zkPHIRE: A Programmable Accelerator for ZKPs over HIgh-degRee, Expressive Gates

Alhad Daftardar*
NYU Tandon
Brooklyn, NY, USA
ajd9396@nyu.edu

Jianqiao Mo*
NYU Tandon
Brooklyn, NY, USA
jm8782@nyu.edu

Joey Ah-kiow
NYU Tandon
Brooklyn, NY, USA
ja4844@nyu.edu

Benedikt Bünz
NYU Courant
Brooklyn, NY, USA
bb@nyu.edu

Siddharth Garg
NYU Tandon
Brooklyn, NY, USA
sg175@nyu.edu

Brandon Reagen
NYU Tandon
Brooklyn, NY, USA
bjr5@nyu.edu

Abstract—**Zero-Knowledge Proofs (ZKPs) have emerged as powerful tools for secure and privacy-preserving computation. ZKPs enable one party to convince another of a statement's validity without revealing anything else. This capability has profound implications in many domains, including: machine learning, blockchain, image authentication, and electronic voting. Despite their potential, ZKPs have seen limited deployment because of their exceptionally high computational overhead, which manifests primarily during proof generation. To mitigate these overheads, a (growing) body of researchers has proposed hardware accelerators and GPU implementations for kernels and complete protocols. Prior art spans a wide variety of ZKP schemes that vary significantly in computational overhead, proof size, verifier cost, protocol setup, and trust. The latest, and widely used ZKP protocols are intentionally designed to balance these trade-offs. A particular challenge in modern ZKP systems is supporting complex, high-degree gates using the SumCheck protocol. We address this challenge with a novel programmable accelerator that efficiently handles arbitrary custom gates via SumCheck. Our accelerator achieves upwards of $1000\times$ geomean speedup over CPU-based SumChecks across a range of gate types. We integrate this unit into a full-system accelerator, zkPHIRE, which achieves $1486\times$ geomean speedup over CPU and $11.87\times$ speedup over the state-of-the-art at iso-area. zkPHIRE is the first accelerator to scale to problem sizes of 2^{30} nominal constraints while maintaining small proof sizes and programmability.**

I. INTRODUCTION

Zero-Knowledge Proofs (ZKPs) are cryptographic protocols that allow a prover to convince one or more verifiers that a claim is correct without revealing any information beyond the claim's validity. For example, a prover can demonstrate possession of a valid authentication key without disclosing the key. ZKPs have many compelling applications, including privacy-preserving authentication, trustworthy AI, blockchain scalability, and recently, private age verification [53]. A major limitation of state-of-the-art ZKP protocols is their high computational cost–generating a single proof on a CPU can take minutes to hours. As a result, there is growing interest in accelerating ZKPs, both with programmable solutions (e.g., GPUs [24], [36]) and ASICs [12], [13], [49], [66], [67].

ZKP protocols continue to evolve and improve. Many have been proposed, offering different trade-offs in runtime, proof size, verifier time, and setup (universal vs. application-specific), and security guarantees. A major family of ZKP protocols relies on the SumCheck protocol [35]. SumCheck-based protocols have $O(N)$ prover time, as compared to NTT-based protocols (a popular alternative [17], [20]), which require $O(N \log(N))$ prover operations. The SumCheck protocol is highly flexible and is at the core of many classic and modern ZKP protocols [6], [9], [19], [50], [56], [61], [64], which have relatively fast prover time. While SumChecks were part of previously accelerated systems [12], [49], [55], they studied specific instantiations of the SumCheck protocol rather than the protocol family more generally. Another major distinguishing factor between protocols is the type of polynomial commitment scheme [25] that is used. Some of these protocols, such as **Spartan** [50]**+Orion** [61] use faster Merkle tree commitments, at the cost of large proofs (many MBs), while **HyperPlonk** [9] uses a slower commitment scheme that produces significantly smaller proofs (a few kBs).

ZKP protocols typically encode computations as arithmetic circuits (ACs) – directed acyclic graphs where each node (or gate) enforces a local algebraic constraint, such as addition ($f = a + b$) or multiplication ($f = a \cdot b$) over a finite field. We generally denote the number of gates by N. These circuits are then arithmetized by translating gates into polynomial constraints, which are later verified using polynomial checks for correctness. The SumCheck protocol is used to reduce checking N gates to checking a single, randomized gate formula. A key feature of newer SumCheck-based protocols is the use of *custom, high degree gates* that encapsulate more computation per gate by using higher-degree polynomial constraints (e.g., $f = a \cdot b^5 + c^2$). This reduces the overall number of gates needed to represent a computation, resulting in large efficient gates. The move to custom gates is exemplified in the Halo2 arithmetization language [63]. Halo2 enables a circuit designer to specify the types of gates they want to use *in each circuit.*

arXiv:2508.16738v1 [cs.AR] 22 Aug 2025

1

However, this level of customizability introduces significant hardware challenges. Unlike NTTs, which have fixed, regular dataflows, SumCheck computations vary with each gate's structure and polynomial degree. High-degree polynomials require more modular multiplications and deeper reductions, leading to increased bandwidth pressure, varying memory access patterns, and more complex scheduling. Designing efficient SumCheck hardware thus involves a tradeoff: either build fixed-function units for specific gate shapes, or develop a general, programmable architecture that can dynamically adapt to diverse polynomials without significantly compromising performance.

zkSpeed [12] exemplifies the first approach, using dedicated hardware for Vanilla gates (additions and multiplications) within the HyperPlonk protocol. While this provides good performance for a fixed gate set, it limits flexibility: zkSpeed's SumCheck unit cannot easily handle complex custom gates like those seen in the Halo2 [63] arithmetization, reducing its applicability across general ZKP workloads. Furthermore, zkSpeed's reliance on large on-chip scratchpads limits scalability, as its memory footprint grows with the number of gates, becoming impractical beyond 2^{24} gates, a scale increasingly common in emerging ZKP applications.

Generalizable SumCheck cores are needed beyond ZKPs, too. In verifiable computing, SumCheck serves as the core verification mechanism in protocols that provide integrity guarantees without requiring zero knowledge. However, previous work in this area, including verifiable ASICs [55], has also relied on custom SumCheck units with fixed functions, again restricting applicability and reuse.

In this work, we adopt the second approach with *zkPHIRE*, introducing a programmable SumCheck accelerator that supports arbitrary polynomial structures and gate types. zkPHIRE addresses the irregularity and data reuse challenges intrinsic to SumCheck with fused compute pipelines, tree-based interconnects for efficient reductions, and flexible scheduling to handle diverse polynomial structures and degrees. This flexibility opens the door to a wide range of applications across ZKPs and verifiable computing. We evaluate our programmable SumCheck unit on a diverse array of gate types, and then embed it into a full-system accelerator, zkPHIRE. zkPHIRE improves upon zkSpeed with more efficient memory organization and module abstraction that allows hardware to be reused across protocol steps. We conduct a protocol-level evaluation using the HyperPlonk with custom gates as a representative case study. Our key contributions are:

- A novel, programmable SumCheck unit that supports arbitrary high-degree multilinear polynomials, enabling compatibility with custom gate types.
- The *first* hardware evaluation of high-degree gates that offer significant performance and scalability benefits
- A unified multifunction forest and optimized modular inverse unit that (respectively) save 15.2% and 4.2× area with improved utilization compared to prior work [12].
- A full-system accelerator, zkPHIRE, that achieves $11.87\times$ geomean speedup over zkSpeed, $1486\times$ geomean

speedup over CPU, and supports succinct proofs for problem sizes up to 2^{30} nominal constraints.

II. BACKGROUND

A. Zero-Knowledge Proofs

Today, the state-of-the-art ZKP protocols are zero-knowledge Succinct Non-interactive Arguments of Knowledge (zkSNARKs). They have three key properties:

- **zero-knowledge:** The verifier learns nothing beyond the validity of the statement.
- **succinctness:** The proof is short and verified quickly.
- **non-interactivity:** The proof is a single message, eliminating the need for back-and-forth communication.

Modern zkSNARKs combine an Interactive Oracle Proof (IOP) and a Polynomial Commitment Scheme (PCS). The IOP encodes the witness and computation as polynomials, which the verifier checks via probabilistic queries. Applying the Fiat-Shamir heuristic [16] makes public-coin IOPs non-interactive. The PCS lets the prover commit to a polynomial and later open it at chosen points, ensuring both binding (the prover cannot change the committed polynomial) and hiding (nothing about the polynomial is revealed until it is opened).

Different zkSNARKs balance proving time, proof size, and security assumptions based on their IOP and PCS designs. Groth16 [20] offers fast verification and tiny proofs, ideal for blockchains, but requires a circuit-specific (i.e., per-application) trusted setup and scales poorly with circuit size. Code-based protocols [61] use error-correcting codes and Merkle commitments to achieve transparency and post-quantum security, but at the cost of multi-megabyte proofs and slower verification. HyperPlonk [9] trades off slightly larger proofs for universal setup and linear-time proving.

B. Multi-Scalar Multiplication

Multi-Scalar multiplication (MSM) is a key kernel used in *polynomial commitments* and relies on elliptic curve cryptography. MSMs involve computing a linear combination (i.e., dot product) of scalars and corresponding (2D or 3D) points on an elliptic curve. Given scalars $k_1, k_2, \ldots, k_n$ and corresponding elliptic curve points $P_1, P_2, \ldots, P_n$, the goal is to efficiently compute $s = \sum_i k_i P_i$. MSMs are computationally intensive because each term $k_i P_i$ involves a point-scalar multiplication – a costly elliptic curve operation. MSMs constitute a significant portion of the total runtime in many zkSNARKs, including Groth16 and HyperPlonk. As a result, prior work has focused on accelerating MSMs using both application-specific and programmable platforms [13], [24], [30], [34], [36], [66], [67]. Rather than computing each scalar multiplication directly, optimized approaches such as Pippenger's algorithm [45] are commonly employed. Pippenger's algorithm improves efficiency by restructuring the computation to rely more heavily on faster operations like point addition and point doubling.

C. SumCheck

SumCheck [35] is a round-based Interactive Proof (IP) protocol where a prover, $\mathcal{P}$, attempts to convince a verifier, $\mathcal{V}$, that the sum of a multivariate polynomial, $f(X_1, \ldots, X_\mu)$, over the *boolean hypercube* (all boolean values of $X_i \ldots X_\mu$) equals a claimed value, C. The protocol proceeds in μ rounds, one for each variable of the polynomial. In each round i, $\mathcal{P}$ sends $\mathcal{V}$ a univariate polynomial:

$$s_i(X_i) = \sum_{x_{i+1} \in \{0,1\}} \cdots \sum_{X_\mu \in \{0,1\}} f(r_1, \ldots, r_{i-1}, X_i, \ldots, X_\mu)$$

where $r_1, \ldots, r_{i-1}$ are $\mathcal{V}$'s random challenges from prior rounds. $\mathcal{V}$ checks that the sum of $s_i(0) + s_i(1)$ equals the value claimed in the previous round (or the initial claim C for the first round). $\mathcal{V}$ then chooses a random field element r_i and sends it to $\mathcal{P}$, fixing the variable $X_i = r_i$ for the next round. After μ rounds, $\mathcal{V}$ evaluates f at the final point $(r_1, \ldots, r_\mu)$. If all checks pass, $\mathcal{V}$ accepts.

