

USENIX

THE ADVANCED COMPUTING SYSTEMS ASSOCIATION

TensorZKP: Repurposing GPU Tensor Cores for High-Performance Zero-Knowledge Proofs

Tao Lu, Jipeng Zhang, and Yanpei Guo, *National University of Singapore;*
Xuanming Liu, *Zhejiang University;* Wenjie Qu, *National University of Singapore;*
Zonghui Wang and Wenzhi Chen, *Zhejiang University;*
Jiaheng Zhang, *National University of Singapore*

https://www.usenix.org/conference/usenixsecurity26/presentation/lu-tao

This paper is included in the Proceedings of the 35th USENIX Security Symposium.

August 12–14, 2026 • Baltimore, MD, USA

ISBN 978-1-939133-58-8

Open access to the Proceedings of the 35th USENIX Security Symposium is sponsored by



جامعة الملك عبدالله للعلوم والتقنية
King Abdullah University of Science and Technology

ARTIFACT EVALUATED
USENIX
AVAILABLE

TensorZKP: Repurposing GPU Tensor Cores for High-Performance Zero-Knowledge Proofs

Tao Lu[1], Jipeng Zhang[1], Yanpei Guo[1], Xuanming Liu[2], Wenjie Qu[1], Zonghui Wang[2], Wenzhi Chen[2], Jiaheng Zhang[1,†]

[1]*National University of Singapore* [2]*Zhejiang University*

Abstract

GPU Tensor Cores, specialized hardware units designed to accelerate matrix multiplication, have served as the primary engine behind the AI revolution. Given the exponential performance gains they have delivered, aligning cryptographic implementations with this hardware evolution is critical. This is particularly acute for zero-knowledge proofs (ZKPs), a cryptographic primitive that currently grapples with high proof generation costs. Existing GPU implementations for ZKPs rely exclusively on general-purpose SIMT cores, leaving the massive computational power of Tensor Cores untapped.

In this paper, we introduce TensorZKP, the first GPU framework to harness Tensor Cores for ZKP acceleration. Since Tensor Cores are designed for low-precision matrix multiplication, mapping ZKP's arithmetic to this hardware is non-trivial. To bridge this gap, we develop Tensor-Core-compatible finite field arithmetic and reformulate ZKP modules, specifically sum-check protocols and Spielman code, into matrix multiplication tasks. Furthermore, we design an asynchronous warp-specialized framework that pipelines memory access, Tensor Core matrix operations, and SIMT-based modular reductions. We instantiate these optimizations with HyperPlonk as the Polynomial Interactive Oracle Proof (PIOP) and Brakedown as the Polynomial Commitment Scheme (PCS) to enable end-to-end proof generation.

The evaluation results show that TensorZKP exhibits remarkable efficiency. At a 2^{25} scale, the underlying building blocks complete in 0.85 ms for inner product, 0.91 ms for scalar-vector multiplication, 4.04 ms for degree-2 sum-check, and 11.58 ms for the encoder. For a circuit with 2^{25} multiplication gates, TensorZKP achieves a proof generation time of only 215.28 milliseconds, representing a 955× speedup over the CPU baseline and a 36.2× improvement over state-of-the-art SIMT-based GPU implementations.

†Corresponding author.

1 Introduction

Zero-knowledge proof (ZKP) [17] is a fundamental cryptographic primitive that enables one party (the prover $\mathcal{P}$) to convince another party (the verifier $\mathcal{V}$) of the validity of a statement x without revealing the underlying secret witness w. Formally, for a relation R, it ensures that $\mathcal{V}$ accepts the proof if and only if $(x,w) \in R$, except with negligible probability. In recent years, ZKPs have attracted growing attention from both academia [25, 37, 47, 55, 57] and industry [15, 40, 46], finding widespread applications in privacy-sensitive domains such as blockchain systems [28, 33, 56], anonymous payment schemes [3, 44], and verifiable machine learning [5, 11, 21, 32, 41, 49].

While ZKPs offer robust guarantees of privacy and verifiability, proof generation remains a computationally expensive task. The computation is typically modeled as an arithmetic circuit over a large finite field, introducing a fundamental performance disparity compared to standard execution. For instance, in the context of verifiable machine learning, standard workloads for neural networks benefit from hardware-accelerated floating-point arithmetic. In contrast, ZKP systems must emulate these operations using modular arithmetic, incurring significant computational overheads. Consequently, proving the correctness of a neural network inference results in costs orders of magnitude higher than its standard execution [5, 11, 32, 41, 49]. This bottleneck highlights the need for accelerating the underlying cryptographic operations to facilitate ZKP deployment.

To address this computational demand, GPUs have evolved into the dominant platform for high-performance computing, propelled by the explosive growth of AI [29]. This revolution has fundamentally transformed GPU architecture, establishing Tensor Cores as the primary engine of computational power, delivering significant performance gains for matrix-centric workloads. Thus, it is critical for cryptographic implementations to align with this hardware evolution. However, the current landscape of GPU-based ZKP implementations lag behind: existing GPU-based work [7, 24, 26, 34, 35, 37] con-

tinues to rely solely on general-purpose SIMT cores, leaving the massive potential of Tensor Cores unexploited.

Utilizing GPU Tensor Cores for ZKPs is far from straightforward due to the fundamental mismatch between hardware capabilities and cryptographic requirements. **Challenge #1: Precision Mismatch.** Tensor Cores are specialized for low-precision arithmetic (e.g., FP8, UINT8), rendering them ill-suited for the finite field arithmetic required by ZKPs. Traditional element-wise implementations decompose high-precision field elements into multiple limbs, processing them via independent scalar operations. This method incurs significant overhead due to fine-grained carry propagation and modular reduction, undermining the hardware's acceleration potential. **Challenge #2: Structural Divergence.** Tensor Cores are fixed-function units designed exclusively for matrix multiplication. In contrast, core ZKP primitives, such as sum-check protocols and linear-time encoders, do not naturally align with this matrix paradigm. Effectively bridging this gap requires a hardware-algorithm co-design approach to reformulate these operations, ensuring compatibility with Tensor Cores. **Challenge #3: System-Level Constraints.** Unlike general-purpose SIMT cores that support flexible memory access and thread execution, Tensor Cores mandate highly structured data layouts with fixed tile sizes, alongside a strict cooperative execution model where threads within a warp must operate synchronously. These architectural discrepancies render state-of-the-art SIMT-based frameworks [7, 24, 26, 34, 35, 37] functionally incompatible with Tensor Core architectures.

1.1 Contributions

We introduce TensorZKP, the first GPU framework leveraging Tensor Cores for ZKP acceleration, which addresses the aforementioned challenges through the following contributions:

- **Matrix-Centric Framework for Field Arithmetic.** We design a novel framework that reformulates finite field arithmetic into a matrix-centric paradigm. Specifically, departing from the traditional approach that treats field arithmetic as a sequence of element-wise scalar operations, we develop specialized formulations for two fundamental primitives over fields: vector inner products and vector-scalar multiplication. By aggregating the decomposed 8-bit limbs of entire input vectors into unified blocks, we transform two primitives into matrix multiplication tasks. This formulation not only maps field arithmetic into low-precision matrix operations but also leverages this collective structure to enable lazy modular reduction, significantly amortizing the cost of expensive modular arithmetic.
- **Algorithm-Hardware Co-design of ZKP Modules.** Beyond low-level arithmetic, we fundamentally redesign complex ZKP modules, including sum-check protocols [36] and linear-time encoders [48], to strictly align with both the compute capability and memory hierarchy of Tensor Cores. We restructure their dataflow into dense matrix multiplication tasks, enforcing tiled memory access patterns that maximize L2 cache locality on GPUs. Furthermore, we instantiate these optimizations within HyperPlonk [6] (PIOP) and Brakedown [18] (PCS) to enable end-to-end proof generation. To the best of our knowledge, this is the first work to map ZKP workloads on Tensor Core matrix units, following the dominant trend of AI-driven hardware evolution.
- **Asynchronous Warp-Specialized Execution.** We architect an asynchronous execution model using warp specialization to maximize heterogeneous GPU utilization. First, we leverage a producer-consumer model to decouple memory access and on-chip computation using different execution warps asynchronously, to effectively masks global memory latency and arithmetic overheads. Furthermore, we manages the fine-grained execution between Tensor-Core-driven matrix accumulation and SIMT-based field modular reduction. Consequently, this architectural optimization enables high-throughput matrix operations, enabling the high-throughput processing required for real-time proof generation.
- **Implementation and Evaluation.** We implement and evaluate TensorZKP across both individual ZKP modules and the end-to-end proof generation workflow. Our experimental results highlight the superior efficiency of our design. At the module level, TensorZKP outperforms state-of-the-art GPU baselines by 88× in sum-check protocols and 22× in linear-time encoders. For end-to-end evaluation, benchmarking against state-of-the-art ZKP implementations with HyperPlonk as the PIOP and Brakedown as the PCS, TensorZKP exhibits remarkable efficiency. For a circuit with 2^{25} multiplication gates, TensorZKP achieves a proof generation time of only 215.28 milliseconds, representing a 955× speedup over the CPU baseline and a 36.2× improvement over SIMT-based GPU implementations.

1.2 Related Work

Modern ZKP protocols. Modern ZKP protocols [6, 8, 13, 18, 19, 47, 52, 55, 57] predominantly focus on Zero-Knowledge Succinct Non-interactive Arguments of Knowledge (zk-SNARKs), which have three core properties: (1) Non-interactivity, where the proof consists of a single message from $\mathcal{P}$ to $\mathcal{V}$, typically achieved via the Fiat-Shamir transformation [12]; (2) Zero-Knowledge, ensuring the proof discloses nothing about the secret witness; and (3) Succinctness, implying the proof size is small, typically logarithmic or constant to the circuit size, and verification is significantly faster than native computation. Structurally, zk-SNARKs are typically constructed using a modular framework that composes a Polynomial Interactive Oracle Proof (PIOP) [4] with a Polynomial Commitment Scheme (PCS) [27].

In the domain of PIOPs, Plonk [13] and Marlin [8] serve as representative protocols that utilize univariate polynomials. However, they rely heavily on Fast Fourier Transforms (FFTs) for polynomial arithmetic, which incur $O(n \log n)$ complexity. To address this bottleneck, recent protocols like Spartan [47] and HyperPlonk [6] have shifted towards multilinear polynomials evaluated over the Boolean hypercube. By replacing FFTs with the sum-check protocol [36], HyperPlonk eliminates the super-linear overheads of univariate protocols, offering more efficient performance with $O(n)$ complexity.

Parallel to PIOP advancements, PCS has also focused on minimizing prover costs. The widely used KZG commitment [27] offers succinct proofs but relies on computationally expensive Multi-Scalar Multiplications (MSM). To mitigate these issues, FRI [2, 20, 58] was introduced, relying purely on hash functions and Reed-Solomon codes to achieve transparent, post-quantum security. More recently, the field has progressed towards linear-time, code-based commitments such as Brakedown [18] and Orion [57]. These schemes employ Spielman code [48] bypass both MSMs and FFTs entirely, to achieve strictly linear-time $O(n)$ commitment generation. Despite these asymptotic improvements, however, the concrete computational overhead of ZKPs remains substantial.

Hardware-Accelerated ZKPs. To address the significant proving overhead in practical applications, hardware acceleration has become indispensable for enabling high-performance ZKPs. Extensive research has explored diverse platforms, ranging from the flexibility of large CPU clusters [30, 31, 54] to the customized datapaths of FPGAs [9, 22, 42, 60] and energy-efficient ASICs [43, 53, 59]. Earlier efforts [25, 35, 37, 54, 59] focus primarily on accelerating MSM for elliptic-curve–based ZKP protocols such as Groth16 [19] and Plonk [13]. More recent works [9, 22, 43], however, have shifted attention to the protocols [6, 18, 57] that replace heavy curve arithmetic with cost-efficient components like sum-check protocols [36], Merkle trees [38], and Spielman code [48].

Among these platforms, GPUs stand out as a particularly attractive option due to their high computational throughput and wide availability as commodity devices. Previous studies [7, 24, 26, 34, 35, 37] on GPU acceleration have demonstrated promising results. For instance, GZKP [37] and cuZK [35] optimize the compute-intensive MSM kernels required by protocols like Groth16 [19], while DistMSM [25] extends these techniques to distributed multi-GPU settings. For multilinear polynomial workloads, frameworks such as BatchZK [34] and Icicle [24] have introduced GPU-accelerated implementations of sum-check modules. However, these prior approaches for ZKPs relied solely on the general-purpose SIMT cores of GPUs. While they effectively leverage thread-level parallelism, their performance is fundamentally constrained by the throughput of scalar instructions, leaving the specialized Tensor Cores in modern GPUs unutilized.

Tensor-Accelerated Cryptography. Beyond ZKPs, a parallel line of research has explored Tensor Cores for other cryptographic workloads, notably homomorphic encryption (HE) and post-quantum cryptography (PQC). TensorFHE [10] and ConvKyber [61] target the Number Theoretic Transform (NTT) used in FHE/PQC by recasting the $O(N \log N)$ FFT-butterfly NTT as a matrix-based NTT of $O(N\sqrt{N})$ and $O(N^2)$ complexity, respectively. While effective for short polynomials, these direct mappings trade asymptotic complexity for Tensor-Core compatibility. In contrast, TensorZKP reformulates sum-check protocols and Spielman encoding into matrix multiplications while preserving the optimal $O(N)$ complexity, making it suitable for the much larger problem sizes encountered in ZKP proof generation.

