Need for zkSpeed: Accelerating HyperPlonk for Zero-Knowledge Proofs

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

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

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

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

Ramesh Karri
rkarri@nyu.edu
NYU Tandon
Brooklyn, NY, USA

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

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

Abstract

(*Preprint*) Zero-Knowledge Proofs (ZKPs) are rapidly gaining importance in privacy-preserving and verifiable computing. ZKPs enable a *proving* party to prove the truth of a statement to a *verifying* party *without* revealing anything else. ZKPs have applications in blockchain technologies, verifiable machine learning, and electronic voting, but have yet to see widespread adoption due to the computational complexity of the proving process. Recent works have accelerated the key primitives of state-of-the-art ZKP protocols on GPU and ASIC. However, the protocols accelerated thus far face one of two challenges: they either require a trusted setup for *each* application, or they generate larger proof sizes with higher verification costs, limiting their applicability in scenarios with numerous verifiers or strict verification time constraints. This work presents an accelerator, *zkSpeed*, for HyperPlonk, a state-of-the-art ZKP protocol that supports both one-time, *universal* setup and small proof sizes for typical ZKP applications in publicly verifiable, consensus-based systems. We accelerate the entire protocol, including two major primitives: SumCheck and Multi-scalar Multiplications (MSMs). We develop a full-chip architecture using 366.46 mm^2 and 2 TB/s of bandwidth to accelerate the entire proof generation process, achieving geometric mean speedups of 801× over CPU baselines.

1 Introduction

Zero knowledge proofs enable a prover to produce a certificate or proof that some computation (possibly with secret inputs) was performed correctly. The proof itself reveals no information about the secret inputs and can be verified much faster than the original computation. Proof systems can be characterized by several important criteria: The prover time, the verifier time, the proof size, and the cryptographic assumptions required for the proof to be secure. Each application of proof systems results in a different set of requirements on these criteria. For instance, private transactions [6] require a proof per transaction to be posted on a blockchain and distributed to all blockchain nodes, thus prioritizing proof size.

Recent years have seen an explosion in proof systems [10, 18, 21, 61]. Each of them achieves a different Pareto optimal point with respect to the criteria. In all applications, prover time is important, and fast provers enable new applications. Recently, two proof systems have been accelerated in hardware: Groth16 [21] in pipeZK [64] and SZKP [12], and Orion with NoCap [49]. Groth16 produces very short proofs (188 bytes) and enables millisecond verification, independent of computation size. However, it relies on a strong cryptographic assumption; a *circuit-specific trusted setup* where a trusted party generates keys using secret randomness. If this randomness is not properly discarded or the party is malicious, the system's security is compromised, allowing false statements to be proven. Hence, Zcash and other applications have moved away from circuit-specific trusted-setup protocols [42]. On the other hand, Orion does not require a trusted setup, has a fast prover, and does not rely on elliptic curve cryptography, but has very large (8MB) proofs. This is 4× larger than the maximum block size of about 2MB in Ethereum cryptocurrency. Thus, a private transaction with an Orion proof would not fit into a single block.

In this work, we accelerate HyperPlonk [10], a recent ZKP system gaining attention for its intriguing set of tradeoffs, making it suitable for many applications. Its proofs are about 5 KB and it uses a so-called universal trusted setup [22]. This is run only *once* and reused for any computation. It is easy to run the setup among many parties. The security is guaranteed as long as at least one party is honest. Unlike Groth16 and others, it does not use the Number Theoretic Transform (NTT), instead using SumCheck. NTTs asymptotically run in time $O(n \log(n))$ (n represents the size of underlying computation), while SumCheck runs in linear time.

Accelerating HyperPlonk presents its own challenges. First, as noted, the protocol replaces NTT, well studied in the hardware literature, with the SumCheck protocol, which is relatively less understood from a hardware standpoint. Although implemented in NoCap [49], as we will shortly discuss, HyperPlonk's SumChecks are significantly more complex, incurring greater bandwidth costs and design complexity. Further, HyperPlonk introduces additional runtime computations like modular inverses that are not needed in

Groth16 and Orion, and thus not studied from a hardware standpoint in this prior body of work. The additional design complexity requires careful architectural design, new hardware optimizations, and thorough design space exploration to identify architectures with area-performance tradeoffs that justify hardware acceleration.

In this paper we present *zkSpeed*, a modular HyperPlonk accelerator that overcomes the above challenges. zkSpeed comprises eight unique accelerator units that are composed into a complete architecture, including shared and local scratchpads to capture on-chip data reuse. Units and SRAMs communicate via a multi-channel shared bus, and we develop a mapping of complete Hyperplonk protocol onto zkSpeed. This paper makes the following contributions:

- A high-throughput, fully-pipelined accelerator to handle three flavors of SumCheck for HyperPlonk-based proofs
- A novel implementation for modular inversion to compute fraction polynomials not seen in other ZKP protocols
- Additional optimizations across all hardware units that save upwards of 50% area per unit and up to 85% in bandwidth.
- A comprehensive design space exploration of all hardware units to investigate design tradeoffs as we scale to high-performance designs and advanced memory technologies.
- A full-chip design that achieves 801× gmean speedup over CPU at iso-compute area

2 Background

2.1 zkSNARKs

State-of-the-art ZKP protocols are zero-knowledge Succinct Non-interactive Argument of Knowledge (zkSNARKs). zkSNARKs have three properties: (i) zero-knowledge, i.e., the proof does not reveal any information about the secret witness w; (ii) succinct, i.e., the proof has a few hundreds of bytes; and (iii) non-interactive, i.e., $\mathcal{P}$ sends the proof to $\mathcal{V}$ in one exchange.

zkSNARKs like HyperPlonk use polynomials to encode the correct execution of the target program. Polynomials can encode complex computations and constraints in a compact form, reducing the computational burden on the $\mathcal{V}$ to a few checks, instead of requiring inspection of individual operations in the program. In HyperPlonk, the two most time consuming kernels are SumCheck and Multiscalar Multiplications (MSM). The SumCheck kernel lets the $\mathcal{P}$ demonstrate knowledge of a polynomial by verifying that its properties hold over a large set of inputs without revealing the polynomial. MSMs ensure that $\mathcal{P}$ is bound to that polynomial, preventing manipulation of the polynomial in the proof.

2.2 The SumCheck Kernel

SumCheck is an interactive protocol between a prover $\mathcal{P}$ and a verifier $\mathcal{V}$. $\mathcal{P}$ demonstrates to $\mathcal{V}$, that it has correctly computed the sum of a polynomial over the *boolean hypercube*, *i.e.*, over all Boolean (0/1) assignments of its variables [57].

Given a multivariate polynomial $P(x_1, x_2, \ldots, x_\mu)$, where $x_i \in \mathbb{F}_q$[1], $\mathcal{P}$ wants to prove to $\mathcal{V}$ that it correctly computed the sum

$$H = \sum_{x_1 \in \{0,1\}} \sum_{x_2 \in \{0,1\}} \cdots \sum_{x_\mu \in \{0,1\}} P(x_1, x_2, \ldots, x_\mu).$$

[1] each variable can be an integer modulo a prime number q.

$\mathcal{P}$ and $\mathcal{V}$ engage in a multi-round protocol. In Round 1, $\mathcal{P}$ computes $g_1(x_1)$, a *univariate* polynomial of x_1 by summing over all binary values of the remaining variables:

$$g_1(x_1) = \sum_{x_2 \in \{0,1\}} \sum_{x_3 \in \{0,1\}} \cdots \sum_{x_\mu \in \{0,1\}} P(x_1, x_2, \ldots, x_\mu)$$

and returns it to $\mathcal{V}$ in the form of its coefficients. $\mathcal{V}$ checks that $g_1(0) + g_1(1) = H$, and if so, generates a random challenge $r_1 \in \mathbb{F}_q$ and sends it to $\mathcal{P}$. $\mathcal{V}$ asks $\mathcal{P}$ to prove that

$$g_1(r_1) = \sum_{x_2 \in \{0,1\}} \sum_{x_3 \in \{0,1\}} \cdots \sum_{x_\mu \in \{0,1\}} P(r_1, x_2, \ldots, x_\mu),$$

which is an instance of *SumCheck*, except over a $(\mu - 1)$-variate polynomial. Thus, in Round 2, $\mathcal{P}$ computes and returns

$$g_2(x_2) = \sum_{x_3 \in \{0,1\}} \sum_{x_4 \in \{0,1\}} \cdots \sum_{x_\mu \in \{0,1\}} P(r_1, x_2, \ldots, x_\mu)$$

to $\mathcal{V}$ and the protocol repeats recursively for a total of μ rounds. If all checks pass, $\mathcal{V}$ accepts $\mathcal{P}$'s claim about H.