ZKPs typically execute the SumCheck protocol over *multilinear extensions* (MLEs) – polynomials with degree at most 1 in each variable. MLEs can represent multivariate functions compactly and are well-suited for hardware because they can be stored and accessed as flat lookup tables indexed by binary-encoded inputs. Throughout this paper, we will use the terms "MLEs", "MLE tables," and "polynomials" interchangeably.

1) Polynomials and Gates: Polynomials are important in ZKP protocols because they encode and help enforce the correctness of the computation being proven. A popular encoding scheme is Plonk [17]. Plonk's *arithmetization* scheme converts the underlying computation into an arithmetic circuit consisting of gates. Plonk circuits support additions, multiplications, conditionality, and equality checks. They can be represented by the composite polynomial

$$f = q_L w_1 + q_R w_2 + q_M w_1 w_2 - q_O w_3 + q_C \quad (1)$$

consisting of individual polynomials q_i and w_i. These are known as selectors (or enables) and witnesses, respectively. Each of these polynomials encodes a function of a multivariate input vector $X = [X_1, X_2, \ldots X_\mu]$, which encodes the gate index in binary. For example, if $\mu = 3$, there are 8 gates, and gate index 3 is represented by evaluating $f(1, 1, 0)$. For a multiply operation, we would set $q_M(1, 1, 0) = 1$, $q_O(1, 1, 0) = 1$ and the other enable polynomials to 0. Then, gate 3 would execute correctly if and only if $f(1, 1, 0) = 0$, that is, $w_1(1, 1, 0) \cdot w_2(1, 1, 0) - w_3(1, 1, 0) = 0$. Representing circuits as polynomials naturally suits SumCheck-based provers where we can check (with additional constraints) that each gate evaluates to 0 and that $\sum_X f(X) = 0$.

2) High-degree, Expressive Gates: The challenge of using Plonk gates is that all computations must be mapped to only what is supported by the Plonk polynomial, similar to mapping program code to a RISC ISA. In some cases, this results in circuits that require millions ($> 2^{20}$) of gates, and in turn, polynomials with millions of evaluations.

Recent protocols and libraries [9], [63] support *expressive, high-degree* gates, which encode complex operations and

significantly reduce overall gate count. For example, Hyper-Plonk's *Jellyfish* gate captures common patterns like hashing; across benchmarks, HyperPlonk demonstrates upwards of $32\times$ reduction in gate counts when transitioning from *Vanilla* (less expressive) Plonk gates to Jellyfish gates. Similarly, Halo2 uses custom gates for elliptic curve operations. However, these expressive gates increase the complexity of the SumCheck algorithm used to verify them.

3) SumCheck on Multilinear Polynomial Products: Sum-Check over a single polynomial, such as $\sum f(X_1, X_2, X_3)$ (with $\mu = 3$) is relatively straightforward. Using the equation for $s_1(X_1)$ from Section II-C, and viewing $f(X)$ in its MLE table form, in round 1, $s_1(0)$ is the sum of all even indices (where $X_1 = 0$), and $s_1(1)$ is the sum of all odd indices ($X_1 = 1$). This gives us two evaluations of the *univariate, linear* polynomial $s_1(X_1)$ to be sent to the verifier. We then perform an *MLE Update* by fixing X_1 to random challenge r_1. At fixed X_2, X_3, entries $(0, X_2, X_3)$ and $(1, X_2, X_3)$ represent a line over X_1, so fixing $X_1 = r_1$ means extrapolating the line to r_1: $f(r_1, X_2, X_3) = f(0, X_2, X_3) \cdot (1 - r_1) + f(1, X_2, X_3) \cdot r_1$. This operation halves the size of each MLE table.

SumCheck over polynomials using Plonkish encodings (like Equation 1) is more complicated because there are multiple terms with products of polynomials. Figure 1 shows the dataflow for a single polynomial product term. For a d-degree polynomial, we require $d+1$ evaluations for $s_i(X_i)$ after each round of SumCheck. This is done by evaluating all terms at $d+1$ points. Each pair of entries in the MLE table are actually evaluations at $X_i = 0$ and 1; these values are then *extended* to $X_i = 2, \ldots, d$, using the same formula as MLE Update. Then, the extensions for all polynomials in a term are multiplied together. These products are summed (accumulated) down the MLE table, yielding $d+1$ values for a given term. When there are more than one term, this process will be repeated across terms with the $d+1$ evaluations summed across terms to form the final polynomial $s_i(X_i)$ for round i.

III. The Programmable SumCheck Accelerator

A. Challenges

At its core, SumCheck offers many degrees of parallelism. Different MLE tables can compute their extensions in parallel, extensions for different MLE pairs *within* a table can be computed in parallel, and the products *across* different extensions can be computed in parallel. Given a SumCheck polynomial, an optimal circuit can be constructed to maximize reuse and throughput, as done in zkSpeed [12].

However, building a generalizable SumCheck unit poses several challenges. For one, the unit must support a diverse set of composite polynomial structures used to express different types of constraints or implement different kinds of operations. The unit must support an arbitrary number of terms and an arbitrary degree. This means that the optimal resource allocation (e.g., number of modular multipliers) for one composite polynomial may be suboptimal or wasteful for another composite polynomial. Additionally, many composite
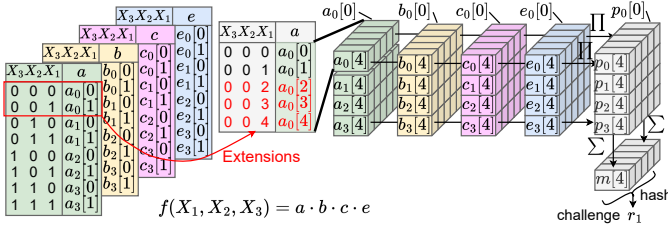
Figure 1. The SumCheck dataflow of size-2^3 polynomial $f(X) = abce$. For a degree-4 polynomial f, every MLE evaluation pair is used to extend to five evaluations. Extended evaluations are multiplied, summed, and hashed to get a challenge that is used to update MLE tables for the next SumCheck round.
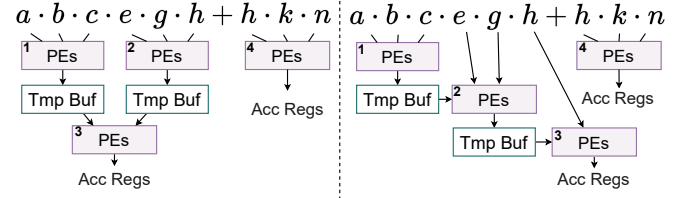

Figure 2. Graph decomposition used to schedule computation on our SumCheck unit. The method on the right minimizes on-chip temporary MLE buffer size. Numbers inside each node indicate the step in the schedule. When processing term hkn, h is already prefetched during the prior step.

polynomials have individual polynomials that repeat. For example, in $f = a \cdot b \cdot e + c \cdot e + e \cdot g$, the e polynomial appears three times. While we could parallelize all computations for just the first term $(a \cdot b \cdot e)$ before proceeding to the next term, this would leave out a clear opportunity for data reuse. Indeed, zkSpeed's design captures immediate reuse, but is tailored to HyperPlonk's vanilla gate polynomials. Their design philosophy overprovisions resources for simpler polynomials, and is unable to support complex polynomials.

B. Datapath

Our programmable SumCheck unit seeks to balance the tradeoff between resource allocation and capturing immediate data reuse. Figure 3 shows the datapath for our programmable SumCheck unit. It consists of several processing elements (PEs), with MLE Update units, Extension Engines (EEs) and Product Lanes. Instead of streaming all individual polynomials' MLE entries simultaneously, we proceed one term at a time, fetching a *tile* of data from each MLE table and storing in local scratchpad buffers. We allocate 16 scratchpad buffers, more than sufficient to accommodate polynomial structures we see in current ZKP systems. The scratchpad buffers are banked to match the number of PEs for parallel accesses. This ensures that individual polynomials reused in subsequent terms need not be fetched from off-chip memory for a given tile.

For a given term, the polynomials are read into the Update units and EEs, where their extensions are generated by a series of modular adders and subtractors. Specifically, in round 1 of SumCheck, two values are read at a time per MLE and directly fed to generate extensions. In all subsequent rounds, four values are read per MLE, because we pipeline the updates into the extension computation. Updating with four inputs ensures that two values are generated and fed to the EEs to match the throughput of round 1. These updated values are also written to FIFOs that buffer writebacks to off-chip, forming the (halved) tables for the next SumCheck round. In later rounds of SumCheck, when MLE tables can fit fully on-chip, writebacks to off-chip are eliminated and updated values are directly routed to the scratchpad banks.

After the extensions for each MLE are generated, they must be packed into their respective groups, e.g., all extensions at 1 are routed together. Each group of extensions is then fed into a Product Lane (PL). Each PL is equipped with $E - 1$ fully pipelined modular multipliers, where E is the number of EEs.

Each PL's output is accumulated into a register corresponding to the extension, e.g., all products at $X_i = 2$ are always routed to register 2. We allocate 32 registers (up to degree 31); higher degrees are supported by storing in scratchpads.

C. Mapping Polynomials to EEs

We devise a scheduler that determines which MLE tables are fed to EEs and PLs using a graph-based decomposition (see Figure 2). In this example, we assume there are 3 EEs, so only 3 MLEs can be handled on-chip at a time. The polynomial is adding a degree-6 term and a degree-3 term. Since there are more MLEs than EEs, we must store the partial products for all extensions; this is done using a dedicated *Tmp* MLE buffer.

A natural way to schedule the MLEs is using the tree-based approach on the left in Figure 2. However, this approach would require storing two distinct intermediate products that would need to be combined. As the polynomial degree grows, this tree-based approach would require more and more intermediate buffers. Instead, we can use the accumulation based schedule on the right. This scheme only requires 1 Tmp MLE buffer that accumulates extension products within the same term. Notice that this scheme uses the same number of steps as the balanced tree while minimizing temporary storage.

In this way, the scheduler traverses the polynomial term-by-term, deciding which MLEs are handled by the PEs in each step. These steps are programmed into a controller on-chip. In each step j, the PEs read the MLEs identified for step j. The controller simultaneously prefetches a tile of data for each MLE in step $j+1$, but only if the MLEs have not already been fetched on-chip. For example, the h MLE would be prefetched during step 2 (on the right), and the only k and n MLEs would be prefetched during step 3. Another advantage of this scheme is that prefetch bandwidth is more balanced over steps.