2 Background

2.1 Notations

We use bold lower-case letters, such as $\mathbf{x}$, to represent vectors, where x_i denotes the i-th element of $\mathbf{x}$. We denote the standard inner product of two vectors as $\langle \mathbf{x}, \mathbf{y} \rangle = \sum x_i y_i$, and the Hadamard product as $\mathbf{x} \circ \mathbf{y}$. Similarly, we use bold upper-case letters, such as $\mathbf{M}$, to represent matrices, where $M_{i,j}$ denotes the element in the i-th row and j-th column. We denote the matrix multiplication of two matrices $\mathbf{A}$ and $\mathbf{B}$ as $\mathbf{A} \times \mathbf{B}$.

We let $\mathbb{F}_p$ denote a prime field of order p and $\mathbb{F}_{p^k}$ denote an extension field. Let λ be the security parameter. To ensure λ-bit security, we require the field size to satisfy $\log_2(p^k) \geq \lambda$, where typically $\lambda \in \{128, 256\}$. To align with low-precision hardware, we use a t-bit limb decomposition. We define a decomposition function $\text{limb}(\cdot, w)$ that maps an element $a \in \mathbb{F}_p$ to a vector of limbs $\mathbf{a} \in [0, 2^w)^l$ with $\mathbf{a} = \text{limb}(a)$. This decomposition satisfies $a = \langle \mathbf{a}, \mathbf{b} \rangle$, where $\mathbf{b} = (1, 2^w, 2^{2w}, \ldots, 2^{(l-1)w})$ is the basis vector. For the extension field $\mathbb{F}_{p^k}$, we represent its element via a polynomial basis with k coefficients in $\mathbb{F}_p$. The notation $s \leftarrow \mathbb{F}_{p^k}$ denotes sampling s uniformly at random from $\mathbb{F}_{p^k}$.

2.2 Zero-Knowledge Proof

Zero-Knowledge Proof (ZKP) [17] is a cryptographic primitive that enables a prover $\mathcal{P}$ to convince a verifier $\mathcal{V}$ of the validity of a statement x without revealing the underlying secret witness w. As discussed in Section 1.2, modern ZKPs are typically constructed by PIOPs and PCSs. In this paper, we target sum-check-based PIOPs and linear-time coding-based PCSs, utilizing HyperPlonk [6] and Brakedown [18] as our case studies. This architectural choice notably eliminates the need for computationally expensive FFTs and MSMs, enabling strictly linear $O(n)$ proof generation complexity.

2.2.1 HyperPlonk

We utilize HyperPlonk [6] as the PIOP to map circuit satisfiability into polynomial relations. Figure 1 outlines an overview of HyperPlonk, which decomposes circuits into two categories: (1) gate identities and (2) wiring identities.

The gate identities enforce arithmetic correctness by unifying addition and multiplication logic into a single polynomial constraint. For a circuit of size $N = 2^\mu$, we associate each gate with a unique coordinate $\mathbf{x}$ on the Boolean hypercube $\{0,1\}^\mu$. Let $w_{\{a,b,c\}}(\mathbf{x})$ denote the polynomials for left, right, and output wires, and $q_{\{A,M,O\}}(\mathbf{x})$ be the selector polynomials acting as structural switches. The system defines gate types by setting $q_A(\mathbf{x}) = q_O(\mathbf{x}) = 1, q_M(\mathbf{x}) = 0$ for the addition gate and $q_M(\mathbf{x}) = q_O(\mathbf{x}) = 1, q_A(\mathbf{x}) = 0$ for the multiplication gate. The circuit validity is encoded as ensuring that $P_{gate}(\mathbf{x}) = 0$ for all $\mathbf{x} \in \{0,1\}^\mu$, where $P_{gate}(\mathbf{x})$ defined as:

$$q_A(\mathbf{x})(w_a(\mathbf{x}) + w_b(\mathbf{x})) + q_M(\mathbf{x})w_a(\mathbf{x})w_b(\mathbf{x}) - q_O(\mathbf{x})w_c(\mathbf{x}) \tag{1}$$

This zero-check condition can be reduced to a sum-check instance using Schwartz-Zippel Lemma [45]:

$$\sum_{\mathbf{x}\in\{0,1\}^\mu} P_{gate}(\mathbf{x}) \cdot eq(\mathbf{x},\mathbf{r}) = 0, \tag{2}$$

where $\mathbf{r} \in (\mathbb{F}_{p^k})^\mu$ are random challenges from verifier, and $eq(\mathbf{x},\mathbf{r})$ is defined as the product of linear terms:

$$eq(\mathbf{x},\mathbf{r}) = \prod_{i=0}^{\mu-1} ((1-x_i)(1-r_i) + x_i r_i). \tag{3}$$

The wiring identities enforce global consistency by ensuring that connected wires across different gates. To verify this, we aggregate the individual wire polynomials $w_{\{a,b,c\}}$ into a single unified witness w by introducing two auxiliary variables $\mathbf{y} \in \{0,1\}^2$. We define polynomial w over the extended hypercube $\{0,1\}^2 \times \{0,1\}^\mu$ as:

$$w(0,0,\mathbf{x}) = w_a(\mathbf{x}), w(0,1,\mathbf{x}) = w_b(\mathbf{x}), w(1,0,\mathbf{x}) = w_c(\mathbf{x}), \tag{4}$$

with $w(1,1,x)$ set to zero padding. The wire connection is then represented as a permutation σ over this extended domain $\{0,1\}^{\mu+2}$. Specifically, if a wire at position $\mathbf{z} \in \{0,1\}^{\mu+2}$ connects to $\mathbf{z}'$, then $\sigma(\mathbf{z}) = \mathbf{z}'$, implying $w(\mathbf{z}) = w(\mathbf{z}')$.

HyperPlonk verifies this via a Grand Product argument. Let $id(\mathbf{y},\mathbf{x})$ be the identity map. Given a random challenges β from the verifier, the protocol checks:

$$\prod_{\mathbf{z}\in\{0,1\}^{\mu+2}} (w(\sigma(\mathbf{z})) + \beta \cdot id(\mathbf{z})) = \prod_{\mathbf{z}\in\{0,1\}^{\mu+2}} (w(\mathbf{z}) + \beta \cdot \sigma(\mathbf{z})). \tag{5}$$

This product check is subsequently reduced to a sum-check protocol based on the GKR protocol [16,47]. Thus, both gate and wiring identities are unified as instances of the sum-check protocol, the details of which are presented in Section 2.3.

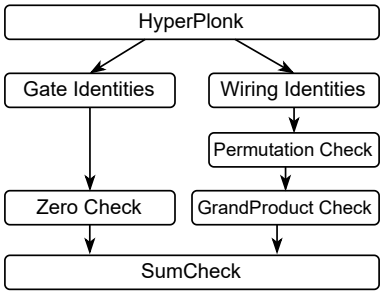


Figure 1: An overview of HyperPlonk.

2.2.2 Brakedown

We use Brakedown [18] as the PCS to bind the polynomials generated by the PIOP. A PCS is a fundamental cryptographic primitive that enables a prover to commit to a large polynomial $w(\mathbf{x})$ via a succinct string cm and subsequently prove evaluations $y = w(\mathbf{x})$ without revealing the underlying polynomial. Brakedown is a code-based scheme leveraging linear error-correcting codes and hashing, with strictly linear-time commitment generation. The commitment procedure transforms the coefficient vector of the multilinear polynomial $w \in (\mathbb{F}_{p^k})^N$ through three logical stages:

1. **Matrix Packing:** The coefficient vector of polynomial w is reshaped into a square matrix $\mathbf{W} \in (\mathbb{F}_{p^k})^{\sqrt{N}\times\sqrt{N}}$ to facilitate row-wise operations.

2. **Linear Encoding:** A linear code $\mathcal{C}$ is applied to each row of $\mathbf{W}$ to produce an extended matrix $\mathbf{W}^*$. To ensure linear-time efficiency, Brakedown utilizes Spielman codes [48] as the underlying code and the detailed construction of Spielman code is discussed in Section 2.4.

3. **Merkle Hashing:** The columns of the extended matrix $\mathbf{W}^*$ are treated as leaves and hashed to construct a Merkle tree. The root of this tree is the final commitment cm.

Consequently, given the relative efficiency of cryptographic hashing, the primary overhead of Brakedown [18] is dominated by the linear encoding process via Spielman code.

2.3 Sum-check Protocol

The sum-check protocol [36] is a fundamental interactive proof system that allows a prover $\mathcal{P}$ to convince a verifier $\mathcal{V}$ that the sum of a polynomial $P : (\mathbb{F}_{p^k})^\mu \to \mathbb{F}_{p^k}$ over the Boolean hypercube is equal to the claimed value h:

$$h = \sum_{\mathbf{x}\in\{0,1\}^\mu} P(\mathbf{x}). \tag{6}$$

In the context of PIOPs, the polynomial P is rarely atomic. Instead, it typically arises as a product of constituent multilinear

polynomials. For instance, if we define $P(\mathbf{x})$ as the product of two polynomials $f(\mathbf{x})$ and $g(\mathbf{x})$, the protocol verifies:

$$h = \sum_{\mathbf{x}\in\{0,1\}^{\mu}} f(\mathbf{x}) \cdot g(\mathbf{x}). \tag{7}$$

Protocol Description. The sumcheck protocol proceeds in μ rounds, in round $i \in [1,\mu]$, $\mathcal{P}$ fixes the first i variables to the random challenges $r_1,\ldots,r_i \in \mathbb{F}_{p^k}$ chosen by $\mathcal{V}$ in previous rounds and sends a univariate polynomial $S_i(x)$ to $\mathcal{V}$:

$$S_i(x) = \sum_{(b_{i+1},\ldots,b_{\mu})\in\{0,1\}^{\mu-i}} P(r_1,\ldots,r_{i-1},x,b_{i+1},\ldots,b_{\mu}). \tag{8}$$

Upon receiving $S_i(x)$, $\mathcal{V}$ performs the following checks:

1. **Consistency Check:** $\mathcal{V}$ verifies that $S_i(0)+S_i(1) = S_{i-1}(r_{i-1})$, where $S_0(r_0) = h$. If this holds, the claim is reduced to checking the correctness of $S_i(x)$.

2. **Challenge Generation:** $\mathcal{V}$ samples a random challenge $r_i \in \mathbb{F}_{p^k}$ and sends it to $\mathcal{P}$, proceeding to the next round.

After μ rounds, $\mathcal{V}$ has collected a vector of random challenges $\mathbf{r} = (r_1,\ldots,r_{\mu})$. The final check requires $\mathcal{V}$ to independently open $P(\mathbf{r})$ using PCS and verify that $S_{\mu}(r_{\mu}) = P(\mathbf{r})$. In practice, we utilize the Fiat-Shamir transform [12] to convert this interactive process into a non-interactive one.

2.4 Spielman Code

The Spielman code [48] is a prominent linear-time error-correcting code capable of encoding data with a linear time complexity of $O(n)$, where n is the input size. Due to this asymptotic efficiency, it has been adopted as the core encoding primitive in state-of-the-art PCS schemes such as Brakedown [18] and Orion [57]. As illustrated in Figure 2, the Spielman encoder operates recursively. Crucially, each recursion stage relies on a pair of bipartite expander graphs. Expander graphs are sparse, constant-degree graphs that exhibit strong connectivity. Acting as efficient mixers, they rapidly propagate information to guarantee sufficient code distance.

Mathematically, each bipartite graph corresponds to a sparse transformation matrix, where left vertices map to column indices, right vertices to row indices, and edges represent non-zero entries in the field. Since the expander graphs have a constant degree, the resulting matrices are highly sparse, containing only $O(n)$ non-zero elements. Consequently, the computational core of the Spielman encoder reduces to a series of Sparse Matrix-Vector Multiplication (SpMV) operations. However, a critical implementation challenge arises: unlike conventional SpMV workloads in scientific computing that utilize floating-point or integer arithmetic [1, 14], the SpMV in Spielman encoding operates over large finite fields. This distinct requirement renders existing SpMV libraries inapplicable to our design.

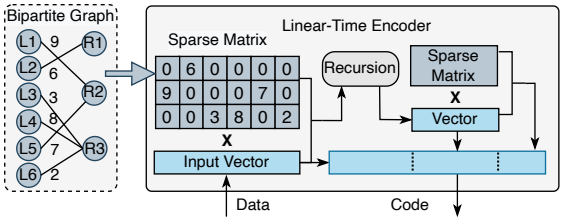


Figure 2: The Spielman encoder using bipartite graphs.

2.5 GPU Architecture and Tensor Cores

Modern GPU architectures are designed for massive parallelism, comprising two distinct types of execution units: general-purpose SIMT cores and specialized Tensor Cores.

