PipeSC: A Resource-efficient and Pipelined Hardware Accelerator for Sumcheck Protocol

Kaixuan Wang[1], Yifan Yanggong[1], Xiaoyu Yang[2], Chenti Baixiao[2] and Lei Wang[1]

[1] Shanghai Jiao Tong University, Shanghai, China
[2] Chipltech, Shanghai, China

Abstract. Zero-knowledge Succinct Non-interactive Arguments of Knowledge (zk-SNARKs) are cryptographic protocols that allow a prover to convince verifiers of the correctness of a statement without revealing any additional information. Recent zk-SNARK constructions have shifted from univariate to multivariate polynomial-based designs, reducing the proving complexity from quasilinear to linear by avoiding costly univariate polynomial interpolation. This shift, however, makes the sumcheck protocol—a core primitive for verifying polynomial relations over the Boolean hypercube—the dominant component in the prover's workload. Consequently, the iterative nature and intensive intermediate data movement of the sumcheck protocol introduce severe performance bottlenecks in CPU- and GPU-based implementations, especially for large-scale multivariate polynomials.

In this paper, we present PipeSC, a resource-efficient, ASIC-based accelerator for sumcheck. PipeSC combines deep pipelining, modular-multiplier reuse, and a finite-state machine-based dependency scheduler to sustain high utilization of computational resources across phases of the protocol. In addition, we introduce an Equality-MLE generation module that employs hierarchical scheduling and multiplier reuse, yielding a unified hardware substrate shared by multiple proving subroutines.

Against state-of-the-art CPU, GPU, and ASIC implementations, PipeSC delivers up to **5.02**× speedup over the GPU implementation and up to **2756.2**× speedup over the CPU implementation, while improving the area–time product (ATP) by up to **3.68**× compared with the ASIC design. These results show that careful hardware–algorithm co-design and conflict-free scheduling substantially accelerate sumcheck, paving the way for fully integrated zk-SNARK hardware pipelines.

Keywords: Hardware acceleration · zk-SNARKs · Sumcheck protocol

1 Introduction

Zero-knowledge Succinct Non-interactive Arguments of Knowledge (zk-SNARKs) are cryptographic tools that allow a prover to convince a verifier of the correctness of a computation without revealing any additional information. Because of their succinctness and zero-knowledge guarantees, zk-SNARKs have found urgent applications in various cutting-edge domains, such as blockchain [BCG+14, XZC+22], verifiable fully homomorphic encryption [TW24, WWX+25], and machine learning [LXZ21, LWQ+24]. Modern zk-SNARKs are typically built on a polynomial interactive oracle proof (PIOP) [BCS16], where the prover extends the intermediate value vectors to polynomials over a large finite field. In Plonk [GWC19], for example, the prover encodes intermediate values (i.e., the trace table) as several univariate polynomials, resulting in quasilinear proving time via fast polynomial interpolation algorithms. To achieve linear proving complexity, recent proof systems have shifted the encoding targets from univariate polynomials to multilinear

extensions (MLEs), which eliminates polynomial interpolation and thereby reduces the computational overhead of polynomial encoding. Several protocols, such as Spartan [Set20], HyperPlonk [CBBZ23], Jolt [AST24], and Binius [DP25], have adopted this MLE-based approach.

These zk-SNARK constructions introduce a core component, called *sumcheck protocol* [LFKN92]. It allows a prover to convince a verifier that the sum of evaluations of a multivariate polynomial over a boolean hypercube equals a claimed value. In multilinear polynomial based proof systems, sumcheck protocol provides an efficient way to verify the correctness of polynomial relations over exponentially large domains with the help of equality MLEs. However, because the sumcheck involves N sequential rounds of pairwise evaluations and updates over exponentially large domains, the prover's workload becomes dominated by modular multiplications and memory accesses. As a result, the sumcheck protocol often emerges as the primary performance bottleneck in multilinear PIOP-based systems. Within the sumcheck, a key subroutine is the generation of equality MLEs that bind verifier challenges to the Boolean hypercube. This step involves structured, data-dependent modular multiplications and contributes a substantial fraction of the runtime. Empirical profiling on HyperPlonk and Spartan shows that the sumcheck phase accounts for 60–80% of the total proving time, among which equality MLE generation alone contributes over 40% due to its data-dependent modular multiplications and high memory bandwidth demands. For a sumcheck instance used to prove a HyperPlonk gate constraint, the gate structure may involve 9 MLEs. At a problem size of 20 variables over a 256-bit prime field, each MLE table contains 2^{20} entries of 256 bits, i.e., approximately 32 MB per table, leading to a total working set of approximately 288 MB. For larger instances with up to 24 variables, the working set exceeds 4.5 GB, which is difficult to accommodate on chip and therefore typically requires external memory. Combined with the sumcheck protocol's repeated streaming accesses and per-round updates, this large memory footprint places substantial pressure on the memory system.

Consequently, the remaining bottleneck lies in the sumcheck protocol, whose iterative and bandwidth-intensive arithmetic is difficult to parallelize efficiently on general-purpose processors. Notably, recent protocol-level advances have increased the relative cost of the sumcheck phase. Earlier proof systems relied on univariate polynomial encodings, where fast polynomial interpolation dominated the prover's workload. The shift to MLE-based designs eliminates the polynomial interpolation bottleneck but replaces it with intensive modular multiplications and large-volume data movement inherent to the sumcheck protocol. As a result, sumcheck now accounts for 60–80% of the total proving time—a share that has grown as protocol complexity moves away from polynomial interpolation toward multilinear reductions. These challenges motivate the exploration of dedicated hardware acceleration, where specialized pipelines and scheduling mechanisms can exploit the protocol's inherent structure and arithmetic regularity.

1.1 Related Work

Recent efforts toward accelerating the sumcheck protocol have primarily focused on GPU- and ASIC-based implementations. We briefly review these works and highlight the remaining limitations that motivate our design.

GPU-based Acceleration. Current GPU-based accelerators for univariate polynomial based zk-SNARK have demonstrated significant progress. However, few works focus on accelerating MLE-based zk-SNARKs, and consequently, the study of the sumcheck protocol remains insufficient. ICICLE [Ing25] provides a CUDA-based implementation of the sumcheck protocol, offering substantial throughput improvement over CPU baselines. BatchZK [LCW+25], a pipelined GPU-accelerated system for batch generation of zero-knowledge proofs, accelerates its sumcheck protocol by independently executing multiple

sumcheck instances. These efforts highlight the benefit of exploiting thread-level parallelism for large-scale proof generation. However, GPU architectures inherently suffer from high control overhead when processing data-dependent recursive computations such as sumcheck. These challenges motivate the exploration of specific hardware platforms, where customized pipelines and dedicated scheduling can fully exploit the protocol's computational structure.

ASIC-based Acceleration. A few ASIC-based accelerators ([SLDS24] and [DMAk+25]) have focused on sumcheck-based zk-SNARKs, leaving this direction largely unexplored. NoCap [SLDS24] introduces a programmable ASIC accelerator for Spartan [Set20] and Orion [XZS22], emphasizing flexibility and on-chip reusability. However, NoCap executes sumcheck as a sequence of arithmetic kernels rather than as a deeply pipelined, dependency-aware dataflow. More recently, zkSpeed [DMAk+25] integrates a dedicated sumcheck computation into a large-scale ASIC architecture that accelerates HyperPlonk [CBBZ23]. However, compute units in zkSpeed for different phases of the sumcheck protocol are instantiated separately rather than time-multiplexed, which leads to replicated modular multipliers across phases and poor hardware utilization (only 40–60% multiplier activity on average). This limited resource reuse drives substantial area overhead and lowers overall resource efficiency. More recently, MTU [MDAK+25] proposes a multifunction tree unit for accelerating sumcheck-based zk-SNARKs, including equality MLE computation. MTU uses a hybrid traversal strategy that combines breadth-first search (BFS) processing for shallow levels with a depth-first search (DFS) accumulator for deeper levels. In contrast, PipeSC focuses on conflict-free DFS scheduling for equality-MLE generation with multiple deeply pipelined modular multipliers, enabling fine-grained dependency-aware execution.

In summary, prior GPU and ASIC designs largely treat sumcheck as a coarse kernel, with limited cross-phase resource reuse and without a fine-grained, dependency-aware schedule for equality MLE generation. Thus, the problem we address in this paper can be stated as follows:

- **Problem 1:** How can we efficiently generate equality MLEs for large-scale sumcheck protocols, minimizing both time and memory overhead?
- **Problem 2:** How can we design a resource-efficient sumcheck pipeline architecture that sustains high bandwidth with bank-parallel memory accesses, and time-multiplexes modular multipliers across phases to reduce area without sacrificing throughput?

1.2 Technique Overview

To address the two problems identified above, our design introduces two key techniques: (1) a conflict-free scheduling algorithm for equality MLE generation, and (2) a resource-efficient and pipelined sumcheck accelerator architecture.

Pipelined Equality MLE Generation Module Design The first challenge lies in generating equality MLEs efficiently without incurring exponential on-chip memory growth. A natural idea is to traverse the binary computation tree in a DFS manner so that intermediate results can be reused on the fly. However, a naive DFS traversal introduces read–write conflicts and pipeline stalls when multiple multipliers operate concurrently. Our key insight is that these space and time conflicts can be formalized as deterministic offset and interval constraints. Based on this observation, we design a conflict-free scheduling algorithm that automatically generates layer-wise access tables ensuring continuous pipeline operation. The resulting equality MLE module reduces the memory footprint from exponential to linear while sustaining full multiplier utilization. Furthermore, a flexible derivation scheme enables the hardware to handle variable-size instances without regenerating scheduling

configurations. This design provides high hardware reusability and scalability, forming the foundation of our unified MLE generation pipeline.

Resource-efficient Sumcheck Accelerator Architecture. The second challenge is how to build a high-throughput yet resource-efficient accelerator for the full sumcheck protocol. Our key insight is that three phases from equality MLE generation to MLE combination and update share the same underlying modular multiplication. This observation allows us to time-multiplex a unified multiplier array across different phases under deterministic finite-state machine (FSM) control. We design a centralized *FSM scheduler* that orchestrates MLE combine, challenge generation, and MLE update in a deeply pipelined manner, overlapping computation and memory transfer. Each module in the accelerator streams data through a multi-bank external memory system, while a shared pool of modular multipliers is time-multiplexed and reorganized to match each module's dataflow, maximizing sustained utilization over the entire sumcheck protocol. Together, these techniques realize a streaming, conflict-free, and memory-optimized accelerator that achieves deterministic throughput and high arithmetic efficiency across large multivariate polynomial instances.

