{i,3} \leftarrow 2^{-13}$, $B_{3,j} \leftarrow 2^{-12}$
4: Compute dot product in the Tensor cores and verify result is 0.
5: **Test 2:** Set $A_{i,3} \leftarrow -2^{-13}$, keep $B_{3,j} \leftarrow 2^{-12}$
6: Compute dot product in the Tensor and verify result is 0 (rules out `nearest_tie_towards_minus_infinity`, `towards_minus_infinity`)
7: **Test 3:** Set $A_{i,3} \leftarrow 2^{-12} + 2^{-13}$, $B_{3,j} \leftarrow 2^{-12}$
8: Compute dot product in the Tensor and verify result is 2^{-24} (rules out `nearest_tie_towards_even`, `towards_even`)

9: **Test 4:** Set $A_{i,3} \leftarrow 2^{-13}$, $B_{3,j} \leftarrow 2^{-12} + 2^{-13}$
10: Compute dot product in the Tensor and verify result is 0 (indicating truncation rather than rounding)

To isolate this behavior, we construct dot products in which a very small value must be aligned against values of magnitude 1 during accumulation. The formal procedure is shown in Algorithm 6. We initialize the matrices so that two products cancel exactly: $A_{i,1} = A_{i,2} = 1$, $B_{1,j} = 1$, and $B_{2,j} = -1$. This ensures that the mathematically correct result of the dot product is determined solely by a carefully chosen small term.

Test 1. We first set $A_{i,3} = 2^{-13}$ and $B_{3,j} = 2^{-12}$, producing the product 2^{-25}. The observed result of the dot product is 0. This outcome is consistent with several rounding strategies during the shift step, including nearest-based modes and directed rounding modes.

Test 2. Next, we set $A_{i,3} = -2^{-13}$ while keeping $B_{3,j} = 2^{-12}$. The result remains 0. If the shift operation used a directed rounding mode toward negative infinity, the discarded bits would produce a negative contribution, so we can rule out rounding modes biased toward $-\infty$.

Test 3. To test nearest-based rounding modes, we construct a product slightly larger than the halfway point by setting $A_{i,3} = 2^{-12} + 2^{-13}$ and $B_{3,j} = 2^{-12}$, producing

$$(2^{-12} + 2^{-13}) \cdot 2^{-12} = 2^{-24} + 2^{-25}.$$

The observed result of the dot product is 2^{-24}, which eliminates rounding strategies such as `nearest_tie_towards_even` and `towards_even`.

Test 4. Finally, we reverse the construction by setting $A_{i,3} = 2^{-13}$ and $B_{3,j} = 2^{-12} + 2^{-13}$. This again produces a value slightly above the halfway threshold. The observed result is 0, indicating that the extra bits introduced during alignment are discarded rather than rounded upward.

Conclusion. Across these experiments, the observed behavior is consistent with a rounding mode of `towards_zero` during the internal shift operation. In other words, when the alignment step discards low-order bits, Tensor Cores truncate these bits rather than rounding them to the nearest representable value.

5.5 Post-Multiplication Normalization

We next investigate whether intermediate products are normalized before being forwarded to the accumulation stage. Normalization means adjusting a number's representation so that the significand (mantissa) falls within a standard range. An alternative design is to defer normalization and pass the raw product significand directly to the accumulator.

To determine which behavior Tensor Cores implement, we construct a dot product that produces large intermediate products that cancel exactly while introducing a very small third term (Algorithm 7). Specifically, we set $A_{i,1} = A_{i,2} = 1.5$, $B_{1,j} = 1.5$, and $B_{2,j} = -1.5$, yielding products of 2.25 and -2.25 that cancel during accumulation. We then introduce a small term by setting $A_{i,3} = 2^{-12}$ and $B_{3,j} = 2^{-12}$, producing the product 2^{-24}.

If products were normalized immediately after multiplication, the renormalization step would increase the exponent of the large intermediate values (e.g., $2.25 = 1.125 \times 2^1$), increasing the exponent difference during accumulation. This larger alignment shift could cause the small value 2^{-24} to be discarded during exponent alignment. Conversely, if normalization is deferred, the raw product significand is accumulated directly, preserving the small contribution.