The fundamental building blocks of the GPU are its SIMT cores, which function as general-purpose scalar processing units. Unlike domain-specific accelerators, SIMT cores support a comprehensive instruction set architecture (ISA), capable of executing arbitrary logic, including arithmetic, bitwise operations, and complex control flow. However, this generality sacrifices the raw computational density achievable by specialized hardware for structured workloads.

To meet the explosive computational demands of AI, modern GPUs, such as NVIDIA Ampere and Hopper series (A100, H100), as well as AMD's MI300 series based on the CDNA3 architecture, integrate Tensor Cores, specialized hardware units that have become the primary engine behind the AI revolution. In contrast to the independent scalar processing of SIMT cores, Tensor Cores utilize a collective execution model tailored for matrix multiplication. For instance, in a `UINT8` configuration commonly used for AI inference, a warp of 32 threads collaborates to perform a $16 \times 32 \times 8$ matrix multiplication in a single hardware step, accumulating results into `SINT32` registers. Despite their massive throughput, Tensor Cores present a challenge for cryptographic applications: they strictly support low-precision arithmetic (e.g., `FP8`, `UINT8`). This design creates a mismatch with ZKPs, which rely on exact modular arithmetic over finite fields.

Beyond execution units, memory access paradigms also differ between SIMT cores and Tensor Cores. While general-purpose SIMT cores rely on thread-centric instructions to fetch scalar elements, modern architectures utilize the Tensor Memory Accelerator (TMA) to sustain Tensor Cores. The TMA functions as an asynchronous background engine that coordinates the bulk transfer of matrix-structured data between global and on-chip memory, effectively decoupling data movement from thread execution to ensure the high-throughput Tensor Cores maintain full saturation.

3 Technical Overview

We introduce TensorZKP, a holistic hardware-algorithm co-design framework that accelerates ZKP on GPU Tensor Cores. Our primary contribution is resolving the fundamental architectural mismatch between the low-precision matrix engines of Tensor Cores and the computation used in ZKPs. Adapting cryptographic workloads to this specialized hardware necessitates converting algebraic operations into matrix-compatible formats without incurring prohibitive overhead. We address this challenge through a bottom-up design methodology, systematically progressing from matrix-centric field arithmetic to Tensor-Core-compatible ZKP modules. In addition, we architect an asynchronous execution model using warp specialization to maximize heterogeneous GPU utilization.

First, matrix-centric finite field arithmetic. The foundation of TensorZKP is a data representation that bridges the granularity gap between cryptographic requirements and hardware constraints. Standard finite field arithmetic operates on elements in $\mathbb{F}_{p^k}$, which are incompatible with the `UINT8`-based Tensor Cores. A naive adaptation is to encode each field element as multiple `UINT8` limbs for scalar operations, such as field multiplication $a \cdot b \in \mathbb{F}_{p^k}$. However, this approach is inefficient: it introduces fine-grained carry-propagation dependencies and a great amount of moduler reduction.

To resolve this, we shift the optimization scope from individual scalar instructions to field vectors, reformulating algebraic operations into uniform matrix workloads. First, we propose a limb-based vector-to-matrix layout transformation. Formally, we reshape a cryptographic input vector $\mathbf{v} \in (\mathbb{F}_{p^k})^N$ into a dense matrix $\mathbf{V} \in \texttt{UINT8}^{N \times kL}$ by decomposing each field element into kL limbs and stacking them as rows, where L denotes the number of limbs required to represent the prime p. Building on this layout, we map vector primitives to matrix instructions. For the vector inner product, we transform the operation $z = \langle \mathbf{v}, \mathbf{u} \rangle$ into a matrix multiplication $\mathbf{Z} = \mathbf{V}^\top \times \mathbf{U}$. This produces a compact $kL \times kL$ result matrix. In addition, we defer the expensive modulo reduction $\mathcal{M}(\cdot)$ until after the full matrix aggregation, computing $z = \mathcal{M}(\mathbf{Z})$ only once. Similarly, for scalar-vector multiplication $\mathbf{z} = c \cdot \mathbf{v}$, we note that field multiplication can be viewed as linear convolution operations. Therefore, we transform the scalar $c \in \mathbb{F}_{p^k}$ into a structured convolution matrix $\mathbf{C}$ and compute $\mathbf{Z} = \mathbf{V} \times \mathbf{C}$. This redesign converts vector arithmetic into matrix operations. We provide further details in Section 4.

Second, Tensor-Core-Compatible ZKP Modules. Although matrix-centric arithmetic establishes the foundational primitives, applying them to complex ZKP protocols necessitates an algorithm-hardware redesign. We restructures two critical ZKP modules, the sum-check protocol and Spielman coding, to fully exploit Tensor Core capabilities.

1. Tensorized Sum-Check Protocol. Take the sum-check protocol applied to the product of two polynomials as an example, checking the identity $h = \sum_{\mathbf{x} \in \{0,1\}^\mu} f(\mathbf{x}) \cdot g(\mathbf{x})$. The protocol operates on two state vectors, $\mathbf{f}$ and $\mathbf{g}$, initialized with 2^μ evaluations over the Boolean hypercube. It proceeds through μ rounds, where each round i is structured into two distinct phases that map directly to our tensorized primitives. In the evaluation phase, to construct the univariate polynomial $S_i(x)$, we decompose the current state vectors into two components, such as $\mathbf{f}_{even} = \{f_{2k} | 0 \le k < 2^{\mu-1}\}$ and $\mathbf{f}_{odd} = \{f_{2k+1} | 0 \le k < 2^{\mu-1}\}$. To define $S_i(x)$, we compute the necessary evaluations at $\{0,1,2\}$ using tensorized inner products. Specifically, $S_i(0)$ is computed as:

$$S_i(0) = \langle \mathbf{f}_{even}, \mathbf{g}_{even} \rangle.$$

We apply similar vector operations to derive $S_i(1)$ and $S_i(2)$.

Subsequently, in the state update phase, the algorithm advances to the next round by "folding" the state vectors using the verifier's random challenge r_i. This reduces the vector dimension by half via the linear combination:

$$\mathbf{f}^{\langle i \rangle} = (1 - r_i) \cdot \mathbf{f}_{even}^{\langle i-1 \rangle} + r_i \cdot \mathbf{f}_{odd}^{\langle i-1 \rangle}.$$

We implement this folding operation using the tensorized vector-scalar multiplication, treating $(1 - r_i)$ and r_i as convolution matrices. We provide further details in Section 5.1.

2. Tensorized Spielman Code. The Spielman code is mathematically defined as a sequence of Sparse Matrix-Vector Multiplication (SpMV): $\mathbf{y} = \mathbf{G} \cdot \mathbf{x}$. Practical PCS instantiations typically reshape the coefficient vector into a square matrix $\mathbf{W}$ of dimension $\sqrt{N} \times \sqrt{N}$. By treating the columns of $\mathbf{W}$ as a batch of independent vectors to be encoded simultaneously, we upgrade the workload to Sparse Matrix-Matrix Multiplication (SpMM): $\mathbf{Y} = \mathbf{G} \cdot \mathbf{W}$. To execute this operation, we iterate through the regularized adjacency matrix $\mathbf{G}$, where each row contains exactly D non-zero elements. For every row i, we utilize the column indices $\mathbf{idx}^{\langle i \rangle}$ to gather the corresponding D operands from the j-th column of the dense matrix $\mathbf{W}$. This operation forms the gathered vector $\mathbf{w}^{\langle i,j \rangle} = \{W_{k,j} \mid k \in \mathbf{idx}^{\langle i \rangle}\}$, which is subsequently consumed via a length-D tensor-compatible inner product $Y_{i,j} = \langle \mathbf{g}^{\langle i \rangle}, \mathbf{w}^{\langle i,j \rangle} \rangle$. However, a naive execution encounters a severe memory bottleneck. Since computing each entry of the $M \times \sqrt{N}$ output matrix $\mathbf{Y}$ requires fetching D elements, the total global memory access volume scales linearly to $M \times \sqrt{N} \times D$, overwhelming the available bandwidth. To resolve this, we introduce a Cache-Resident Blocking strategy that strictly partitions $\mathbf{W}$ into horizontal panels of size $B \times \sqrt{N}$. By synchronizing SM to process one resident panel at a time, we ensure the active working set satisfies the L2 cache constraint $B \times \sqrt{N} \times \text{sizeof}(\mathbb{F}_{p^k}) < \mathcal{C}_{\text{L2}}$. We provide further details in Section 5.2.

Third, Asynchronous Warp-Specialized Execution. While algorithmic restructuring enables Tensor Core compatibility, realizing end-to-end efficiency requires addressing the disparity between the compute throughput of Tensor Cores and the off-chip memory bandwidth. To improve this, we

design a fine-grained scheduling paradigm based on asynchronous warp specialization. We partition thread warps into distinct functional roles to decouple memory access from computation. Producer warps act as dedicated data movers, utilizing TMA to pre-fetch data tiles from global memory into on-chip memory. Consumer warps function as pure compute engines, consuming this pre-loaded data to feed the Tensor Cores. By pipelining these stages, fetching stage $k+1$ while processing stage k, we effectively hide global memory latency. Furthermore, we maximize hardware occupancy by concurrent execution on heterogeneous cores, utilizing SIMT units for modular reduction while Tensor Cores execute the matrix multiplications. We provide further details in Section 6.

4 Matrix-Centric Finite Field Arithmetic

ZKPs rely heavily on arithmetic over finite fields, typically prime fields $\mathbb{F}_p$ or extension fields $\mathbb{F}_{p^k}$. To ensure λ-bit security, we require the field size to satisfy $\log_2(p^k) \geq \lambda$, where typically $\lambda \in \{128, 256\}$. However, such field arithmetic is fundamentally incompatible with Tensor Cores, which support only low-precision data formats such as `UINT8`.

To overcome this, we shift the optimization target from individual scalar instructions to vector-level operations:

(i) The first operation is the *vector inner product*, where

$$\mathbf{v} = (v_0, v_1, \ldots, v_{N-1}), \mathbf{u} = (u_0, u_1, \ldots, u_{N-1}), v_i, u_i \in \mathbb{F}_{p^k},$$

and the computation is

$$z = \langle \mathbf{v}, \mathbf{u} \rangle = \sum_{i=0}^{N-1} v_i u_i.$$

(ii) The second operation is *vector-scalar multiplication* , where a vector

$$\mathbf{v} = [v_0, v_1, \ldots, v_{N-1}], \qquad v_i \in \mathbb{F}_{p^k}$$

is multiplied by a field scalar $c \in \mathbb{F}_{p^k}$, resulting in

$$\mathbf{z} = c \cdot \mathbf{v} = [c \cdot v_0, c \cdot v_1, \ldots, c \cdot v_{N-1}].$$

We reformulate two vector primitives to enable their acceleration on Tensor Cores. Building on this foundation, Section 5 further details how we reconstruct higher-level ZKP modules, including sum-check protocols and Spielman codes, in terms of these two vector operations. Moreover, because these primitives capture fundamental arithmetic patterns over finite fields, the same reformulation strategy can be extended to accelerate other cryptographic algorithms on Tensor Cores.

4.1 Matrix Construction for Finite Fields

To enable vector arithmetic on Tensor Cores, we must bridge the layout gap by mapping both field scalars and vectors into `UINT8` matrix structures. This layout transformation is essential to adapt the cryptographic operands to the native input requirements of the Tensor Core units.

Encode field elements as multiple `UINT8` limbs. For extension fields $\mathbb{F}_{p^k}$ utilized in ZKP protocols, the cryptographic security level is related to the bit-length of the field size p^k. Without loss of generality, we use a configuration where p is a 32-bit prime and $k = 4$. This yields a 128-bit extension field $\mathbb{F}_{p^4}$, offering sufficient security strength for modern protocols. Consequently, we represent each element $a \in \mathbb{F}_{p^4}$ as a degree-3 polynomial over the base field $\mathbb{F}_p$:

$$a = a^{\langle 0 \rangle} + a^{\langle 1 \rangle} X + a^{\langle 2 \rangle} X^2 + a^{\langle 3 \rangle} X^3, \quad \text{where } a^{\langle s \rangle} \in \mathbb{F}_p, \tag{9}$$

and X denotes the root of the irreducible polynomial $P(X)$.

However, the base field coefficients $a^{\langle s \rangle}$ are 32-bit integers, which exceed the native `UINT8` precision of Tensor Cores. To bridge this gap, we further decompose each 32-bit coefficient into a sequence of 8-bit limbs. Let L denote the number of limbs required to represent the prime p (for a 32-bit prime, $L = 4$). We express each coefficient $a^{\langle s \rangle}$, for $s \in \{0, 1, 2, 3\}$, using a radix-2^8 representation:

$$a^{\langle s \rangle} = \sum_{j=0}^{L-1} a_j^{\langle s \rangle} \cdot 2^{8j}, \quad \text{with } a_j^{\langle s \rangle} \in [0, 2^8 - 1]. \tag{10}$$

Through this encoding, a single high-precision element $a \in \mathbb{F}_{p^4}$ is mapped to $4L$ `UINT8` values, denoted as:

$$[a_0^{\langle 0 \rangle}, \ldots, a_{L-1}^{\langle 0 \rangle}, \ldots, a_0^{\langle 3 \rangle}, \ldots, a_{L-1}^{\langle 3 \rangle}].$$

Vector-to-Matrix Layout Transformation. Consider a vector $\mathbf{a} \in (\mathbb{F}_{p^k})^N$ consisting of N elements, denoted as $\mathbf{a} = [a_0, a_1, \ldots, a_{N-1}]$. To fully exploit matrix arithmetic used in Tensor Cores, we transform this one-dimensional vector structure into a two-dimensional matrix layout suitable for hardware matrix units.