1.3 Our Contributions

The main contributions of this work are summarized as follows:

- **Conflict-free DFS-based scheduling.** We model equality MLE generation as a modular-multiplication scheduling problem and reveal the exponential memory growth of the conventional BFS approach. To address this, we design a DFS-based scheduling algorithm that eliminates both space and time conflicts under formalized dependency constraints. It reduces on-chip memory from $O(2^N)$ to $O(N)$ while sustaining full pipeline utilization. We further introduce a derivation method that reuses precomputed scheduling tables for smaller instances, enabling flexible and scalable deployment.
- **Resource-efficient and reusable Sumcheck architecture.** Building upon our conflict-free DFS scheduler, we design a pipelined Equality MLE Generator that reuses modular multipliers efficiently under deterministic timing control. We integrate this generator with MLE Combine and Update modules into a unified accelerator, PipeSC. The MLE Combine and MLE Update stages produce one result per cycle while overlapping computation and data transfer. The Equality MLE Generator sustains one output every two cycles, requiring only $O(N)$ on-chip memory.
- **Implementation and evaluation.** We implement and evaluate PipeSC on both FPGA and ASIC platforms. A functional prototype is deployed on a Xilinx Alveo U280 FPGA, while the ASIC implementation is synthesized under a TSMC 12 nm standard-cell library targeting 1.0 GHz, and constitutes the primary performance evaluation platform for the parameterized cycle model. The design achieves up to **2756.2**× speedup over CPU (Arkworks), **5.02**× speedup over GPU (BatchZK), and **3.68**× **higher area–time efficiency** than ASIC (zkSpeed, 7 nm, 1 GHz) after process normalization.

2 Preliminary

In this section, we review the fundamental background on zero-knowledge succinct non-interactive arguments of knowledge (zk-SNARKs), multilinear extensions, and the sumcheck protocol. The sumcheck protocol serves as a core component in constructing many modern zk-SNARKs and is the algorithmic foundation of our proposed hardware accelerator designed to reduce the proof generation time of sumcheck-based zk-SNARKs.

2.1 Zero-Knowledge Succinct Non-interactive Arguments of Knowledge

Zero-Knowledge Succinct Non-Interactive Arguments of Knowledge (zk-SNARKs) are cryptographic proof systems that enable a prover to convince a verifier of the validity of a statement without revealing any additional information about the witness. A zk-SNARK satisfies five key properties: (1) *completeness*, i.e., an honest prover with a valid witness can always convince the verifier; (2) *knowledge soundness*, i.e., any prover convincing the verifier must possess a valid witness; (3) *zero-knowledge*, i.e., the proof reveals no information beyond the statement's truth; (4) *succinctness*, i.e., proofs are short, and verification is efficient; and (5) *non-interactivity*, i.e., a single proof message suffices for verification.

Modern zk-SNARKs are typically built upon probabilistic proof systems where computation is expressed as relations among polynomials. These polynomials are often referred to as extensions of Boolean functions to a finite field. In particular, multilinear polynomials used in zk-SNARKs are known as Multilinear Extensions (MLEs). The consistency of these relations is verified through a sequence of probabilistic checks over the polynomial evaluations. To remove interaction between prover and verifier, the Fiat–Shamir transformation [FS86] replaces verifier-generated randomness with hash-derived challenges, yielding non-interactive proofs. Among the various algebraic sub-protocols that constitute zk-SNARKs, the sumcheck protocol plays a central role. This protocol forms the backbone of many modern multilinear polynomial–based proof systems, such as GKR [GKR15], Spartan [Set20], and HyperPlonk [CBBZ23]. The sumcheck protocol efficiently verifies polynomial relations of exponentially large domains using only $O(n)$ rounds of interaction, dramatically reducing the verifier's workload.

From an implementation perspective, sumcheck introduces intensive arithmetic workloads—repeated polynomial evaluations and multilinear updates across multiple rounds. These characteristics make sumcheck the dominant contributor to zk-SNARK proving time, motivating specialized hardware acceleration to fully exploit its algebraic regularity and parallelism.

2.2 Multilinear Extension

An n-variate polynomial $p : \mathbb{F}^n \to \mathbb{F}$ is *multilinear* if it has degree at most one in each variable. Suppose $f : \{0,1\}^n \to \mathbb{F}$ is a function that maps the n-dimensional Boolean hypercube into a finite field $\mathbb{F}$. Given a Boolean function $f : \{0,1\}^n \to \mathbb{F}$, its *multilinear extension* (MLE) [CMT12], denoted $\tilde{f}(X)$, is the unique multilinear polynomial over $\mathbb{F}$ satisfying $\tilde{f}(x) = f(x)$ for all $x \in \{0,1\}^n$. The existence and uniqueness follow from the Schwartz–Zippel Lemma.

Equality MLE A particularly important multilinear extension used in SNARK constructions is the *equality MLE* $\tilde{eq}$, which is the MLE of the function $eq : \{0,1\}^m \times \{0,1\}^m \to \mathbb{F}$ defined as follows:

$$eq(\mathbf{x}, \mathbf{y}) = \begin{cases} 1 & \text{if } \mathbf{x} = \mathbf{y}, \\ 0 & \text{otherwise.} \end{cases} \tag{1}$$

An explicit form for equality MLE $\tilde{eq}(\mathbf{b}, \mathbf{X})$ is:

$$\tilde{eq}(\mathbf{b}, \mathbf{X}) = \prod_{i=0}^{m-1} (b_i X_i + (1 - b_i)(1 - X_i)). \tag{2}$$

The expression for MLEs of any functions is given by the following lemma (see [Tha22], Lemma 3.6). The multilinear extension of any function f is:

$$\tilde{f}(\mathbf{X}) = \sum_{b \in \{0,1\}^n} f(\mathbf{b}) \cdot \tilde{eq}(\mathbf{b}, \mathbf{X}) \tag{3}$$

Here, the equality MLEs $\{\tilde{eq}(\mathbf{b}, \mathbf{X}) : \mathbf{b} \in \{0,1\}^n\}$ are the *Lagrange basis polynomials* for multilinear polynomials of variable n. The evaluations $\{\tilde{f}(\mathbf{X}) : \mathbf{X} \in \{0,1\}^n\}$ are the coefficients of $\tilde{f}$ in the Lagrange basis.

2.3 Sumcheck Protocol

In modern zk-SNARK constructions, the sumcheck protocol [LFKN92] serves as an efficient mechanism for verifying algebraic relations among multilinear extensions (MLEs) by proving that the summation of a target polynomial over the Boolean hypercube equals a claimed value. Instead of directly evaluating all 2^n points, the protocol iteratively reduces an n-variate polynomial to an $(n-1)$-variate one through MLE combine, challenge generation, and MLE update phases. After n rounds, the verifier only needs to check a single evaluation, thereby reducing the verification complexity from $O(2^n)$ to $O(n)$. To adapt the protocol to non-interactive zk-SNARK settings, modern implementations follow a *hash-based Fiat–Shamir transcript*, where the verifier's random challenges are derived from cryptographic hashes. The corresponding proof generation process is shown in Algorithm 1. We denote by $k = k_{in} + k_{eq}$ the total number of multilinear tables involved in the protocol. For notational convenience, we reindex all MLEs as a single family $\{F_\ell\}_{l=1}^{k}$, where

$$F_\ell = \tilde{f}_\ell \quad \text{for} \quad 1 \le \ell \le k_{in}, \quad F_{k_{in}+j} = \tilde{eq}_j \quad \text{for} \quad 1 \le j \le k_{eq}. \tag{4}$$

In Algorithm 1, the arrays $F_\ell[\cdot]$ therefore represent the current values of these MLE tables as they are updated across the sumcheck rounds.

Each sumcheck instance can be characterized by four key parameters:

- n: the number of variables in the target multilinear extensions (MLEs);
- $k = k_{\text{in}} + k_{\text{eq}}$: the total number of MLEs involved in the relation, including k_{in} input MLEs and k_{eq} equality MLEs;
- d: the degree of the combine logic, which determines how many terms are produced per round; and
- m: the number of modular multiplications required per combine operation.

Among these parameters, k determines the number of modular multiplications required in the *MLE update* phase, while m and d determine the number of modular multiplications in the *MLE combine* phase. Compared to hash computations in challenge generation and modular additions in MLE combine, modular multiplications constitute the dominant time-consuming operations. These parameters therefore serve as the foundation for estimating the cycle count in our pipeline-level hardware analysis presented in Section 4.6.

The *combine logic* defines how multiple multilinear extensions are algebraically aggregated to form the target polynomial being proved in the sumcheck protocol. Its form is highly protocol-dependent: in different zk-SNARK systems, it encodes different algebraic relations, such as gate constraints, lookup constraints, or custom arithmetic checks. For example, in the HyperPlonk system [CBBZ23], the combine logic expresses the *gate constraint* that enforces the correctness of circuit operations:

$$\begin{aligned} 0 &= \sum_{X \in \{0,1\}^n} \text{combine}(\tilde{eq}, \tilde{q}_l, \tilde{q}_r, \tilde{q}_o, \tilde{q_m}, \tilde{q}_c, \tilde{w}_l, \tilde{w}_r, \tilde{w}_o) \\ &= \sum_{X \in \{0,1\}^n} \tilde{eq}\big(\tilde{q}_l \tilde{w}_l + \tilde{q}_r \tilde{w}_r + \tilde{q}_m \tilde{w}_l \tilde{w}_r + \tilde{q}_o \tilde{w}_o + \tilde{q}_c\big) \end{aligned} \tag{5}$$

Here, the combine logic combine$(\cdot)$ represents the linear combination and multiplication between selector polynomials $\{\tilde{q}_l, \tilde{q}_r, \tilde{q}_o, \tilde{q}_m, \tilde{q}_c\}$, wire values $\{\tilde{w}_l, \tilde{w}_r, \tilde{w}_o\}$, and the equality polynomial $\tilde{eq}$ that binds the verifier's random challenge $\mathbf{r}$ to the Boolean domain.