2.3 Multilinear Polynomials

ZKPs like HyperPlonk use *multilinear polynomials*, that are *linear* in each of their variables. For example,

$$f(x_1, x_2, x_3) = x_1x_2 + 2x_1x_3 + 3x_1x_2x_3$$

is multilinear because the maximum degree of individual variables is one. ZKP protocols use a general representations of multilinear polynomials, shown below for a 2-variable multilinear polynomial:

$$f(x_1, x_2) = (1-x_2)(1-x_1)a_0 + x_2(1-x_1)a_1 + (1-x_2)x_1a_2 + x_2x_1a_3.$$

These representations map cleanly to hardware, since we can store polynomials as lookup tables indexed by the binary values of (x_1, x_2). In general, a multilinear polynomial with μ variables $x_1 \ldots x_\mu$ can be stored in a lookup table of 2^μ entries. As we will discuss in Section 3, HyperPlonk uses multilinear polynomials as building blocks for higher-degree polynomials on which we then perform SumCheck and other computations. In the rest of the paper, we will use the term *MLE table* to refer to these lookup data structures (MLE stands for "multilinear extensions" [57]; the word “extension" reflects the fact that these polynomials can also be evaluated at non-binary, or extended, values).

2.4 The MSM Kernel

MSMs are dot products between a vector of scalars $\vec{s}$ and a vector of 2D or 3D points $\vec{P}$ on an elliptical curve, e.g. $\sum_{i=0}^{n-1} s_iP_i$. MSMs are used in ZKPs to perform *commitments*. A commitment is a cryptographic primitive that binds a prover to a value without revealing it. In zkSNARK protocols like HyperPlonk [10], the scalars are the entries of the MLE tables. Computing a dot-product reduces these polynomials to a single value, i.e., the commitment.

MSMs are a bottleneck in ZKP provers and recent work has focused on accelerating them on ASIC and GPU [12, 25, 31, 35, 36, 64, 66]. The bottleneck is due to the extremely expensive elliptic curve point multiplications that they use. To reduce this cost, MSMs use Pippenger's algorithm [44], which performs point multiplications via several point additions (PADDs). PADDs are still expensive, typically tens of regular multiplications. In the context of ASIC accelerators, the state-of-the-art (SZKP [12]) presents a framework for building scalable MSM architectures.

3 The HyperPlonk Protocol

3.1 Plonk-based encodings

This is the *first* work to accelerate HyperPlonk, a zkSNARK protocol that replaces Number Theoretic Transform (Fast Fourier Transforms over finite fields) with the SumCheck protocol. In ZKPs, the program being proven must be converted into a specific form before generating the proof. While most prior works use *Rank-1 Constraint System (R1CS)*—a series of sparse matrix-vector encodings—HyperPlonk adopts a Plonk-based structure [18], which we describe shortly. R1CS and Plonk encodings map all program values into addition, multiplication, and boolean operations, with nonlinear operations (branches) resolved via bit-wise decompositions.

Plonk-based encodings mapping each operation in a program's execution to an arithmetic logic unit (or "gate") that supports only addition, multiplication, and equality checks, as shown in Eq. 1:

$$f = q_L w_1 + q_R w_2 + q_M w_1 w_2 - q_O w_3 + q_c \tag{1}$$

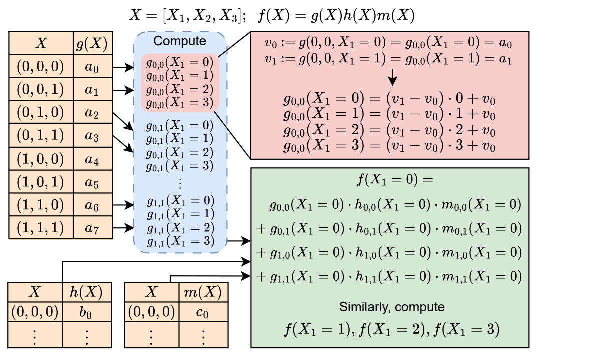
Terms q_L, q_R, q_M, q_O are binary "control" signals for left, right, multiply, and output ports. Terms w_1, w_2 represent the gate's inputs, and w_3 is its output. q_c represents a constant input. An addition operation, for instance, is implemented by setting $q_L = 1$, $q_R = 1$, $q_O = 1$ and other control inputs to 0. This yields: $f = w_1 + w_2 - w_3$. That is, $f = 0$ if and only if the addition is correctly performed.

Each operation in a program is mapped to a Plonk gate. A program with 2^μ operations has 2^μ Plonk gates interconnected to form a "circuit." A μ-dimensional polynomial $f(X_1, X_2 \dots, X_\mu)$ represents the entire circuit, with binary assignments to variables $\{X_1, X_2 \dots, X_\mu\}$ representing individual gates. For example, if $\mu = 4$, the circuit has 16 gates, and $f(X_1 = 0, X_2 = 0, X_3 = 0, X_4 = 0)$ represents the first gate.

$f(X)$ is constructed using *multilinear* polynomials representing every term from Eq. 1. For example, $q_L(X)$ is the left control input polynomial, $w_1(X)$ is the first data input polynomial, and so on. Each of these polynomials also takes $X = [X_1, X_2, \dots, X_4]$ as an "index". For example, in a program that maps to 16 gates, $q_M([0, 1, 0, 0])$ will tell us whether or not the "multiply" control signal is enabled in gate 2. $w_3([1, 1, 1, 1])$ tells us what the output data signal is at gate 15. All individual polynomials are encoded in this fashion, and can be stored as MLE tables. These MLE tables are populated from the program trace in software before running the HyperPlonk prover. The MLEs are then used by the steps of the HyperPlonk protocol, which we discuss in the following sections. A consequence is that the SumCheck algorithm (Section 2.2) has to be modified to account for Plonk polynomials.

3.2 SumCheck on Plonk-based Encodings

The key difference between the SumCheck used in HyperPlonk and the example shown in Section 2.2 is in the polynomial structure. While that example shows how SumCheck runs on a single polynomial of arbitrary degree, HyperPlonk runs SumChecks on the Plonk polynomial, which involves the *products* of multilinear polynomials. The polynomial in Eq. 1, for example, consists of 5 terms, each of a different degree. Figure 1 shows an example of how a SumCheck of a three-polynomial product is computed.



$X = [X_1, X_2, X_3]$; $f(X) = g(X)h(X)m(X)$
X | $g(X)$
$(0,0,0)$ a_0
$(0,0,1)$ a_1
$(0,1,0)$ a_2
$(0,1,1)$ a_3
$(1,0,0)$ a_4
$(1,0,1)$ a_5
$(1,1,0)$ a_6
$(1,1,1)$ a_7
Compute
$g_{0,0}(X_1 = 0)$ $g_{0,0}(X_1 = 1)$ $g_{0,0}(X_1 = 2)$ $g_{0,0}(X_1 = 3)$
$g_{0,1}(X_1 = 0)$ $g_{0,1}(X_1 = 1)$ $g_{0,1}(X_1 = 2)$ $g_{0,1}(X_1 = 3)$
$g_{1,1}(X_1 = 0)$ $g_{1,1}(X_1 = 1)$ $g_{1,1}(X_1 = 2)$ $g_{1,1}(X_1 = 3)$
$v_0 := g(0,0,X_1 = 0) = g_{0,0}(X_1 = 0) = a_0$
$v_1 := g(0,0,X_1 = 1) = g_{0,0}(X_1 = 1) = a_1$
$g_{0,0}(X_1 = 0) = (v_1 - v_0) \cdot 0 + v_0$
$g_{0,0}(X_1 = 1) = (v_1 - v_0) \cdot 1 + v_0$
$g_{0,0}(X_1 = 2) = (v_1 - v_0) \cdot 2 + v_0$
$g_{0,0}(X_1 = 3) = (v_1 - v_0) \cdot 3 + v_0$
$f(X_1 = 0) =$
$g_{0,0}(X_1 = 0) \cdot h_{0,0}(X_1 = 0) \cdot m_{0,0}(X_1 = 0)$
$+ g_{0,1}(X_1 = 0) \cdot h_{0,1}(X_1 = 0) \cdot m_{0,1}(X_1 = 0)$
$+ g_{1,0}(X_1 = 0) \cdot h_{1,0}(X_1 = 0) \cdot m_{1,0}(X_1 = 0)$
$+ g_{1,1}(X_1 = 0) \cdot h_{1,1}(X_1 = 0) \cdot m_{1,1}(X_1 = 0)$
Similarly, compute
$f(X_1 = 1), f(X_1 = 2), f(X_1 = 3)$
X | $h(X)$
$(0,0,0)$ b_0
X | $m(X)$
$(0,0,0)$ c_0

Figure 1: SumCheck Example. The subscripts $(0, 0$ etc.) refer to the specific instances of X_2, X_3.

In the example from Section 2.2, we computed the evaluations at 0 and 1, because the result of summing over all variables yielded a univariate polynomial of degree 1. In this example, however, when we sum over all variables, we have a univariate *degree 3* polynomial. From elementary algebra, such a polynomial must be evaluated at *4* unique points to fully characterize the polynomial. In the figure example, in the blue region, we are iterating over $[X_2, X_3] = [0, 0], [0, 1], [1, 0], [1, 1]$ to compute the evaluations. For each iteration (red region), we must evaluate each polynomial at $X_1 = 0, 1, 2, 3$. Then, in the green region, we compute the product for $X_1 = 1$ across all polynomials, and then sum across iterations. This is repeated for $X_1 = 2, 3, 4$. After the summations, each polynomial's MLE table is updated with a random challenge r from the verifier. For example, a table t' (for round 2) can be constructed from an table t (from round 1) using the formula $t'[i] \leftarrow (t[2i+1] - t[2i])r + t[2i]$.