We construct a matrix $\mathbf{A} \in \texttt{UINT8}^{N \times kL}$ by stacking the limb-decomposed representation of each element a_i as a distinct row. Recall that each element $a_i \in \mathbb{F}_{p^k}$ consists of k coefficients, and each coefficient is decomposed into L limbs. Consequently, the matrix $\mathbf{A}$ is defined such that the i-th row corresponds to the i-th element of the vector $\mathbf{a}$, and the columns store the flattened sequence of limbs. Formally, each entry of the matrix $\mathbf{A}$ is given by:

$$A_{i,(sL+j)} = (a_i)_j^{\langle s \rangle}, \tag{11}$$

where $i \in [0, N-1]$ denotes the element index, $s \in [0, k-1]$ denotes the coefficient degree, and $j \in [0, L-1]$ denotes the limb significance.

Visually, this transformation reshapes the abstract vector

into a dense integer matrix:

$$\mathbf{A} = \begin{bmatrix} \text{—} & \text{limbs of } a_0 & \text{—} \\ \text{—} & \text{limbs of } a_1 & \text{—} \\ & \vdots & \\ \text{—} & \text{limbs of } a_{N-1} & \text{—} \end{bmatrix}_{N \times kL} \tag{12}$$

For our specific configuration with $k = 4$ and $L = 4$, each field element occupies one row of 16 `UINT8` columns. This layout aligns with the dimension requirements of Tensor Cores.

Scalar-to-Matrix Layout Transformation. In addition to vector operands, field scalars $c \in \mathbb{F}_{p^k}$ must also be mapped to the matrix domain to leverage Tensor Cores. Recall that our elements are encoded as sequences of limbs. Mathematically, the multiplication of two multi-limb integers is equivalent to the *discrete linear convolution* of their limb sequences. To perform this convolution using hardware matrix multipliers, we transform each constituent base-field coefficient $c^{\langle s \rangle} \in \mathbb{F}_p$ of the scalar c into a structured matrix.

Specifically, let the coefficient $c^{\langle s \rangle}$ be decomposed into L limbs: $[c_0^{\langle s \rangle}, \ldots, c_{L-1}^{\langle s \rangle}]$. To compute the product of each row in the data matrix with this scalar, we construct a scalar matrix $\mathbf{C}^{\langle s \rangle} \in \texttt{UINT8}^{L \times 2L}$. The matrix $\mathbf{C}^{\langle s \rangle}$ encapsulates the shift-and-add logic of convolution by arranging the limbs. Formally, the entry at row u and column v is defined as:

$$\mathbf{C}_{u,v}^{\langle s \rangle} = \begin{cases} c_{v-u}^{\langle s \rangle} & \text{if } 0 \le v - u < L, \\ 0 & \text{otherwise.} \end{cases} \tag{13}$$

With the base-field scalar matrix $\mathbf{C}^{\langle s \rangle}$ defined, we now extend this construction to support the full arithmetic of the extension field $\mathbb{F}_{p^k}$. Multiplication in $\mathbb{F}_{p^k}$ involves multiplying two polynomials: $a(X) = \sum a^{\langle i \rangle} X^i$ and $c(X) = \sum c^{\langle i \rangle} X^i$. The coefficients of the product $d(X) = a(X) \cdot c(X)$ are obtained via the convolution of the coefficient sequences of a and c. Since our data matrix $\mathbf{A}$ stores the coefficients $[a^{\langle 0 \rangle}, \ldots, a^{\langle k-1 \rangle}]$ horizontally, we can compute the polynomial product $d(X)$ by right-multiplying $\mathbf{A}$ with a extension scalar matrix $\mathbf{C}$.

For our specific configuration with $k = 4$, the scalar c consists of coefficients $c^{\langle 0 \rangle}, \ldots, c^{\langle 3 \rangle}$. We construct the global matrix $\mathbf{C}$ by arranging the sub-matrices $\mathbf{C}^{\langle s \rangle}$ as follows:

$$\begin{bmatrix} \mathbf{C}^{\langle 0 \rangle} & \mathbf{C}^{\langle 1 \rangle} & \mathbf{C}^{\langle 2 \rangle} & \mathbf{C}^{\langle 3 \rangle} & \mathbf{0} & \mathbf{0} & \mathbf{0} & \mathbf{0} \\ \mathbf{0} & \mathbf{C}^{\langle 0 \rangle} & \mathbf{C}^{\langle 1 \rangle} & \mathbf{C}^{\langle 2 \rangle} & \mathbf{C}^{\langle 3 \rangle} & \mathbf{0} & \mathbf{0} & \mathbf{0} \\ \mathbf{0} & \mathbf{0} & \mathbf{C}^{\langle 0 \rangle} & \mathbf{C}^{\langle 1 \rangle} & \mathbf{C}^{\langle 2 \rangle} & \mathbf{C}^{\langle 3 \rangle} & \mathbf{0} & \mathbf{0} \\ \mathbf{0} & \mathbf{0} & \mathbf{0} & \mathbf{C}^{\langle 0 \rangle} & \mathbf{C}^{\langle 1 \rangle} & \mathbf{C}^{\langle 2 \rangle} & \mathbf{C}^{\langle 3 \rangle} & \mathbf{0} \end{bmatrix} \tag{14}$$

The dimensions of this block matrix are $kL \times 4kL$. Specifically, for $k = 4$ and $L = 4$, $\mathbf{C}$ is a 16×64 matrix.

4.2 Tensor-Compatible Vector Arithmetic

With the data operands encoded into the matrix domain, we now detail how to execute the two fundamental ZKP primitives using matrix instructions.

Vector inner product. The inner product of two vectors $\mathbf{v}, \mathbf{u} \in (\mathbb{F}_{p^k})^N$ is defined as $z = \langle \mathbf{v}, \mathbf{u} \rangle = \sum_{i=0}^{N-1} v_i \cdot u_i$. In TensorZKP, we reformulate this operation as a matrix multiplication to exploit the linearity of the limb decomposition.

Let $\mathbf{V}, \mathbf{U} \in \texttt{UINT8}^{N \times kL}$ be the matrix representations of vectors $\mathbf{v}$ and $\mathbf{u}$ respectively, constructed via the *Vector-to-Matrix Layout Transformation*. We compute the inner product by executing:

$$\mathbf{Z} = \mathbf{V}^\top \times \mathbf{U}, \tag{15}$$

where the resulting matrix $\mathbf{Z}$ is of size $kL \times kL$.

We can view $\mathbf{Z}$ as a logical $k \times k$ grid of sub-blocks, denoted as $\mathbf{Z}^{(s,t)}$ for $s, t \in [0, k-1]$:

$$\mathbf{Z} = \begin{bmatrix} \mathbf{Z}^{(0,0)} & \mathbf{Z}^{(0,1)} & \ldots & \mathbf{Z}^{(0,k-1)} \\ \mathbf{Z}^{(1,0)} & \mathbf{Z}^{(1,1)} & \ldots & \mathbf{Z}^{(1,k-1)} \\ \vdots & \vdots & \ddots & \vdots \\ \mathbf{Z}^{(k-1,0)} & \mathbf{Z}^{(k-1,1)} & \ldots & \mathbf{Z}^{(k-1,k-1)} \end{bmatrix}_{kL \times kL}. \tag{16}$$

Physically, each sub-block $\mathbf{Z}^{(s,t)}$ of size $L \times L$ captures the cross-products between the coefficient of degree s from the first vector and the coefficient of degree t from the second vector. Consequently, each element in $\mathbf{Z}$ serves by compressing each integer sub-block into a scalar base field element $z^{(s,t)} \in \mathbb{F}_p$:

$$z^{(s,t)} = \left(\sum_{i=0}^{L-1} \sum_{j=0}^{L-1} Z_{i,j}^{(s,t)} \cdot 2^{8(i+j)} \right) \pmod{p}. \tag{17}$$

This operation maps the dense integer matrix into k^2 discrete coefficients. Since the term associated with the pair (s,t) contributes to the polynomial degree $s+t$, the extension field element z can be aggregated by:

$$z = \sum_{s=0}^{k-1} \sum_{t=0}^{k-1} z^{(s,t)} \cdot X^{s+t} \pmod{P(X)}, \tag{18}$$

where $P(X)$ denotes the irreducible polynomial.

By reformulating the inner product as a matrix multiplication task, we not only map this operation to low-precision Tensor Core instructions but also enable lazy modular reduction. Specifically, instead of performing modular reduction on each intermediate vector element, this approach allows us to defer the operation, applying the reduction solely to the final result $z \in \mathbb{F}_{p^k}$ reconstructed from $\mathbf{Z}$.

Vector-scalar multiplication. The multiplication of a vector $\mathbf{v} \in (\mathbb{F}_{p^k})^N$ by a field scalar $c \in \mathbb{F}_{p^k}$ is defined as $\mathbf{z} = c \cdot \mathbf{v} = [c \cdot v_0, c \cdot v_1, \ldots, c \cdot v_{N-1}]$. We reformulate this operation as a matrix multiplication. Let $\mathbf{V} \in \texttt{UINT8}^{N \times kL}$ be the matrix representation of vector $\mathbf{v}$, and let $\mathbf{C} \in \texttt{UINT8}^{kL \times 4kL}$ be the structured matrix derived from the scalar c, constructed via the *Scalar-to-Matrix Layout Transformation*. We compute the operation by executing:

$$\mathbf{Z} = \mathbf{V} \times \mathbf{C}, \tag{19}$$

where the resulting matrix $\mathbf{Z}$ is of size $N \times 4kL$.

Similarly, $\mathbf{Z}$ can be viewed as a logical sequence of $4k$ blocks, denoted as $\mathbf{Z}^{\langle d \rangle}$ for $d \in [0, 4k-1]$:

$$\mathbf{Z} = \left[\mathbf{Z}^{\langle 0 \rangle}, \quad \mathbf{Z}^{\langle 1 \rangle}, \quad \ldots, \quad \mathbf{Z}^{\langle 4k-1 \rangle}\right]. \tag{20}$$

Physically, each blocks $\mathbf{Z}^{\langle d \rangle}$ is a sub-matrix of size $N \times L$, which captures the limb-decomposed integer values for a specific polynomial coefficient term of the product. The reduced value can be represented as $z_i^{\langle d \rangle} \in \mathbb{F}_p$:

$$z_i^{\langle d \rangle} = \left(\sum_{j=0}^{L-1} Z_{i,j}^{\langle d \rangle} \cdot 2^{8j} \right) \pmod{p}. \tag{21}$$

This operation maps the dense integer matrix into $N \times 4k$ discrete coefficients. Since the term associated with the layer d contributes to the polynomial degree corresponding to that layer's index, the final extension field element z_i is aggregated using these scalar values:

$$z_i = \sum_{d=0}^{4k-1} z_i^{\langle d \rangle} \cdot X^d \pmod{P(X)}. \tag{22}$$

5 Tensor-Core-Compatible ZKP Modules

In this section, we propose an algorithm-hardware co-design strategy to adapt core ZKP modules, including sum-check protocols [36] and Spielman coding [48], for Tensor Cores. By restructuring these workloads to utilize the tensorized field arithmetic primitives established in Section 4, we align the algorithmic dataflow with both the compute capability and memory hierarchy of Tensor Cores. In addition, we instantiate these optimizations within HyperPlonk [6] and Brakedown [18] to enable end-to-end proof generation.

5.1 Tensorized Sum-Check Protocol

As detailed in Section 2.3, the sum-check protocol [36] proves that an aggregated sum over the Boolean hypercube matches a claimed value h. We first address that the protocol verifies the identity:

$$h = \sum_{\mathbf{x} \in \{0,1\}^\mu} f(\mathbf{x}) \cdot g(\mathbf{x}).$$

The computational bottleneck of the sum-check protocol lies in the generation of the univariate polynomial $S_i(x)$ at each round $i \in [1, \mu]$, as defined in Eq. 8. While the state-of-the-art approach [50] achieves an optimal time complexity of $O(N)$, where $N = 2^\mu$, its design is inherently incompatible with Tensor Core architectures. Building on this algorithmic foundation, we propose an improved algorithm realized through tensor-compatible vector arithmetic over finite fields. This adaptation preserves the original $O(N)$ complexity while unlocking efficient execution on Tensor Cores.

Algorithm 1 Tensorized Sum-check Module

Input: Initial vectors $\mathbf{f}^{\langle 0 \rangle}$ and $\mathbf{g}^{\langle 0 \rangle}$ with $f_b^{\langle 0 \rangle} = f(b_1, \ldots, b_\mu)$ and $g_b^{\langle 0 \rangle} = g(b_1, \ldots, b_\mu)$, where b_i is i-th bit of b, and $f_b^{\langle 0 \rangle}, g_b^{\langle 0 \rangle} \in \mathbb{F}_{p^k}$. Random elements $r_1, r_2, \ldots, r_\mu \in \mathbb{F}_{p^k}$.