D. Mapping Extensions to PLs

Since our SumCheck unit must handle any number of extensions, we may run into scenarios where there are K extensions but only $P < K$ lanes. In this case, we have an initiation interval $II = K/P$ and use delay buffers to queue unmatched extensions. In Figure 3, with $K = 5$ and $P = 3$, the first cycle maps extensions 0–2 to PLs, while extensions 3–4 are buffered for the next cycle. Scheduling then proceeds in a pipelined fashion, with buffered and incoming extensions interleaved across PLs. While K depends on polynomial degree, the
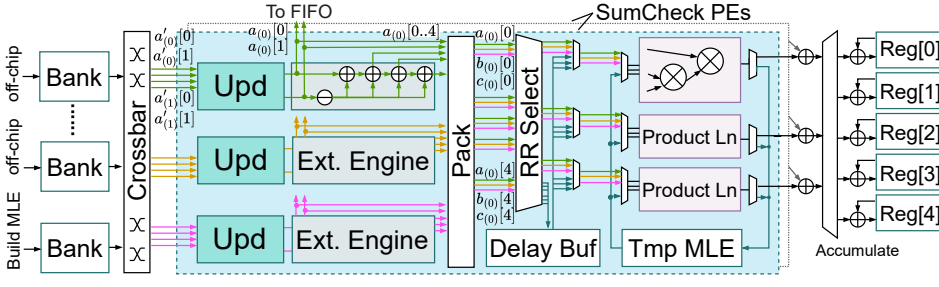
Figure 3. Programmable SumCheck Architecture. The Update unit updates the MLE a' to halved-size MLE a for the next SumCheck round. For deg-4 term $abce$, Extension Engines generate five evaluations 0-4.
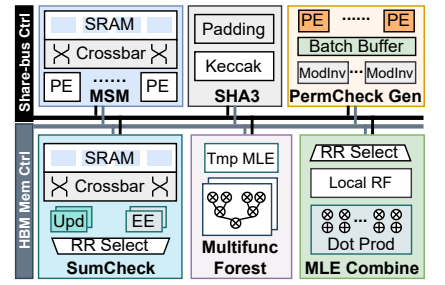
Figure 4. Overall Architecture of zkPHIRE.

schedule for a given (K, P) is static and programmed into the controller at the start of each SumCheck.

E. Built-in support for ZeroChecks

Many ZKP protocols use SumCheck to verify circuit correctness by checking that a polynomial $f(X)$ evaluates to zero across all inputs. However, simply verifying $\sum_X f(X) = 0$ is insufficient. Individual gates could evaluate to nonzero values (meaning they were incorrectly executed), but cancel each other out, so the total sum would still be zero. To prevent this, protocols like [7], [50], [61] use *ZeroCheck*, which multiplies $f(X)$ with a randomized auxiliary polynomial $f_r(X)$ and checks that $\sum_X (f(X) \cdot f_r(X)) = 0$. $f_r(X)$ is constructed on-the-fly from a vector of random challenges generated by a SHA module (referred to as *Build MLE* in zkSpeed), and zkPHIRE integrates this auxiliary polynomial construction directly into the SumCheck datapath. During the first round, one product lane is dedicated to computing $f_r(X)$, with its outputs routed to scratchpad banks and the EEs in parallel with the fetched MLE elements. This reduces the number of lanes for product computations by one in that round but avoids separate $O(N)$ precomputation and memory overhead, reusing existing PL multipliers to mask the cost.

IV. ZKPHIRE ARCHITECTURE

We include the programmable SumCheck unit as part of a larger accelerator framework for the HyperPlonk protocol. We describe the steps of the HyperPlonk protocol and its mapping onto zkPHIRE units. HyperPlonk is based on the BLS12-381 elliptic curve, which uses 255-bit scalars (datatypes used by MLEs) and 381-bit elliptic curve points (used for point addition in MSMs). We explore the use of both arbitrary and fixed prime modular multipliers (as done in [49]); the latter saves us roughly 50% on area and improves our computational density by roughly $2\times$. Figure 4 shows the overall architecture.

A. HyperPlonk Protocol Overview

HyperPlonk has a very complex dataflow. Here, we describe the high-level protocol and what hardware modules each step requires. The 5 main steps are Witness Commitments, Gate Identity Check, Wire Identity Check, Batch Evaluations, and Polynomial Opening. In **Witness Commitments**, the prover commits to their private input values (witnesses) using multiscalar multiplications (MSMs) so that they cannot claim other

witness values later. We run this step with the MSM module. In **Gate Identity Checks**, the prover shows that all gates in the circuit were computed correctly via ZeroChecks. We run this step with our SumCheck unit and a Multifunction Forest unit. In **Wire Identity Checks**, the prover proves that the wiring between gates is consistent with the circuit's intended connections. This is done by constructing wiring permutation polynomials and running a PermutationCheck (henceforth called a PermCheck) and then committing to those polynomials. We run this step with a PermCheck Generator unit, the SumCheck and Multifunction Forest units, and the MSM unit. In **Batch Evaluations**, the prover evaluates all committed polynomials at various points provided by the verifier. We run this step with the Multifunction Forest unit. In the **Polynomial Opening**, the prover reveals the values of multiple committed polynomials at specific points and proves that it is consistent with the original commitment. The prover then runs a SumCheck to combine these polynomial evaluation claims into one (we call this *OpenCheck* as per zkSpeed). The combined polynomial commitment is then opened using the MSM unit. Overall this step uses an MLE Combine unit, the SumCheck and Multifunction Forest units, and the MSM unit.

Masking ZeroCheck: In zkSpeed and the HyperPlonk baseline, Gate Identity and Wire Identity steps run serially, despite verifying independent circuit properties. zkPHIRE includes support to overlap the Gate Identity's ZeroCheck with the MSMs during Wiring Identity. Since MSMs dominate runtime and have low bandwidth pressure due to high data reuse [12], [13], this scheduling effectively masks ZeroCheck latency.

B. Accelerator Units

*1) **Sparsity Handling for SumCheck**:* ZKP protocols often exhibit significant sparsity: enable MLEs (q_i) are mostly binary, while constant and witness MLEs (q_c, w_i) are typically 90% sparse [12], [13], [66]. zkPHIRE extends the SumCheck unit to exploit this sparsity for improved efficiency.

zkSpeed opts to use large on-chip scratchpads to store these MLEs, because they are reused throughout steps of the HyperPlonk protocol. However, as the problem size scales, the area consumed by the scratchpad grows, with 10s of MBs dedicated to address translation buffers alone, and a total on-chip storage of nearly 300 MB. Furthermore, the MLEs are only useful for round 1 of SumChecks. For subsequent rounds, the MLEs
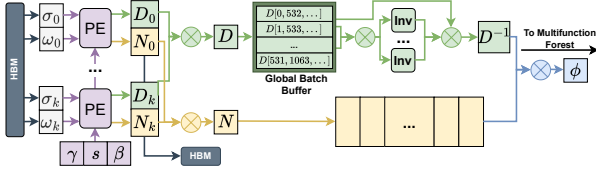
Figure 5. The PermCheck Generator Module.

are updated and written to off-chip memory, so the global scratchpad is idle for roughly 50% of the SumCheck runtime. zkPHIRE opts to use smaller scratchpad buffers to store tiles of each MLE. While this results in more accesses to off-chip memory and we incur more fill/drain latency penalties, it allows for more area to be dedicated core computation structures within SumCheck and other accelerator modules. The improved speedups offset the latency hits incurred by eliminating large scratchpads. To eliminate address translation overheads, zkPHIRE fetches lightweight *per-tile offset buffers* for witness MLEs. The SumCheck controller decodes 255-bit elements using these offsets, while binary MLEs are stored directly without translation. This scheme incurs minimal bandwidth overhead while enabling more compact storage.

2) Multifunction Forest: In HyperPlonk, there are several sub-steps that can be mapped to a binary-tree computation, including MLE evaluation, building MLEs, computing the product MLE (π), and multiplication tree. We adopt the same base tree unit architecture as zkSpeed's MTU [38]. However, unlike zkSpeed – which employs a single tree unit for all sub-step computations while maintaining a separate, dedicated SumCheck module – zkPHIRE introduces a more unified and efficient approach. In zkPHIRE, we implement the product lanes of the programmable SumCheck unit using a *multifunction forest*. This allows modular multipliers initially provisioned just for SumCheck to be shared across other protocol steps. The shared multifunction forest optimization helps zkPHIRE achieve the same latency as zkSpeed at iso-workload-length with 15.2% fewer modular multipliers.

3) MSM: We use the same MSM architecture as zkSpeed, since we target the same protocol, HyperPlonk. In Hyper-Plonk, the number of MSMs only depends on the number of witnesses. The underlying hardware is unchanged, we just compute more MSMs using it. Jellyfish gates have five witnesses, so we run five Sparse MSMs and three Dense MSMs.

4) MLE Combine: The MLE Combine module, shown in Figure 4, is used for various element-wise operations and dot products before and after the OpenCheck in Polynomial Opening. For these steps, we fetch MLE tables from off-chip into up to 6 local SRAM buffers and perform the operations between MLE entries and/or challenges from prior protocol steps stored in registers. All operations are fully pipelined.

5) PermCheck Generator: PermCheck requires constructing Numerator (N), Denominator (D), Fraction (ϕ), and Product (π) MLEs [12]. Since ϕ is computed on-the-fly from N and D, we design a pipelined unit that generates all three in parallel, producing one element per cycle after warmup

(Figure 5). The Multifunction Forest is used to construct π from ϕ, as in zkSpeed. To accommodate both vanilla and custom gates, we allocate 5 PEs (one per witness), with the ability to support more than 5 through overlapped scheduling and cyclic reuse of PEs. Intermediate $N_{1 \ldots k}$ and $D_{1 \ldots k}$ MLEs are written to HBM and combined via modular multiplications to produce the final N and D MLEs, followed by modular inversion of D.

We build on zkSpeed's Montgomery batching approach [40], which uses a batch size of 64 and dedicated multipliers to mask inversion latency at high area cost. To improve efficiency, we reduce the batch size to 2 and eliminate per-inverse multipliers, instead using two shared multipliers, one for batching and one for output isolation. Since zkSpeed's modular multipliers are $17.7\times$ larger than their inverse units (0.478 mm^2 vs. 0.027 mm^2 in TSMC 22nm), we increase the number of inverse units and schedule them in a round-robin fashion to initiate one inversion every two cycles without backpressure. We find that 266 inverse units is sufficient for this. A shared batch buffer stores inputs; once an inverse completes, it is multiplied by its paired batch element and the slot is reused. Our design achieves a $4.2\times$ area reduction over zkSpeed, without sacrificing throughput.

6) Interconnect and Memory: HyperPlonk's stage-level data obliviousness enables static scheduling of computation and communication via a centralized controller. zkPHIRE comprises six major modules connected by a multi-channel shared bus, which is provisioned to meet peak data movement demands. During SumCheck and Wiring Identity, where bidirectional transfers occur between the SumCheck unit and Multifunction Forest, and one-way from PermCheck Generator to MSM unit, up to three channels are needed to avoid stalls. At our 294mm^2 design point, the peak bandwidth requirement reaches 19 TB/s, comparable to state-of-the-art accelerators [28], [47].

Each module includes local SRAM scratchpads to buffer intermediate data. Smaller buffers (6 MB) serve Perm-Check Generator, MLE Combine, and the Multifunction Forest. Heavier modules—MSM (43 MB) and SumCheck (6 MB)—use highly banked SRAM to support multiple PEs and exploit data reuse. For instance, MSM stores elliptic curve points, while SumCheck stores frequently accessed MLE tiles. All modules connect to off-chip PHYs via the shared bus, which is coordinated to avoid contention and sustain peak throughput.