Algorithm 1 Sumcheck Algorithm

Define: $s = \sum_{\mathbf{b}\in\{0,1\}^n} \text{combine}(\tilde{f}_1(\mathbf{b}), \tilde{f}_2(\mathbf{b}), \cdots, \tilde{f}_{k_{\text{in}}}(\mathbf{b}), \tilde{eq}_1(\mathbf{b}), \ldots, \tilde{eq}_{k_{\text{eq}}}(\mathbf{b}))$

Input: Target sum s; input MLEs $\tilde{f}_1, \ldots, \tilde{f}_{k_{\text{in}}}$ over n variables; random vectors $\mathbf{r}^{(1)}, \ldots, \mathbf{r}^{(k_{\text{eq}})}$, where the superscript (j) indexes the equality MLE and the subscript i indexes the variable position; and the combining function combine$(\cdot)$ of degree d.

Output: Proof T.

▷ **Step 1: Equality MLE Generation**

for $j = 1$ **to** k_{eq} **do**

  $\tilde{eq}_j \leftarrow \bigotimes_{i=0}^{n-1} [1 - r_i^{(j)} \quad r_i^{(j)}]$

end for

Let $k \leftarrow k_{\text{in}} + k_{\text{eq}}$

Let $F_1, \ldots, F_k \leftarrow \tilde{f}_1, \ldots, \tilde{f}_{k_{\text{in}}}, \tilde{eq}_1, \ldots, \tilde{eq}_{k_{\text{eq}}}$

▷ **Step 2: Sumcheck Iteration**

Initialize transcript T with the public inputs

let $\alpha_0 \leftarrow \text{hash}(T)$

for $p = 0$ **to** $n - 1$ **do**

  ▷ **Phase 1: MLE Combine**

  Initialize $r_p[0], \ldots, r_p[d] \leftarrow 0$

  for $i = 0$ **to** d **do**

    for $j = 0$ **to** $2^{n-p-1} - 1$ **do**

      $r_p[i]$ += combine$(\{F_l[j] \cdot (1 - i) + F_l[j + 2^{n-p-1}] \cdot i\}_{l\in\{1,\cdots,k\}})$

    end for

  end for

  ▷ **Phase 2: Challenge Generation**

  T.append(r_p)

  $\alpha_{p+1} \leftarrow \text{hash}(\alpha_p, r_p)$

  ▷ **Phase 3: MLE Update**

  for $i = 0$ **to** $2^{n-p-1} - 1$ **do**

    for $l = 1$ **to** k **do**

      $F_l[i] \leftarrow (1 - \alpha_{p+1}) \cdot F_l[i] + \alpha_{p+1} \cdot F_l[i + 2^{n-p-1}]$

    end for

  end for

end for

return T

Other systems may define different combine functions (e.g., for lookup consistency checks, permutation arguments, or range constraints), yet all share the same iterative reduction structure of sumcheck.

In this work, we adopt the standard vanilla-gate combine logic used in HyperPlonk [CBBZ23], as given in Eq. 5. This is also the combine logic used in our direct ASIC comparison with zkSpeed [DMAk+25], which evaluates HyperPlonk-based sumcheck instances under the same functional setting.

3 A Conflict-free Modular Scheduling Algorithm for Equality MLE Generation

In this section, we focus on the algorithmic aspect of accelerating the equality multilinear extension (MLE) generation. Specifically, we formalize the equality MLE generation as a modular multiplication scheduling problem under dependency and memory constraints. Then, we propose a conflict-free scheduling algorithm. This algorithm ensures that the

equality MLE generation can be completed using only $O(N)$ on-chip memory units while sustaining a high-throughput pipeline that outputs one result every two cycles. However, directly recomputing a full scheduling plan for every N configuration is costly. To address this, we further propose a derivation algorithm: given a precomputed scheduling solution for the fixed parameter N, it can efficiently derive valid scheduling plans for any smaller N without recomputation, where N is the length of $\mathbf{r}$. This property allows scalable reuse of scheduling patterns across varying problem sizes, significantly improving algorithmic flexibility and design efficiency.

3.1 Baseline Analysis and Challenges

Generating $\tilde{eq}(\mathbf{r}, \mathbf{X})$ entails a sequence of *data-dependent* modular multiplications. In naive per-output product method, the total number of modular multiplications is $(N-1) \cdot 2^N$. We assume T fully-pipelined modular multipliers (MMs), each with a pipeline depth of D_{MM} cycles, and an on-chip memory unit with a capacity of M words whose width is equal to the bitwidth of the field elements r_i. The key question is therefore:

> *How can we, under a fixed number of modular multipliers T, achieve the minimal total execution cycles and memory capacity M required for generating $\tilde{eq}(\mathbf{r}, \mathbf{X})$, while maintaining continuous throughput and high modular multiplier utilization?*

A standard construction expands this product in a binary tree with $N+1$ layers: each node in the ℓ-th layer where $\ell \in [2, N-1]$ represents a modular multiplication to compute a partial product and the leaves of the final layer (the N-th layer) produce all 2^N outputs. We index the nodes of the tree in a breadth-first manner, where the root node is labeled as 1, and the left and right children of node j are labeled as $2j$ and $2j+1$, respectively. By sharing partial products across layers, the total number of modular multiplications is reduced from $(N-1) \cdot 2^N$ to $2^{N+1} - 4$.[1]

A straightforward approach based on breadth-first search. For layer ℓ ($2 \le \ell \le N$), take all $2^{\ell-1}$ partial products from layer $\ell - 1$ and produce 2^ℓ children by multiplying each parent once by $1 - r_{\ell-1}$ and once by $r_{\ell-1}$. Before proceeding to layer $\ell + 1$, store the entire set of 2^ℓ results in layer ℓ. Repeat the procedure until layer N. The 2^N values are then emitted. This schedule performs $\sum_{\ell=1}^{N} 2^\ell = 2^{N+1} - 2$ multiplications in the pure tree expansion; with the standard first-layer special case, this matches the $2^{N+1} - 4$ count noted above. This approach can be efficiently implemented on a pipelined architecture. Except for the first few layers—where the multiplications in layer $\ell + 1$ must wait for the outputs of layer ℓ—the computation is conflict-free, since the number of modular multiplications in each subsequent layer ($2^{\ell+1}$ multiplications in the ℓ layer) exceeds the pipeline depth of the modular multiplier D_{MM}, allowing fully pipelined execution. For simplicity, we first consider a single modular multiplier ($T = 1$). The total number of cycles required by this approach can be expressed as

$$C_{BFS} = \lceil \log D_{\mathsf{MM}} \rceil \cdot D_{\mathsf{MM}} + \sum_{i=\lceil \log D_{\mathsf{MM}} \rceil - 1}^{N} 2^i \tag{6}$$

The first term represents the initial pipeline filling latency of the first $\lceil \log D_{\mathsf{MM}} \rceil$ layers, while the summation term accounts for computation cost. When T multipliers are available, the total latency can be approximately reduced to C_{BFS}/T, assuming ideal load balancing achieved by partitioning the binary tree into subtrees evenly assigned to all multipliers.

[1] Because the value of root node is 1, and the two nodes in the first layer is $1 - r_0$ and r_0, these two layers can be implemented without multiplications, saving three modular multiplications.

Although the BFS-based approach is conceptually simple to schedule the modular multiplications, the *memory footprint* reaches its peak at layer $N-1$, where 2^{N-1} partial products must reside on-chip simultaneously. The exponential growth of the memory requirement makes the hardware design lack scalability. Such exponential memory growth renders the BFS approach impractical for large instances. For example, in a sumcheck instance with $N = 30$ variables and 256-bit field elements, the hardware would require over 16 GB of on-chip storage to hold intermediate values.

3.2 A Conflict-free Multiplication Scheduling Algorithm

The large memory footprint limitation of the BFS-based approach motivates a depth-first search (DFS) based scheduling strategy that traverses the computation tree along root-to-leaf paths to reuse intermediate results efficiently. Instead of executing the modular multiplications layer by layer, the DFS-based scheduler explores one branch of the tree at a time, computes its contributions, and then backtracks, allowing only $O(N)$ partial products to reside in on-chip registers. This design maintains a linear on-chip memory footprint.

However, implementing this DFS-based strategy on hardware introduces two potential hazards: *space conflicts* and *time conflicts*. In the terminology of processor pipelines, space conflicts correspond to *data hazards* (analogous to read-after-write (RAW) and read-before-write (RBW) hazards), while time conflicts correspond to *structural hazards* where multiple operations compete for the same pipeline resource. To illustrate, consider the case where the memory unit holds $N-2$ intermediate layers (storing the values of layer ℓ, where $2 \le \ell \le N-1$) and only one modular multiplier is available.

1. A *space conflict* occurs when the same memory location is accessed by multiple nodes before the previous data is safely consumed or correctly updated. This can lead to two types of data hazards: (1) a RAW hazard, where a value that has not yet been reused is prematurely overwritten by the result of a subsequent computation; and (2) a RBW hazard, where a computation attempts to read a value that has not yet been produced. For example, suppose the j-th node in layer ℓ has been computed and stored in memory. If the $(j+1)$-th node is computed early and its result is written back to the same memory location before the j-th value is reused to compute nodes $2j$ or $2j+1$ in layer $\ell+1$, a RAW hazard occurs, leading to incorrect results. Conversely, if the $(j+1)$-th node has not yet been computed while the $2(j+1)$-th node in the next layer starts execution and attempts to read its input, an RBW hazard arises due to missing data dependency enforcement.

2. A *time conflict* arises in a fully pipelined execution of the modular multiplication, where multiplications from different layers may attempt to occupy the modular multiplier pipeline in the same clock cycle. Such contention between operations from distinct layers must be avoided through careful scheduling to ensure correct data dependencies and to sustain continuous pipeline throughput.

Formalization of DFS-based Scheduling Problem To systematically address these two types of conflicts, we first formalize the scheduling constraints of the DFS-based multiplier execution. Let S_j denote the *start time* (in clock cycles) of the modular multiplication corresponding to node j in the computation tree. And the node j is in the layer ℓ where $\ell = \lfloor \log j \rfloor$. Each node j produces an output value v_j that may be consumed by its child nodes at layer $\ell+1$.

To prevent the space conflicts, we impose the following constraints:

$$\begin{aligned} &(1) \quad S_{2j-1} \le S_j + D_{\mathsf{MM}} - 1, \quad \forall j \in [4, 2^{N-1}-1], \\ &(2) \quad S_{2j} \ge S_j + D_{\mathsf{MM}}, \quad \forall j \in [4, 2^{N-1}-1], \end{aligned} \tag{7}$$

the first constraint enforces that the finish time of node j (i.e., $S_j + D_{\mathsf{MM}}$) does not occur earlier than the start time of node $2j-1$ (the right child of its sibling $j-1$). This prevents the partial result v_{j-1} required by v_{2j-1} from being overwritten by subsequent computations of node j. Conversely, the second constraint ensures that the start time of the right child node $2j$ is not earlier than the completion of its parent j, thereby preventing node $2j$ from incorrectly using v_{j-1} before v_j has been fully produced. Together, these constraints guarantee temporal data consistency and eliminate hazards during equality MLE generation.