Output: $\mathbf{S} = \{(S_i(0), S_i(1), S_i(2)) \mid i \in [1, \mu]\}$, where the triplet $(S_i(0), S_i(1), S_i(2))$ uniquely determines the degree-2 polynomial $S_i(x)$ via interpolation.

for $i = 1, 2, \ldots, n$ **do**
  $t = 2^{\mu - i}$
  # Decompose vectors into even/odd halves.
  $\mathbf{f}_{even}^{\langle i-1 \rangle} = \{f_0^{\langle i-1 \rangle}, f_2^{\langle i-1 \rangle}, \ldots, f_{2t-2}^{\langle i-1 \rangle}\}$
  $\mathbf{f}_{odd}^{\langle i-1 \rangle} = \{f_1^{\langle i-1 \rangle}, f_3^{\langle i-1 \rangle}, \ldots, f_{2t-1}^{\langle i-1 \rangle}\}$
  $\mathbf{g}_{even}^{\langle i-1 \rangle} = \{g_0^{\langle i-1 \rangle}, g_2^{\langle i-1 \rangle}, \ldots, g_{2t-2}^{\langle i-1 \rangle}\}$
  $\mathbf{g}_{odd}^{\langle i-1 \rangle} = \{g_1^{\langle i-1 \rangle}, g_3^{\langle i-1 \rangle}, \ldots, g_{2t-1}^{\langle i-1 \rangle}\}$
  # Compute $S_i(x)$ via vector inner product.
  $S_i(0) = \langle \mathbf{f}_{even}^{\langle i-1 \rangle}, \mathbf{g}_{even}^{\langle i-1 \rangle} \rangle$
  $S_i(1) = \langle \mathbf{f}_{odd}^{\langle i-1 \rangle}, \mathbf{g}_{odd}^{\langle i-1 \rangle} \rangle$
  $S_i(2) = 4S_i(0) + S_i(1) - 2\langle \mathbf{f}_{odd}^{\langle i-1 \rangle}, \mathbf{g}_{even}^{\langle i-1 \rangle} \rangle - 2\langle \mathbf{f}_{even}^{\langle i-1 \rangle}, \mathbf{g}_{odd}^{\langle i-1 \rangle} \rangle$
  # Fold vectors via vector-scalar multiplication.
  $\mathbf{f}^{\langle i \rangle} = (1 - r_i) \cdot \mathbf{f}_{odd}^{\langle i-1 \rangle} + r_i \cdot \mathbf{f}_{even}^{\langle i-1 \rangle}$
  $\mathbf{g}^{\langle i \rangle} = (1 - r_i) \cdot \mathbf{g}_{odd}^{\langle i-1 \rangle} + r_i \cdot \mathbf{g}_{even}^{\langle i-1 \rangle}$
end for
return $\mathbf{S} = \{(S_i(0), S_i(1), S_i(2)) \mid i \in [1, \mu]\}$

Algorithm 1 outlines the detailed execution of the tensorized sum-check protocol for the product of two polynomials. The procedure initializes with two bookkeeping vectors, $\mathbf{f}^{\langle 0 \rangle}$ and $\mathbf{g}^{\langle 0 \rangle}$, which contain 2^μ evaluations over the Boolean hypercube. The algorithm then iterates through μ rounds, maintaining these vectors as they strictly halve in size at each step. Within each round $i \in [1, \mu]$, the workflow is structured into two sequential phases. First, the algorithm performs Evaluation Generation by logically decomposing the current vectors into "even" and "odd" components, such as $\mathbf{f}_{even} = \{f_{2k} | 0 \le k < 2^{\mu-1}\}$ and $\mathbf{f}_{odd} = \{f_{2k+1} | 0 \le k < 2^{\mu-1}\}$. These sub-vectors are then operated upon to compute the interpolation points for the round polynomial $S_i(x)$, specifically $S_i(0)$, $S_i(1)$, and $S_i(2)$, by calculating inner products. Crucially, these computationally intensive reductions are executed using the Tensor-Compatible Vector Inner Product primitive. Subsequently, the algorithm proceeds to the state update phase to advance to the next round. Here, the vectors are folded using the verifier's random challenge r_i, reducing the problem size by half via a linear combination $\mathbf{f}^{\langle i \rangle} = (1 - r_i) \cdot \mathbf{f}_{even}^{\langle i-1 \rangle} + r_i \cdot \mathbf{f}_{odd}^{\langle i-1 \rangle}$. This folding operation is efficiently implemented using the Tensor-Compatible Vector-Scalar Multiplication primitive, ensuring the data remains aligned for the subsequent round.

Practical Lazy Modular Reduction on GPUs. As detailed in Section 4, we execute the tensor-compatible vector inner product, such as $S_i(0) = \langle \mathbf{f}_{even}^{\langle 0 \rangle}, \mathbf{g}_{even}^{\langle 0 \rangle} \rangle$ using the matrix multiplication $\mathbf{S}_0 = \mathbf{F}_{even}^{\top} \times \mathbf{G}_{even}$, where $\mathbf{F}_{even}$ and $\mathbf{F}_{even}$ is $\texttt{UINT8}^{N \times kL}$. $\mathbf{S}_0$ is stored in $\texttt{SINT32}^{kL \times kL}$. The selection of SINT32 as the accumulation format is dictated by the architecture of Tensor Cores, which are designed to accumulate products of low-precision operands into 32-bit registers.

Given that the input limbs are strictly bounded by $2^8 - 1$, the maximum value for a single product term is $(255)^2$. Consequently, the accumulated sum in any entry of the result matrix is bounded by $N \cdot (255)^2$. To ensure numerical integrity within the SINT32 accumulator, the vector length N must satisfy:

$$N \cdot (255)^2 < 2^{31} - 1 \implies N < 33,025. \tag{23}$$

This bound implies that for vector chunks up to approximately 3.3×10^4 elements, matrix multiplication can be performed entirely over the integers without information loss. This property allows us to bypass expensive modular reduction operations during the accumulation loop. Instead, we perform modular reduction only once.

However, in practice, PIOP instantiations often involve vectors of massive magnitude. The polynomial size N typically exceeds the derived safety threshold of $33,025$. To address this scalability challenge without sacrificing numerical stability, we employ a parallel block-wise accumulation strategy. We partition the large input vectors into sub-segments of fixed size B (e.g., $B = 32,768$) that strictly satisfy the overflow bound. This decomposition enables a natural mapping to the GPU's hardware parallelism, specifically utilizing its Streaming Multiprocessors (SMs). An SM is a self-contained execution unit within the GPU, equipped with its own set of Tensor Cores and shared memory. By assigning each safe sub-segment to different SMs, we process these chunks concurrently across the device. Finally, the global inner product is computed by aggregating the partial results using SINT64, performing the modular reduction at the final stage.

Adapting Higher-Degree Sum-Check. In HyperPlonk's execution trace, the sum-check protocol for multiplication gates involves degree-3 terms, with $q_M(\mathbf{x})w_a(\mathbf{x})w_b(\mathbf{x})$. Since the Tensorized Sum-Check Protocol detailed in Algorithm 1 is natively tailored for degree-2 operations, we must reduce the polynomial degree before processing. To handle degree-3 terms involving polynomials f, g, h, we first aggregate two of the multiplicands using an element-wise multiplication, which produces a temporary vector $\mathbf{t} = \mathbf{g} \circ \mathbf{h}$. Next, the univariate polynomial $S_i(x)$ is computed using $\mathbf{f}$ and $\mathbf{t}$. The vector folding process remains unchanged for $\mathbf{f}$, $\mathbf{g}$ and $\mathbf{h}$, utilizing the same vector-scalar multiplication with the random challenges.

5.2 Tensorized Spielman Code

Section 2.4 details the encoding process for the Spielman code. Distinguished by its linear time complexity of $O(N)$, this error-correcting code serves as the foundational primitive for state-of-the-art PCS schemes such as Brakedown [18] and Orion [57]. Mathematically, the encoding pipeline consists of a sequence of linear transformations over a finite field $\mathbb{F}_{p^k}$. In a standard setting, this reduces to a series of Sparse Matrix-Vector Multiplications (SpMV): $\mathbf{y} = \mathbf{G} \cdot \mathbf{x}$, where $\mathbf{G}$ represents the sparse adjacency matrix derived from the bipartite expander graph used in the Spielman code.

Practical PCS instantiations allow us to optimize this further. As detailed in Section 2.2.2, the coefficient vector of the polynomial w is typically reshaped into a square matrix $\mathbf{W}$ of dimension $\sqrt{N} \times \sqrt{N}$, where N denotes the polynomial size. Consequently, rather than encoding a single vector, we encode multiple rows simultaneously. This data layout enables us to perform the encoding in batch, upgrading the workload to Sparse Matrix-Matrix Multiplication (SpMM): $\mathbf{Y} = \mathbf{G} \cdot \mathbf{W}$.

Crucially, our tensorized SpMM differs from conventional sparse linear algebra workloads in two fundamental aspects:

- **Arithmetic Domain:** Unlike scientific computing kernels that utilize floating-point or integer arithmetic, all elements in this setting are performed over the extension field $\mathbb{F}_{p^k}$. This distinct arithmetic requirement leads to state-of-the-art SpMM libraries like cuSPARSE [39] inapplicable.
- **Structural Regularity:** The sparsity pattern of $\mathbf{G}$ is dictated by the underlying bipartite expander graph. Unlike general sparse matrices with arbitrary non-zero distributions, $\mathbf{G}$ can exhibit a fixed row sparsity [22], where every row contains exactly D non-zero elements, corresponding to the graph's degree D. This strict regularity allows us to bypass the load-balancing issues typical of SpMM.

Algorithm 2 details the tensorized SpMM tailored for the Spielman code. To align with the constant-degree property of the expander graph, $\mathbf{G}$ is stored in a regularized ELL-like format. Specifically, for each row i, the non-zero weights are packed into a vector $\mathbf{g}^{\langle i \rangle}$, and their column coordinates are stored in $\mathbf{idx}^{\langle i \rangle}$, both of fixed length D. The algorithm iterates through the output matrix $\mathbf{Y}$ to compute each entry $Y_{i,j}$. First, it performs a gather operation, where the column indices $\mathbf{idx}^{\langle i \rangle}$ are used to fetch the corresponding operands from the matrix $\mathbf{W}$ to the gathered vector $\mathbf{w}^{\langle i,j \rangle} = \{W_{k,j} \mid k \in \mathbf{idx}^{\langle i \rangle}\}$. Subsequently, the algorithm executes the compute stage by calculating the Tensor-Compatible Vector Inner Product between $\mathbf{g}^{\langle i \rangle}$ and the gathered vector $\mathbf{w}^{\langle i,j \rangle}$.

The Memory Bottleneck. While transforming the core arithmetic of the Spielman code into tensor-compatible operations is algorithmically straightforward, the decisive factor for hardware performance lies in memory bottleneck. A naive implementation of Algorithm 2 treats each inner product independently. To compute the full result matrix $\mathbf{Y}$ of size

Algorithm 2 Tensorized SpMM in the Spielman Code

Input: A sparse matrix $\mathbf{G}$ with M rows. For each row $i \in [0, M-1]$, we define $\mathbf{g}^{\langle i \rangle}$ as the vector containing the non-zero elements and $\mathbf{idx}^{\langle i \rangle}$ as the vector of their corresponding column indices, where the constant D denotes the fixed number of non-zero elements per row. $\mathbf{W}$ is a square matrix of dimension $\sqrt{N} \times \sqrt{N}$. All entries in both matrices belong to the extension field $\mathbb{F}_{p^k}$.

Output: A result matrix $\mathbf{Y}$ of dimension $M \times \sqrt{N}$.

for $i = 0, 1, \ldots, M-1$ **do**
  for $j = 0, 1, \ldots, \sqrt{N}-1$ **do**
    # Fetch $\mathbf{W}$ values via $\mathbf{idx}$.
    $\mathbf{w}^{\langle i,j \rangle} = \left[W_{idx_0^{\langle i \rangle}, j}, W_{idx_1^{\langle i \rangle}, j}, \ldots, W_{idx_{D-1}^{\langle i \rangle}, j} \right]$
    # Compute Y via vector inner product.
    $Y_{i,j} = \langle \mathbf{g}^{\langle i \rangle}, \mathbf{w}^{\langle i,j \rangle} \rangle$
  end for
end for
return $\mathbf{Y}$

$M \times \sqrt{N}$, the system must perform a gather operation for $\mathbf{W}$. Specifically, calculating a single entry of $\mathbf{Y}$ requires fetching D non-contiguous elements from $\mathbf{W}$. Consequently, the total volume of data read from global memory scales to:

$$\text{Total Access Volume} = M \times \sqrt{N} \times D \tag{24}$$

This creates a severe discrepancy: the total data requested is orders of magnitude larger than the actual storage footprint of the matrix $\mathbf{W}$, where $D \cdot M \gg \sqrt{N}$ in Spielman configurations. If these requests are served from global memory, the latency will stall the Tensor Cores.