V. METHODOLOGY

We use Catapult HLS 2024 to generate the RTL for Montgomery multipliers with arbitrary primes (as in prior work [12], [13]) and fixed primes [8], [37], fully-pipelined PADD units, modular inverse units, and the SumCheck PEs. We assume the same elliptic curve as HyperPlonk (BLS12-381), where all MLE datatypes (e.g., in the SumCheck unit) are 255-bit, and all elliptic curve points (e.g., in the PADD) are 381-bit. We use Design Compiler with TSMC 22nm. Our 255b modular multipliers are 0.264/0.478 mm^2 (fixed/arbitrary); our
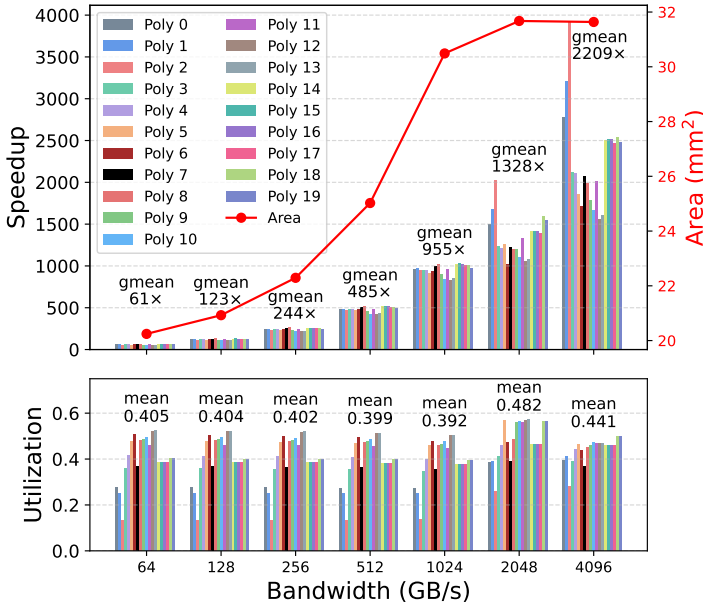
Figure 6. Speedups over 4-threaded CPU implementations of SumChecks on the polynomials in Table I. We pick a different optimal design across each bandwidth technology. Utilization across the design points shown

The HyperPlonk CPU performance is measured on an AMD EPYC 7502 32-core processor running up to 3.35GHz, and the total die size is 296 mm² [5], [41], [54]. We benchmark our SumCheck unit performance relative to prior work [12] and a 4-threaded CPU implementation. We then use workloads from prior work [1], [9] to model the performance of our full architecture vs. 32 CPU threads, up to problem sizes of 2^{30} nominal gates. We show these results in Section VI. We use a cycle-accurate simulator to model runtimes for MSMs. For SumCheck and tree-based computations, we model performance using analytical models, since these steps are data-oblivious. For sparse computations, we use the same workload statistics from prior work [1], [13], [36], [66].

VI. EVALUATION

We present our evaluation in two parts. First, we evaluate our programmable SumCheck unit as a standalone unit in Section VI-A. We use polynomial gates seen in prior works [50], [55] as well as gates used for elliptic curve addition in the Halo2 [63] library. We then evaluate it as part of a broader HyperPlonk accelerator, zkPHIRE, in Section VI-B.

A. Programmable SumCheck

1) Speedup over various gate types: Table I enumerates the polynomials we use in our evaluation. We first benchmark the performance of SumChecks on each polynomial by extending

Table I
LIST OF POLYNOMIAL CONSTRAINTS.

ID	Name	Polynomial
0	Verifiable ASICs [55]	$q_{\text{add}} \cdot (a+b) + q_{\text{mul}} \cdot (a \cdot b)$
1	Spartan 1 [50]	$(A \cdot B - C) \cdot f_\tau$
2	Spartan 2 [50]	$(\text{Sum}_{ABC}) \cdot Z$
Halo2 Constraints [63]		
3	Nonzero Point Check	$q_{\text{point}}^{\text{non-id}} \cdot (y^2 - x^3 - 5)$
4	x-gated Curve Check	$(q_{\text{point}} \cdot x) \cdot (y^2 - x^3 - 5)$
5	y-gated Curve Check	$(q_{\text{point}} \cdot y) \cdot (y^2 - x^3 - 5)$
6	Incomplete Addition 1	$q_{\text{add-incomplete}} \cdot ((x_r + x_q + x_p) \cdot (x_p - x_q)^2 - (y_p - y_q)^2)$
7	Incomplete Addition 2	$q_{\text{add-incomplete}} \cdot (y_r + y_q)(x_p - x_q) - (y_p - y_q)(x_q - x_r)$
8	Complete Addition 1	$q_{\text{add}} \cdot (x_q - x_p)((x_q - x_p)\lambda - (y_q - y_p))$
9	Complete Addition 2	$q_{\text{add}} \cdot (1 - (x_q - x_p)\alpha) (2y_p\lambda - 3x_p^2)$
10	Complete Addition 3	$q_{\text{add}} \cdot x_p x_q (x_q - x_p) (\lambda^2 - x_p - x_q - x_r)$
11	Complete Addition 4	$q_{\text{add}} \cdot x_p x_q (x_q - x_p) (\lambda(x_p - x_r) - y_p - y_r)$
12	Complete Addition 5	$q_{\text{add}} \cdot x_p x_q (y_q + y_p) (\lambda^2 - x_p - x_q - x_r)$
13	Complete Addition 6	$q_{\text{add}} \cdot x_p x_q (y_q + y_p) (\lambda(x_p - x_r) - y_p - y_r)$
14	Complete Addition 7	$q_{\text{add}} \cdot (1 - x_p \cdot \beta)(x_r - x_q)$
15	Complete Addition 8	$q_{\text{add}} \cdot (1 - x_p \cdot \beta)(y_r - y_q)$
16	Complete Addition 9	$q_{\text{add}} \cdot (1 - x_q \cdot \gamma)(x_r - x_p)$
17	Complete Addition 10	$q_{\text{add}} \cdot (1 - x_q \cdot \gamma)(y_r - y_p)$
18	Complete Addition 11	$q_{\text{add}} \cdot (1 - (x_q - x_p)\alpha - (y_q + y_p)\delta)x_r$
19	Complete Addition 12	$q_{\text{add}} \cdot (1 - (x_q - x_p)\alpha - (y_q + y_p)\delta)y_r$
HyperPlonk Polynomials [9]		
20	Vanilla ZeroCheck	$(q_L w_1 + q_R w_2 - q_O w_3 + q_M w_1 w_2 + q_C)f_r$
21	Vanilla PermCheck (α is a scalar)	$(\pi - p_1 p_2 + \alpha(\phi D_1 D_2 D_3 - N_1 N_2 N_3))f_r$
22	Jellyfish ZeroCheck	$(q_1 w_1 + q_2 w_2 + q_3 w_3 + q_4 w_4 + q_{M_1} w_1 w_2 + q_{M_2} w_3 w_4 + q_{H_1} w_1^5 + q_{H_2} w_2^5 + q_{H_3} w_3^5 + q_{H_4} w_4^5 - q_O w_5 + q_{ecc} w_1 w_2 w_3 w_4 + q_C)f_r$
23	Jellyfish PermCheck (α is a scalar)	$(\pi - p_1 p_2 + \alpha(\phi D_1 D_2 D_3 \cdot D_4 D_5 - N_1 N_2 N_3 N_4 N_5))f_r$
20	OpenCheck	$y_1 f_{r_1} + y_2 f_{r_2} + y_3 f_{r_3} y_4 f_{r_4} + y_5 f_{r_5} + y_5 f_{r_5}$

HyperPlonk's existing code base [9]. We then run each polynomial on our programmable SumCheck unit. In this evaluation, we seek to optimize both performance and utilization across all polynomials. We can always allocate EEs and PLs to handle the worst-case polynomial degree; however, this would lead to underutilization on low-degree polynomials. In our design space search, we specify an area and bandwidth constraint and then use the following objective function:

$$\min_{\text{design}} \lambda \cdot (1 - f_{\text{util}}(u_{d,i})) + \lambda \cdot f_{\text{slowdown}}(s_{d,i})$$

Here, $s_{d,i}$ refers to the slowdown of the ith polynomial for a given design d, relative to the fastest runtime for that polynomial in the area-constrained design space. Similarly, $u_{d,i}$ reflects the utilization for design d on polynomial i. f_{slowdown} and f_{util} are aggregation functions; we employ geomean for
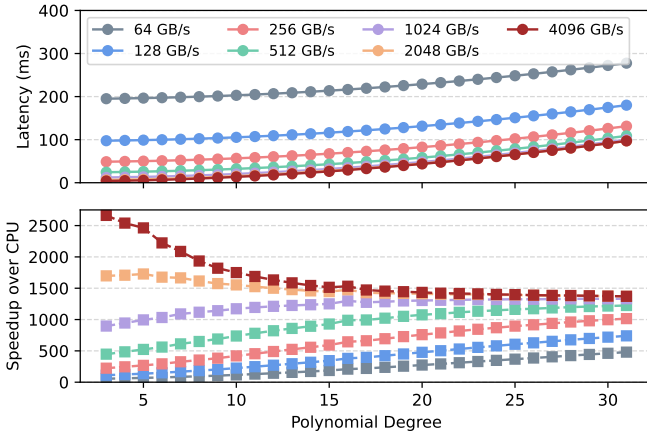
Figure 7. Performance of a fixed SumCheck configuration on high-degree polynomials at different bandwidths. Speedup relative to CPU is also plotted.

Figure 8. Scheduler-induced behavior of SumCheck at fixed bandwidth and PL settings.

slowdown and arithmetic mean for utilization. λ controls the tradeoff between the minimizing slowdown and maximizing utilization. For this analysis, we focus on utilization, using $\lambda = 0.8$. We compare the runtime against the CPU running each polynomial's SumCheck on 4 threads, for which we estimate the total core area to be 37 mm^2 in 7nm [5], [41], [54] and use that as our area constraint (after scaling from 22nm). Across various bandwidths, we find the optimal design point and report speedups and utilizations in Figure 6. For most designs, two EEs and five PLs is optimal.

We observe 50% utilization across benchmarks at each bandwidth, due to three main factors: (1) MLE Update units are inactive during the first round of SumCheck, which accounts for about half of total runtime; (2) lower-degree polynomials require fewer product lanes; and (3) repeating MLEs (e.g., q_{add}) reduce update modmul demand since they are reused across terms. Consequently, when optimizing for utilization, we observed a tendency to limit the number of EEs. Further parallelism and potentially higher utilization can be achieved by mapping multiple terms to EEs when possible, but this would introduce additional complexity in interconnect and increase bandwidth demand. These tradeoffs can be explored further in future work. In contrast, product lane modmuls are usually fully utilized, provided the polynomial degree and memory bandwidth are high enough. Despite moderate utilization, we achieve significant speedups over CPU baselines, reaching nearly 1000× at 1 TB/s of bandwidth.