Algorithm 7 Detecting Post-Multiplication Normalization

1: Initialize matrices A, B, C to zeros
2: Set $A_{i,1}, A_{i,2} \leftarrow 1.5$, $B_{1,j} \leftarrow 1.5$, $B_{2,j} \leftarrow -1.5$
3: Set $A_{i,3} \leftarrow 2^{-12}$, $B_{3,j} \leftarrow 2^{-12}$
4: Compute in the Tensor cores:

$$D \leftarrow C + A \cdot B$$

5: Check whether $D_{i,j} = 2^{-24}$

With Amper FP16, we observed $D_{i,j} = 2^{-24}$, indicating that the small term survives accumulation. This behavior is consistent with a design in which intermediate products are not normalized prior to summation. Instead, the raw multiplication result is forwarded directly to the accumulation pipeline, allowing the internal product significand to temporarily exceed the normalized range.

5.6 Normalization After Subnormal Multiplication

We next investigate how Tensor Cores handle products involving subnormal inputs. In IEEE floating-point arithmetic, subnormal numbers do not contain the implicit leading 1 in the significand, which effectively reduces the available precision. Some floating-point pipelines renormalize such values during multiplication, restoring a normalized representation before forwarding the result to the accumulation stage. Alternatively, the hardware may propagate the reduced-precision significand directly into the accumulator.

To distinguish between these behaviors, we construct a test in which a subnormal value is multiplied by a large normal value and then combined with a smaller term whose preservation depends on the effective precision of the intermediate product (Algorithm 8).

Specifically, we set $A_{i,1} = 2^{-14-k}$, which is a subnormal FP16 value, and $B_{1,j} = 2^{14}$. Their product equals

$$2^{-14-k} \cdot 2^{14} = 2^{-k}.$$

Although the resulting value is normal, its significand originates from a subnormal input and therefore contains only k significant bits of precision. We then introduce a smaller value by setting $A_{i,2} = 2^{-24}$ and $B_{2,j} = 1$, allowing us to observe whether this contribution survives the subsequent accumulation.

Algorithm 8 Detecting Normalization After Subnormal Multiplication

1: Initialize matrices A, B, C to zeros
2: Set $A_{i,1} \leftarrow 2^{-14-k}$ (subnormal FP16 value)
3: Set $B_{1,j} \leftarrow 2^{14}$ (normal FP16 value)
4: Set $A_{i,2} \leftarrow 2^{-24}, \quad B_{2,j} \leftarrow 1$
5: Compute in the Tensor cores:

$$D \leftarrow C + A \cdot B$$

6: Check whether the term 2^{-24} is preserved in $D_{i,j}$

Conclusion. We observe that the product derived from the subnormal operand behaves as if its significand contains only the reduced precision inherited from the input. The smaller term is therefore more easily lost during alignment in the accumulation stage. This indicates that Tensor Cores do not renormalize the result of a subnormal multiplication prior to accumulation; instead, the reduced-precision significand produced by the multiplication is propagated directly into the summation pipeline.

5.7 Rounding Mode of the Final Summation Result

Finally, we determine the rounding mode applied when the internal accumulation result is converted to the output precision. Earlier experiments showed that the Tensor Core accumulator maintains higher precision than the target FP16 format. When the final result is written back, excess mantissa bits must therefore be discarded or rounded.

To determine how this conversion is performed, we construct a dot product in which a very large intermediate value is perturbed by a small correction term (Algorithm 9). The large product establishes the dominant magnitude of the result, while the smaller term introduces low-order bits whose treatment reveals the rounding policy.

Test 1 (baseline magnitude). We first set

$$A_{i,1} = B_{1,j} = 1.5 \cdot 2^{12}, \quad A_{i,2} = 3, \quad B_{2,j} = 1.$$

The dominant product is

$$(1.5 \cdot 2^{12})^2 = 2.25 \cdot 2^{24},$$

while the smaller product contributes 3. The observed result is

$$D_{i,j} = 2.25 \cdot 2^{24},$$

indicating that the small contribution is eliminated during the final precision reduction.