L2 Cache Utilization on GPUs. To employ L2 cache on GPUs, we fundamentally restructure the access pattern of the matrix $\mathbf{W}$. Since the full matrix dimensions ($\sqrt{N} \times \sqrt{N}$) generate a memory footprint exceeding the L2 cache capacity, we adopt a Cache-Resident Blocking strategy. Instead of treating $\mathbf{W}$ as a monolithic entity, we partition it into sub-blocks of dimension $B \times \sqrt{N}$ and process them iteratively. In this execution model, all SMs synchronize to operate on the currently resident $B \times \sqrt{N}$ block. Only after the computations for this specific batch are fully resolved does the system proceed to load and process the subsequent $B \times \sqrt{N}$ block. The block height B serves as a decisive tuning parameter and is rigorously selected to satisfy the hardware constraint $B \times \sqrt{N} \times \text{sizeof}(\mathbb{F}_{p^k}) < \mathcal{C}_{\text{L2}}$, where $\mathcal{C}_{\text{L2}}$ represents the total L2 cache capacity of the target GPU. By enforcing this bound, we guarantee that the active working set fits entirely within the high-bandwidth L2 cache throughout the computation.

5.3 End-to-End Proof Generation

Benefiting from the modular design of our proposed framework, integrating the optimized modules into the end-to-end proof generation pipeline, comprising HyperPlonk [6] as the PIOP and Brakedown [18] as the PCS, is straightforward. We achieve this by directly substituting the standard sum-check protocols and Spielman coding with our Tensor-Core-compatible counterparts. For the Merkle Tree commitment phase within Brakedown. In contrast to the linear encoding step, cryptographic hash functions rely heavily on non-linear bitwise operations, including logical shifts and S-box substitutions, which do not map efficiently to matrix multiplication operations. We adopt a existing, highly optimized SIMT-based GPU implementations [34] for the hashing workload.

6 Asynchronous Warp-Specialized Execution

In addition to adapting ZKP algorithms for Tensor Cores, we introduce system-level optimizations to ensure end-to-end efficiency. These measures are necessitated by the significant disparity between Tensor Core throughput and memory bandwidth. Since Tensor Cores process data orders of magnitude faster than standard SIMT cores, unoptimized designs risk hitting the "memory wall", resulting in hardware underutilization. Thus, establishing an efficient execution model that balances memory access with on-chip computation is critical.

To address this problem, we design a fine-grained scheduling paradigm based on Asynchronous Warp Specialization. Unlike traditional bulk-synchronous kernels where all threads stall during memory fetches, we partition the warps into two distinct functional roles:

Producer Warps. A dedicated subset of warps functions exclusively as data movers, tasked with managing TMA descriptors to fetch fixed-size data tiles from global memory into shared memory. For a vector $\mathbf{v}$ of size N over $\mathbb{F}_{p^k}$, the producer warps load a block of size B at each pipeline stage. By decoupling memory operations from computation, these warps minimize register pressure and maintain sufficient in-flight memory transactions.

Consumer Warps. The remaining warps act as pure computational engines. They read pre-fetched data directly from the low-latency on-chip memory buffer to execute compute-intensive Tensor Core matrix multiplications. Because the Producers actively pre-fill the pipeline, loading stage $k+1$ while stage k is being processed, the Consumer warps are effectively decoupled from global memory latency. This ensures a continuous stream of valid data.

Beyond memory-compute overlapping, we further maximize hardware occupancy by enabling concurrent execution of heterogeneous core types. Specifically, we employ Tensor Cores for the primary matrix operations while simultaneously utilizing SIMT cores for modular reduction, allowing for asynchronous execution across different compute units.

7 Evaluation

In this section, we provide a comprehensive evaluation of TensorZKP. We begin by outlining our experimental setup in Section 7.1. Subsequently, we benchmark the performance of matrix-centric finite field arithmetic in Section 7.2 and individual ZKP modules in Section 7.3, followed by an analysis of the end-to-end system efficiency in Section 7.4.

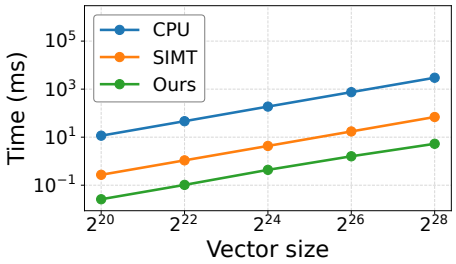
7.1 Experimental Setup

Setup. We implement TensorZKP in the C++/CUDA language and compile it with nvcc 12.6. All GPU experiments are conducted on an NVIDIA H100 equipped with 80 GB of high-bandwidth memory (HBM). Although our primary benchmarks target the H100 architecture, the implementation is compatible with Tensor-Core–enabled GPUs, such as the NVIDIA A100. CPU baselines are run on Xeon® Platinum 8480+. We utilize the BabyBear prime field extended to degree $k = 8$ as our underlying finite field. This configuration yields a security level of more than 200 bits. Specifically, we use the base prime $p = 2^{31} - 2^{27} + 1$ and define the extension via the irreducible polynomial $P(x) = x^8 - 11$ over $\mathbb{F}_p$.

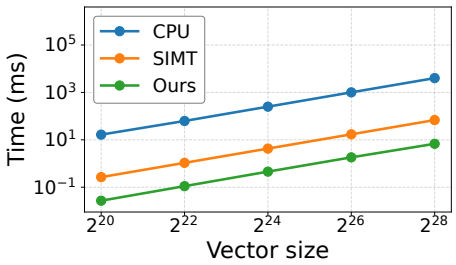
Baselines. We compare TensorZKP against state-of-the-art implementations on both CPUs and GPUs. For GPU baselines, we utilize Ingonyama's Icicle [24], a production-grade library optimized for ZKP acceleration [23], targeting the sum-check protocol. We also compare against BatchZK [34], the first scheme to implement batched Spielman codes on GPUs. Crucially, these existing GPU baselines rely on standard SIMT instructions. In contrast, our implementation is explicitly designed to leverage the higher throughput of Tensor Cores. Regarding CPU comparisons, the absence of a functional Brakedown implementation led us to adopt Orion [51] as our primary CPU baseline for PCS. Orion provides a robust implementation of Spielman codes and serves as a faithful performance proxy for the Brakedown-style PCS. We adopt the HyperPlonk protocol used in DeepFold [20] as our CPU-based PIOP. Both DeepFold and Orion are single-threaded reference implementations released by their original authors; we use them as published without additional multi-threaded modification to preserve the reported behavior. We focus on prover runtime as the main bottleneck. Since we only accelerate prover-side computation without changing the protocol, proof size and verifier time are identical to those of the standard CPU-based HyperPlonk+Brakedown implementation.

7.2 Evaluating Field Arithmetic

We initiate our evaluation by micro-benchmarking the two core primitives designed in Section 4: the Tensor-Compatible Vector Inner Product and Vector-Scalar Multiplication. All experiments utilize a consistent finite field implementation, with input vector sizes N ranging from 2^{20} to 2^{28}. We benchmark our Tensor Core approach against two baselines: a CPU implementation and a standard CUDA SIMT kernel. **Vector Inner Product.** Figure 3a presents the throughput comparison for computing the vector inner product $\langle \mathbf{v}, \mathbf{u} \rangle$. As the results indicate, our Tensor Core-based primitive delivers substantial performance gains, achieving a speedup of up to 563× over the CPU baseline and 13.1× over the SIMT baseline. At a 2^{25} scale, our implementation completes one inner product in only 0.85 ms. **Vector-Scalar Multiplication.** Figure 3b details vector-scalar multiplication, $\mathbf{z} = c \cdot \mathbf{v}$. We observe speedups of 613× and 10.0× over CPU and SIMT-based implementations, with one multiplication finishing in 0.91 ms at a 2^{25} scale.



(a) Vector Inner Product



(b) Vector-Scalar Multiplication

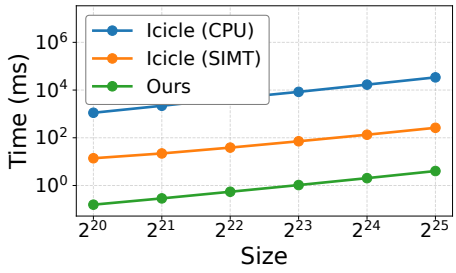
Figure 3: The performance of field arithmetic.

7.3 Evaluating ZKP Modules

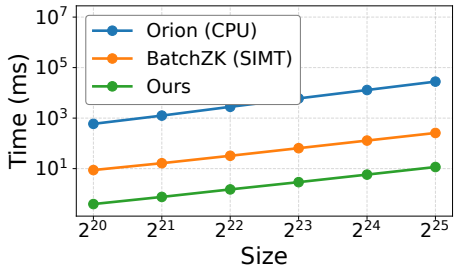
We benchmark the performance of our pipelined ZKP modules, including the Merkle tree, sum-check protocol, and linear-time encoder. For each module, we compare the state-of-the-art CPU and GPU implementations.

Sum-check. Figure 4a gives the evaluation results of our sum-check modules. This experiment benchmarks the performance for generating sum-check proofs for the the product of two polynomials $h = \sum_{\mathbf{x} \in \{0,1\}^\mu} f(\mathbf{x}) g(\mathbf{x})$, with the number of variables μ ranging from 20 to 25. We compare our scheme to Icicle [24], which has both CPU and GPU implementations.

Our sum-check module demonstrates superior performance. We achieved a speedup of more than 8,435× compared to Icicle's CPU-based implementation, and more than 88× over Icicle's GPU implementation. At a 2^{25} scale, our sum-check completes in only 4.04 ms. Icicle [24] is an industrial product

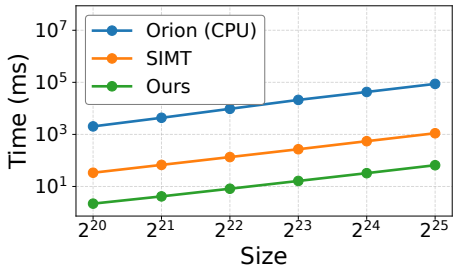


(a) Sum-Check Module

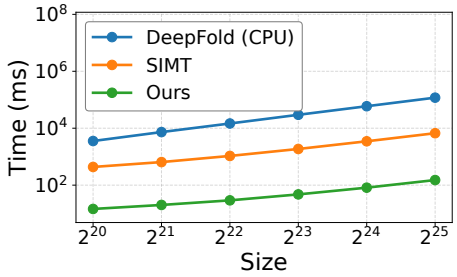


(b) Spielman Coding

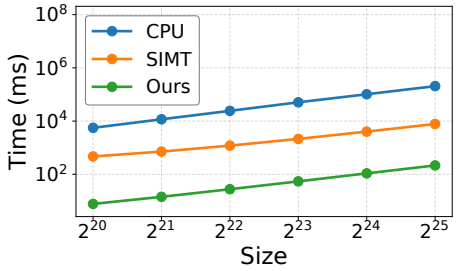
Figure 4: The performance of ZKP modules.



(a) PCS



(b) PIOP



(c) Total

Figure 5: The performance of ZKP system.

developed by Ingonyama [23], which focuses on the acceleration of ZKPs. Therefore, achieving such a speedup means that our implementation has great commercial value.

Linear-time encoder. Figure 4b gives the evaluation results of our linear-time encoder modules. This experiment benchmarks the performance for encoding N finite field elements. To faithfully replicate the encoding scheme in PCS, we reshape the input vector of size N into a square matrix of dimension $\sqrt{N} \times \sqrt{N}$. For input sizes, where $\log_2 N$ is odd, we decompose the dimensions into the nearest integer powers, such as $2^{12} \times 2^{11}$ for $N = 2^{23}$. We achieve speedups of more than $2,407\times$ compared to the CPU-based Spielman implementation in Orion [57], and more than $22\times$ over the state-of-the-art SIMT-based GPU baseline, BatchZK [34]. At a 2^{25} scale, one encoding completes in only 11.58 ms.

7.4 Evaluating ZKP System

This section evaluates the end-to-end performance of our ZKP system, employing a proving stack that integrates HyperPlonk [6] as the PIOP backend and Brakedown [18] as the PCS. We benchmark our system against two hybrid baselines: a GPU baseline utilizing Icicle [24] for the HyperPlonk's sum-check module and BatchZK [34] for Spielman coding, representing state-of-the-art SIMT performance, as well as a CPU baseline that integrates DeepFold's HyperPlonk implementation [20] for the PIOP and Orion [57] for the PCS.

Figure 5 gives the evaluation results, where the size denotes the number of multiplication gates in the circuit. In the experiments, we assess the time required for producing each proof for the circuits with S multiplication gates, where S ranges from 2^{20} to 2^{25}. The evaluation results show that our system is very efficient. For a circuit with 2^{25} multiplication gates, TensorZKP achieves a proof generation time of only 215.28 milliseconds, representing a $955\times$ speedup over the CPU baseline and a $36.2\times$ improvement over state-of-the-art SIMT-based GPU implementations.