2) High degree sweep: We analyze the performance scaling of SumChecks as we increase polynomial degree. We sweep polynomial degrees ($d = 2 \ldots 30$) for $f = q_1 w_1 + q_2 w_2 + q_3 w_1^{d-1} w_2 + q_c$, as in HyperPlonk, and measure speedup over CPU. We use the same objective as before and pick a high-performance design under same area constraints (Figure 7). Our results show that low-degree polynomials require HBM-scale bandwidths to achieve 1000× speedups, while higher-degree polynomials can reach similar speedups with DDR5-level bandwidths (e.g., 256 GB/s). Higher-degree polynomials in this scenario exhibit more computations on the same amount
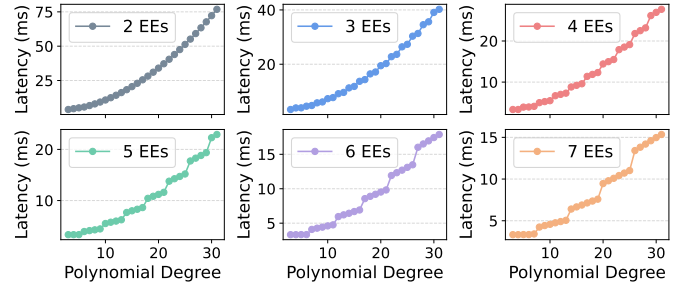
of data, which is why the disparity in speedups across bandwidths is less compared to low-degree polynomials.

At fixed bandwidth, PE count, and EE-per-PE settings, increasing the number of PLs per PE improves performance roughly proportionally. Similar gains are observed when increasing EE count at fixed PLs. Interestingly, as shown in Figure 8, runtime increases in discrete jumps due to the graph decomposition (see Figure 2) the scheduler uses to map polynomials to EEs. For example, under 6 EEs, degree-1–6 polynomials have 1 node in their schedule graph, while degree-7–11 polynomials require 2. Each added node incurs a sharp runtime jump, as it introduces additional products across all MLEs and terms. Within each node cluster, runtime grows more gradually with degree due to early exit optimizations. These trends may help optimize the SumCheck unit when the degree distribution of polynomials is known, by minimizing high-degree outliers that cause runtime spikes.

3) Comparison with Prior ASICs: Figure 9 details our speedups relative to a prior ASIC, zkSpeed. zkSpeed implements the SumChecks by using a custom unified core that maximizes immediate reuse. We also compare with zkSpeed+, which gives zkSpeed the benefits of pipelining MLE updates directly into the extensions and product computations (as we do in zkPHIRE). We assume the same modular multiplier area and 2 TB/s as zkSpeed. We use the same objective function from Section VI-A1 on the same "training set" of polynomials in Table I with a roughly iso-zkSpeed area constraint. We pick a 35.24 mm^2 design to compare with zkSpeed's 30.8 mm^2 SumCheck and MLE Update area. Our design has 4.870 mm^2 of SRAM, while zkSpeed used a much larger global SRAM for storing MLEs, hence we believe this comparison is fair.

We run the same polynomials from zkSpeed's evaluation on our SumCheck unit, and find that compared to the zkSpeed+, we are only 30% slower at iso-area and iso-bandwidth, while offering programmability to support other types of polynomials beyond those in HyperPlonk. We then use the same hardware design point for evaluation on Jellyfish polynomials. The advantage of using Jellyfish gates is that for a given application, while the polynomial complexity grows, the total gate count (i.e., workload length) decreases. We simulate three degrees of gate count reduction, $2\times, 4\times,$ and $8\times$.

We find that, for ZeroCheck, because the Jellyfish polynomial is more complex, the $2\times$ is not sufficient to bring
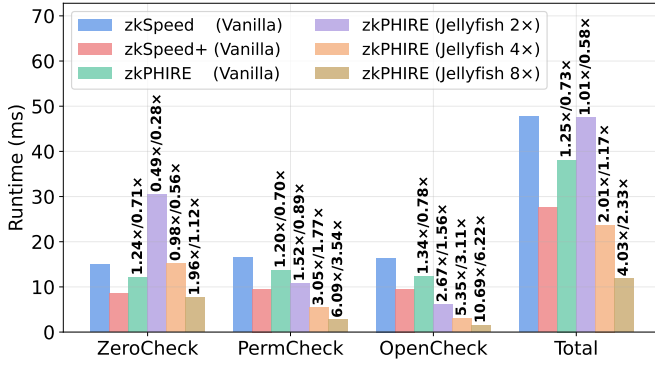
Figure 9. Comparison with prior ASICs. Bars are annotated with speedup numbers over baseline zkSpeed / zkSpeed+.

TABLE II
DESIGN SPACE EXPLORATION PARAMETERS FOR ZKPHIRE.

Module	Design Knob	Values
SumCheck	PEs	1, 2, 4, 8, 16, 32
SumCheck	Extension Engines	2, 3, 4, 5, 6, 7
SumCheck	Product Lanes	3, 4, 5, 6, 7, 8
SumCheck	SRAM Bank Size	$2^{10} - 2^{15}$
MSM	PEs	1, 2, 4, 8, 16, 32
MSM	Window Size	7, 8, 9, 10
MSM	Points/PE	1K, 2K, 4K, 8K, 16K
FracMLE	PEs	1, 2, 3, 4
—	Bandwidth (GB/s)	64, 128, 256, 512, 1T, 2T, 4T

speedups over the vanilla counterpart. However, PermCheck's increase in complexity *is* outweighed by a $2\times$ workload reduction. OpenCheck, which is the same between Jellyfish and Vanilla, exhibits $1.12\times, 3.54\times$ and $6.22\times$ speedup for each gate reduction over zkSpeed+. Combining all three SumChecks, a Jellyfish gate count reduction by $4\times$ is sufficient to outperform vanilla polynomials on both zkSpeed+ and zkPHIRE. These gains are observed despite the inherent overheads of programmability in our accelerator.

B. zkPHIRE

We now incorporate the programmable SumCheck unit into a larger accelerator that accelerates the HyperPlonk protocol, zkPHIRE. Here, our analysis focuses on yielding high-performance designs. In these comparisons, we assume the CPU runs with 32 threads, roughly 296 mm^2 in 7nm. These experiments assume fixed-prime modular multipliers, which is a common technique seen in prior cryptographic accelerators and libraries [1], [9], [49].

1) Pareto Frontier Analysis: We sweep the parameters in Table II to obtain Pareto-optimal designs for each bandwidth, and then construct a global Pareto frontier across all bandwidths. Figure 10 illustrates the design space for a 2^{24} Jellyfish gate workload under seven bandwidth scenarios, ranging from DDR-class to HBM-scale bandwidths. We account for memory PHY overheads [15], [22], [26], [27], [39], [47]–[49], assuming 14.9 mm^2 per HBM2 PHY and 29.6 mm^2 per HBM3 PHY [2]. For 2^{24} Jellyfish gates, the CPU runtime is roughly 182.896 seconds. We can hit roughly $1000\times$ speedup
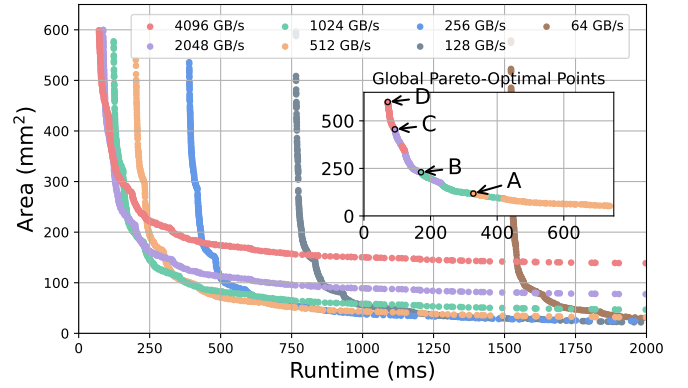

Figure 10. Pareto Frontiers for 2^{24} gates. We plot the individual Pareto curves for each bandwidth tier and the global Pareto frontier in the inset.
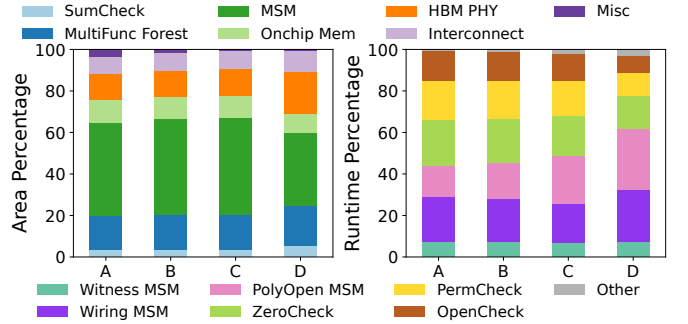

Figure 11. Area (left, top legend) and runtime (right, bottom legend) breakdowns for the selected Pareto points in Figure 10.
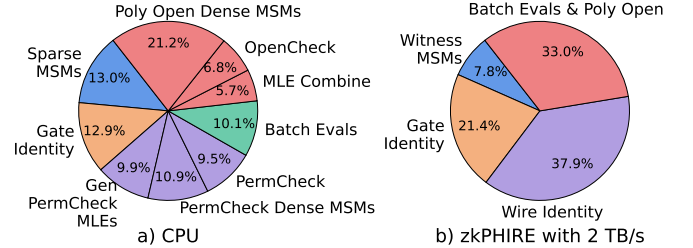

Figure 12. Runtime breakdown for CPU and zkPHIRE for 2^{24} Jellyfish gates. CPU's sequential execution enables finer breakdowns compared to zkPHIRE.

with a 207 mm^2 design and 1 TB/s off-chip bandwidth, and with 294 mm^2 and 2 TB/s, around $1400\times$ at iso-CPU area. For this high-performance regime, a Pareto design around 250 mm^2 with 1 TB/s (design B) achieves nearly $2\times$ and $3\times$ speedup over 512 GB/s and 256 GB/s configurations. These results highlight a key trend in recent cryptographic accelerators where high performance demands investments in high-bandwidth memory technologies (even for small accelerators like NoCap, where the PHY cost alone is $3\times$ the compute area). That being said, with our accelerator framework, for applications with looser prover constraints, sub-DDR-scale bandwidths can still yield $100-200\times$ speedups over multi-threaded CPU baselines.

2) Area, Runtime, and Power: In Figure 11, we analyze selected Pareto points from Figure 10. Across all points, MSM

Table III
AREA AND POWER OF zkPHIRE. OTHER INCLUDES THE PERMCHECK
GENERATOR, MLE COMBINE AND SHA3 UNITS.