These core computational steps are more expensive than the baseline SumCheck. An additional layer of complexity is that in the polynomial shown in Eq. 1, there are 5 terms of varying degrees. The varying degrees leads to imbalance in how many evaluations are needed (i.e. $q_L w_1$ needs 3 evaluations, while $q_M w_1 w_2$ needs 4). This is handled by the HyperPlonk SumCheck protocol with a fixed interpolation step before the MLE tables are reduced. While these computations are expensive, many of these operations can be performed in parallel. All polynomials can compute their evaluations independently, and within a polynomial, evaluations at different binary values can also be parallelized. Thus, HyperPlonk's SumChecks have high compute and degrees of parallelism.

3.3 HyperPlonk Protocol Steps

In this section, we outline the steps of the HyperPlonk protocol, as seen in Figure 8, what each step achieves, and what hardware units are needed to perform each step.

3.3.1 Commit Witnesses. The first step in HyperPlonk's prover involves witness commitments. In this step, we compute MSMs between elliptic curve points and the witness polynomials w_1, w_2, w_3 to reduce each polynomial to individual commitments. The witness polynomials in HyperPlonk are typically "Sparse", meaning that 90% of the values are either 0 or 1, and 10% are up to the full scalar width. SZKP [12] handles this by using *Sparse MSMs.* We implement Sparse MSMs using the MSM unit for this step.

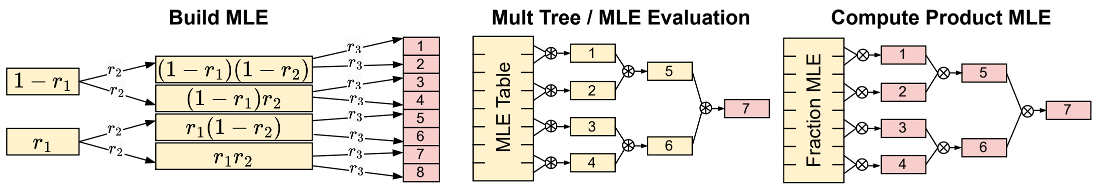


Figure 2: Different computation patterns that fit in the multifunction tree unit. Computation outputs are marked in red.

3.3.2 Gate Identity. The Gate Identity step confirms that each gate in the circuit performs its computation correctly. The intuition is that if so, then $f(X)$ should evaluate to 0 for each gate, equivalently, for each binary assignment of X. Further, the *sum* of $f(X)$ should also equal 0, which can be confirmed by running SumCheck on $f(X)$. However, since the prior statement is only a necessary but not sufficient condition for correct computation, HyperPlonk performs SumCheck over a polynomial $f(X)r(X)$, where $r(X)$ is a multilinear polynomial with coefficients computed from μ random challenges as shown in Figure 2 (left). This is known as a ZeroCheck.

The Gate Identity step relies on three units: the "multifunction tree unit" (Section 4.3) to build $r(X)$, a SumCheck unit, and the MLE Update unit to update MLE tables between SumCheck rounds.

3.3.3 Wiring Identity. This step verifies that the outputs of each gate are routed correctly to inputs of downstream gates. This is achieved via a PermutationCheck, which is performed by constructing a numerator polynomial N that encodes the "natural order" of the outputs and a denominator polynomial D that encodes the permuted ordering, and then checking that $\phi = N/D$, called the fraction polynomial, equals 1. As we will discuss, computing D^{-1} is time consuming, since inverting (or, in effect, performing divisions on) elements of a finite field is difficult. From ϕ, a product polynomial π is constructed by computing the product of all elements of ϕ. The ϕ and π polynomials are both *committed* to via MSM computations. A final ZeroCheck is also performed to ensure that the structure imposed upon the ϕ and π polynomials was not violated during their creation, i.e., the sum of these polynomial should be zero everywhere. All polynomials are stored in the form of MLE tables. The Wiring Identity step uses the construct N&D unit, the FracMLE unit, the Multifunction Tree unit (to construct the π MLE and later within the ZeroCheck step), the MSM unit, the SumCheck unit, and the MLE Update unit.

3.3.4 Batch Evaluations. The SumCheck protocols require evaluating the MLE at a specific point. Batch evaluation reduces the evaluation of multiple polynomials at multiple points into a single polynomial evaluated at a single point. The batch evaluation step consists of querying the input MLEs, as well as ϕ and π, at a set of 6 points, some of which are derived from round challenges in the SumCheck portion of the Gate and Wiring Identity steps, and some of which are fixed at compile time. In total, 22 total evaluations are performed among 13 polynomials using 6 distinct points. This step is almost the exact reverse of the step in 3.3.2 used to construct $r(x)$, since we are compressing an entire MLE into 1 value, while $r(x)$ builds an entire MLE from μ values. These polynomial queries are then used by the verifier to check that the prover obeyed the protocol. The Batch Evaluation step uses the Multifunction Tree unit to compute each MLE query.

3.3.5 Polynomial Opening. This step is used to succinctly verify the correctness of the prover's batch evaluations. For brevity, we will omit details of thepolynomial opening step. To summarize, it involves first computing 6 MLEs as a linear combination of the MLEs from Eq. 1, ϕ and π. Then, 6 more MLEs are constructed from the query points in Section 3.3.4. These 12 MLEs are combined into a MLE "dot-product", upon which a final SumCheck is computed. To avoid confusion, we will refer to this final SumCheck as "OpenCheck" and "SumCheck" to refer to the underlying computation.

After OpenCheck, the first 6 MLEs used as OpenCheck's inputs are linearly combined with the OpenCheck's round challenges to construct a final MLE, which is denoted as g'. This MLE is first reduced to half its size, and it is used as the scalar set for a $2^{\mu-1}$-point MSM. We then halve the scalar set and perform MSMs. For example, if $\mu = 10$, we compute a 2^9-point MSM, then a 2^8-point MSM, all the way to a 2^0-point MSM. The Polynomial Opening uses the MLE combine unit to compute the linear MLE combinations, the Multifunction Tree unit to build the 6 MLEs, the SumCheck unit to perform OpenCheck, and the MSM Unit to compute MSMs.

3.3.6 SHA3. zkSNARKs are non-interactive and use SHA3 to generate challenges as well as transcripts. (Transcripts are logs of proof-related computations checked by the verifier.) Because SHA3 is used in-between HyperPlonk steps to log computed values, SHA3 acts as an order-enforcing mechanism. This means most of the protocol steps must be executed in series, as shown in Figure 8.

3.3.7 Compute Demands of HyperPlonk. Table 1 summarizes profiling results to characterize key functions and understand the sources of performance overhead and hardware needs when accelerating HyperPlonk. There are too many functions to list, and we present the twelve with the highest computational density, which is defined as modular multiplications (modmuls) per Byte, as done in prior work [13, 27] (note SHA3 has no modmuls). Additionally, the reference CPU implementation is provided as a link. First, we observe that all functions require an immense amount of computation: ranging from millions to billions of 255/351b modmuls (comprising three integer multiplications) over all invocations of each function. For example, the data for Wire Identity MSMs modmuls reflects two function calls; for ZeroCheck Rounds there is one function call (see links in table). This motivates the need for both specialized modmul units and a high degree of parallelism to mitigate overhead. Second, compute intensity drops off sharply after the third function (since data reuse is limited), and the data input/output sizes for all functions are large, typically hundreds of megabytes up to terabytes as problem sizes scale to more gates. This motivates the need for large on-chip scratchpads to mitigate off-chip data movement when possible and high off-chip (i.e., HBM) bandwidth.

[2]SumCheck code (used by Zero, Perm, and OpenCheck) is at L154-182 and L123-181

Table 1: Modular multiplications, memory requirements, and arithmetic intensity of select functions for 2^{20} gates. Links to the source code are provided[2].