To prevent the *time conflict*, the following constraint must hold: $S_i \neq S_j, \forall i \neq j$, or more generally,

$$|S_i - S_j| \geq 1, \quad \forall i \neq j \tag{8}$$

for all multiplications scheduled on the same pipeline. In the case of multiple multipliers ($T > 1$), this constraint can be relaxed to nodes in layers 2 through $N + 1 - \lfloor \log T \rfloor$ only as:

$$|S_i - S_j| \geq 1, \quad \forall i \neq j \text{ and } i, j \in [4, 2^{N+1-\lfloor \log T \rfloor} - 1] \tag{9}$$

In addition, we impose a linear memory-capacity constraint on the DFS schedule. Because DFS stores only the active partial products along the current root-to-leaf traversal path, the number of simultaneously live intermediate values is bounded by the tree depth. Therefore, the required on-chip memory grows only linearly with N.

Under these constraints, the overall scheduling problem can be expressed as:

$$\begin{aligned} &\text{Minimize:} && C_{\text{total}} = S_{2^{N+1}-1} + D_{\mathsf{MM}}, \\ &\text{Subject to:} && \text{(i) Space-conflict constraint,} \\ &&& \text{(ii) Time-conflict constraint,} \\ &&& \text{(iii) Memory-capacity constraint.} \end{aligned} \tag{10}$$

Conflict-Free Scheduling Algorithm. To solve the scheduling problem, we propose an algorithm that guarantees conflict-free execution while maximizing pipeline utilization. Leveraging the DFS traversal strategy, the start times of all nodes can be deterministically assigned according to the pipeline depth of the modular multiplier (D_{MM}). More generally, for the layer indexed by 2^i with $N \geq i \geq 2$, the first node starts at $S_{2^i} = (i-2)D_{\mathsf{MM}}$.

The core idea of our algorithm is to sequentially apply two types of constraints: first, the *space-conflict constraint*, which ensures that no partial results are overwritten before being fully consumed; and second, the *time-conflict constraint*, which prevents simultaneous access to the modular multiplier pipeline from different layers. We begin by assigning start times to the leaf nodes (in the N-th layer) and then iteratively propagate scheduling decisions upward, layer by layer, from layer $N-1$ to layer 2. According to the space-conflict constraint (Eq. 7), each parent node can be safely scheduled only after both of its child nodes have completed their computations. Once the space dependencies are resolved, the time-conflict constraint is enforced to ensure that no two layers issue multiplication operations to the modular multiplier pipeline in the same cycle.

This hierarchical scheduling process guarantees that all memory accesses respect data dependencies and that the pipeline remains continuously active without stalling. Algorithm 2 summarizes the proposed conflict-free scheduling method. In this algorithm, we encode the schedule with three tables: $\{\mathsf{Base}[i], \mathsf{Offset}[i], \mathsf{Interval}[i]\}_{i \in [2,N]}$. For layer i, the first node starts at the pivot time $\mathsf{Pivot}[i] = (i-2)D_{\text{MM}}$. The second node is placed at $\mathsf{Base}[i] + \mathsf{Offset}[i]$, and all subsequent nodes in that layer are launched periodically with stride $\mathsf{Interval}[i]$.

Firstly, we set $\mathsf{Base}[i] = (i-2)(D_{\text{MM}} - 1) + N$ to satisfy space constraints (parents are scheduled only after both children complete), and we choose $\mathsf{Interval}[i] = 2^{N-i+1}$ to

Algorithm 2 Conflict-free Scheduling Algorithm

Input: Input size N, multiplier delay D_{MM};
Output: Base table $\{\mathsf{Base}[i]\}$, offset table $\{\mathsf{Offset}[i]\}$, interval table $\{\mathsf{Interval}[i]\}$, shift table $\{\mathsf{Shift}[i]\}$;
Initialize $\mathsf{Base}[2..N] \leftarrow 0$, $\mathsf{Offset}[2..N] \leftarrow 0$, $\mathsf{Interval}[2..N] \leftarrow 0$, and $\mathsf{Shift}[2..N] \leftarrow 0$
$\mathsf{Base}[N] \leftarrow (N-2)D_{\mathrm{MM}} + 2$
$\mathsf{Offset}[N] \leftarrow 0$
$\mathsf{Interval}[N] \leftarrow 2$
$\mathsf{Shift}[N] \leftarrow \mathsf{Base}[N] + \mathsf{Offset}[N]$
for $i \leftarrow N-1$ **down to** 2 **do**
  $\mathsf{Pivot}[i] \leftarrow (i-2) \cdot D_{\mathrm{MM}}$
  $\mathsf{Base}[i] \leftarrow \mathsf{Pivot}[i] - i + N + 2$
  $\mathsf{Interval}[i] \leftarrow 2^{(N-i+1)}$
  $\mathsf{Offset}[i] \leftarrow (\mathsf{Interval}[i]/4 + (D_{\mathrm{MM}} - 1)) \bmod \mathsf{Interval}[i]/2$
  $\mathsf{Shift}[i] \leftarrow \mathsf{Base}[i] + \mathsf{Offset}[i]$
end for
return $\mathsf{Base}[2..N], \mathsf{Offset}[2..N], \mathsf{Interval}[2..N], \mathsf{Shift}[2..N]$

prevent simultaneous issues across layers. Finally, we define

$$\mathsf{Offset}[i] = \left(\frac{\mathsf{Interval}[i]}{4} + D_{\mathrm{MM}} - 1\right) \bmod \frac{\mathsf{Interval}[i]}{2},$$

a key deterministic shift that eliminates residual time conflicts (proved in the next section).

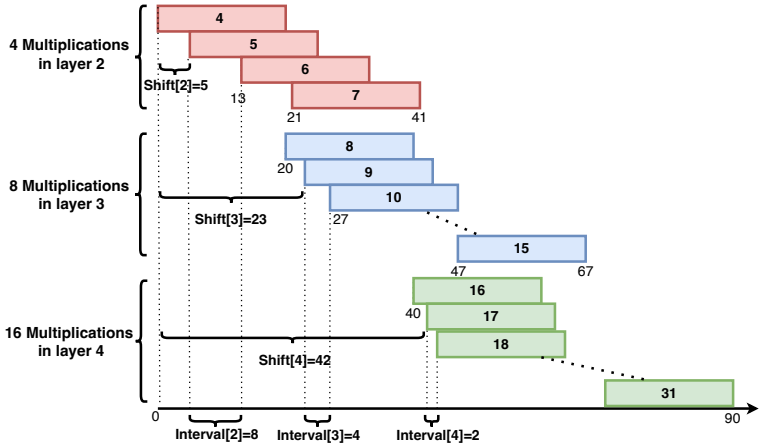

Figure 1: Conflict-free scheduling timeline for $N = 4$ and $D_{\mathsf{MM}} = 20$ cycles.

Taking $N = 4$ and $D_{\mathrm{MM}} = 20$ as an example, we obtain the shift table $\mathsf{Shift}[i] = \mathsf{Base}[i] + \mathsf{Offset}[i]$ for $i \in \{2, 3, 4\}$. The resulting scheduling timeline is illustrated in Fig. 1. Each colored bar represents a layer, where the left edge of each bar indicates the start cycle of a modular multiplication. Within i-th layer, the second node is started at the shift time $\mathsf{Shift}[i]$, and subsequent nodes follow periodically with stride $\mathsf{Interval}[i]$. As shown, the start times are staggered across layers to ensure that no two layers request access to the modular multiplier pipeline simultaneously, achieving a conflict-free schedule with continuous pipeline utilization.

Analysis of Conflict-free Scheduling Algorithm. We show that the schedule $S[\cdot]$ assigns pairwise distinct execution times to all modular-multiplication tasks, ensuring that no two

tasks are issued in the same cycle.

For each level $i \in [2, N]$ of the computation tree, tasks are indexed by $j \in [2^i,\, 2^{i+1} - 1]$. Their scheduled issue times are defined as:

$$S[j] = \begin{cases} \mathsf{Pivot}[i], & j = 2^i, \\ \mathsf{Shift}[i] + (j - 2^i - 1) \cdot \mathsf{Interval}[i], & j > 2^i. \end{cases}$$

For any two levels $i < k$, we define the relative offset:

$$\delta_{i,k} = \mathsf{Offset}[i] - \mathsf{Offset}[k] - (k - i) \cdot (D_{\mathsf{MM}} - 1).$$

The scheduling constraint enforces:

$$\delta_{i,k} \bmod \mathsf{Interval}[k] \neq 0, \qquad \forall\, k \in [i + 1, N]. \tag{11}$$

We now prove time-conflict freedom in two parts: intra-level conflict freedom and inter-level conflict freedom. For a fixed level i, the subsequence $\{S[j]\}_{j=2^i+1}^{2^{i+1}-1}$ forms an arithmetic progression with common difference $\mathsf{Interval}[i] > 0$, so all values are distinct. To check for collisions with the first element $S[2^i] = \mathsf{Pivot}[i]$, note that by construction $\mathsf{Pivot}[i] < \mathsf{Shift}[i]$. Therefore,

$$S[2^i] \neq S[j], \quad \forall j > 2^i,$$

and thus all $S[j]$ in level i are unique.

To prove inter-level conflict freedom, consider any two levels $i < k$ and tasks a and b belonging to them respectively. Each level produces an arithmetic sequence of times (plus one pivot element). If a time collision occurred, it would satisfy

$$S[a] - S[b] \equiv 0 \pmod{\mathsf{Interval}[k]}.$$

Expanding the DFS timing relation gives $S[a] - S[b] \equiv \delta_{i,k} \pmod{\mathsf{Interval}[k]}$. However, by condition in Eq. 11, we have $\delta_{i,k} \bmod \mathsf{Interval}[k] \neq 0$, which is a contradiction. Hence, no two tasks from levels i and k share the same issue time.

In conclusion, the mapping $j \mapsto S[j]$ is injective over the entire task set $j \in [4,\, 2^{N+1} - 1]$. Equivalently,

$$S[a] \neq S[b], \quad \forall\, a \neq b \in [4,\, 2^{N+1} - 1]. \tag{12}$$

Thus, the DFS-derived scheduling scheme is time-conflict free. The modular condition $\delta_{i,k} \bmod \mathsf{Interval}[k] \neq 0$ guarantees that the latency-adjusted inter-level offsets $(k - i)(D_{\mathsf{MM}} - 1)$ fully prevent pipeline collisions across all levels.