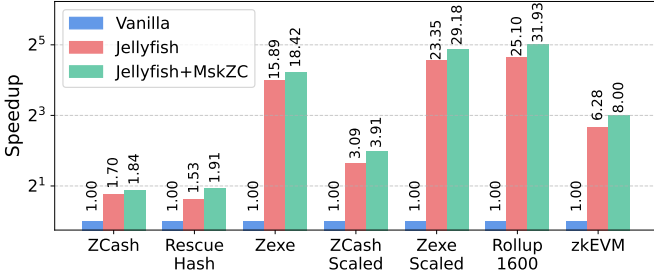|	Area (mm²)	Average Power (W)
MSM (32 PEs)	105.69	58.99
Multifunc Forest (80 Trees)	48.18	40.69
SumCheck (16 PEs)	16.65	14.43
Other	10.64	6.17
Total Compute	**181.15**	**120.29**
SRAM	27.55	3.56
Interconnect	26.42	14.83
HBM3 (2 PHYs)	59.20	63.60
Total Memory System	**113.18**	**81.99**
Total	**294.32**	**202.28**

Figure 13. zkPHIRE speedups across workloads relative to Vanilla Gates. MskZC = Masked ZeroCheck optimization.

Figure 14. High-degree sweep on overall HyperPlonk Protocol. Crossover point is noted. $f = q_1 w_1 + q_2 w_2 + q_3 w_1^{d-1} w_2 + qc$.

dominates the area as expected. As the bandwidth increases, both MultiFunction Forest and MSM portions increase, but from C to D, the absolute MSM area remains unchanged while the SumCheck and Forest area increase. This is also evident in the runtime breakdown: from C to D, the SumCheck portions (ZeroCheck, PermCheck, and OpenCheck) become smaller. Because SumCheck is memory-bound, higher bandwidth incentivizes more resource allocation towards SumCheck-related compute. For low-performance designs, less bandwidth is required, so the MSM is allocated a higher area portion.

Table III shows an exemplar zkPHIRE design point we choose for the following analysis. In this 294 mm² design, there are 32 MSM PEs, with 8 modular multipliers per Multifunction tree PE, and 7 EEs and 5 PLs per SumCheck PE. The design runs on 2 TB/s HBM bandwidth. We use two 32×32 bit-sliced crossbars [28], [44], [47] in the MSM and SumCheck unit, with a shared-bus to form the on-chip bandwidth (up to 19 TB/s) between modules. Figure 12 further breaks down the runtime to compare with HyperPlonk on CPU. The colors indicate the same HyperPlonk step between CPU and zkPHIRE running at the configuration of Table III. We show the latency breakdown for Gate and Wire Identities before performing ZeroCheck masking to show their initial runtime proportions. MSMs dominate runtime before and after acceleration, though compared to zkSpeed's CPU baseline [12], SumChecks take more runtime because the polynomials are complex, exerting more memory pressure on the CPU.

3) Performance from Protocol-level Optimizations: Figure 13 quantifies the benefits of using Jellyfish gates and
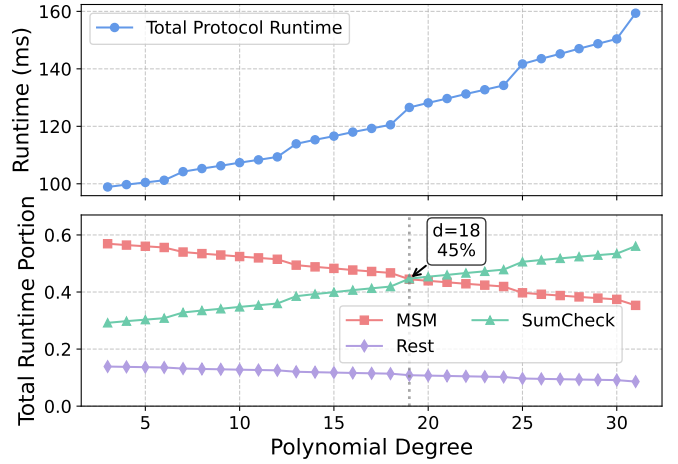
masking ZeroCheck latency across several workloads, since different workloads may respond differently to Jellyfish-style mappings. For large workloads, where the Jellyfish gate count is $> 2^{20}$, the speedups from Jellyfish alone reasonably reach what we expect from table size reduction alone. For smaller workloads, we see lower speedups from Jellyfish alone due to serialization overheads of MSMs and other off-chip memory overheads (e.g. fill/drain latencies) that are dwarfed for large workloads. To highlight this, we test scaled up versions of Zcash and Zexe to 2^{24} and 2^{25} (as done previously [49]), and observe $3.1\times$ and $23.3\times$ speedups from Jellyfish alone. For zkEVM, no estimation of vanilla gates is available [9]; we estimate an $8\times$ reduction to highlight our speedup trends for a hypothetical case. ZeroCheck masking provides additional gains of roughly 25-27% for most workloads, which is what we expect from Amdahl's law by inspecting Figure 12.

4) High-Degree Sweep: In Figure 14, we plot the runtime of the highlighted design point over custom gates represented by $f = q_1 w_1 + q_2 w_2 + q_3 w_1^{d-1} w_2 + qc$. For this family of gates, the number of witnesses is fixed, so the total MSM runtime is constant across all values of d. Consequently, there is a crossover point after which SumChecks dominate the total protocol runtime. This highlights a key design tradeoff: while higher-degree gates reduce overall gate count, they can potentially shift the performance bottleneck from MSMs to SumChecks. Protocol designers must therefore balance gate expressivity against the rising cost of polynomial verification.

5) Performance Comparison: Table IV compares a 300 mm² zkPHIRE design using the same modmuls as zkSpeed+ (366 mm² with MLE Updates pipelined into SumCheck) and without ZeroCheck masking. This is to provide as fair a comparison as possible (zkSpeed+ is roughly 10% faster than zkSpeed). Speedups are reported relative to CPU. zkPHIRE is about 10% slower than zkSpeed+, while offering flexibility to support arbitrary, custom gates. Table V shows zkPHIRE's speedups at iso-CPU area (294 mm²) with fixed primes and ZeroCheck masking. We achieve $1486\times$ geomean speedup and, for the first time, bring workloads on the order of 2^{30}

Table IV
COMPARISON WITH ZKSPEED+ AND CPU FOR VANILLA GATES.

Workload	Gates	Runtime (ms)		
		CPU	zkSpeed+	zkPHIRE
ZCash	2^{17}	1429	1.825 (783×)	2.012 (710×)
Auction	2^{20}	8619	10.171 (847×)	10.88 (792×)
2^{12} Rescue Hashes	2^{21}	18637	19.631 (888×)	20.977 (1822×)
Zexe's Recursive Ckt	2^{22}	37469	38.535 (972×)	41.117 (911×)
Rollup of 10 Pvt Tx	2^{23}	74052	76.356 (969×)	81.362 (910×)
Rollup of 25 Pvt Tx	2^{24}	145500	151.973 (957×)	161.876 (898×)
Rollup of 50 Pvt Tx	2^{25}	325048	—	322.922 (1006×)
Rollup of 100 Pvt Tx	2^{26}	640987	—	645.029 (994×)

Table V
COMPARISON WITH CPU FOR JELLYFISH GATE RUNTIME. VANILLA GATE
COUNTS ARE SHOWN TO SHOW ORIGINAL PROBLEM SIZE.

Workload	Gates		Runtime (ms)	
	Vanilla	Jellyfish	CPU	zkPHIRE
ZCash	2^{17}	2^{15}	701	0.750 (934×)
Zexe Recursive Ckt	2^{22}	2^{17}	1951	1.440 (1354×)
Rollup of 10 Pvt Tx	2^{23}	2^{18}	3339	2.269 (1471×)
Rollup of 25 Pvt Tx	2^{24}	2^{19}	6161	3.874 (1590×)
2^{12} Rescue Hashes	2^{21}	2^{20}	11532	7.114 (1621×)
Rollup of 50 Pvt Tx	2^{25}	2^{20}	11533	7.114 (1621×)
Rollup of 100 Pvt Tx	2^{26}	2^{21}	24071	13.614 (1474×)
Rollup of 1600 Pvt Tx	2^{30}	2^{25}	355406	207.673 (1711×)
zkEVM	—	2^{27}	25 min	828.948 (1809×)

nominal gates within reach for proof generation. Lastly, in Table VI, we compare the same design (with ZeroCheck masking and fixed primes) with zkSpeed+ at iso-application, achieving geomean 11.87× speedup.

VII. RELATED WORK

There is a large body of work accelerating next-generation cryptography, but they have largely focused on Multi-Party Computation and Homomorphic Encryption [14], [18], [23], [26]–[28], [39], [42], [47], [48], [52]. Research on ZKP acceleration is relatively newer, and has focused primarily on MSMs and NTTs [10], [21], [24], [30], [31], [31], [32], [34], [36], [46], [46], [57], [58], [62], [65]–[68]. The community has broadened its focus to include other ZKP kernels such as the SumCheck protocol, Merkle tree-based commitments, and hash functions [4], [51], [58]. Prior work has explored accelerating these kernels on GPUs [29], [33] and ASICs [12], [49]. We compare zkPHIRE with three ASICs that accelerate proof-generation end-to-end: SZKP, NoCap, and zkSpeed. **SZKP** is a Groth16 [20] accelerator speeding all MSMs, including Sparse G2 MSMs, achieving geomean speedups of

Table VI
ISO-APPLICATION COMPARISON OF ZKSPEED+ (VANILLA) AND ZKPHIRE
(JELLYFISH).

Workload	Gates		Runtime (ms)	
	Vanilla	Jellyfish	zkSpeed+	zkPHIRE
ZCash	2^{17}	2^{15}	1.825	0.750 (2.43×)
2^{12} Rescue Hashes	2^{21}	2^{20}	19.631	7.114 (2.75×)
Zexe Recursive Circuit	2^{22}	2^{17}	38.535	1.440 (26.76×)
Rollup of 10 Pvt Tx	2^{23}	2^{18}	76.356	2.269 (33.65×)
Rollup of 25 Pvt Tx	2^{24}	2^{19}	151.973	3.874 (39.23×)

Table VII
COMPARISON OF ZKPHIRE WITH PRIOR ZKP ACCELERATORS. N = NTT,
S = SUMCHECK, M = MSM. AREAS ARE IN 7NM.