Kernel	Source Code	Modmuls (millions)	Input Size (MB)	Output Size (MB)	Arithmetic Intensity (modmul/byte)
Poly Open MSMs	↗	1160	127	0.00	8.70
Wire Identity MSMs	↗	2290	254	0.00	8.59
Witness MSMs	↗	1370	167	0.00	7.83
Batch Evaluations	↗	23.1	77.5	0.00	0.28
ZeroCheck Rounds	↗	77.6	332	0.00	0.22
Fraction MLE (FracMLE)	↗	5.19	0.00	31.9	0.16
PermCheck Rounds	↗	94.4	701	0.00	0.13
Linear Combine	↗	18.9	77.5	191	0.07
OpenCheck Rounds	↗	31.5	765	0.00	0.04
Construct N & D	↗	10.5	18.2	255	0.04
Product MLE (ProdMLE)	↗	1.05	0.00	31.9	0.03
All MLE Updates	↗	33.6	1800	900	0.01

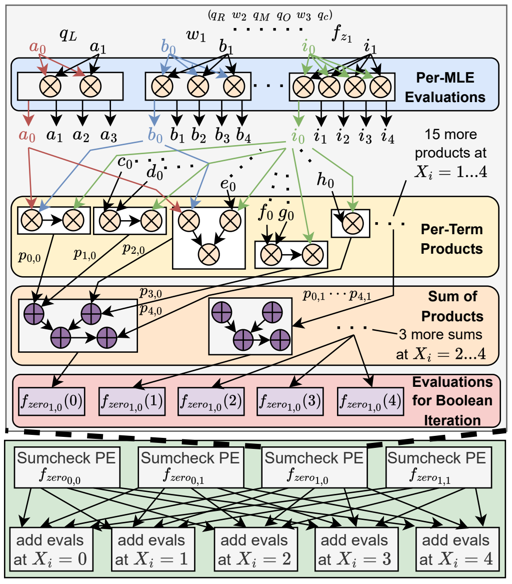


Figure 3: SumCheck Round unit example. Subscripts of f indicate which X_2, X_3 instance is being handled by the PE. For simplicity, each MLE is renamed from $a - i$. Their subscripts indicate the evaluation, e.g., $X_1 = 0 \ldots 4$. The products $p_{j,k}$ are labeled to indicate the i^{th} term evaluated at $X_1 = j$.

4 HyperPlonk Accelerator Units

HyperPlonk's protocol is based on the BLS12-381 elliptical curve. Here, all MLE datatypes are 255 bits wide, and all elliptical curve points in the MSMs are 381 bits wide. All MLE and MSM operations involve modular arithmetic primitives. These are built into each accelerator unit that requires them. In total, zkSpeed comprises eight accelerator units, we describe each below.

4.1 SumCheck and MLE Update

Three HyperPlonk steps use SumCheck: ZeroCheck (Section 3.3.2), PermutationCheck (Section 3.3.3), and OpenCheck (Section 3.3.5). Each step has a unique polynomial shown, respectively, below:

$$f_{zero} = q_L w_1 f_{z_1} + q_R w_2 f_{z_1} + q_M w_1 w_2 f_{z_1} - q_O w_3 f_{z_1} + q_c f_{z_1} \quad (2)$$

$$f_{perm} = \pi f_{z_2} - p_1 p_2 f_{z_2} + \alpha(\phi D_1 D_2 D_3) f_{z_2} - \alpha(N_1 N_2 N_3) f_{z_2} \quad (3)$$

$$f_{open} = y_1 k_1 + y_2 k_2 + y_3 k_3 + y_4 k_4 + y_5 k_5 + y_6 k_6 \quad (4)$$

In these equations, α is a challenge from the SHA3 unit, and all other symbols represent multilinear polynomials. These polynomials share a common sum-of-products representation, but are each unique and require slightly different datapaths. We develop a unified SumCheck Unit that can handle each of these polynomials. We highlight the key contributions of our architecture next.

4.1.1 Sumcheck Round PE Microarchitecture. In ZeroCheck and PermutationCheck, there are polynomials that appear multiple times across terms. In HyperPlonk's CPU baseline, the boolean hypercube summations are performed iteratively term-by-term, incurring redundant computation for these repeating polynomials. We address this by computing all evaluations for each polynomial in parallel. For example, in Equation 2, the polynomial f_{z_1} has to be evaluated at $X_i = 0, 1, 2, 3$ and 4. This computation only needs to be performed once before being used to compute the product in each term of f_{zero}. Figure 3 shows an example of how our Sumcheck PEs handle ZeroCheck computation for f_{zero}. Each unique MLE takes its respective $X_i = 0$ and 1 values (e.g., $c_0 = q_R(X_i = 0)$), and uses these values to construct the needed X_i evaluations. Then, each product in Equation 2 must be computed for each X_i evaluation. The products at the evaluation points are then accumulated into registers. Due to the inherent degree imbalance, some terms have fewer evaluations; the additional evaluations are computed via Barycentric Interpolation [7]. This is omitted in the figure since it only adds a fixed cost at the end of each round (23 modmuls for ZeroCheck and 46 modmuls for PermCheck). Each MLE observes different datapaths, so we use a specialized design to exploit high reuse, full-pipelining, and high levels of parallelism.

4.1.2 Streaming approach. At the start of the protocol, the MLE tables that are provided to the prover can be stored on-chip. However, as these MLEs undergo rounds of SumCheck, the process of incorporating challenges into the MLE values expands binary values to the full 255 bits. Though the number of MLE table entries reduces by half each round, the data itself grows by over 100×, so the total storage cost for storing all MLEs is intractable. However, each round, the intermediate values of MLE tables are only used by the main SumCheck computation and then by the MLE Update to be halved in size. Since there is no data reuse in-between rounds, we adopt a streaming-based solution to alleviate the pressure on on-chip SRAM storage. The key tradeoff here is that our SumCheck and MLE Update units become memory-bound, since each MLE in Equations 2-4 must be updated and written back to off-chip memory after SumCheck rounds, necessitating off-chip traffic. Fortunately, recent advances in high-bandwidth memory (HBM) can supply very high bandwidths to offset this. This is inline with many other cryptographic computing accelerators, which also rely on

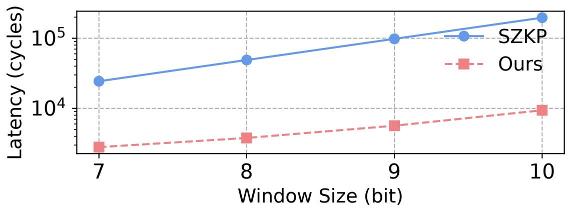


Figure 4: MSM bucket aggregation comparison.

HBM [12, 17, 26–28, 47, 48]. We analyze the bandwidth sensitivity of the SumCheck computations in our evaluation section.

4.1.3 Scaling to Multiple PEs. Each SumCheck PE handles the product and sum for one iteration of the boolean hypercube sum. For example, in Figure 1, this corresponds to one red region and the first product in the green region for each $X_1 = 0, 1, 2, 3$. These iterations run in parallel with multiple PEs storing their own accumulation registers. The bottom of Figure 3 shows the accumulation of each evaluation across boolean hypercube values (equivalently, across indices of the MLE tables). After all evaluations are done, an MLE Update PE handles the updates for one MLE, and can provision multiple modmuls. Multiple PEs can run in parallel, handling each MLE independently.

4.1.4 Unified SumCheck PE. The three polynomials f_{zero}, f_{perm}, f_{open} have different datapaths. We use HLS to generate a unified PE that handles each SumCheck variation used in HyperPlonk. Each PE requires 94 modular multipliers, compared to 184 modular multipliers without resource sharing, saving 48.9% on area. HyperPlonk's CPU baseline is designed to support *any* composition of multilinear polynomials for different protocols, not just the three we have shown. In software, repeating polynomial computations greatly improves programmability as opposed to having specialized functions to handle the specific computation patterns we optimize.

NoCap [49] is a recent accelerator that also accelerates SumCheck, but there are critical differences at the protocol-level that motivate our architecture. NoCap implements Spartan [51], which uses R1CS encodings resulting in two SumCheck instances that look as follows: $f_1 = g_1g_2g_4 - g_3g_4$ and $f_2 = g_5g_6$. NoCap uses a vector architecture with 2048 PEs to process boolean hypercube instances with a Beneš network to sum across PEs. This makes sense for NoCap because Spartan's polynomials are degree 2 and 3 with up to two terms. In contrast, HyperPlonk's polynomials have a more heterogeneous structure; there are more terms of varying degree. This complexity arises from the usage of the control signal MLEs (q_L, q_R, q_M, q_O, q_C) to represent gates. These MLEs are required to keep HyperPlonk verifier costs low (in Section 7, we see NoCap's verifier is slower). Consequently, mapping HyperPlonk's polynomials to a vector architecture would put more pressure on vector register files because of the high amount of intermediate values needed to be read. As seen in Figure 3, our SumChecks also require very complex communication, which can increase bandwidth pressure and may not be efficient to implement with a Beneš network. Further, our specialized PEs immediately reuse values without relying on register files to store the numerous amount of intermediates. We have fewer, heftier PEs so data movement costs are less relative to those incurred by a vector processor.

4.2 MSMs

MSMs are used in three Hyperplonk steps: Witness Commitments, the Wiring Identity, and Polynomial Opening. Several prior works accelerated MSMs. We begin with the base MSM design from SZKP [12], the state-of-the-art for accelerating Groth16, and propose two optimizations for better performance and area efficiency.