3.3 Flexible Scheduling Scheme Derivation for Variable-Size Instances

Although the conflict-free scheduling algorithm efficiently generates a static scheduling table for a given polynomial size N, real zkSNARK workloads often vary in input size, where $n < N$. Regenerating a new configuration for every n requires rerunning the scheduling logic and reinitializing on-chip control signals. To improve adaptability, we introduce a flexible derivation method. It allows scheduling tables for smaller instances to be directly derived from a precomputed large-N table.

The main observation is that the dependency pattern in equality MLE generation exhibits a recursive binary-tree structure. Thus, the scheduling configuration for an instance of size n can be extracted from that of size N by pruning redundant layers and remapping memory offsets. In other words, all configurations for smaller instances are substructures of the N-point dependency graph.

Algorithm 3 Derivation of Scheduling Tables for Smaller Instances

Input: Maximum size N_{max}, Pipeline depth of modular multiplier D_{max}, scheduling tables $\{\mathsf{Interval}[2..N_{max}], \mathsf{Base}[2..N_{max}]\}$, target size N_{sub}, target delay D_{sub};
Output: Derived tables $\{\mathsf{SubBase}[2..N_{sub}], \mathsf{SubOffset}[2..N_{sub}], \mathsf{SubInterval}[2..N_{sub}]\}$;
for $i \leftarrow N_{sub}$ **down to** 2 **do**
  $\mathsf{SubInterval}[i] \leftarrow \mathsf{Interval}[N_{max} - N_{sub} + i]$
  $\mathsf{SubBase}[i] \leftarrow \mathsf{Base}[i] - (i-2) \cdot (D_{max} - D_{sub}) - (N_{max} - N_{sub})$
  $\mathsf{SubOffset}[i] \leftarrow (\mathsf{SubInterval}[i]/4 + (D_{sub} - 1)) \bmod (\mathsf{SubInterval}[i]/2)$
end for
return SubBase, SubOffset, SubInterval

We propose a derivation algorithm that generates a reduced configuration scheduling scheme from a precomputed scheduling scheme. The derivation algorithm is shown in Algorithm 3.

The proposed derivation method offers two key advantages: flexibility, as it supports multiple instance sizes without regenerating configuration tables, and hardware reusability, as it preserves identical FSM timing and dataflow without reconfiguration overhead. By enabling on-demand derivation of configuration tables for variable-size instances, our approach significantly enhances the reusability and flexibility of the scheduling algorithm. It allows the same hardware design to adapt to different zk-SNARK workloads while preserving conflict-free execution and pipeline efficiency.

4 A Resource-efficient Sumcheck Accelerator Architecture

4.1 Design Overview

The hardware accelerator is built on Algorithm 1. The architecture is shown in Fig.2. It contains a centralized *FSM controller*, a shared *Modular Multiplier* array, three major computation modules (*Equality MLE Generation*, *MLE Combine*, and *MLE Update*), and a *Challenge Generation* module for Fiat–Shamir challenge derivation in the sumcheck algorithm.

Figure 2 illustrates the overall architecture of our proposed ASIC-based sumcheck accelerator, PipeSC. The accelerator follows a pipelined design that tightly integrates control, computation, and memory subsystems to sustain continuous dataflow throughout the sumcheck protocol.

At the top level, a centralized *FSM Controller* orchestrates the execution of each protocol phase—*MLE Combine*, *Challenge Generation*, and *MLE Update*—by issuing fine-grained control signals and synchronization commands. The *Memory I/O* module serves as the streaming interface between on-chip logic and external memory, performing burst-aligned read and write operations under FSM supervision. A *Width Adapter* within this module converts 256-bit field element operands into Advanced eXtensible Interface (AXI)-aligned burst transfers compatible with the external memory interface, handling data packing and unpacking between the computation core and the memory subsystem. This design allows polynomial data to be continuously streamed into and out of the accelerator without intermediate buffering.

The core computation pipeline consists of three major modules. The *MLE Combine* module performs evaluation, combination, and accumulation of combined MLE evaluations to generate each round's proof message. The *MLE Update* module executes the polynomial folding step based on the random challenge, updating coefficients for the next round. The *Equality MLE Generation* module constructs the equality polynomial $eq(r, X)$ using a conflict-free modular multiplication schedule; it is time-multiplexed onto the shared

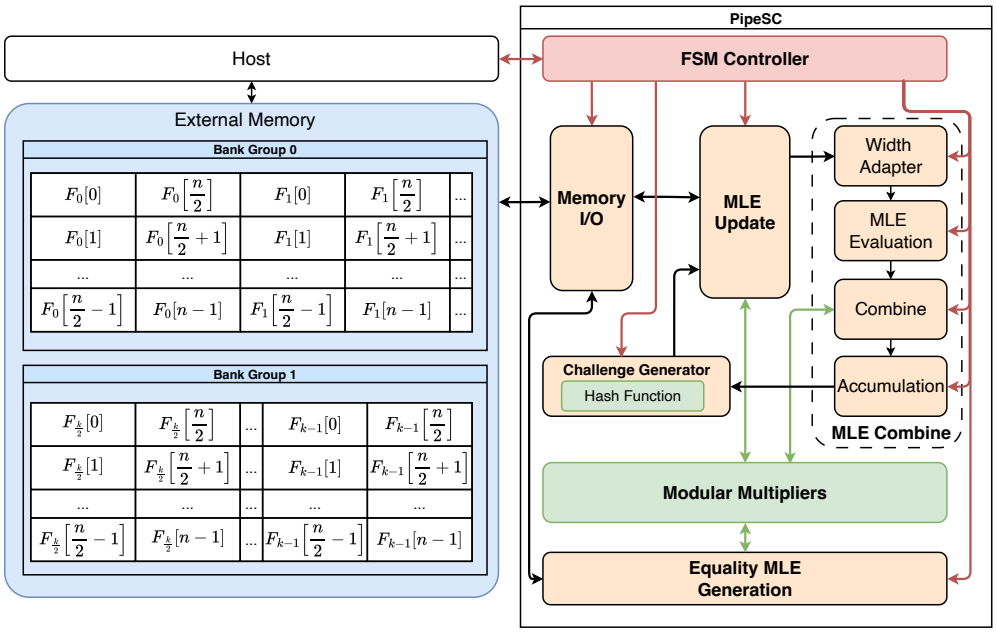

Figure 2: Overall architecture of PipeSC. The accelerator consists of a centralized FSM controller, a Memory I/O interface to external memory, three major computation modules (Equality MLE Generation, MLE Combine, and MLE Update), and a Challenge Generation module, all sharing modular multipliers. The memory layout and access pattern are described in Section 4.5.

multiplier array during a designated FSM state rather than executing simultaneously with other modules. Beside the three major modules, a *Challenge Generation* module based on SHA3-256 hash function provides round challenges for both the update phase and equality polynomial generation.

All arithmetic operations across these modules are executed by a shared array of T deeply pipelined *Modular Multipliers*, which serve as the accelerator's primary compute units. The primary architectural parameter T determines both the performance and area of PipeSC. By coordinating computation and data movement through the FSM-based scheduler, PipeSC achieves near-ideal pipeline utilization and fully overlaps modular arithmetic with off-chip data transfers, enabling high-throughput and scalable sumcheck acceleration. The cycle analysis in Section 4.6 is parameterized by T and the pipeline depths, enabling straightforward scaling analysis across different hardware budgets.

4.2 Finite-state Machine based Scheduler Design

The scheduler based on a compact finite-state machine (Fig. 3) coordinates memory I/O and arithmetic. It overlaps computation with transfers so the multiplier array stays busy.

From *Idle*, the machine waits for a start signal and enters *Combine*. A batch of combine operations runs; on a batch-done event, the machine moves to *Index-Check*. If more combine batches remain in the current round, it returns to *Combine*. If the round is complete but further rounds remain, it proceeds to *Challenge*. If all rounds are complete, it goes to *Finish*. In *Challenge*, the machine derives the Fiat–Shamir challenge in the Challenge Generation module and waits until the challenge is ready, then transitions to *Update*. *Update* performs in-place, size-halving updates; on an update-done event, it returns

to *Combine* to start the next round. *Finish* emits a completion signal and returns to *Idle*.

At each state, the controller programs round-specific strides and offsets for external memory access. This enforces data dependencies, keeps accesses coalesced, reduces stalls, and maintains high utilization of arithmetic units.

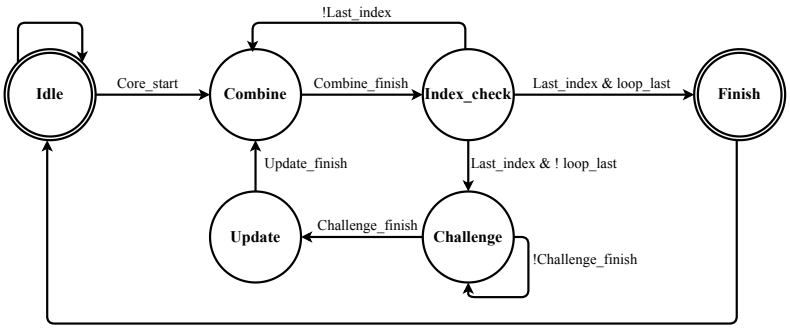

Figure 3: State transition diagram of PipeSC

4.3 Pipeline Sumcheck Modules Design

To accelerate the sumcheck protocol, PipeSC employs three major computation modules: *Equality MLE Generation*, *MLE Combine*, and *MLE Update*. In addition, the architecture includes a *Challenge Generation* module that performs several SHA3-256 hash computations to derive round challenges. Since its computational cost occupies only a negligible fraction of the total execution cycles, we treat it separately from the three major computation modules.

4.3.1 MLE Combine Module

The *MLE Combine* module is designed to perform the core combination phase of each sumcheck round through a fully pipelined computation process. It executes three sequential operations corresponding to the mathematical steps of the sumcheck algorithm.

First, the module evaluates the point-wise values of the multilinear extension at the points $(0, 1, 2, \cdots, d)$ where d denotes the degree of combine logic. Second, these evaluation results are combined according to the round-specific *combine logic*, which merges multiple intermediate values into a single combined set. Third, the combined results are accumulated to produce a partial sum that contributes to the final round message (proof message).