Test 2 (sign-changing correction). Next, we replace the correction term with

$$A_{i,2} = 1, \quad B_{2,j} = -1,$$

producing a product of -1. The observed result becomes

$$D_{i,j} = 2.25 \cdot 2^{24} - 5.$$

The magnitude of the adjustment corresponds to the truncation of low-order mantissa bits rather than rounding to the nearest representable value.

Algorithm 9 Recovering Rounding Mode of Final Result

1: Initialize matrices A, B, C to zero
2: **Test 1:**
3: $A_{i,1} \leftarrow 1.5 \cdot 2^{12}, \quad B_{1,j} \leftarrow 1.5 \cdot 2^{12}$
4: $A_{i,2} \leftarrow 3, \quad B_{2,j} \leftarrow 1$
5: Compute $D \leftarrow C + A \cdot B$
6: **Reset matrices**
7: **Test 2:**
8: $A_{i,1} \leftarrow 1.5 \cdot 2^{12}, \quad B_{1,j} \leftarrow 1.5 \cdot 2^{12}$
9: $A_{i,2} \leftarrow 1, \quad B_{2,j} \leftarrow -1$
10: Compute $D \leftarrow C + A \cdot B$

Conclusion. The observed behavior indicates that the final conversion from the internal accumulator representation to the output format uses `towards_zero` rounding. In other words, excess mantissa bits are truncated rather than rounded to the nearest representable value.

5.8 Recovered Tensor Core Computation Pipeline

Based on the experiments described in the previous sections, we reconstruct the numerical pipeline used by NVIDIA Ampere Tensor Cores for FP16 matrix multiply-accumulate operations. The resulting model captures the sequence of transformations applied to each partial product and the accumulation strategy used to produce the final output.

Our findings indicate that Tensor Cores follow a structured pipeline consisting of three stages: (1) multiplication with expanded internal precision, (2) grouped accumulation with truncation-based exponent alignment, and (3) final normalization and precision reduction to the output format. The recovered computational model is summarized in Algorithms 12, 13, and 14, with the corresponding computational graph illustrated in Figure 1. This recovered model captures the key numerical behaviors observed in our experiments, including delayed normalization, truncation-based alignment

during summation, and the hierarchical grouping structure used for accumulation.

6 Reproducing FP16 Workflow on Hopper

To extend our analysis, we applied the same computational neutrality tests to the Hopper architecture Tensor Cores. The results revealed a different, more unified accumulation strategy. In contrast to the two-stage process found in the Ampere architecture, our tests on Hopper show that all 16 products are summed together with the initial accumulator in a single stage, as depicted in Figure 2.

Furthermore, our analysis indicates that the Hopper architecture utilizes an internal mantissa representation of 25 bits for this accumulation compared to 24 in Ampere. Consistent with the Ampere architecture, the final sum is normalized only after accumulation is complete, and the rounding mode is round-towards-zero.

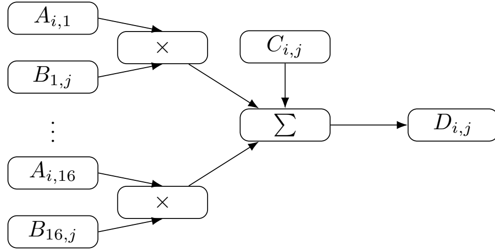

Figure 2. Computational graph of Hopper Tensor Core accumulation in a pyramid structure. The initial accumulator $C_{i,j}$ and the 16 products are summed into the final result

7 Reproducing BF16 Workflow on Ampere

We now extend our investigation of the dot product behavior to the BF16 format. Initial experiments show that all previously observed behaviors in FP16, including summation order, accumulator integration, rounding modes, and normalization patterns, are preserved when operating in BF16 on Ampere GPUs. This confirms that the underlying dot product algorithm is format-agnostic in terms of execution strategy.

However, BF16 introduces two new cases that do not occur in FP16-based tile multiplications. First, due to its wide exponent range, multiplications within a tile can result in products that fall below the smallest normalized FP32 value, producing sub-normal results in the accumulation path. Second, it is also possible for intermediate products or accumulations to exceed the dynamic range of FP32, resulting in overflow beyond the representable range of the FP32 format. These scenarios require additional tests to understand how such edge cases are handled, including whether rounding, saturation, or flushing to zero is applied by the hardware.