In HyperPlonk, the sparse MSMs run in series are on the critical path. (This is unlike prior protocols like Groth16 where their execution could be masked via parallel processing with the dense MSMs.) Consequently, we opt to use the same MSM hardware unit for both sparse and dense MSMs in zkSpeed (as opposed to separate units in SZKP). zkSpeed uses a similar scheme as SZKP for handling sparse computations. First, we compute the sum of all points corresponding to 1-valued scalars. This is done by fetching the points corresponding to 1-valued scalars into the MSM unit's SRAM banks. Then, using a tree-based approach, we feed two points at a time to the pipelined Point Adder (PADD) unit, with the result written back to the SRAM banks. This process repeats until we have reduced all points to the final sum for points corresponding to 1-valued scalars. Note that in this step, we need not fetch scalars into the MSM's SRAM banks since they are all 1. Then, we use Pippenger's algorithm [44] on the remaining $\approx$ 10% of (dense) scalars. For the Dense MSMs in the Wiring Identity and Polynomial Opening steps, we use Pippenger's algorithm for the full MSM computation.

We improve upon SZKP's architecture in two ways. First, we note that elliptical curve points, while being three-dimensional, are initialized as $(X, Y, 1)$ coordinates in HyperPlonk. Thus, we only fetch two coordinates per point, saving off-chip bandwidth. We further save on-chip SRAM area. SZKP provisions one scalar memory bank and three point memory banks, to hold X, Y, Z coordinates. In Hyperplonk Sparse MSMs do not need to store scalars, however, the tree-based addition partials still need buffering (their Z coordinates are no longer 1-valued) necessitating a Z coordinate memory bank. Therefore, zkSpeed allocates three SRAM memory banks. In dense MSM operation, the Z memory bank is reused to store the scalars, and since partial sums are *only* stored in bucket registers, we do not need to provision another dedicated memory bank, as in SZKP. This represents a savings of 18% in on-chip SRAM area compared to having a dedicated scalar memory bank.

The second optimization addresses a runtime bottleneck in SZKP's bucket aggregation step. After sorting points into buckets and computing each bucket's sum, SZKP employs a naive aggregation algorithm to calculate the sum $\sum_{i=1}^{2^W-1} iB_i$, where W is the window size [12, 44], and B_i represents the accumulated sum of the i-th bucket. This is inefficient when processing smaller MSMs, e.g., 32-point MSMs, which are prominent in Polynomial Opening. The fixed bucket aggregation latency becomes a performance bottleneck because the point additions are serially performed and do not leverage the pipelining available in the PADD unit. Consequently, the PADD is underutilized in this step. To address this, zkSpeed adapts bucket aggregation introduced in [31]. This scheme divides aggregation into smaller groups, computes the partial sums within each group in parallel, and finally combines the results. As shown in Figure 4, it reduces the bucket aggregation latency by an average of 92% across all window sizes compared to SZKP. We select a group size of 16, which provides the best overall performance and ensures the aggregation step no longer dominates runtime for small MSMs.

4.3 Multifunction Tree Unit

Many HyperPlonk functions exhibit binary-tree compute patterns, including building MLE, MLE evaluation, and constructing Product MLE (π). zkSpeed supports these in hardware with our multifunction tree unit, specially designed to handle these compute patterns effectively. Building MLE is a function used in ZeroCheck and Opencheck steps. It constructs a table with 2^N entries from random values r_1 to r_N, where 2^N represents problem size. The computation is divided into $N-1$ layers to reduce the number of modular multiplications from $(N-1)2^N$ times to $2^{N+1}-4$. The multiplier tree is used for batch inversion during Fraction MLE generation, where it efficiently calculates the product $D[0]D[1]\cdots D[n-1]$ for an inversion batch size, n. Similarly, the MLE evaluation in the Batch Evaluation operates like a multiplier tree but includes additional modular additions in each operation. The Product MLE generation in the Wiring Identity outputs all layer results. The functions' dataflow are presented in Figure 2.

The original (CPU) HyperPlonk implementation uses breadth-first (level-order) tree traversal (BFS). This is inappropriate for hardware acceleration as it puts increased pressure on SRAM capacity and off-chip bandwidth. For example, a problem size of 2^{23} requires up to 2^{22} intermediate elements in a layer, each 255 bits wide, which would require 128MB for the intermediates alone. We propose to use depth-first traversal (DFS) to address these challenges. This approach reuses and consumes intermediate results as they are computed, reducing the up to 255MB intermediates that must be stored on-chip or spilled to DRAM for problem size 2^{23}.

As shown in Figure 2, for Build MLE, if the PEs produce two outputs per cycle at the last layer, other PEs will keep generating one element each cycle in the previous layer, and one element every two cycles of the first layer. Outputs are produced continuously in the last layer, and intermediate results like $(1-r_1)(1-r_2)$ are discarded after use. The multiplier tree (in FracMLE) and MLE evaluation also benefit from DFS. For instance, as long as the unit reads two MLE table entries each cycle, it can keep generating one item of the first layer per cycle, one item every two cycles of the second layer, and so on to sustain computation similarly to a pipeline. For ProdMLE, the design outputs all layers. Since we generate an item of the second layer during the first layer, we label each item's layer index to store results correctly.

Figure 5 shows how the multifunction tree unit works. Each PE includes a modular multiplier and modular adder. (E.g., for Build MLE, $(1-r_1)r_2$ and $(1-r_1)(1-r_2)$ only require one modular multiplication since $(1-r_1)(1-r_2)$ is computed as $(1-r_1)-(1-r_1)r_2$). The hardware supports three processing modes of Figure 2. For the Inverse Tree (e.g., MLE Evaluation), the unit accepts p inputs in parallel and reduces tree-level partials via a matching hardware dataflow; in Figure 5, $p=8$, and the data flows from left to right. If the tree has more levels than hardware supports (three levels in the example; a problem size 2^{20} workload has 20 levels), the remaining levels are scheduled and processed via the accumulator. Outputs from the hardware tree (Level 3 in Figure 5) are pushed in an accumulator-local register file and once operands are ready, popped to the accumulator PE. The bottom of Figure 5 shows its

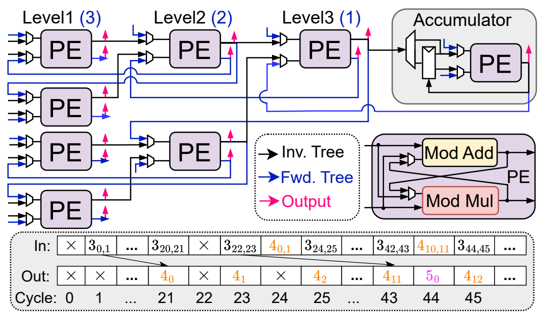


Figure 5: Hardware structure of the multifunction tree unit and accumulator schedule (blue: forward, black: inverted, red: outputs to HBM/other modules. Level for binary tree). Accumulator schedule shows the level and node index.

schedule. Initially, there are gaps as the accumulator must wait two cycles for each input pair. However, once multiple levels are processed, the gaps are filled; this can be seen at cycle 44, where layer 4 and 5 are processed by the PE at the same time (i.e., both in the pipeline). Thus the PEs in tree have high utilization: for a 2^{20} workload, they are over 99% utilized during the whole computation. The red arrows coming out of each PE show how computations are sampled and output to support ProdMLE. Forward Tree (Build MLE) support is shown in blue and data flows from right to left. Each PE takes the previous level and a challenge r_i as inputs and generates two outputs that are fed to the next level. Similarly, if the tree has more levels, the accumulator PE is scheduled to generate (roughly) one output per cycle to feed the rest of the PEs, which correspond to the last 3 levels, outputting 8 results in the end. Switching is supported by muxes at the PE inputs and configuring them.

The advantages of DFS are noticeable: it eliminates the need to store entire intermediate layers, making it practical for large problem sizes, and provides the ability to rate-match with upstream or downstream units and maintain throughput. By adjusting the number of PEs, the unit can handle varying input and output rates, forming a full pipeline with other units. The ability to reuse across multi-functions eliminates the need to allocate multiple dedicated units, saving 41.6% area across global Pareto design points in Section 7. The HyperPlonk code uses BFS. BFS has greater dependence distance; we tile DFS and schedule work to avoid dependence stalls. Executing nodes already stresses CPU's limited compute resources, and we expect DFS to have little performance impact in software.

4.4 Construct N&D and Fraction MLE

The construct N&D stage generates the N and D MLEs discussed in Section 3.3.3. Elements of six intermediate MLEs, $D_{0\ldots2}$, and $N_{0\ldots2}$, are computed in parallel from modular additions and multiplications of the witness and wiring permutation MLEs stored in on-chip SRAM, and two random challenges from SHA3. These intermediate MLEs are written off-chip for the subsequent PermutationCheck. The intermediate MLEs are multiplied to obtain the D and N MLE (e.g., $D[i]=D_0[i]D_1[i]D_2[i]$) elements, and fed to FracMLE unit.