This entire process (i.e., evaluation, combination, and accumulation) is implemented as a streaming pipeline. Each stage of the pipeline is carefully balanced such that once the pipeline is filled, one set of combined values is accumulated into the output every clock cycle (i.e., the initiation interval equals one). The pipeline depth is determined by the number of pipeline stages in the modular multiplier and adder chain; since all stages operate with single-cycle latency, the pipeline depth equals the pipeline latency in clock cycles. Throughput is sustained continuously without stalls once the pipeline is filled.

The *MLE Combine* module repeats this $(d + 1)$-stage pipelined procedure to generate the point-value representation of the round polynomial required for proof generation. By maintaining constant throughput and overlapping arithmetic operations across $(d + 1)$ iterations, the module achieves high hardware utilization and deterministic latency, serving as the front-end engine of the pipelined sumcheck architecture.

4.3.2 MLE Update Module

Following the combination phase and challenge generation phase, the *MLE Update* module performs the coefficient update stage of the sumcheck protocol. Its primary function is to update the multilinear extension (MLE) representation according to the random challenge α_i generated by *Challenge Generation* module, preparing the new coefficient vector for the next round of computation.

The *MLE Update* module is also implemented as a fully pipelined streaming engine. Each pipeline stage consists of modular multipliers and adders that perform the weighted combination between corresponding coefficients of $f^{(i)}(X|X_i = 0)$ and $f^{(i)}(X|X_i = 1)$. Once the pipeline is filled, one updated coefficient is produced per clock cycle, maintaining an initiation interval of one. To preserve data dependency and scheduling determinism, the *MLE Update* module is also controlled by the FSM-based scheduler. Through this design, the *MLE Update* module achieves continuous throughput, low latency, and efficient reuse of modular arithmetic units.

Overall, the *MLE Update* module transforms the combined evaluations from the current round into the updated coefficient vector for the next, thereby closing the computation loop of one complete sumcheck iteration.

4.3.3 Modular Multiplier Module

The modular multiplier is implemented as a pipelined composite module consisting of the *Karatsuba Multiplier* followed by the *Barrett Reduction* unit. The datapath first computes the full-width integer product of two 256-bit field elements and then reduces the result modulo the target prime.

In the multiplication stage, each operand is decomposed into high and low limbs, and the product is computed using the Karatsuba method. Specifically, the datapath generates three sub-products for the limb pairs and recombines them with add/shift logic to form the full intermediate product.

The intermediate product is then passed to the Barrett reduction stage, which performs modular reduction using precomputed constants associated with the modulus. This stage consists of a sequence of pipelined multiplications, additions, and subtractions, followed by a final correction step to guarantee that the output lies in the field range. The entire modular multiplier is fully pipelined.

4.4 Pipelined Equality MLE Generation

Based on the conflict-free modular multiplication scheduling scheme in Algorithm 2, we design a pipelined equality MLE generation module. The overall architecture of the equality MLE generation module is illustrated in Fig. 4. The module computes the equality MLE $eq(r, X)$ by progressively combining intermediate products across binary-tree layers. It is composed of three main submodules: the Controller, the Register Array, and the Computation Core.

The controller orchestrates the generation process through multiple tables which are generated by the Algorithm 2 and timers. The Base Table and Offset Table store layer-specific addressing information, while the Interval Table defines the update intervals for the intermediate registers. The Global Timer and Level Timers are cycle counters that synchronize the computation sequence across different pipeline stages by counting elapsed clock cycles from configurable initial values and generating deterministic read/write addresses and trigger signals aligned with the modular multiplier pipeline depth D_{MM}. The Update Control unit generates read/write addresses and triggers register updates according to the scheduling pattern derived from (n, D_{MM}) parameters.

The register subsystem contains two major parts: Random Registers and Intermediate Registers. The random registers store the random vector $r = (r_0, r_1, \ldots, r_{n-1})$, divided

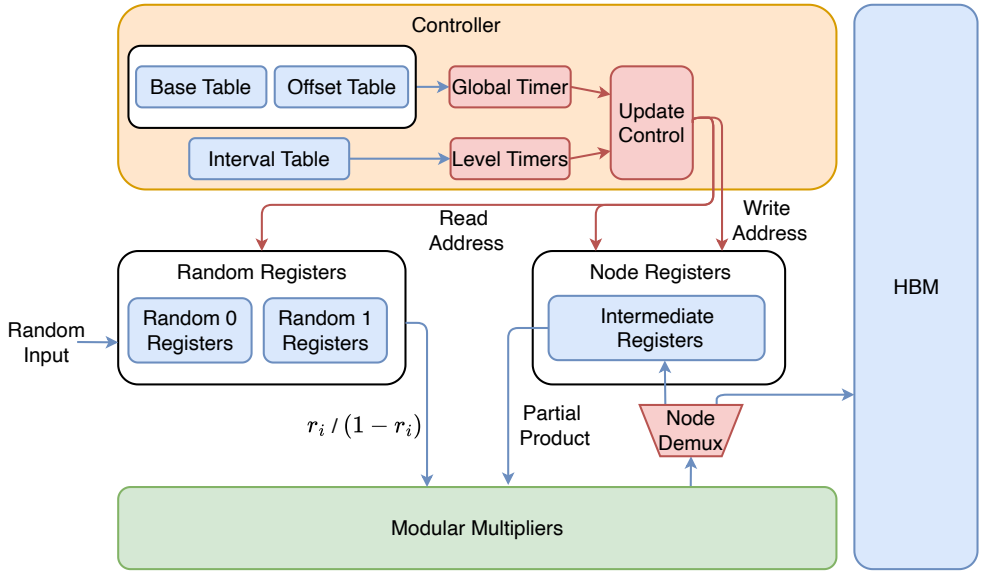

Figure 4: The architecture of equality MLE generator

into r_i and $(1 - r_i)$ forms to support the computation of $r_i/(1 - r_i)$ terms. The Node Registers hold intermediate results of the binary-tree computation. Each layer's partial products are written back to the Intermediate Registers under the control of the FSM, ensuring dependency correctness between consecutive levels.

The computation core consists of several modular multipliers, whose number is configurable and shared across the Equality MLE generation module and other submodules for subsequent operations. These multipliers perform the core arithmetic operations for generating equality polynomials. Intermediate values are passed through a node demultiplexer, which routes the partial products either to the on-chip node registers used by the Equality MLE generation module or to the external memory system used by the streaming sumcheck datapath. This modular design supports a streaming-style dataflow and maintains high pipeline utilization, enabling scalability under limited on-chip resources.

The design integrates on-chip register storage and explicit address scheduling. The FSM issues deterministic read and write commands to maintain conflict-free data transfer between layers, enabling scalability to large 2^n-point equality polynomial instances.

Overall, this architecture achieves efficient and pipelined evaluation of the equality MLE, minimizing redundant modular multiplications while maintaining deterministic timing control across computation layers.

Total Cycle Analysis. Our equality polynomial generation module produces the final results (leaf nodes) with an output interval of two cycles per node. In the last computation layer, the first leaf node (i.e., node $j = 2^N$) starts execution at time $(N - 2) \cdot D_{\mathsf{MM}}$. The modular multiplication corresponding to the last leaf node begins $(2^N - 1)$ cycles after the first leaf node starts, and completes after an additional D_{MM} cycles. Therefore, the total execution cycles can be expressed as:

$$C_{\text{ours}} = (N - 1) \cdot D_{\mathsf{MM}} + (2^N - 1) \cdot 2. \tag{13}$$

4.5 Memory Layout and Access Pattern Design

Because the working set of large-size sumcheck instances exceeds on-chip capacity, PipeSC relies on a multi-bank external memory system to store intermediate MLE values across sumcheck rounds. The memory layout and access pattern are therefore designed to exploit bank-level parallelism, support burst-oriented transfers, and enable in-place updates while keeping the compute pipeline utilized.

We store all k_{in} input MLEs and k_{eq} equality MLEs in external memory as contiguous, bank-aligned arrays (Fig. 2). The key idea is to split each MLE table into two halves and map these halves to different memory banks. This layout matches the operand access pattern of sumcheck, where both *MLE Combine* and *MLE Update* consume paired entries from each MLE in every cycle. By storing the two halves of each MLE in different banks, the datapath can fetch the two operands associated with the same MLE simultaneously, while different MLEs are distributed across bank groups whenever possible. This bank-aware placement exposes bank-level parallelism to the memory system and avoids access conflicts that would otherwise arise if both operands of an MLE were served by the same bank. Within each bank, data are organized in fixed-size chunks chosen as an integer multiple of the AXI burst length, yielding long sequential bursts and supporting in-place, size-halving write-back during *MLE Update*.

The access patterns are designed for three phases. In *Equality MLE Generation*, Memory I/O streams $eq(\mathbf{r}, X)$ once to external memory in index order using aligned, maximum-length bursts. In the *MLE Combine* phase, Memory I/O issues synchronized read streams for the active MLEs, fetching the two operands of each MLE from its two assigned banks in the same cycle and delivering them to the pipeline without bank conflicts. In the *MLE Update* phase, Memory I/O uses the same two-bank organization to read the front and back halves of each MLE concurrently and writes the size-halved results *in place*. The updated MLE is again split into two halves, which are written back to the front regions of the same bank pair so that the two-operand access pattern is preserved for the next round. Because the paired operands are distributed across different banks and the read and write indices differ, the two reads and one write can be pipelined without conflicts.

In particular, our architecture adopts a streaming-based method in which all intermediate MLE values across sumcheck rounds are stored in external memory, regardless of the instance size. This applies uniformly to all configurations: MLE table entries are read from external memory, processed through the modular multiplier pipeline, and updated results are written back to external memory. On-chip memory is reserved exclusively for the Equality MLE generation module, which requires only $O(N)$ registers (where N is the number of variables), and for small control and buffering structures.

4.6 Cycle Analysis of PipeSC

The total cycle count of PipeSC depends on two sets of parameters: instance-dependent parameters (n, k, k_{eq}, d, m) describing the characteristics of the sumcheck protocol, and architecture-dependent parameters $(T, D_{\mathsf{MM}}, D_{\mathsf{combine}}, D_{\mathsf{update}})$ determined by the hardware configuration. Recall n is the number of variables, k the total number of multilinear extensions including k_{eq} equality MLEs, d the degree of the combine logic, m the number of modular multiplications per combine operation, and T the number of parallel multipliers; $D_{\mathsf{MM}}, D_{\mathsf{combine}}, D_{\mathsf{update}}$ denote the pipeline depths of the modular multiplier, MLE combine, and MLE update modules, respectively.