7.1 Handling of Out-of-Range Values in Tile Multiplication

To investigate how the GPU handles intermediate values that exceed the representable FP32 range during BF16 tile multiplication, we conducted two targeted tests. In the first test, we set $A_{i,1} = A_{i,2} = A_{i,3} = 2^{127}$ and $B_{1,j} = 2$, $B_{2,j} = -2$, $B_{3,j} = 1$. The expected contribution from the first two terms cancels out, and the result is $D_{i,j} = 2^{127}$. the intermediate values exceeds the maximum finite FP32 number ($\approx 3.4 \times 10^{38}$), yet the output was correctly returned as 2^{127}, indicating that the hardware internally allows intermediate values to temporarily exceed the FP32 range during summation, as long as the final result falls back within the FP32 representable range.

In the second test, we set $A_{i,1} = A_{i,9} = 2^{127}$, with $B_{1,j} = 2$ and $B_{9,j} = -2^{127}$. The resulting dot product was $D_{i,j} = \infty$, suggesting that the intermediate result exceeded the FP32 dynamic range and was ultimately cast to infinity. This confirms that after the accumulation step, the final result is cast back to FP32, and any value that falls outside the FP32 representable bounds is mapped to positive or negative infinity according to the IEEE 754 standard.

Algorithm 10 Handling Out-of-Range in BF16

1: **Require:** Matrices $A, B \in \mathbb{R}^{16\times16}$, accumulator $C \in \mathbb{R}^{16\times16}$
2: Initialize all entries of A, B, C to 0
3: **Test 1 (temporary extended-range accumulation):**
4: $A_{i,1}, A_{i,2}, A_{i,3} \leftarrow 2^{127}$ *(BF16 inputs)*
5: $B_{1,j} \leftarrow 2, \quad B_{2,j} \leftarrow -2, \quad B_{3,j} \leftarrow 1$
6: Compute in the Tensor cores:

$$D \leftarrow C + A \cdot B$$

7: **Verify:** $D_{i,j} = 2^{127}$ *(final FP32 within range despite intermediate overflow)*
8: **Reset:** Set all entries of A, B, C back to 0
9: **Test 2 (final cast to FP32 with overflow):**
10: $A_{i,1}, A_{i,9} \leftarrow 2^{127}$ *(BF16 inputs)*
11: $B_{1,j} \leftarrow 2, \quad B_{9,j} \leftarrow -2^{127}$
12: Compute in the Tensor cores:

$$D \leftarrow C + A \cdot B$$

13: **Verify:** $D_{i,j} = \infty$ *(magnitude exceeds FP32; saturated to IEEE 754 infinity)*

Conclusion: These results indicate that while the GPU supports extended-range accumulation internally when using

BF16, the final output still conforms to FP32 limits, and extreme overflows are handled via saturation to infinity.

7.2 Minimal Effective Contribution from Sub-Normal Products

As established in previous sections, the GPU performs internal summation using an extended exponent range beyond the standard FP32 format. This higher internal precision allows the accumulation of intermediate values with exponents far outside the final representable FP32 range. Motivated by this behavior, we tested what is the smallest multiplicative contribution that can still affect the final dot-product result. Since the internal summation uses rounding towards zero, the only way a small product can influence the result is by shifting into the least significant bits of the FP32 accumulation and reducing a bit that is still within the representable range.

We found that the smallest effective contribution corresponds to a product with exponent 2^{-156}. This implies that the summation logic tracks enough significant bits to detect changes down to this level. In terms of group-based alignment, this threshold is equivalent to taking the maximum exponent across all products and the number -132 (For Hopper GPUs, the corresponding number is -133). To generalize this observation, we designed a reproducible test framework that identifies the minimal maximal exponent during accumulation across architectures, This procedure is formalized in Algorithm 11.