Beyond the end-to-end headline, Figure 5a and Figure 5b report the per-component speedups. For the Brakedown PCS commit phase, TensorZKP attains up to $1{,}329\times$ over the Orion CPU baseline and $16.9\times$ over the SIMT-GPU baseline. For the HyperPlonk PIOP, which combines the degree-3 mul-gate sum-check with the permutation product-check, TensorZKP attains up to $785\times$ over the DeepFold CPU baseline and $44.3\times$ over the SIMT-GPU baseline. Both module-level speedups grow monotonically with circuit size, indicating that the Tensor-Core advantage compounds as the matrix-multiplication workload increasingly dominates the prover.

Performance breakdown. Table 1 reports a per-component breakdown of the prover at 2^{25}. Within the PCS, the Spielman encoder dominates, followed by the Merkle

Table 1: Performance breakdown at 2^{25} (ms).

Stage	Merkle	Encoder	Open	Perm-Check	Zero-Check	Subtotal
PCS	8.59	50.64	6.30	–	–	65.53
PIOP	–	–	–	134.00	16.96	150.96

Table 2: Real-world circuits (batch size = 64).

Circuit	SIMT (ms)	Ours (ms)	vs. SIMT
AES-128 (113,762 gates)	2,400	99.8	24.0×
SHA-256 (301,634 gates)	8,052	233.0	34.6×

tree and the polynomial evaluation. Within the PIOP, the permutation product-check dominates the zero-check. Two observations follow. First, Merkle hashing, the only hash-bound component in our pipeline, accounts for less than 5% of the total proving time, confirming that arithmetic, rather than hashing, remains the dominant cost on Tensor Cores. Second, the chunking overhead arising from accumulation-bound limb decompositions in our permutation-check and zero-check kernels stays below 2% across all evaluated scales, with no growth observed from 2^{20} to 2^{25}. Together, these two observations suggest that further speedups will come from further saturating the Tensor-Core pipeline rather than from optimizing hashing or limb arithmetic.

Real-world applications. To validate TensorZKP on circuits drawn from real-world workloads rather than synthetic benchmarks, Table 2 reports end-to-end proving time on AES-128 (113,762 gates/block) and SHA-256 (301,634 gates/block) circuits in batched mode. TensorZKP delivers 24× and 35× speedups over the SIMT-GPU baseline on AES-128 and SHA-256, respectively, with proof generation completing in roughly 100 and 233 ms.

Comparison against ASIC and FPGA accelerators. We contrast TensorZKP with two recent dedicated-silicon ZKP accelerators: zkSpeed [9], a 22 nm ASIC for HyperPlonk, and HKPR [22], an FPGA design on Xilinx Alveo U280 for an Orion-style code-based PCS. Table 3 compares the prover-side time at 2^{24}. Despite being a programmable general-purpose platform, TensorZKP outperforms the 22 nm ASIC on PIOP (1.2×) and the FPGA on PCS (9.8×), indicating that the Tensor-Core formulation is competitive with — or superior to — dedicated silicon while preserving deployment flexibility.

Memory footprint. Our matrix-centric layout incurs no asymptotic overhead: an N-element field vector occupies exactly $N \cdot kL$ bytes (Section 4), identical to the limb-decomposed representation used by SIMT baselines. The ∼2× constant-factor overhead in Table 4 reflects the in-flight pipeline buffers of our producer–consumer warp specialization (Section 6); the end-to-end footprint remains well within the H100's 80 GB HBM budget.

Resource-normalized comparison. To address the concern that wall-clock comparisons conflate raw performance

Table 3: ASIC/FPGA comparison at 2^{24}.

Scheme	Component	Hardware	Time (ms)
zkSpeed [9]	PIOP	ASIC (22 nm)	97.30
Ours	PIOP	GPU (H100)	**81.58**
HKPR [22]	PCS	FPGA (U280)	316.30
Ours	PCS	GPU (H100)	**32.44**

Table 4: Memory footprint (GB).

$\log_2 N$	Ours (GB)	SIMT-based (GB)
2^{22}	15.1	6.4
2^{25}	52.9	28.2

with hardware capacity, Table 5 reports throughput per Watt and per dollar, normalized to a single-core CPU baseline. Hardware power and price are taken from publicly priced cloud instances, namely GCP `c4d-highmem-16` for the CPU and Vast.ai H100 SXM5 for the GPU. Even after this normalization, TensorZKP delivers 5.1× per Watt and 53.6× per dollar, confirming that the speedup stems from genuine throughput-per-resource gains rather than from using a larger device.

Scale limit. Our evaluation extends up to a circuit size of 2^{25} multiplication gates, at which the end-to-end memory footprint already reaches about 53 GB on a single H100 (Table 4). Beyond 2^{25}, the pipeline buffers and intermediate matrices exceed the 80 GB HBM budget of a single device and trigger an out-of-memory failure, which prevents us from reporting larger single-GPU numbers. A multi-GPU version is left to future work.

8 Future Work

In this work, we introduced TensorZKP, a hardware-algorithm co-design framework that leverages GPU Tensor Cores to accelerate ZKPs. TensorZKP successfully demonstrates linear-time proof generation for sum-check protocols, which are integral to PIOPs such as Libra [55], Spartan [47], and HyperPlonk [6], as well as Spielman coding for PCS including Orion [57] and Brakedown [18].

These systems achieve optimal $O(N)$ prover time complexity, they inherently incur a trade-off regarding proof size, yielding larger proofs compared to cryptographic pairing-based schemes. Extending the TensorZKP to support the primitives required for these succinct proof systems, specifically those achieving constant $O(1)$ proof sizes like the combination of Plonk [13] with KZG commitments [27], presents a significant opportunity for future exploration. Therefore, two promising research directions include:

- **Tensor-Compatible MSM:** The computational bottleneck of KZG commitments is Multi-Scalar Multiplication (MSM) over elliptic curves. Mapping MSM to Tensor Cores

Table 5: Per-Watt and per-dollar throughput at 2^{25}.

Scheme	per-Watt	per-dollar
CPU	1.2×10^{-3} (1×)	146 (1×)
SIMT-GPU	1.8×10^{-4} (0.15×)	226 (1.55×)
Ours	$\mathbf{6.4\times10^{-3}}$ **(5.1×)**	**7,832 (53.6×)**

is non-trivial because elliptic-curve point operations involve line-curve intersections and frequent carry propagation, both of which break the regular dataflow expected by Tensor Cores. A promising direction is to adopt Twisted Edwards curves with Extended (projective) Coordinates, which unify point addition and doubling into a single formula and expose a more regular dataflow. Even so, multi-precision carry propagation still requires SIMT cores, suggesting a heterogeneous design in which Tensor Cores handle the field-multiplication kernels while SIMT cores manage carry chains and curve-specific control flow.

- **Tensor-Compatible FFT:** Protocols using FRI and Plonk-style PIOPs rely heavily on Fast Fourier Transforms (FFT). The fundamental obstacle is that each butterfly $x = e + \omega \cdot o$ couples a unique twiddle factor ω to a single pair of operands, which resists direct matrix-multiplication mapping. One promising opening is the batched FFT setting that arises in proof aggregation and recursive composition: sharing the twiddle factor ω across many independent inputs yields $\mathbf{x} = \mathbf{e} + \omega \cdot \mathbf{o}$, which is precisely the Tensor-Compatible vector-scalar primitive developed in Section 4. This observation suggests that the techniques in this paper can be reused to accelerate the butterfly stage of batched FFTs without redesigning the underlying primitive.

9 Conclusion

We introduce TensorZKP, a novel framework that unlocks the untapped potential of Tensor Cores for ZKPs, moving beyond traditional SIMT-based approaches. By redesigning sum-check protocols and linear-time encoders to fit the matrix-multiplication paradigm of Tensor Cores, and introducing a warp-specialized memory management design, TensorZKP overcomes the inherent limitations of low-precision hardware. Validated through end-to-end proof generation with HyperPlonk and Brakedown, our framework delivers a substantial performance improvement over existing GPU baselines.

Acknowledgments

This work was supported in part by the CRPO under WBS A-8004052-01-00 and in part by the Tier 2 grant MOE-T2EP20125-0015. We also thank Yuxiang Qiu for his valuable help during the early stage of this work.

Ethical Considerations

Stakeholders. We distinguish direct stakeholders from indirect stakeholders. Direct stakeholders include the authors and the organizations that integrate TensorZKP into ZKP-based applications, such as blockchain rollups, privacy-preserving payment systems, and verifiable machine learning pipelines. Indirect stakeholders include end users whose data or assets rely on these platforms, regulators and auditors responsible for oversight, and adversaries who may attempt to abuse or subvert such systems. Our work involves no human subjects, no physical experiments, and no collection of real-world private data. All benchmarks are conducted on synthetic finite-field workloads and on standard cryptographic circuits (AES-128 and SHA-256) using publicly available open-source baselines.

Ethical Principles (Menlo Report). We considered the four principles of the Menlo Report in turn. *Beneficence*: TensorZKP reduces the computational cost of verifiable computation, making privacy-preserving and integrity-preserving systems more practical without requiring access to users' raw data. *Respect for Persons*: we cite prior work for accurate attribution and use third-party baselines in accordance with their licenses. *Justice*: TensorZKP targets commodity NVIDIA GPUs and is released as open source, with no proprietary cryptographic dependencies, so that the benefits remain accessible to researchers and developers worldwide. *Respect for Law and Public Interest*: our research complies with relevant academic and cryptographic research policies, and advancing ZKP efficiency serves the public interest by enabling more trustworthy digital infrastructure.

Potential Harms and Mitigations. Faster ZKP proving could in principle be misused by illicit actors to conceal unlawful activities within privacy-preserving systems, and immature or misconfigured deployments could expose honest users to financial loss, privacy breaches, or service disruption. To mitigate these risks, we (i) document our hardware assumptions and parameter choices, including the BabyBear field extension and the Spielman code parameters, to guide responsible adoption; (ii) preserve, rather than alter, the soundness and security properties of the underlying PIOP (HyperPlonk) and PCS (Brakedown) that we accelerate, so that our performance optimizations introduce no new cryptographic vulnerabilities; and (iii) encourage deployers to pair TensorZKP with appropriate governance, monitoring, and legal compliance processes.

Decision to Publish. After internal review, we judged that the benefits to the security, privacy, and cryptography communities, together with the benefits to legitimate applications such as blockchain scaling, verifiable machine learning, and privacy-preserving authentication, outweigh the residual dual-use risk when deployment follows appropriate legal and technical controls. The decision to publish was made after correctness checks against reference baselines and end-to-end benchmarking. We did not identify ethical concerns sufficient

to warrant withholding publication. We believe that open sharing of our results promotes accountability, reduces duplication of effort, and supports innovation in cryptography research, in line with community norms.

Open Science

The source code of TensorZKP is publicly available at https://doi.org/10.5281/zenodo.20337923.

References

[1] Nathan Bell and Michael Garland. Implementing sparse matrix-vector multiplication on throughput-oriented processors. In *Proceedings of the conference on high performance computing networking, storage and analysis*, pages 1–11, 2009.

[2] Eli Ben-Sasson, Iddo Bentov, Yinon Horesh, and Michael Riabzev. Fast Reed-Solomon interactive oracle proofs of proximity. In *45th international colloquium on automata, languages, and programming (icalp 2018)*, pages 14–1. Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik, 2018.

[3] Benedikt Bünz, Shashank Agrawal, Mahdi Zamani, and Dan Boneh. Zether: Towards privacy in a smart contract world. In *International Conference on Financial Cryptography and Data Security*, pages 423–443. Springer, 2020.

[4] Benedikt Bünz, Ben Fisch, and Alan Szepieniec. Transparent SNARKs from DARK compilers. In *Advances in Cryptology–EUROCRYPT 2020: 39th Annual International Conference on the Theory and Applications of Cryptographic Techniques, Zagreb, Croatia, May 10–14, 2020, Proceedings, Part I 39*, pages 677–706. Springer, 2020.

[5] Bing-Jyue Chen, Suppakit Waiwitlikhit, Ion Stoica, and Daniel Kang. Zkml: An optimizing system for ML inference in zero-knowledge proofs. In *Proceedings of the Nineteenth European Conference on Computer Systems*, pages 560–574, 2024.

[6] Binyi Chen, Benedikt Bünz, Dan Boneh, and Zhenfei Zhang. Hyperplonk: Plonk with linear-time prover and high-degree custom gates. In *Annual International Conference on the Theory and Applications of Cryptographic Techniques*, pages 499–530. Springer, 2023.

[7] Yutian Chen, Cong Peng, Yu Dai, Min Luo, and Debiao He. Load-balanced parallel implementation on GPUs for multi-scalar multiplication algorithm. *IACR Transactions on Cryptographic Hardware and Embedded Systems*, 2024(2):522–544, 2024.

[8] Alessandro Chiesa, Yuncong Hu, Mary Maller, Pratyush Mishra, Noah Vesely, and Nicholas Ward. Marlin: Preprocessing zkSNARKs with universal and updatable SRS. In *Annual International Conference on the Theory and Applications of Cryptographic Techniques*, pages 738–768. Springer, 2020.

[9] Alhad Daftardar, Jianqiao Mo, Joey Ah-kiow, Benedikt Bünz, Ramesh Karri, Siddharth Garg, and Brandon Reagen. Need for zkSpeed: Accelerating HyperPlonk for zero-knowledge proofs. In *Proceedings of the 52nd Annual International Symposium on Computer Architecture*, pages 1986–2001, 2025.