Metric	NoCap	SZKP+	zkSpeed+	zkPHIRE
Workload	Scaled-up AES	Rollup 25 Pvt Tx	Rollup 25 Pvt Tx	Rollup 25 Pvt Tx
Protocol	Spartan+Orion	Groth16	HyperPlonk	HyperPlonk
Main Kernels	N & S	N & M	S & M	S & M
Gates	2^{24}	2^{24}	2^{24}	2^{19}
Encoding	R1CS	R1CS	Plonk (Vanilla)	Plonk (Jellyfish)
Proof Size	8.1 MB	0.18 KB	5.09 KB	4.41 KB
Setup	none	circuit-specific	universal	universal
Prime	fixed	arbitrary	arbitrary	fixed
Bitwidth	64	255/381	255/381	255/381
SW Prover (s)	94.2	51.18	145.5	6.161
HW Prover (ms)	151.3	28.43	151.973	3.874
SW Verifier (ms)	134	4.2	26	19
Chip Area (mm²)	38.73	353.2	366.46	294.32
# Modmuls	2432	1720	1206	2267
Modmul (mm²)	0.002	0.133 / 0.314	0.133 / 0.314	0.073/0.162
Power (W)	62	>220	171	202

493× over a CPU. SZKP builds upon **PipeZK** [66], the first Groth16 accelerator. HyperPlonk and Groth16 target similar applications, since both provide small and cheaply verifiable proofs, but HyperPlonk offers universal setup and removes the expensive trusted setup ceremony [3]. **Protocol Acceleration:** Although a direct apples-to-apples comparison with Groth16 accelerators is not possible because the two protocols differ in proof sizes and trusted setup assumptions, we nevertheless report SKZP's characteristics (comparisons with other Groth16 accelerators will have the same qualitative trends). Similar to zkSpeed+, we compare with a scaled-up SZKP+ design that adopts zkSpeed+'s more efficient MSMs and upgrades to the *BLS12-381* curve. **NoCap** is a vector processor that accelerates Spartan+Orion proofs with a different application space from zkPHIRE. NoCap uses Merkle tree-based commitments and generates larger proofs with more expensive verifier runtime. NoCap is well-suited for scenarios where succinct proofs are less prioritized or the number of verifiers is limited. **zkSpeed** is the only other accelerator that targets HyperPlonk, to our knowledge. It accelerates proof generation end-to-end and achieves geometric mean speedups of 801× over CPU baseline. We do a detailed comparison with zkSpeed+ in Sec. VI-B5. Table VII summarizes the comparison across the four accelerators and reports iso-workload performance numbers. We use numbers for a scaled-up AES workload from NoCap's evaluation as it has similar number of gates and their area numbers scaled to 7nm, from zkSpeed. zkPHIRE's proving time is 39×, 7×, and 39× faster than NoCap, SZKP+, and zkSpeed+, respectively. Jellyfish's advantages are application-dependent: not all workloads map efficiently to Jellyfish gates. In such cases, different custom gates (which zkPHIRE can support) may yield faster proving times. Finally, while SZKP+ still yields the smallest proofs among all accelerators, it suffers from a major usability drawback due to the costly circuit-specific trusted setup needed for each application.

Nocap [49] implements SumChecks using their vector architecture and chip. While SumCheck is highly parallelizable, its core operation—reducing a vector—can require repeated

data access and movement on a vector processor. Reductions over a vector of length M require $\log_2(M)$ folding steps, typically implemented as shift-and-add instructions, that must be serialized and access the vector register file. Furthermore, higher degree polynomials introduce more extensions, each of which requires a dedicated vector reduction, causing the total reduction overhead grows with degree. In contrast, zkPHIRE uses fused pipelines with tree-like, direct connections to support efficient reductions with minimal data movement. Modern ZKP protocols rely on distinct kernels like SumCheck and MSM, which differ in compute and communication patterns. zkPHIRE's heterogeneous architecture provides specialized hardware for each, avoiding the inefficiencies of a general vector-based design balanced to handle both.

VIII. Conclusion

This paper presents zkPHIRE, the first accelerator to generate succinct proofs for problem sizes reaching 2^{30} nominal constraints. zkPHIRE supports general, programmable SumCheck, enabling ZKPs on custom, high-degree gates. zkPHIRE outperforms state-of-the-art accelerators for Hyper-Plonk, achieving iso-application geomean speedups of $11.87\times$ over ASIC and $1486\times$ over CPU.

References

[1] "libsnark: a c++ library for zksnark proofs." 2018. [Online]. Available: https://github.com/scipr-lab/libsnark

[2] "High bandwidth memory dram (hbm3)," JEDEC, Technical Report JESD238A, 2023, [Online]. Available: https://www.jedec.org/standards-documents/docs/jesd238a.

[3] "What is the zcash ceremony? the complete beginners guide," https://coinbureau.com/education/zcash-ceremony/, 2023.

[4] A. Ahmed, N. Sheybani, D. Moreno, N. B. Njungle, T. Gong, M. Kinsy, and F. Koushanfar, "Amaze: Accelerated mimc hardware architecture for zero-knowledge applications on the edge," 2024, accepted to ICCAD 2024. [Online]. Available: https://arxiv.org/abs/2411.06350

[5] P. AMD, "Server processor specifications," Mar 2024. [Online]. Available: https://www.amd.com/en/products/specifications/server-processor.html

[6] G. Arnon, A. Chiesa, G. Fenzi, and E. Yogev, "Whir: Reed–solomon proximity testing with super-fast verification," in *Annual International Conference on the Theory and Applications of Cryptographic Techniques*. Springer, 2025, pp. 214–243.

[7] A. J. Blumberg, J. Thaler, V. Vu, and M. Walfish, "Verifiable computation using multiple provers," Cryptology ePrint Archive, Paper 2014/846, 2014. [Online]. Available: https://eprint.iacr.org/2014/846

[8] S. Bowe, "Bls12-381: New zk-snark elliptic curve construction," Mar 2017. [Online]. Available: https://electriccoin.co/blog/new-snark-curve/

[9] B. Chen, B. Bünz, D. Boneh, and Z. Zhang, "Hyperplonk: Plonk with linear-time prover and high-degree custom gates," in *Annual International Conference on the Theory and Applications of Cryptographic Techniques*. Springer, 2023, pp. 499–530.

[10] X. Chen, B. Yang, W. Zhu, H. Wang, Q. Tao, S. Yin, M. Zhu, S. Wei, and L. Liu, "A high-performance ntt/msm accelerator for zero-knowledge proof using load-balanced fully-pipelined montgomery multiplier," *IACR Transactions on Cryptographic Hardware and Embedded Systems*, vol. 2025, no. 1, p. 275–313, Dec. 2024. [Online]. Available: https://tches.iacr.org/index.php/TCHES/article/view/11930

[11] T. S. M. Company, "Tsmc 22nm technology," 2021. [Online]. Available: https://www.tsmc.com/english/dedicatedFoundry/technology/logic/l_22nm

[12] A. Daftardar, J. Mo, J. Ah-kiow, B. Bünz, R. Karri, S. Garg, and B. Reagen, "Need for zkspeed: Accelerating hyperplonk for zero-knowledge proofs," in *Proceedings of the 52nd Annual International Symposium on Computer Architecture*, ser. ISCA '25. New York, NY, USA: Association for Computing Machinery, 2025, p. 1986–2001. [Online]. Available: https://doi.org/10.1145/3695053.3731021

[13] A. Daftardar, B. Reagen, and S. Garg, "Szkp: A scalable accelerator architecture for zero-knowledge proofs," in *Proceedings of the 2024 International Conference on Parallel Architectures and Compilation Techniques*, 2024, pp. 271–283.

[14] A. Ebel, K. Garimella, and B. Reagen, "Orion: A fully homomorphic encryption framework for deep learning," in *Proceedings of the 30th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 2*, ser. ASPLOS '25. New York, NY, USA: Association for Computing Machinery, 2025, p. 734–749. [Online]. Available: https://doi.org/10.1145/3676641.3716008

[15] A. Ebel and B. Reagen, "Osiris: A systolic approach to accelerating fully homomorphic encryption," 2024. [Online]. Available: https://arxiv.org/abs/2408.09593

[16] A. Fiat and A. Shamir, "How to prove yourself: Practical solutions to identification and signature problems," in *Advances in Cryptology — CRYPTO' 86*, A. M. Odlyzko, Ed. Berlin, Heidelberg: Springer Berlin Heidelberg, 1987, pp. 186–194.

[17] A. Gabizon, Z. J. Williamson, and O. Ciobotaru, "Plonk: Permutations over lagrange-bases for oecumenical noninteractive arguments of knowledge," *Cryptology ePrint Archive*, 2019.

[18] K. Garimella, Z. Ghodsi, N. K. Jha, S. Garg, and B. Reagen, "Characterizing and optimizing end-to-end systems for private inference," in *Proceedings of the 28th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 3*, ser. ASPLOS 2023. New York, NY, USA: Association for Computing Machinery, 2023, p. 89–104. [Online]. Available: https://doi.org/10.1145/3582016.3582065

[19] S. Goldwasser, Y. T. Kalai, and G. N. Rothblum, "Delegating computation: interactive proofs for muggles," *Journal of the ACM (JACM)*, vol. 62, no. 4, pp. 1–64, 2015.

[20] J. Groth, "On the size of pairing-based non-interactive arguments," in *Advances in Cryptology – EUROCRYPT 2016*, M. Fischlin and J.-S. Coron, Eds. Berlin, Heidelberg: Springer Berlin Heidelberg, 2016, pp. 305–326.

[21] F. Hirner, F. Krieger, and S. S. Roy, "Chiplet-based techniques for scalable and memory-aware multi-scalar multiplication," Cryptology ePrint Archive, Paper 2025/252, 2025. [Online]. Available: https://eprint.iacr.org/2025/252

[22] R. Inc., "White paper: Hbm2e and gddr6: Memory solutions for ai," White Paper, Rambus Inc., 2020.

[23] N. K. Jha and B. Reagen, "Deepreshape: Redesigning neural networks for efficient private inference," *Transactions on Machine Learning Research (TMLR)*, 2024.

[24] Z. Ji, Z. Zhang, J. Xu, and L. 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*, ser. ASPLOS '24. New York, NY, USA: Association for Computing Machinery, 2024, p. 57–70. [Online]. Available: https://doi.org/10.1145/3620666.3651364

[25] A. Kate, G. M. Zaverucha, and I. Goldberg, "Constant-size commitments to polynomials and their applications," in *Advances in Cryptology - ASIACRYPT 2010*, M. Abe, Ed. Berlin, Heidelberg: Springer Berlin Heidelberg, 2010, pp. 177–194.

[26] J. Kim, S. Kim, J. Choi, J. Park, D. Kim, and J. H. Ahn, "Sharp: A short-word hierarchical accelerator for robust and practical fully homomorphic encryption," in *Proceedings of the 50th Annual International Symposium on Computer Architecture*, 2023, pp. 1–15.

[27] J. Kim, G. Lee, S. Kim, G. Sohn, M. Rhu, J. Kim, and J. H. Ahn, "Ark: Fully homomorphic encryption accelerator with runtime data generation and inter-operation key reuse," in *2022 55th IEEE/ACM International Symposium on Microarchitecture (MICRO)*, 2022, pp. 1237–1254.

[28] S. Kim, J. Kim, M. J. Kim, W. Jung, J. Kim, M. Rhu, and J. H. Ahn, "Bts: An accelerator for bootstrappable fully homomorphic encryption," in *Proceedings of the 49th annual international symposium on computer architecture*, 2022, pp. 711–725.

[29] M. Li, Y. Yu, B. Wang, X. Fan, and S. Deng, "ZKPoG: Accelerating WitGen-incorporated end-to-end zero-knowledge proof on GPU," Cryptology ePrint Archive, Paper 2025/765, 2025. [Online]. Available: https://eprint.iacr.org/2025/765

[30] C. Liu, H. Zhou, P. Dai, L. Shang, and F. Yang, "Priormsm: An efficient acceleration architecture for multi-scalar multiplication," *ACM*

Trans. Des. Autom. Electron. Syst., vol. 29, no. 5, Aug. 2024. [Online]. Available: https://doi.org/10.1145/3678006

[31] C. Liu, H. Zhou, L. Yang, Z. Wu, P. Dai, Y. Li, S. Wu, and F. Yang, "Myosotis: An efficiently pipelined and parameterized multi-scalar multiplication architecture via data sharing," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, pp. 1–1, 2024.