Fraction MLE, ϕ, requires computing the modular inverse of every element of the Denominator MLE table (D^{-1}), and multiplying each inverted element with the corresponding element of the Numerator MLE table (N). Given x, modular inversion outputs a

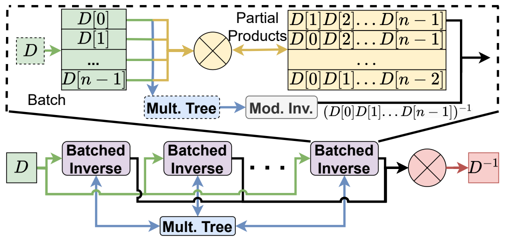


Figure 6: Modular inverse unit using multiple batched inverse units and a shared multiplier tree.

y, such that $x \cdot y \bmod p = 1$. We use the constant time *Binary Extended Euclidean Algorithm (BEEA)* [45] for this operation to make it data-oblivious and ensure constant scheduling. The algorithm requires $2W - 1$ iterations of a loop consisting of shifts and subtractions, where W is the number of input bits ($W = 255$), resulting in a 509-cycle latency. A data-dependent implementation is faster for smaller input values, but since we are computing on random inputs (derived from SHA3 hashes), the average latency of this implementation ($2(\sum_{i=1}^{255} \frac{255-i}{2^i}) - 1 \approx 505$ cycles) is only 1% better. In exchange for this negligible latency overhead, our constant-time implementation reduces design complexity as this ensures elements of D^{-1} (and thus FracMLE) are generated in-order if we execute multiple inverses in parallel (as we will discuss shortly). Parallel execution of the data-dependent algorithm may output elements out-of-order and would require buffering or stalling to resolve.

This is an expensive operation for HyperPlonk as 255-bit elements are used, and each element of an MLE must be inverted, so we heavily optimize it. We leverage Montgomery batching [38], which allows us to compute the inverse of multiple elements at the cost of one inversion and additional multiplications, to amortize the cost of one inversion across multiple elements and improve per-inversion throughput. The common optimized approach to batching, stores the partial products generated while computing the overall product using sequential multiplications. After inversion, the inverted output is sequentially multiplied by batch elements to isolate inverted batch elements. The sequential multiplications, combined with the latency of modular multiplication, can limit performance for hardware implementations. We address these limitations with two modifications. First, we use a multiplier tree, detailed in Section 4.3; this significantly reduce latency and improve scaling for large values of n, from $O(n)$ to $O(\log_2 n)$ multiplications. Second, we pre-compute the sequential multiplications needed to obtain the partial products in parallel to the modular inverse operation. This hides the latency of sequential multiplications and maximizes the work done during the long latency of the modular inverse.

Figure 6 illustrates the architecture of our modular inverse unit using batching. We use multiple batched inversion units in round-robin fashion to completely mask long inversion latencies and enable the FracMLE unit to accept one input and generate one output per cycle, behaving as a pipeline with depth $batch\ size \times number\ of\ units$. We achieve this by using enough batched inverse units to mask the latency of one batch inversion. The multiplier tree and one multiplier are reused across all units to compute their inputs and produce the individual elements of D^{-1}, respectively. One batch inversion latency is the maximum of the parallel partial

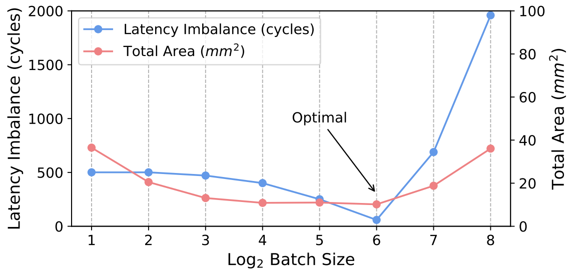


Figure 7: Latency imbalance of batched inverse unit on the left (blue) and area cost of the FracMLE unit on the right (red). Both are optimal at 64. The area includes all hardware resources needed and does not account for area savings from reuse in the overall architecture.

product latency and the combined multiplier tree & modular inverse latencies. The former scales $O(n)$ while the latter scales $O(\log_2 n)$.

We frame choosing the ideal n as an optimization problem to minimize the imbalance between the two latencies. The left y-axis of Figure 7 plots the latency imbalance as a function of n. The initial latency imbalance is due to the high constant latency of modular inverse, which decreases as the partial product latency increases – reaching a minimum at 2^6 (64) – then increases as the partial product latency overtakes the multiplier tree and modular inverse latency. Another consideration when choosing n is the area cost due to the multiplier and storage overhead. The right y-axis of Figure 7 plots the total area of the Modular Inverse unit for different batch sizes, including the multiplier tree, partial products SRAM, batch element registers, and multipliers. The minimum area is also seen at 64 because more inverse units are needed to mask the latency of one inversion at smaller batch sizes. For example, $n = 2$ requires 256 batched inverse units while only 12 units are required at $n = 64$. Also, starting at $n = 64$, we can reuse the multiplier tree across all units since the tree can compute a batch's product before the next batch is ready thanks to its $O(\log_2 n)$ scaling. The total area increases past $n = 64$ because the latency of the multiplications for the partial products overtakes the latency of modular inversion (hence latency imbalance also starts to increase), and the number of batched inverse units no longer decreases while the area overhead of batching (e.g., SRAM) continues to increase without additional benefit. Using these metrics, we get a batch size of 64.

4.5 MLE Combine Unit

The MLE Combine Unit is used in the Polynomial Opening step. As mentioned in Section 3.3.5, there are several linear combinations of MLEs that are performed before OpenCheck and before the MSMs. These operations are straightforward, and use a combination of MLEs stored in on-chip SRAM and off-chip memory to construct the MLEs used in subsequent steps. Because Opencheck happens in series with the MSMs, the respective MLE Combine operations also happen in sequence. Consequently, we can share resources between these two operations. For the design point we highlight in Table 5, without sharing, we would require 122 modular multipliers. With sharing, we require only 72, representing a 41% area savings.

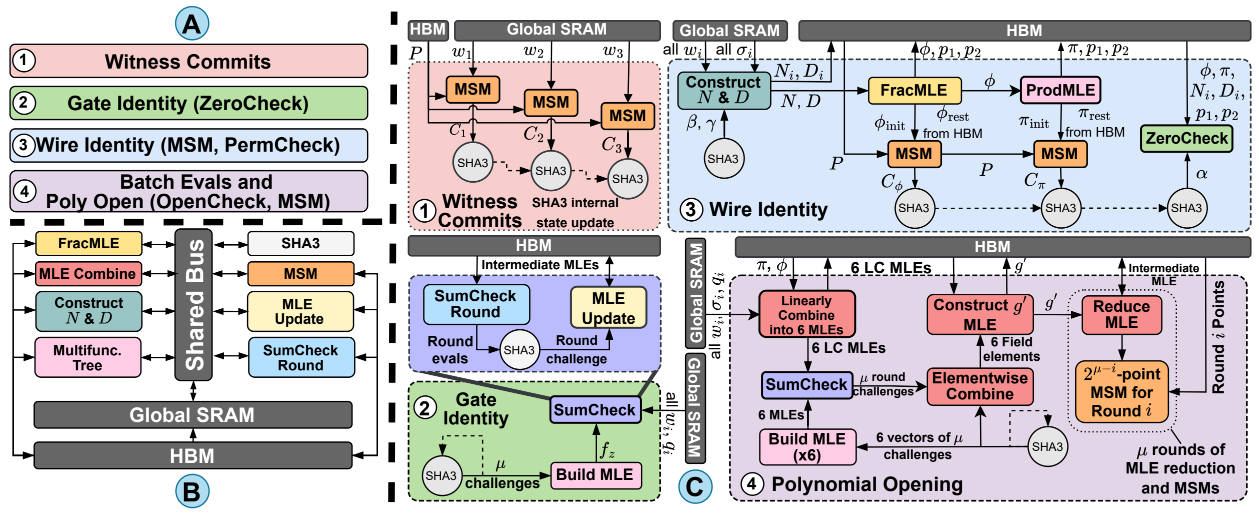


Figure 8: A) HyperPlonk Protocol Steps, B) zkSpeed Architecture, and C) Step Dataflow. Each step is executed in the numerical order listed. Colors of each module in the zkSpeed architecture directly correspond with the operations in the dataflow. Batch Evals only use the Multifunction Tree unit, they are omitted for space.

4.6 On-Chip MLE SRAM

The MLE table size scales with the number of gates, i.e., for a problem size of 2^μ gates, each MLE table has 2^μ entries. In practice, input MLEs are sparse. Control MLEs q_L, q_R, q_M, q_O are all binary, and q_C, w_1, w_2, w_3 are roughly 90% 1s and 0s and 10% full bit-width. These MLEs get reused throughout the protocol, so we store them on-chip in Global SRAM. We compress the tables, packing together control MLEs and using address translation units to perform lookups to either binary or 255-bit data. These compression strategies save 10 to 11× on MLE storage across problem sizes.