To confirm this, we ran the following test: we set $A_{i,1} = A_{i,2} = B_{j,1} = 2^{-74}$, and $B_{2,j} = -2^{-82}$. The result of the dot product was $D_{i,j} = 2^{-149}$. However, when we modified $B_{2,j}$ to -2^{-83}, the result increased to $D_{i,j} = 2^{-148}$. This change confirms that a contribution at the 2^{-156} level still affects rounding behavior due to its interaction with the mantissa bits of the aligned sum. Any smaller contribution would be fully truncated and have no effect.

Algorithm 11 Reproducing GPU Minimal max exponent

1: **Input:** m - GPU internal representation
2: Initialize matrices A, B, C to zeros
3: **for** each integer $e \leq 0$ **do**
4: Set $A_{i,1} \leftarrow 2^{\lceil \frac{e}{2} \rceil}$, $B_{1,j} \leftarrow 2^{\lfloor \frac{e}{2} \rfloor}$
5: Set $A_{i,2} \leftarrow 2^{\lceil \frac{e-m}{2} \rceil}$, $B_{2,j} \leftarrow 2^{\lfloor \frac{e-m}{2} \rfloor}$
6: Compute in the Tensor cores:

$$D \leftarrow C + A \cdot B$$

7: **if** $D_{i,j} = 2^e$ **then**
8: Record $e+1$ as the minimal option for the maximal exponent during accumulation.
9: **break**
10: **end if**
11: **end for**

8 Empirical Results

Recall that the primary motivation of Hawkeye is verifiable machine learning: given a server's computation on an arbitrary GPU, can an auditor replicate the computation such that any discrepancies must be due to incorrect behavior versus GPU nondeterminism? Due to scope of our paper, we leave integrating matrix multiplication implementations identified by Hawkeye into the architecture of an existing model for future work. However, we provide simulators across the different GPU architectures and precision types we study in a public repository available here: https://github.com/badasherez/gpu-simulator.

To test our simulators, we create bit-exact reproduction tests across architectures (e.g. H100) and precision types (e.g. FP16), even extending our prior analysis to FP8 (E4M3 type) precision. These tests compare the outputs of our simulator with outputs of the custom kernels invoking `mma` on the hardware the simulator is run on. Across 100,000 randomly generated 16x16 matrix multiplication, our tests demonstrate 100% bit-exact replication.

We further report the average CPU execution times for sample matrix multiplications in Table 1, noting that our implementation provides a performance baseline, and further optimizations (such as a integer-based GPU implementation of the simulators) are deferred to future work. However, we note that for applications in verifiable ML, it suffices for an auditor to reproduce a computation only once, and therefore a slow CPU-side execution can be tolerated.

9 Future Work and Conclusion

While Hawkeye demonstrates that GPU-level numerical behaviors can be faithfully reproduced on CPUs (or other GPU hardwares), several challenges remain before such techniques can be fully integrated into verifiable machine learning pipelines, including proof systems and distributed training settings. First, our current framework focuses on a subset of NVIDIA architectures that utilize Tensor Cores; extending this approach to other hardware vendors and accelerator designs is an important direction for future work. Although vendor-specific implementations may introduce subtle differences, we believe that our test suite captures the fundamental design choices underlying 16×16 tiled matrix multiplication, providing a strong basis for generalization.

Second, while Hawkeye accurately reproduces low-level matrix multiplication behavior, higher-level operations commonly used in modern ML workloads—such as convolutions or attention mechanisms —require additional reverse engineering of implementation details beyond the scope of this work. Bridging this gap is essential for enabling end-to-end reproducibility of full training and inference pipelines.

Finally, integrating Hawkeye into practical verification frameworks raises systems-level challenges, including scalability in distributed environments and compatibility with cryptographic proof systems. Addressing these challenges will be critical for deploying Hawkeye in real-world auditing and verification workflows. Overall, Hawkeye provides a foundation for understanding and reproducing hardware-level numerical behavior, paving the way toward reproducible and hardware-agnostic machine learning systems.

ACKNOWLEDGEMENTS

Special thanks to Itamr Schen and Yoray Herzberg for their technical assistance in writing and implementing the CUDA kernels used in this work.