[32] C. Liu, H. Zhou, L. Yang, J. Xu, P. Dai, and F. Yang, "Gypsophila: A scalable and bandwidth-optimized multi-scalar multiplication architecture," in *Proceedings of the 61st ACM/IEEE Design Automation Conference*, ser. DAC '24. New York, NY, USA: Association for Computing Machinery, 2024. [Online]. Available: https://doi.org/10.1145/3649329.3658259

[33] T. Lu, Y. Chen, Z. Wang, X. Wang, W. Chen, and J. 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*, ser. ASPLOS '25. New York, NY, USA: Association for Computing Machinery, 2025, p. 100–115. [Online]. Available: https://doi.org/10.1145/3669940.3707270

[34] T. Lu, C. Wei, R. Yu, C. Chen, W. Fang, L. Wang, Z. Wang, and W. Chen, "cuzk: Accelerating zero-knowledge proof with a faster parallel multi-scalar multiplication algorithm on gpus," *IACR Transactions on Cryptographic Hardware and Embedded Systems*, vol. 2023, no. 3, p. 194–220, Jun. 2023. [Online]. Available: https://tches.iacr.org/index.php/TCHES/article/view/10961

[35] C. Lund, L. Fortnow, H. Karloff, and N. Nisan, "Algebraic methods for interactive proof systems," in *Proceedings [1990] 31st Annual Symposium on Foundations of Computer Science*, 1990, pp. 2–10 vol.1.

[36] W. Ma, Q. Xiong, X. Shi, X. Ma, H. Jin, H. Kuang, M. Gao, Y. Zhang, H. Shen, and W. 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*, ser. ASPLOS 2023. New York, NY, USA: Association for Computing Machinery, 2023, p. 340–353. [Online]. Available: https://doi.org/10.1145/3575693.3575711

[37] S. Masson, A. Sanso, and Z. Zhang, "Bandersnatch: a fast elliptic curve built over the bls12-381 scalar field," *Designs, Codes and Cryptography*, vol. 92, no. 12, pp. 4131–4143, 2024.

[38] J. Mo, A. Daftardar, J. Ah-kiow, K. Guo, B. Bünz, S. Garg, and B. Reagen, "Mtu: The multifunction tree unit in zkspeed for accelerating hyperplonk," *arXiv preprint arXiv:2507.16793*, 2025.

[39] J. Mo, J. Gopinath, and B. Reagen, "Haac: A hardware-software co-design to accelerate garbled circuits," in *Proceedings of the 50th Annual International Symposium on Computer Architecture*, 2023, pp. 1–13.

[40] P. L. Montgomery, "Speeding the pollard and elliptic curve methods of factorization," *Mathematics of Computation*, vol. 48, no. 177, p. 243–264, 1987.

[41] T. P. Morgan, "Amd doubles down – and up – with rome epyc server chips," Aug 2019. [Online]. Available: https://www.nextplatform.com/2019/08/07/amd-doubles-down-and-up-with-rome-epyc-server-chips/

[42] N. Neda, A. Ebel, B. Reynwar, and B. Reagen, "Ciflow: Dataflow analysis and optimization of key switching for homomorphic encryption," in *2024 IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS)*, 2024, pp. 61–72.

[43] OpenCores, "Sha-3 ip core," https://opencores.org/projects/sha3, 2013, accessed: November 16, 2024.

[44] G. Passas, M. Katevenis, and D. Pnevmatikatos, "Crossbar nocs are scalable beyond 100 nodes," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 31, no. 4, pp. 573–585, 2012.

[45] N. Pippenger, "On the evaluation of powers and related problems," in *17th Annual Symposium on Foundations of Computer Science (sfcs 1976)*, 1976, pp. 258–263.

[46] P. Qiu, G. Wu, T. Chu, C. Wei, R. Luo, Y. Yan, W. Wang, and H. Zhang, "Msmac: Accelerating multi-scalar multiplication for zero-knowledge proof," in *Proceedings of the 61st ACM/IEEE Design Automation Conference*, ser. DAC '24. New York, NY, USA: Association for Computing Machinery, 2024. [Online]. Available: https://doi.org/10.1145/3649329.3655672

[47] N. Samardzic, A. Feldmann, A. Krastev, S. Devadas, R. Dreslinski, C. Peikert, and D. Sanchez, "F1: A fast and programmable accelerator for fully homomorphic encryption," in *MICRO-54: 54th Annual IEEE/ACM International Symposium on Microarchitecture*, 2021, pp. 238–252.

[48] N. Samardzic, A. Feldmann, A. Krastev, N. Manohar, N. Genise, S. Devadas, K. Eldefrawy, C. Peikert, and D. Sanchez, "Craterlake: a hardware accelerator for efficient unbounded computation on encrypted data," in *Proceedings of the 49th Annual International Symposium on Computer Architecture*, 2022, pp. 173–187.

[49] N. Samardzic, S. Langowski, S. Devadas, and D. Sanchez, "Accelerating zero-knowledge proofs through hardware-algorithm co-design," in *2024 57th IEEE/ACM International Symposium on Microarchitecture (MICRO)*, 2024, pp. 366–379.

[50] S. Setty, "Spartan: Efficient and general-purpose zksnarks without trusted setup," in *Advances in Cryptology – CRYPTO 2020*, D. Micciancio and T. Ristenpart, Eds. Cham: Springer International Publishing, 2020, pp. 704–737.

[51] N. Sheybani, T. Gong, A. Ahmed, N. B. Njungle, M. Kinsy, and F. Koushanfar, "Gotta hash 'em all! speeding up hash functions for zero-knowledge proof applications," 2025. [Online]. Available: https://arxiv.org/abs/2501.18780

[52] D. Soni, N. Neda, N. Zhang, B. Reynwar, H. Gamil, B. Heyman, M. Nabeel, A. A. Badawi, Y. Polyakov, K. Canida, M. Pedram, M. Maniatakos, D. B. Cousins, F. Franchetti, M. French, A. Schmidt, and B. Reagen, "Rpu: The ring processing unit," in *2023 IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS)*, 2023, pp. 272–282.

[53] A. Stapelberg, "Opening up 'zero-knowledge proof' technology to promote privacy in age assurance," Google Blog (The Keyword), Jul 2025, "Safety & Security / Google in Europe" blog post.

[54] p. tech, "Amd epyc 7502 specs," Sep 2024. [Online]. Available: https://www.techpowerup.com/cpu-specs/epyc-7502.c2250

[55] R. S. Wahby, M. Howald, S. Garg, A. Shelat, and M. Walfish, "Verifiable asics," in *2016 IEEE Symposium on Security and Privacy (SP)*, 2016, pp. 759–778.

[56] R. S. Wahby, I. Tzialla, A. Shelat, J. Thaler, and M. Walfish, "Doubly-efficient zksnarks without trusted setup," in *2018 IEEE Symposium on Security and Privacy (SP)*. IEEE, 2018, pp. 926–943.

[57] C. Wang and M. Gao, "Sam: A scalable accelerator for number theoretic transform using multi-dimensional decomposition," in *2023 IEEE/ACM International Conference on Computer Aided Design (ICCAD)*, 2023, pp. 1–9.

[58] C. Wang and M. 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*, ser. ASPLOS '25. New York, NY, USA: Association for Computing Machinery, 2025, p. 1101–1117. [Online]. Available: https://doi.org/10.1145/3669940.3707228

[59] S.-Y. Wu, C. Y. Lin, M. Chiang, J. Liaw, J. Cheng, S. Yang, M. Liang, T. Miyashita, C. Tsai, B. Hsu *et al.*, "A 16nm finfet cmos technology for mobile soc and computing applications," in *2013 IEEE International Electron Devices Meeting*. IEEE, 2013, pp. 9–1.

[60] S.-Y. Wu, C. Lin, M. Chiang, J. Liaw, J. Cheng, S. Yang, C. Tsai, P. Chen, T. Miyashita, C. Chang *et al.*, "A 7nm cmos platform technology featuring 4 th generation finfet transistors with a 0.027 um 2 high density 6-t sram cell for mobile soc applications," in *2016 IEEE International Electron Devices Meeting (IEDM)*. IEEE, 2016, pp. 2–6.

[61] T. Xie, Y. Zhang, and D. Song, "Orion: Zero knowledge proof with linear prover time," in *Advances in Cryptology – CRYPTO 2022: 42nd Annual International Cryptology Conference, CRYPTO 2022, Santa Barbara, CA, USA, August 15–18, 2022, Proceedings, Part IV*. Berlin, Heidelberg: Springer-Verlag, 2022, p. 299–328. [Online]. Available: https://doi.org/10.1007/978-3-031-15985-5_11

[62] Z. Yang, L. Zhao, P. Li, H. Liu, K. Li, B. Zhao, D. Meng, and R. Hou, " LegoZK: A Dynamically Reconfigurable Accelerator for Zero-Knowledge Proof ," in *2025 IEEE International Symposium on High Performance Computer Architecture (HPCA)*. Los Alamitos, CA, USA: IEEE Computer Society, Mar. 2025, pp. 113–126. [Online]. Available: https://doi.ieeecomputersociety.org/10.1109/HPCA61900.2025.00020

[63] Zcash, "Elliptic curve cryptography - the halo2 book," 2025. [Online]. Available: https://zcash.github.io/halo2/design/gadgets/ecc.html

[64] H. Zeilberger, B. Chen, and B. Fisch, "Basefold: efficient field-agnostic polynomial commitment schemes from foldable codes," in *Annual International Cryptology Conference*. Springer, 2024, pp. 138–169.

[65] N. Zhang and F. Franchetti, "Code generation for cryptographic kernels using multi-word modular arithmetic on gpu," in *Proceedings of the 23rd ACM/IEEE International Symposium on Code Generation and Optimization*, ser. CGO '25. New York, NY, USA: Association for Computing Machinery, 2025, p. 476–492. [Online]. Available: https://doi.org/10.1145/3696443.3708948

[66] Y. Zhang, S. Wang, X. Zhang, J. Dong, X. Mao, F. Long, C. Wang, D. Zhou, M. Gao, , and G. Sun, "Pipezk: Accelerating zero-knowledge proof with a pipelined architecture," in *2021 ACM/IEEE 48th Annual International Symposium on Computer Architecture (ISCA)*, 2021.

[67] H. Zhou, C. Liu, L. Yang, L. Shang, and F. Yang, "Rezk: A highly reconfigurable accelerator for zero-knowledge proof," *IEEE Transactions on Circuits and Systems I: Regular Papers*, 2024.

[68] X. Zhu, H. He, Z. Yang, Y. Deng, L. Zhao, and R. Hou, "Elastic MSM: A fast, elastic and modular preprocessing technique for multi-scalar multiplication algorithm on GPUs," Cryptology ePrint Archive, Paper 2024/057, 2024. [Online]. Available: https://eprint.iacr.org/2024/057