[10] Shengyu Fan, Zhiwei Wang, Weizhi Xu, Rui Hou, Dan Meng, and Mingzhe Zhang. TensorFHE: Achieving practical computation on encrypted data using GPGPU. In *IEEE International Symposium on High-Performance Computer Architecture (HPCA)*, pages 922–934, 2023.

[11] Boyuan Feng, Zheng Wang, Yuke Wang, Shu Yang, and Yufei Ding. Zeno: A type-based optimization framework for zero knowledge neural network inference. In *Proceedings of the 29th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 1*, pages 450–464, 2024.

[12] Amos Fiat and Adi Shamir. How to prove yourself: Practical solutions to identification and signature problems. In *Conference on the theory and application of cryptographic techniques*, pages 186–194. Springer, 1986.

[13] Ariel Gabizon, Zachary J Williamson, and Oana Ciobotaru. Plonk: Permutations over lagrange-bases for oecumenical noninteractive arguments of knowledge. *Cryptology ePrint Archive*, 2019.

[14] Michael Garland. Sparse matrix computations on manycore GPU's. In *Proceedings of the 45th annual design automation conference*, pages 2–6, 2008.

[15] Giza. Actionable ai for decentralized applications. Build reliable, scalable and easy to integrate AI solutions for Web3. https://www.gizatech.xyz/.

[16] Shafi Goldwasser, Yael Tauman Kalai, and Guy N Rothblum. Delegating computation: interactive proofs for muggles. *Journal of the ACM (JACM)*, 62(4):1–64, 2015.

[17] Shafi Goldwasser, Silvio Micali, and Chales Rackoff. The knowledge complexity of interactive proof-systems. In *Providing sound foundations for cryptography: On the work of shafi goldwasser and silvio micali*, pages 203–225. 2019.

[18] Alexander Golovnev, Jonathan Lee, Srinath Setty, Justin Thaler, and Riad S Wahby. Brakedown: Linear-time and field-agnostic SNARKs for R1CS. In *Annual International Cryptology Conference*, pages 193–226. Springer, 2023.

[19] Jens Groth. On the size of pairing-based non-interactive arguments. In *Annual international conference on the theory and applications of cryptographic techniques*, pages 305–326. Springer, 2016.

[20] Yanpei Guo, Xuanming Liu, Kexi Huang, Wenjie Qu, Tianyang Tao, and Jiaheng Zhang. {DeepFold}: Efficient multilinear polynomial commitment from {Reed-Solomon} code and its application to zero-knowledge proofs. In *34th USENIX Security Symposium (USENIX Security 25)*, pages 3497–3516, 2025.

[21] Meng Hao, Hanxiao Chen, Hongwei Li, Chenkai Weng, Yuan Zhang, Haomiao Yang, and Tianwei Zhang. Scalable zero-knowledge proofs for non-linear functions in machine learning.

[22] Florian Hirner, Florian Krieger, Constantin Piber, and Sujoy Sinha Roy. Accelerating hash-based polynomial commitment schemes with linear prover time. *IACR Transactions on Cryptographic Hardware and Embedded Systems*, 2025(4):341–385, 2025.

[23] ingonyama. Hardware accelerators for zero knowledge cryptography. https://www.ingonyama.com/.

[24] Ingonyama. Icicle is a library for ZK acceleration using CUDA-enabled GPUs. https://github.com/ingonyama-zk/icicle.

[25] Zhuoran Ji, Zhiyuan Zhang, Jiming Xu, and Lei Ju. Accelerating multi-scalar multiplication for efficient zero knowledge proofs with multi-GPU systems. In *Proceedings of the 29th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 3*, pages 57–70, 2024.

[26] Zhuoran Ji, Jianyu Zhao, Peimin Gao, Xiangkai Yin, and Lei Ju. Accelerating number theoretic transform with multi-GPU systems for efficient zero knowledge proof. In *Proceedings of the 30th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 1*, pages 1–14, 2025.

[27] Aniket Kate, Gregory M Zaverucha, and Ian Goldberg. Constant-size commitments to polynomials and their applications. In *International conference on the theory and application of cryptology and information security*, pages 177–194. Springer, 2010.

[28] Ahmed Kosba, Andrew Miller, Elaine Shi, Zikai Wen, and Charalampos Papamanthou. Hawk: The blockchain model of cryptography and privacy-preserving smart contracts. In *2016 IEEE symposium on security and privacy (SP)*, pages 839–858. IEEE, 2016.

[29] Alex Krizhevsky, Ilya Sutskever, and Geoffrey E Hinton. Imagenet classification with deep convolutional neural networks. *Advances in neural information processing systems*, 25, 2012.

[30] Chongrong Li, Pengfei Zhu, Yun Li, Cheng Hong, Wenjie Qu, and Jiaheng Zhang. Hyperpianist: Pianist with linear-time prover and logarithmic communication cost. In *2025 IEEE Symposium on Security and Privacy (SP)*, pages 3383–3401. IEEE, 2025.

[31] Tianyi Liu, Tiancheng Xie, Jiaheng Zhang, Dawn Song, and Yupeng Zhang. Pianist: Scalable zkrollups via fully distributed zero-knowledge proofs. *Cryptology ePrint Archive*, 2023.

[32] Tianyi Liu, Xiang Xie, and Yupeng Zhang. Zkcnn: Zero knowledge proofs for convolutional neural network predictions and accuracy. In *Proceedings of the 2021 ACM SIGSAC Conference on Computer and Communications Security*, pages 2968–2985, 2021.

[33] Tianyi Liu, Zhenfei Zhang, Yuncong Zhang, Wenqing Hu, and Ye Zhang. Ceno: Non-uniform, segment and parallel zero-knowledge virtual machine. *Journal of Cryptology*, 38(2):17, 2025.

[34] Tao Lu, Yuxun Chen, Zonghui Wang, Xiaohang Wang, Wenzhi Chen, and Jiaheng Zhang. BatchZK: A fully pipelined GPU-accelerated system for batch generation of zero-knowledge proofs. In *Proceedings of the 30th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 1*, pages 100–115, 2025.

[35] Tao Lu, Chengkun Wei, Ruijing Yu, Chaochao Chen, Wenjing Fang, Lei Wang, Zeke Wang, and Wenzhi Chen. Cuzk: Accelerating zero-knowledge proof with a faster parallel multi-scalar multiplication algorithm on GPUs. *Cryptology ePrint Archive*, 2022.

[36] Carsten Lund, Lance Fortnow, Howard Karloff, and Noam Nisan. Algebraic methods for interactive proof systems. *Journal of the ACM (JACM)*, 39(4):859–868, 1992.

[37] Weiliang Ma, Qian Xiong, Xuanhua Shi, Xiaosong Ma, Hai Jin, Haozhao Kuang, Mingyu Gao, Ye Zhang, Haichen Shen, and Weifang Hu. Gzkp: A GPU accelerated zero-knowledge proof system. In *Proceedings of the 28th ACM International Conference on Architectural*

Support for Programming Languages and Operating Systems, Volume 2, pages 340–353, 2023.

[38] Ralph C Merkle. A digital signature based on a conventional encryption function. In *Conference on the theory and application of cryptographic techniques*, pages 369–378. Springer, 1987.

[39] Nvidia. A high-performance CUDA library for sparse matrix-matrix multiplication. https://docs.nvidia.com/cuda/cusparselt/, 2025.

[40] Polyhedra. Empowering interoperability and computation via ZK. Bringing interoperability and scalability to Web3 with cutting-edge zero-knowledge proof systems. https://www.polyhedra.network/.

[41] Wenjie Qu, Yijun Sun, Xuanming Liu, Tao Lu, Yanpei Guo, Kai Chen, and Jiaheng Zhang. zkgpt: An efficient non-interactive zero-knowledge proof framework for LLM inference. In *34st USENIX Security Symposium (USENIX Security 25)*, 2025.

[42] Andy Ray, Benjamin Devlin, Fu Yong Quah, and Rahul Yesantharao. Hardcaml MSM: A high-performance split CPU-FPGA multi-scalar multiplication engine. In *Proceedings of the 2024 ACM/SIGDA International Symposium on Field Programmable Gate Arrays*, pages 33–39, 2024.

[43] Nikola Samardzic, Simon Langowski, Srinivas Devadas, and Daniel Sanchez. Accelerating zero-knowledge proofs through hardware-algorithm co-design. In *2024 57th IEEE/ACM International Symposium on Microarchitecture (MICRO)*, pages 366–379. IEEE, 2024.

[44] Eli Ben Sasson, Alessandro Chiesa, Christina Garman, Matthew Green, Ian Miers, Eran Tromer, and Madars Virza. Zerocash: Decentralized anonymous payments from bitcoin. In *2014 IEEE symposium on security and privacy*, pages 459–474. IEEE, 2014.

[45] Jacob T Schwartz. Fast probabilistic algorithms for verification of polynomial identities. *Journal of the ACM (JACM)*, 27(4):701–717, 1980.

[46] Scroll. Scroll seamlessly extends Ethereum's capabilities through zero knowledge tech and EVM compatibility. the L2 network built by Ethereum devs for Ethereum devs. https://scroll.io/.

[47] Srinath Setty. Spartan: Efficient and general-purpose zkSNARKs without trusted setup. In *Annual International Cryptology Conference*, pages 704–737. Springer, 2020.

[48] Daniel A Spielman. Linear-time encodable and decodable error-correcting codes. In *Proceedings of the twenty-seventh annual ACM symposium on Theory of computing*, pages 388–397, 1995.

[49] Haochen Sun, Jason Li, and Hongyang Zhang. zkllm: Zero knowledge proofs for large language models. In *Proceedings of the 2024 on ACM SIGSAC Conference on Computer and Communications Security*, pages 4405–4419, 2024.

[50] Justin Thaler. Time-optimal interactive proofs for circuit evaluation. In *Annual cryptology conference*, pages 71–89. Springer, 2013.

[51] Riad S. Wahby. Polynomial commitments from linear codes. https://github.com/conroi/lcpc.

[52] Riad S Wahby, Ioanna Tzialla, Abhi Shelat, Justin Thaler, and Michael Walfish. Doubly-efficient zkSNARKs without trusted setup. In *2018 IEEE Symposium on Security and Privacy (SP)*, pages 926–943. IEEE, 2018.

[53] Cheng Wang and Mingyu Gao. Unizk: Accelerating zero-knowledge proof with unified hardware and flexible kernel mapping. In *Proceedings of the 30th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 1*, pages 1101–1117, 2025.

[54] Howard Wu, Wenting Zheng, Alessandro Chiesa, Raluca Ada Popa, and Ion Stoica. {DIZK}: A distributed zero knowledge proof system. In *27th USENIX Security Symposium (USENIX Security 18)*, pages 675–692, 2018.

[55] Tiacheng Xie, Jiaheng Zhang, Yupeng Zhang, Charalampos Papamanthou, and Dawn Song. Libra: Succinct zero-knowledge proofs with optimal prover computation. In *Advances in Cryptology–CRYPTO 2019: 39th Annual International Cryptology Conference, Santa Barbara, CA, USA, August 18–22, 2019, Proceedings, Part III 39*, pages 733–764. Springer, 2019.

[56] Tiancheng Xie, Jiaheng Zhang, Zerui Cheng, Fan Zhang, Yupeng Zhang, Yongzheng Jia, Dan Boneh, and Dawn Song. zkbridge: Trustless cross-chain bridges made practical. In *Proceedings of the 2022 ACM SIGSAC Conference on Computer and Communications Security*, pages 3003–3017, 2022.

[57] Tiancheng Xie, Yupeng Zhang, and Dawn Song. Orion: Zero knowledge proof with linear prover time. In *Annual International Cryptology Conference*, pages 299–328. Springer, 2022.

[58] Jiaheng Zhang, Tiancheng Xie, Yupeng Zhang, and Dawn Song. Transparent polynomial delegation and its applications to zero knowledge proof. In *2020 IEEE Symposium on Security and Privacy (SP)*, pages 859–876. IEEE, 2020.

[59] Ye Zhang, Shuo Wang, Xian Zhang, Jiangbin Dong, Xingzhong Mao, Fan Long, Cong Wang, Dong Zhou, Mingyu Gao, and Guangyu Sun. Pipezk: Accelerating zero-knowledge proof with a pipelined architecture. In *2021 ACM/IEEE 48th Annual International Symposium on Computer Architecture (ISCA)*, pages 416–428. IEEE, 2021.

[60] Baoze Zhao, Wenjin Huang, Tianrui Li, and Yihua Huang. Bstmsm: A high-performance FPGA-based multi-scalar multiplication hardware accelerator. In *2023 International Conference on Field Programmable Technology (ICFPT)*, pages 35–43. IEEE, 2023.

[61] Tian Zhou, Fangyu Zheng, Guang Fan, Lipeng Wan, Wenxu Tang, Yixuan Song, Yi Bian, and Jingqiang Lin. ConvKyber: Unleashing the power of AI accelerators for faster Kyber with novel iteration-based approaches. *IACR Transactions on Cryptographic Hardware and Embedded Systems*, 2024(2):25–63, 2024. doi:10.46586/tches.v2024.i2.25-63.