REFERENCES

Together AI. Together ai documentation and inference services. https://docs.together.ai/ and https://www.together.ai/inference, 2025. Serverless inference, fine-tuning, dedicated endpoints.

Jeremy Appleyard, Scott Yokim, and Paulius Micikevicius. Programming tensor cores in cuda 9. NVIDIA Developer Blog, 2017. October 2017. Available at: https://developer.nvidia.com/blog/programming-tensor-cores-cuda-9/.

Arasu Arun, Adam St. Arnaud, Alexey Titov, Brian Wilcox, Viktor Kolobaric, Marc Brinkmann, Oguzhan Ersoy, Ben Fielding, and Joseph Bonneau. Verde: Verification via refereed delegation for machine learning programs. In *CoRR*, volume abs/2502.19405, 2025.

Eli Ben-Sasson, Iddo Bentov, Yinon Horesh, and Michael Riabzev. Scalable, transparent, and post-quantum secure computational integrity (starks). IACR ePrint 2018/046, 2018a.

Eli Ben-Sasson, Iddo Bentov, Yinon Horesh, and Michael Riabzev. Fast reed–solomon interactive oracle proofs of proximity. In *ICALP*, 2018b.

Eli Ben-Sasson, Alessandro Chiesa, Michael Riabzev, Nicholas Spooner, Madars Virza, and Nicholas P. Ward. Aurora: Transparent succinct arguments for r1cs. In *EUROCRYPT*, 2019.

Benedikt Bünz, Jonathan Bootle, Dan Boneh, Andrew Poelstra, Pieter Wuille, and Greg Maxwell. Bulletproofs: Short proofs for confidential transactions and more. In *IEEE S&P*, 2018.

Alessandro Chiesa, Youssef Nir, Nicholas Sullivan Ward, et al. Marlin: Preprocessing zksnarks with universal and updatable srs. In *EUROCRYPT*, 2020a.

Alessandro Chiesa, Dev Ojha, and Nicholas Spooner. Fractal: Post-quantum and transparent recursive proofs from holography. In *EUROCRYPT*, 2020b.

Google Cloud. Vertex ai documentation. https://cloud.google.com/vertex-ai/docs, 2025. Platform docs and API reference.

Ariel Gabizon, Zachary J. Williamson, and Oana Ciobotaru. Plonk: Permutations over lagrange-bases for oecumenical noninteractive arguments of knowledge. IACR ePrint 2019/953, 2019.

Bianca-Mihaela Ganescu and Jonathan Passerat-Palmbach. Trust the process: Zero-knowledge machine learning to enhance trust in generative AI interactions. *arXiv preprint arXiv:2402.06414*, 2024.

Rosario Gennaro, Craig Gentry, Bryan Parno, and Mariana Raykova. Quadratic span programs and succinct nizks without PCPs. In *EUROCRYPT*, page 626–645, 2013.

Shafi Goldwasser, Yael Tauman Kalai, and Guy N. Rothblum. Delegating computation: Interactive proofs for muggles. In *STOC*, page 113–122, 2008.

Jens Groth. On the size of pairing-based non-interactive arguments. In *EUROCRYPT*, page 305–326, 2016.

Lukas Heumos, Philipp Ehmele, Luis Kuhn Cuellar, Kevin Menden, Edmund Miller, Steffen Lemke, Gisela Gabernet, and Sven Nahnsen. mlf-core: a framework for deterministic machine learning. *Bioinform.*, 39(4), 2023. doi: 10.1093/BIOINFORMATICS/BTAD164. URL https://doi.org/10.1093/bioinformatics/btad164.

Jordan Hoffmann, Sebastian Borgeaud, Arthur Mensch, Elena Buchatskaya, Trevor Cai, Eliza Rutherford, Diego de Las Casas, Lisa Anne Hendricks, Johannes Welbl, Aidan Clark, et al. Training compute-optimal large language models. *arXiv preprint arXiv:2203.15556*, 2022.

Matthew Hutson. Artificial intelligence faces reproducibility crisis, 2018.

William Kahan. IEEE standard 754 for binary floating-point arithmetic. *Lecture Notes on the Status of IEEE*, 754 (94720-1776):11, 1996.