5 zkSpeed Architecture & HyperPlonk Mapping

Figure 8 provides an overview of the HyperPlonk protocol and zkSpeed architecture. HyperPlonk is expressed as a series of five steps, with the final steps (4A and 4B) executed in parallel. Colors indicate the mapping of protocol steps (Section 3.3) to the accelerator units (Section 4) they run on. The zkSpeed architecture is streaming in nature and captures on-chip data reuse when feasible via explicitly managed scratchpad memories. The architecture has four major components: accelerator units (Section 4), local and global SRAM, a multi-channel shared bus, and HBM interface. HBM is needed to feed the chip with high bandwidths needed by HyperPlonk, and we conduct bandwidth sensitivity studies in Section 7.

zkSpeed uses a shared bus rather than a crossbar or NoC. This design choice was made after rigorously analyzing the HyperPlonk dataflow. The dataflow of each HyperPlonk protocol step is shown in Figure 8(C). Colors indicate the hardware module each kernel executes on and annotated wires show data movement. We observed that at any given time, only 1-2 zkSpeed units typically communicate, and at most 4 independent bus channels are needed to avoid stalling – this is during Wire Identity where Construct N&D sends results to FracMLE, FracMLE simultaneously feeds ProdMLE and MSM units, and ProdMLE streams to MSM. Units overlap computation with each other, e.g., enabling MSM to start processing partial outputs from FracMLE, effectively masking latency. Without bus stalls, we are able to rate match each accelerator unit to pipeline across modules when possible, further improving performance.

zkSpeed deploys a highly banked global SRAM and two local SRAMs for FracMLE and MSM units that store data unused by other units (the FracMLE SRAM captures MLE table reuse, the MSM's SRAM reuses elliptical curve points). All other units share the Global SRAM, which stores input MLEs. At the start of execution, these MLEs are prefetched from HBM and remain unchanged on-chip throughout execution. They are read at the beginning of multiple protocol steps, thereby reducing HBM pressure. zkSpeed allocates a single-channel shared bus for units to read the global SRAM since only one unit requires access at any given time. HBM access is managed by a memory controller that has dedicated point-to-point connections to each module and the Global SRAM, with enough wires to a given component to accommodate the widest access needed. The controller interfaces with the HBM PHYs and arbitrates access to the HBM channels to ensure that no channel is being used more than once simultaneously (i.e., no channel conflict).

Dataflow. HyperPlonk is data oblivious at the stage granularity. This allows it to statically schedule computations and manage units, SRAM, and the buses via a simple controller. The Batch Evaluation and Polynomial Opening steps run in parallel because the same MLEs used prior to Opencheck also undergo MLE evaluations. Both steps require simultaneous access to 13 MLE tables. With our on-chip MLE storage scheme, only two of these MLEs are stored off-chip, π and ϕ. This saves bandwidth by 84%. By synchronizing the off-chip accesses between the two protocol steps, zkSpeed further halves the bandwidth pressure. We provision six hardware units for computing Batch Evaluations, since this step is not on the critical path, and the remaining MLE evaluations only rely on MLEs already stored on-chip. Thus we eliminate batch evaluations from the critical path without incurring additional bandwidth costs.

zkSpeed Programmability. zkSpeed modules are programmed by instructions specifying problem sizes and configuring local controllers and address data from the buses, SRAM, and HBM. Due to the ASIC nature, much of the fine-grained control is handled by FSMs within each unit. For each HyperPlonk protocol step, each unit (e.g., bus, modules, Global SRAM, and HBM controller) are configured with complex instructions and run to completion. Then next set of instructions are loaded to execute the next protocol step.

Hyperplonk Trends and Outlook. Zero-knowledge protocols are still continuously evolving and improving in several aspects. However, zkSpeed, with its focus on Hyperplonk still has significant stability. Firstly, Hyperplonk has seen adoption both in industry implementations [15, 65] and academic research [15, 16, 29]. In addition, our design is modular, and the key components of Hyperplonk SumChecks, MSMs, and MLEs, are present in essentially any modern SNARK protocol [8, 21, 30, 51, 52, 56]. Therefore, zkSpeed can also be targeted to new protocols so long as they comprise the fabricated modules, including proof composition methods [58] that seek to compose protocols like Orion and Hyperplonk.

6 Methodology

6.1 Performance Modeling

SumCheck has a fixed dataflow and is data oblivious. Thus, we model performance using analytical models. Modules that feed directly to the SumCheck units are handled in a data-oblivious manner, so they are also modeled analytically since we assume rate-matching. For the MSM, we use a cycle-accurate simulator to model performance. Each unit within zkSpeed is modularized. To understand full-chip performance, we conduct a design-space exploration of all combinations of design parameters detailed in Table 2 and then analyze the pareto-optimal space to pick a suitable configuration for profiling runtimes. We also construct power traces to estimate average power for the full-chip architecture.

We use Catapult HLS 2023 to generate the RTL for Montgomery multipliers (as done in prior work [12]), the fully-pipelined, unified SumCheck unit (to handle ZeroCheck, PermutationCheck, and OpenCheck), and the fully-pipelined PADD unit. Consistent with HyperPlonk, we use the BLS12-381 elliptical curve, where all MLE datatypes (e.g., in the SumCheck unit) are 255b, and all elliptic curve points (e.g., in the PADD) are 381b. Using Design Compiler with TSMC 22nm, we find the critical path in our design is the 381b PADD unit at 1.05ns. We use Synopsys 22nm Memory Compiler to generate SRAM estimates. For SHA3, we use the publicly available IP block from OpenCores [43]. We scale down to 7nm using scale factors of 3.6× for area, 3.3× for power, and 1.7× for delay (as in prior work [12]), and clock all zkSpeed accelerators at 1 GHz.

6.2 Benchmarks

HyperPlonk was evaluated using *mock circuit* workloads [10], as there is no publicly available compiler to generate real workloads. We similarly use synthetically generated workloads to model the performance of our architecture, which is standard for ZKP benchmarks, as performance primarily depends on the size of the workload. Similarly, GZKP [36] uses synthetic workloads to benchmark their implementation for workload sizes of 2^{22} and higher. NoCap [49] uses workloads from libsnark [1] that prove relatively small circuits and scales them up to larger problem sizes because their proving times are dominated by fixed overheads on smaller ZKP circuits. In the context of HyperPlonk, workload statistics primarily affect the witness commit (sparse MSM) step. The scalar distributions of the MSMs in the Wiring Identity and Polynomial Opening are random because they are constructed in part from SHA3 challenges. Therefore, while the MSMs are data-dependent, they are not workload dependent and their runtimes are roughly the same at iso-problem size. All other steps in HyperPlonk are data-oblivious. From prior work [1, 12, 36, 64], we know that in sparse MSMs, the scalars are typically 5-10% dense (i.e., full bitwidth), and 95-99% sparse. Since the dense components of sparse MSMs are still runtime dominant, we assume a pessimistic upper bound of 10% dense scalars and 90% sparse scalars, of which 45% are 1s and 45% are 0s. The rest of the protocol steps operate on 100% dense scalars/MLE values. In our evaluation, we use five workloads from prior work [1, 10], and show our results in Table 3.

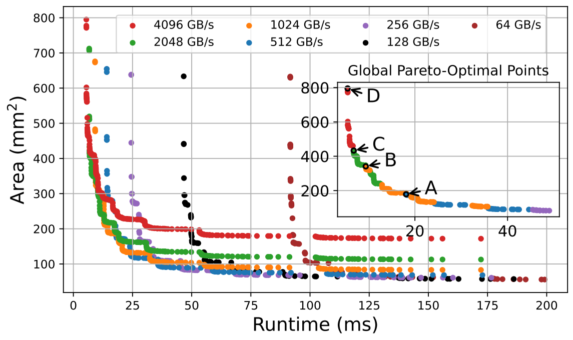


Figure 9: Pareto Frontiers for 2^{20} Gates. We plot the individual pareto curves for each bandwidth model and the globally optimal pareto curve (for designs under 50ms) in the inset.

7 Evaluation

7.1 Pareto Space Analysis

Figure 9 shows the design space for a problem size of 2^{20} gates under four bandwidth scenarios. We sweep all the parameters in Table 2 and obtain Pareto curves for each bandwidth individually, and then construct the global Pareto curve from these four local Pareto curves. The key highlight from this plot is that HBM3-scale bandwidths (e.g. 1-4 TB/s [2]) *do* yield significant performance gains over, e.g., an HBM2-scale bandwidth (0.5 TB/s [24]). Beyond 300 mm^2, the globally pareto-optimal design configurations yield over 2× speedups compared to 512 GB/s designs and over 700× speedups over the CPU baseline. This is because high-performance SumCheck designs quickly saturate 512 GB/s of bandwidth. In this analysis, we also include the cost of HBM PHYs [17, 24, 26–28, 37, 47–49] where the PHY cost is 14.9 mm^2 for a single HBM2 PHY and 29.6 mm^2 for a single HBM3 PHY. For low-performance designs (below 100mm^2), higher bandwidth becomes less effective because of the related PHY area costs. Figure 9 shows that zkSpeed remains viable even at bandwidths typical of DDR5 [50] (256 GB/s and below). While HBM allows exploring scalability benefits, less expensive memory technologies can be used to achieve Pareto-optimality within a target performance range (e.g., within 50 ms).

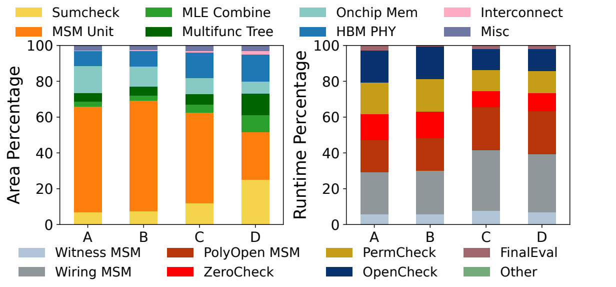


Figure 10: Area (left, top legend) and runtime (right, bottom legend) breakdown for the chosen Pareto points in Figure 9.