Let $\theta_{\text{inst}} = (n, k, k_{eq}, m, d)$ and $\theta_{\text{arch}} = (T, D_{\mathsf{MM}}, D_{\mathsf{combine}}, D_{\mathsf{update}})$. The total cycle count of PipeSC can be approximated as the sum of three major components: the equality MLE generation, the combine operation, and the MLE update stage. Each term consists of two parts: (1) the computation latency determined by the total number of modular

multiplications divided by the available multiplier parallelism T, and (2) the pipeline fill/drain latency proportional to the number of protocol rounds n and the pipeline depth of each stage. The total cycle count is then given by:

$$\begin{aligned} C_{\mathsf{sumcheck}}(\theta_{\text{inst}}, \theta_{\text{arch}}) &\approx C_{\mathsf{eqGen}} + C_{\mathsf{combine}} + C_{\mathsf{update}} \\ &= \left(\frac{2 \cdot (2^n - 1) k_{eq}}{T} + (n-1) D_{\mathsf{MM}} \right) \\ &\quad + \left(\frac{(2^n - 1) m (d+1)}{T} + n D_{\mathsf{combine}} \right) \\ &\quad + \left(\frac{(2^n - 1) k}{T} + n D_{\mathsf{update}} \right). \end{aligned} \tag{14}$$

This analytical model helps guide our architectural optimizations and parameter tuning by making the cost of each protocol stage explicit. The model focuses on arithmetic pipeline cycles and assumes sustained bandwidth of the external memory as discussed in Section 4.5. In Section 5, we validate this model against empirical measurements on our hardware prototype.

5 Evaluation and Performance

5.1 Evaluation Methodology

Our hardware accelerator PipeSC was designed in SystemVerilog and evaluated on both ASIC and FPGA platforms. The FPGA prototype is used to validate functionality. The performance and efficiency results are reported based on the ASIC implementation.

FPGA Prototype. We implemented a prototype on the Xilinx Alveo U280 FPGA, which provides 9024 DSP slices, 1304K lookup tables (LUTs), 2547 36 Kb BRAMs, and 1280 288 Kb URAMs. The U280 also features 8 GB of HBM2 and 32 GB of DDR4 memory, enabling high-throughput off-chip data access. Our FPGA prototype instantiates $T = 5$ modular multipliers, constrained by the available DSP resources on the U280, and operates at 200 MHz. The FPGA implementation was synthesized and implemented using Xilinx Vivado 2023.2 with default optimization settings. We measured resource utilization and timing after place-and-route to confirm the correctness and scalability of our pipeline design. The FPGA results serve as hardware-level validation consistent with the ASIC RTL implementation.

ASIC Implementation. Our ASIC implementation uses $T = 16$ modular multipliers and was synthesized using Synopsys Design Compiler with a 12 nm TSMC standard-cell library targeting a 1.0 GHz frequency. All reported area, timing, and power results are based on post-synthesis estimates. The reported results include the PipeSC compute core and the core-side Memory I/O logic only; the HBM2 controller and PHY are not included in the synthesis results. For external memory access, we use an HBM2-based bandwidth model rather than a fully instantiated HBM subsystem. Our Memory I/O module is therefore evaluated as core-side RTL logic that generates burst-aligned request streams, and the external memory system is modeled by its sustained bandwidth.

Bandwidth Requirement Analysis. With $T = 16$ modular multipliers operating at 1 GHz, each consuming two 256-bit operands per cycle, the peak read bandwidth requirement is = 1024 GB/s. Thus, we assume a multi-stack HBM2 organization with pseudo-channels in the performance model, and use a sustained bandwidth target of approximately = 1024 GB/s to feed all modular multipliers.

In particular, the latency model assumes that long burst transfers and small on-chip FIFOs can amortize memory-access latency and keep the arithmetic pipeline supplied. If the realized memory subsystem delivers lower sustained bandwidth or cannot hide read latency, the end-to-end latency would increase accordingly. By contrast, the bandwidth provided by a single DDR4 interface is lower than the bandwidth requirement and would therefore cause severe pipeline stalls under the same compute configuration. More generally, any external memory system that can provide sufficient sustained bandwidth could support the design. However, under the bandwidth target considered in this work, HBM is a more practical choice than a conventional DDR configuration.

Baselines. We compare our design against three representative state-of-the-art implementations across different platforms. For fair comparison, all are evaluated under the same 256-bit prime field configuration and using the same sumcheck instance defined by the gate-constraint relation in HyperPlonk, as described in Eq. 5. For the CPU baseline, we evaluate the sumcheck implementation from the Arkworks library [ark24], which employs the same algorithm as ours. The evaluation is conducted on an Amazon EC2 `c5a.8xlarge` instance equipped with 32 vCPUs and 64 GB of memory. For the GPU baseline, we adopt BatchZK [LCW+25], a GPU-accelerated zkSNARK framework that includes an optimized sumcheck module implemented in CUDA. For the ASIC baseline, we use zkSpeed [DMAk+25], an ASIC accelerator for HyperPlonk [CBBZ23] that integrates a dedicated sumcheck submodule. For fairness, we extract and report the Sumcheck-specific area and power results from their ASIC implementation for direct comparison with ours.

5.2 FPGA Prototype Validation

Table 1 lists the FPGA implementation results. The Equality MLE generation module produces results every two cycles once the pipeline is filled, and the entire sumcheck instance with $n = 20$ variables completes in 4.21×10^6 cycles (approximately 21.1 ms at 200 MHz). These results verify that our control logic and pipeline scheduling function correctly in hardware and that the latency behavior is consistent with our analytical model in Section 4.6.

Table 1: FPGA prototype results (Five modular multipliers). The latency values are measured under a sumcheck instance with $n = 20$. † indicates that the latency refers only to the cycle count of a single modular multiplier and is not included in the total cycle count.

Component	LUTs	Registers	DSPs	BRAMs	Latency (cycles)
Equality MLE Generation	10,552	14,588	0	0	532,652
MLE Combine	26166	30947	0	20	2,626,848
MLE Update	906	1574	0	0	1,051,984
Challenge Generation	2324	2441	0	1	2860
Modular Multipliers	120,729	161,113	1650	0	58†
Control	321	131	0	0	-
Total	191,177	223,924	1650	21	4,214,344

5.3 ASIC Performance Analysis

Table 2 reports the detailed ASIC synthesis results under a 12 nm TSMC process. The core module dominates both area (94.5%) and power (85.6%), mainly due to the modular

Table 2: ASIC synthesis results of PipeSC (16 modular multipliers). "Other" includes the Interconnect and Control submodules.

Module	Area (mm^2)	Average Power (mW)
Core	2.162	1033.08
Challenge Generation	0.009	12.05
MLE Combine	0.127	171.88
MLE Update	0.101	170.54
Multiplier Array	1.904	1154.88
Modular Multiplier (×16)	0.119	72.18
Adapter	0.021	28.99
Equality MLE Generation	0.005	5.57
Other	0.122	167.17
Total	**2.289**	**1205.82**

multiplier array. Each modular multiplier contributes roughly 0.12 mm^2 and 72 mW, indicating a strong coupling between arithmetic throughput and hardware cost. The remaining submodules (e.g., MLE Combine, Update, and Challenge Generation) together account for less than 15% of the total area, demonstrating that our architectural optimization primarily benefits computational density and modular multiplier utilization.

Table 3: Cycle breakdown of PipeSC ASIC across different task sizes (16 modular multipliers, $D_{\mathsf{MM}} = 20$). All cycle counts are reported in million cycles; percentages are shown in parentheses.

Size	Total Mcycles	Eq. MLE Gen. Mcycles (%)	MLE Combine Mcycles (%)	MLE Update Mcycles (%)	Challenge Gen. Mcycles (%)
2^{18}	0.355	0.045 (12.6%)	0.221 (62.3%)	0.089 (25.0%)	0.0002 (<1%)
2^{20}	1.321	0.167 (12.6%)	0.823 (62.3%)	0.330 (25.0%)	0.0009 (<1%)
2^{22}	4.985	0.630 (12.6%)	3.107 (62.3%)	1.244 (25.0%)	0.0034 (<1%)
2^{24}	17.408	2.200 (12.6%)	10.851 (62.3%)	4.345 (25.0%)	0.0118 (<1%)

Runtime Breakdown Analysis Table 3 presents both the cycle counts and percentage-based runtime breakdown of PipeSC across varying task sizes. The MLE Combine phase consistently dominates the total execution time (∼62%), due to the $m \cdot (d+1)$ modular multiplications required per entry. Under the gate-combination logic of HyperPlonk, we have $m = 5$ and $d = 3$. The MLE Update phase accounts for approximately 25%, while Equality MLE Generation contributes around 13%. Challenge Generation is negligible (<1%). Although the cycle counts increase with the task size, the percentage distribution remains stable because all three major phases scale linearly with 2^n. As indicated by Eq. (14), under a fixed protocol setting and combine logic, the parameters k, k_{eq}, m, and d are fixed, so the relative contributions of C_{eqGen}, C_{combine}, and C_{update} are preserved as n grows. By contrast, under different combine logics, the parameters may change, which in turn alters the relative cost of each phase and leads to a different runtime percentage breakdown. The pipeline fill/drain overhead (proportional to n) is negligible for large instances.

5.4 Comparison with Prior ASIC Design

Table 4 compares the latency and Area–Time Product (ATP) between our design and zkSpeed. Since zkSpeed adopts a 7 nm process and reports an area of 52.38 mm^2, we normalize our 12 nm results to the same technology node for a fair comparison, using a scaling factor of 2.0× for area (as zkSpeed scales from TSMC 22 nm down to 7nm using a scaling factor of 3.6× for area). We note that cross-node scaling may introduce residual uncertainty due to differences in standard-cell libraries and routing density across foundries. After normalization, our design uses 1.15 mm^2. Both areas exclude external memory. In terms of raw latency, our design is 12.37–16.16× slower than zkSpeed, which is expected given the significant area disparity: zkSpeed occupies 52.38 mm^2, whereas our design uses only 1.15 mm^2, a ∼45× difference in area. Our design achieves **3.22×–3.68× ATP improvement**, demonstrating a superior balance between latency and area. This efficiency stems from the conflict-free scheduling and reuse of modular multipliers, which sustain high utilization without proportional area scaling.

Table 4: Latency and area-time product (ATP) comparison with zkSpeed. The Slowdown metric is computed as the latency of our design divided by that of zkSpeed. Entries marked with † are linearly scaled w.r.t. size.