Jared Kaplan, Sam McCandlish, Tom Henighan, Tom B. Brown, Benjamin Chess, Rewon Child, Scott Gray, Alec Radford, Jeffrey Wu, and Dario Amodei. Scaling laws for neural language models. *arXiv preprint arXiv:2001.08361*, 2020.

Aniket Kate, Gregory M. Zaverucha, and Ian Goldberg. Constant-size commitments to polynomials and their applications. In *ASIACRYPT*, page 177–194, 2010.

Joe Kilian. A note on efficient zero-knowledge proofs and arguments (extended abstract). In *STOC*, page 723–732, 1992.

Hidde Lycklama, Alexander Viand, Nikolay Avramov, Nicolas Küchler, and Anwar Hithnawi. Artemis: Efficient commit-and-prove snarks for zkml. In *arXiv preprint arXiv:2409.12055*, 2024.

Mary Maller, Sean Bowe, Markulf Kohlweiss, and Sarah Meiklejohn. Sonic: Zero-knowledge snarks from linear-size universal and updatable srs. In *ACM CCS*, 2019.

Microsoft. Azure machine learning documentation. https://learn.microsoft.com/azure/machine-learning/, 2024. Service overview, training and deployment guides.

NVIDIA. Nvidia a100 tensor core gpu architecture. https://images.nvidia.com/aem-dam/en-zz/Solutions/data-center/nvidia-ampere-architecture-whitepaper.pdf, 2020.

NVIDIA. Nvidia h100 tensor core gpu architecture. https://www.hpctech.co.jp/catalog/gtc22-whitepaper-hopper_v1.01.pdf, 2022.

Jack Min Ong, Matthew Di Ferrante, Aaron Pazdera, Ryan Garner, Sami Jaghouar, Manveer Basra, and Johannes Hagemann. TOPLOC: A locality sensitive hashing scheme for trustless verifiable inference. In *CoRR*, volume abs/2501.16007, 2025.

Bryan Parno, Jon Howell, Craig Gentry, and Mariana Raykova. Pinocchio: Nearly practical verifiable computation. In *IEEE S&P*, page 238–252, 2013.

Zhizhi Peng, Taotao Wang, Chonghe Zhao, Guofu Liao, Zibin Lin, Yifeng Liu, Bin Cao, Long Shi, Qing Yang, and Shengli Zhang. A survey of zero-knowledge proof based verifiable machine learning. *arXiv preprint arXiv:2502.18535*, 2025.

PyTorch Contributors. torch.use_deterministic_algorithms. https://docs.pytorch.org/docs/stable/generated/torch.use_deterministic_algorithms.html, 2026. Accessed: 2026-03-17.

pyxis-roc. sass-math: Bit-accurate implementations of elementary function approximations in nvidia's sass isa. https://github.com/pyxis-roc/sass-math, 2021.

Samyam Rajbhandari, Jeff Rasley, Olatunji Ruwase, and Yuxiong He. Zero: Memory optimizations toward training trillion parameter models. *arXiv preprint arXiv:1910.02054*, 2020.

Replicate. Replicate documentation. https://replicate.com/docs/, 2025. Cloud API for running and hosting ML models.

Amazon Web Services. Amazon sagemaker documentation. https://docs.aws.amazon.com/sagemaker/, 2025. Product docs (overview and user guide).

Srinath Setty. Spartan: Efficient and general-purpose zksnarks without trusted setup. In *CRYPTO*, 2020.

Adi Shamir. Ip = pspace. *Journal of the ACM*, 39(4): 869–877, 1992. doi: 10.1145/146585.146609.

Sanjif Shanmugavelu, Mathieu Taillefumier, Christopher Culver, Oscar R. Hernandez, Mark Coletti, and Ada Sedova. Impacts of floating-point non-associativity on reproducibility for HPC and deep learning applications. In *SC Workshops*, pages 170–179. IEEE, 2024.

Mohammad Shoeybi, Mostofa Patwary, Raul Puri, Patrick LeGresley, Jared Casper, and Bryan Catanzaro. Megatron-lm: Training multi-billion parameter language models using model parallelism. *arXiv preprint arXiv:1909.08053*, 2019.

Cathie So, KD Conway, Xiaohang Yu, Suning Yao, and Kartin Wong. opp/ai: Optimistic privacy-preserving AI on blockchain. In *arXiv preprint arXiv:2402.15006*, 2024.

Megha Srivastava, Simran Arora, and Dan Boneh. Optimistic verifiable training by controlling hardware nondeterminism. In *NeurIPS*, 2024.

Haochen Sun, Jason Li, and Hongyang Zhang. zkllm: Zero knowledge proofs for large language models. In *CCS*, 2024. ACM CCS 2024 (salt lake city), 4405–4419.

Thinking Machines Lab. Defeating non-determinism in llm inference. https://thinkingmachines.ai/blog/defeating-nondeterminism-in-llm-inference/, September 2025. Accessed: 2026-03-17.

Dennis Wüst, Alex Ozdemir, Riad S. Wahby, Abhi Shelat, et al. Hyrax: Doubly-efficient zksnarks without trusted setup. In *IEEE S&P*, 2018.

Tiancheng Xie, Jiaheng Zhang, Yupeng Zhang, Charalampos Papamanthou, and Dawn Song. Libra: Succinct zero-knowledge proofs with optimal prover computation. In *CRYPTO*, 2019.

Algorithm 12 Floating-Point Multiplication (Recovered Ampere Model for FP16)

Table 1. Average execution time (in seconds) and standard deviation for 4096×4096 matrix multiplication on an Apple M4 Pro CPU, calculated over 10 independent runs.

PRECISION	ARCHITECTURE	TIME (SEC.)	STD (SEC.)
FP16	AMPERE	50.8	3.2
FP16	HOPPER	47.2	2.5
BF16	AMPERE	52.5	2.9
BF16	HOPPER	48.2	2.6
FP8	HOPPER	40.6	0.6

Require: Floating-point inputs (s_A, e_A, m_A) and (s_B, e_B, m_B)
Ensure: Product $(s_{\text{result}}, e_{\text{result}}, m_{\text{result}})$
1: $s_{\text{result}} \leftarrow s_A \oplus s_B$
2: $e_{\text{result}} \leftarrow e_A + e_B$
3: $m_{\text{raw}} \leftarrow m_A \cdot m_B$
4: $m_{\text{result}} \leftarrow m_{\text{raw}} \ll 3$ // internal precision expansion
5: **return**$(s_{\text{result}}, e_{\text{result}}, m_{\text{result}})$

Algorithm 13 Grouped Summation (Recovered Ampere Model for FP16)

Require: Floating-point values $\{(s_i, e_i, m_i)\}_{i=1}^n$
Ensure: Result $(s_{\text{out}}, e_{\text{out}}, m_{\text{out}})$
1: **for** $i = 1$ to n **do**
2: $m_i \leftarrow m_i \ll 1$ // expand to internal precision
3: **end for**
4: $e_{\max} \leftarrow \max_i e_i$
5: **for** $i = 1$ to n **do**
6: $\Delta e \leftarrow e_{\max} - e_i$
7: $m_i \leftarrow m_i \gg \Delta e$ // truncating alignment shift
8: **end for**
9: $M \leftarrow \sum_{i=1}^n (-1)^{s_i} \cdot m_i$
10: Normalize $(e_{\max}, M)$ to floating-point form
11: Truncate significand to output precision
12: **return**$(s_{\text{out}}, e_{\text{out}}, m_{\text{out}})$

Algorithm 14 Dot Product Computation (Recovered Ampere Model for FP16)

Require: Accumulator C and vectors $A[1..16], B[1..16]$
Ensure: Dot product result
1: **for** $i = 1$ to 16 **do**
2: $P[i] \leftarrow$ **Multiply**$(A[i], B[i])$
3: **end for**
4: $G_1 \leftarrow$ GroupSum$(\{C, P[1], \ldots, P[8]\})$
5: $G_2 \leftarrow$ GroupSum$(\{G_1, P[9], \ldots, P[16]\})$
6: **return**G_2