Size	Latency (ms)			ATP		
	zkSpeed	Ours	Slowdown	zkSpeed	Ours	Improv.
2^{18}	0.022†	0.355	16.14×	1.15	0.41	2.80×
2^{19}	0.044†	0.711	16.16×	2.30	0.82	2.80×
2^{20}	0.094	1.321	14.05×	4.92	1.52	3.24×
2^{21}	0.150	2.125	14.17×	7.86	2.44	3.22×
2^{22}	0.356	4.985	14.00×	18.65	5.73	3.25×
2^{23}	0.707	9.665	13.67×	37.03	11.11	3.33×
2^{24}	1.407	17.408	12.37×	73.70	20.02	3.68×

Notably, we do not include NoCap as a direct quantitative baseline because its evaluation setting is not aligned with ours. In particular, NoCap targets a simpler sumcheck combine logic, whereas our evaluation and the zkSpeed baseline both use the HyperPlonk-style combine logic considered in Eq. 5. In addition, NoCap is evaluated over a 64-bit finite field, while both PipeSC and zkSpeed are evaluated for 256-bit prime-field sumcheck. Because these differences materially affect both arithmetic cost and memory behavior, the resulting latency and ATP numbers are not directly comparable. We therefore discuss NoCap in related work for architectural context, while using zkSpeed as the primary prior-ASIC baseline under the matched 256-bit setting.

5.5 Performance Comparison Across Platforms

Table 5 presents the latency comparison with CPU and GPU implementations under varying instance sizes. Our ASIC achieves **1710.1×–2756.2× speedup** over the CPU baseline (Arkworks) and **2.39×–5.02× speedup** over the GPU baseline (BatchZK), demonstrating the effectiveness of our conflict-free pipeline and resource reuse strategy. We note that the speedup comparison between our ASIC (Area is 2.29 mm^2 and average power is 1.21 W) and the GPU platform (NVIDIA GH200 [NVI24], die area ∼814 mm^2, configurable power from 450 W to 1000 W) is made across substantially different resource envelopes, involves a ∼350× area disparity and a roughly 373×–829× power disparity. This large area gap indicates substantial headroom to scale our ASIC architecture, but the achievable speedup would not grow linearly with silicon area because off-chip memory bandwidth and data movement would become limiting factors. The decreasing speedup at

Table 5: Latency comparison with state-of-the-art CPU and GPU implementations. The Speedup columns report the ratio of the baseline latency to our ASIC latency. Entries marked with † are linearly scaled w.r.t. size (fit through the origin).

Size	Arkworks [ark24] (CPU)		BatchZK [LCW+25] (GPU)		Ours (ASIC)
	Latency (ms)	Speedup	Latency (ms)	Speedup	Latency (ms)
2^{18}	607.1	**1710.1**×	1.782	**5.02**×	0.355
2^{19}	1291.3	**1816.2**×	2.713	**3.82**×	0.711
2^{20}	2810.8	**2127.8**×	3.699	**2.80**×	1.321
2^{21}	5856.9	**2756.2**×	6.392	**3.01**×	2.125
2^{22}	11536.6	**2314.3**×	10.817	**2.47**×	4.985
2^{23}	23073.2	**2387.3**×	23.105†	**2.39**×†	9.665
2^{24}	46146.3	**2650.9**×	46.210†	**2.65**×†	17.408

larger proof sizes reflects the fixed parallelism of our current 16-multiplier configuration; scaling the number of on-chip multipliers together with the memory system would improve this trend. The rationale for ASIC acceleration lies in its deterministic latency, significantly lower energy consumption per proof, and the potential for integration into larger SoC-based proving systems where area and power budgets are constrained.

6 Conclusion

We presented PipeSC, a resource-efficient hardware accelerator for the sumcheck protocol. By modeling modular-multiplication dependencies and introducing a conflict-free DFS-based scheduler, we achieved memory-efficient and partial pipelined equality-MLE generation with high hardware utilization. Implemented on FPGA and ASIC, PipeSC delivers up to 2756.2× and 5.02× speedups over CPU and GPU baselines, and 3.68× higher area–time efficiency than prior ASICs. This work demonstrates that algorithm–architecture co-design can substantially advance zk-SNARK acceleration.

References

[ark24] arkworks contributors. arkworks zksnark ecosystem. https://arkworks.rs, 2024. Project website.

[AST24] Arasu Arun, Srinath T. V. Setty, and Justin Thaler. Jolt: SNARKs for virtual machines via lookups. In Marc Joye and Gregor Leander, editors, *EUROCRYPT 2024, Part VI*, volume 14656 of *LNCS*, pages 3–33. Springer, Cham, May 2024.

[BCG+14] 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 Computer Society Press, May 2014.

[BCS16] Eli Ben-Sasson, Alessandro Chiesa, and Nicholas Spooner. Interactive oracle proofs. In Martin Hirt and Adam D. Smith, editors, *TCC 2016-B, Part II*, volume 9986 of *LNCS*, pages 31–60. Springer, Berlin, Heidelberg, October / November 2016.

[CBBZ23] Binyi Chen, Benedikt Bünz, Dan Boneh, and Zhenfei Zhang. HyperPlonk: Plonk with linear-time prover and high-degree custom gates. In Carmit Hazay

and Martijn Stam, editors, *EUROCRYPT 2023, Part II*, volume 14005 of *LNCS*, pages 499–530. Springer, Cham, April 2023.

[CMT12] Graham Cormode, Michael Mitzenmacher, and Justin Thaler. Practical verified computation with streaming interactive proofs. In *Proceedings of the 3rd Innovations in Theoretical Computer Science Conference*, pages 90–112, 2012.

[DMAk+25] 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.

[DP25] Benjamin E. Diamond and Jim Posen. Succinct arguments over towers of binary fields. In Serge Fehr and Pierre-Alain Fouque, editors, *EUROCRYPT 2025, Part IV*, volume 15604 of *LNCS*, pages 93–122. Springer, Cham, May 2025.

[FS86] Amos Fiat and Adi Shamir. How to prove yourself: Practical solutions to identification and signature problems. In Andrew M. Odlyzko, editor, *Advances in Cryptology - CRYPTO '86, Santa Barbara, California, USA, 1986, Proceedings*, volume 263 of *Lecture Notes in Computer Science*, pages 186–194. Springer, 1986.

[GKR15] Shafi Goldwasser, Yael Tauman Kalai, and Guy N. Rothblum. Delegating computation: Interactive proofs for muggles. *Journal of the ACM*, 62(4):27:1–27:64, 2015.

[GWC19] Ariel Gabizon, Zachary J. Williamson, and Oana Ciobotaru. PLONK: Permutations over Lagrange-bases for oecumenical noninteractive arguments of knowledge. Cryptology ePrint Archive, Report 2019/953, 2019.

[Ing25] Ingonyama. Icicle: a cuda-enabled gpu library for zk acceleration. https://github.com/ingonyama-zk/icicle, 2025. GitHub repository.

[LCW+25] 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 Lieven Eeckhout, Georgios Smaragdakis, Kaitai Liang, Adrian Sampson, Martha A. Kim, and Christopher J. Rossbach, editors, *Proceedings of the 30th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 1, ASPLOS 2025, Rotterdam, The Netherlands, 30 March 2025 - 3 April 2025*, pages 100–115. ACM, 2025.

[LFKN92] Carsten Lund, Lance Fortnow, Howard J. Karloff, and Noam Nisan. Algebraic methods for interactive proof systems. *Journal of the ACM*, 39(4):859–868, 1992.

[LWQ+24] Tao Lu, Haoyu Wang, Wenjie Qu, Zonghui Wang, Jinye He, Tianyang Tao, Wenzhi Chen, and Jiaheng Zhang. An efficient and extensible zero-knowledge proof framework for neural networks. Cryptology ePrint Archive, Report 2024/703, 2024.

[LXZ21] Tianyi Liu, Xiang Xie, and Yupeng Zhang. zkCNN: Zero knowledge proofs for convolutional neural network predictions and accuracy. In Giovanni Vigna and Elaine Shi, editors, *ACM CCS 2021*, pages 2968–2985. ACM Press, November 2021.

[MDAK+25] Jianqiao Mo, Alhad Daftardar, Joey Ah-Kiow, Kaiyue Guo, Benedikt Bünz, Siddharth Garg, and Brandon Reagen. Mtu: The multifunction tree unit for accelerating zero-knowledge proofs. In *Proceedings of the 14th International Workshop on Hardware and Architectural Support for Security and Privacy*, pages 19–27, 2025.

[NVI24] NVIDIA. *NVIDIA GH200 Grace Hopper Superchip Datasheet*. NVIDIA, 2024.

[Set20] Srinath Setty. Spartan: Efficient and general-purpose zkSNARKs without trusted setup. In Daniele Micciancio and Thomas Ristenpart, editors, *CRYPTO 2020, Part III*, volume 12172 of *LNCS*, pages 704–737. Springer, Cham, August 2020.

[SLDS24] Nikola Samardzic, Simon Langowski, Srinivas Devadas, and Daniel Sánchez. Accelerating zero-knowledge proofs through hardware-algorithm co-design. In *57th IEEE/ACM International Symposium on Microarchitecture, MICRO 2024, Austin, TX, USA, November 2-6, 2024*, pages 366–379. IEEE, 2024.

[Tha22] Justin Thaler. Proofs, arguments, and zero-knowledge. *Foundations and Trends® in Privacy and Security*, 4(2–4):117–660, 2022.

[TW24] Louis Tremblay Thibault and Michael Walter. Towards verifiable FHE in practice: Proving correct execution of TFHE's bootstrapping using plonky2. Cryptology ePrint Archive, Report 2024/451, 2024.

[WWX+25] Yuanju Wei, Kaixuan Wang, Binwu Xiang, Xinxuan Zhang, Yi Deng, Hailong Wang, and Xudong Zhu. Packed sumcheck over fields of small characteristic with application to verifiable FHE. *IACR Cryptol. ePrint Arch.*, page 719, 2025.

[XZC+22] 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 Heng Yin, Angelos Stavrou, Cas Cremers, and Elaine Shi, editors, *ACM CCS 2022*, pages 3003–3017. ACM Press, November 2022.

[XZS22] Tiancheng Xie, Yupeng Zhang, and Dawn Song. Orion: Zero knowledge proof with linear prover time. In Yevgeniy Dodis and Thomas Shrimpton, editors, *CRYPTO 2022, Part IV*, volume 13510 of *LNCS*, pages 299–328. Springer, Cham, August 2022.