Table 2: Design Space of zkSpeed Architecture.

Module	Design Knob	Values
MSM	PEs	1, 2, 4, 8, 16
MSM	Window Size	7, 8, 9, 10
MSM	Points/PE	1K, 2K, 4K, 8K, 16K
FracMLE	PEs	1, 2, 4
SumCheck	PEs	1, 2, 4, 8, 16
MLE Update	PEs	1, 2, . . . , 11
MLE Update	Modmuls/PE	1, 2, 4, 8, 16
—	Bandwidth (GB/s)	64, 128, 256, 512, 1T, 2T, 4T

We analyze the area and runtime of selected points on the Pareto curve in Figure 10 to further understand bandwidth sensitivity. We pick Pareto points representing the highest-performing design point for each bandwidth level. In the area breakdown, moving from low to high-performance design points (A to D), the proportion of the SumCheck area increases significantly, because SumCheck is bandwidth intensive, and higher bandwidth allows more parallel SumCheck PEs, boosting throughput. The MSM unit accounts for a large portion of the total area, but its absolute area remains unchanged when switching to high performance. This trend is also evident in the runtime breakdown: total runtime decreases as bandwidth increases, and the runtime contributions of the SumCheck-related processes (ZeroCheck, PermutationCheck, and OpenCheck) become smaller. Our analysis of the Pareto design points shows that high performance significantly depends on sufficient bandwidth, particularly improving the SumCheck computation. Conversely, for low-performance designs, the system utilizes less bandwidth and allocates more resources to MSM computation.

7.2 Bandwidth Sensitivity

Figure 12 shows how the speedups for MSM-related computation and SumCheck-related computation scale with increased PE count and bandwidth. These are two runtime-dominant components in our design points shown in Figure 10. We take the runtime of all MSM and SumCheck operations for 1 PE under 512 GB/s, and compute respective speedups to these numbers. Because MSMs are compute-bound, adding compute resources improves the speedups significantly, while adding bandwidth does not. We do not see perfectly linear speedup because of the serialization incurred in PolyOpen MSMs. SumChecks, which rely on a streaming-based approach, are memory-bound. As we add compute, we see linear speedups initially and then diminishing returns after saturating bandwidth. Consequently, in our pareto-optimal design space, we see most points along the curve outside high-performance regimes use at most 2 SumCheck PEs compared to 8 or 16 MSM PEs.

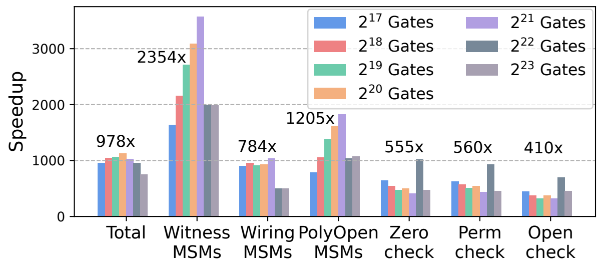


Figure 11: Speedup over CPU at Iso-CPU Area Designs. Each problem size has a different pareto-optimal point. Each bar reflects absolute speedup, while the annotated speedup is the gmean computed across gate counts for each kernel.

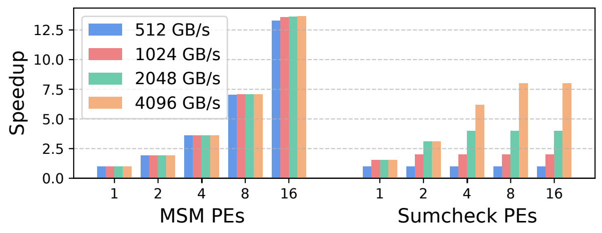


Figure 12: Performance scaling with different memory technologies and PEs for MSM and SumCheck. Speedup normalized to 1 PE at 512 GB/s.

7.3 Iso-CPU-Area Comparisons

Our CPU is an AMD EPYC 7502 32-core processor [5, 12, 39, 55]. The total die size is 296 mm^2. We sweep the problem sizes, and for each problem size, pick a pareto-optimal design point that is close to 296 mm^2. In these comparisons, we exclude the PHY cost, since the AMD EPYC processor has its own separate die for I/O [40]. Therefore, we compare our total compute and on-chip memory area with the CPU's total core area, including on-chip caches. We assume 2 TB/s HBM to achieve pareto-optimality in Figure 9.

After picking each design, we run synthetic benchmarks over problem sizes $2^{17} - 2^{23}$. Figure 11 shows the speedup of each design over CPU baseline, and the breakdown across different steps of the protocol to understand where our speedups come from. In general, we get more speedup from our MSM units than the SumCheck units. This intuitively follows, given our observations that MSMs are compute-bound and more robust to bandwidth constraints. Additionally, the CPU poorly handles sparse computations because it serially computes the point addition for 1-valued scalars. PolyOpen MSMs also incur serialization costs that we reduce by overlapping MSM executions where possible. The variations in speedups over different problem sizes are an artifact of our choice to highlight different Pareto points per problem size; for example, at 2^{20} size problems, a dual-core MSM is used, while at 2^{22} size a single-core MSM is chosen. This is because the area costs of on-chip MLE SRAM begins to dominate; choosing a weaker MSM in turn reduces achievable speedup. Storing MLE tables entirely off-chip may yield larger MSM speedups at higher bandwidth costs for SumChecks. These tradeoffs can be explored in future work.

Table 3: zkSpeed evaluation on real-world workloads.

Workload	Problem Size	Runtime (ms)	
		CPU	zkSpeed
Zcash	2^{17}	1429	1.984 (**720×**)
Auction	2^{20}	8619	11.405 (**755×**)
2^{12} Rescue-Hash Invocations	2^{21}	18637	18.335 (**844×**)
Zexe's Recursive Circuit	2^{22}	37469	43.451 (**862×**)
Rollup of 10 Pvt Tx	2^{23}	74052	86.181 (**859×**)

Table 4: Comparison of zkSpeed with Prior ZKP Accelerators on 2^{24} Constraints/Gates. N = NTT, S = SumCheck, M = MSM

Accelerator	NoCap	SZKP+	zkSpeed
Protocol	Spartan+Orion	Groth16	HyperPlonk
Main Kernels	N & S	N & M	S & M
Encoding	R1CS	R1CS	Plonk
Proof Size	8.1 MB	0.18 KB	5.09 KB
Setup	none	circuit-specific	universal
Prime	fixed	arbitrary	arbitrary
Bitwidth	64	255b/381b	255b/381b
CPU Prover (s)	94.2	51.18	145.5
HW Prover (ms)	151.3	28.43	171.61
Verifier (ms)	134	4.2	26
Chip Area (mm^2)	38.73	353.2	366.46
# Modmuls	2432	1720	1206
Modmul (mm^2)	0.002	0.133 / 0.314	0.133 / 0.314
Power (W)	62	>220 W	170.88

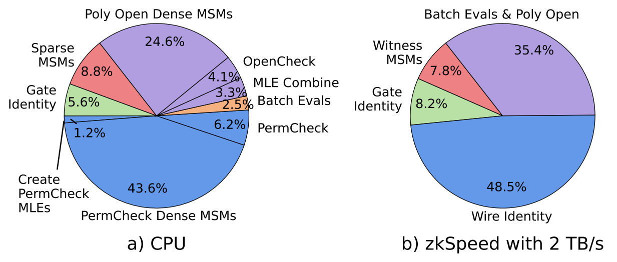


Figure 13: Runtime breakdown for CPU and zkSpeed at 2^{20} gates. CPU's sequential kernel execution enables finer breakdowns; aggregate step times are presented for zkSpeed.

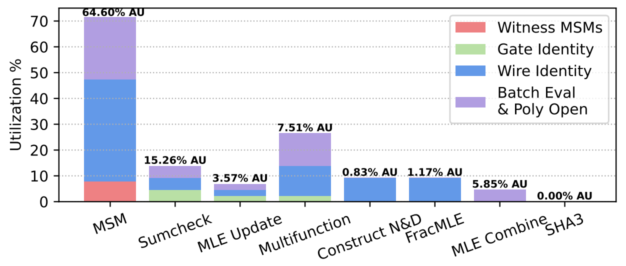


Figure 14: Utilization of zkSpeed modules, with compute area utilization (AU) listed on top of each bar. Stacked bar colors reflect the protocol steps in which each module is active.

7.3.1 Runtime Breakdown and Utilization. Figure 13 shows the latency breakdown of HyperPlonk on a CPU and zkSpeed. The CPU executes kernels sequentially, enabling detailed profiling; we report step latency for zkSpeed due to its parallel scheduling of kernels. As expected, the majority of time goes to processing MSMs, while a handful of other kernels account for single percentage points of runtime. Figure 14 presents the utilization of each unit and relative (datapath) area allocation (design in Table 5). The utilizations vary from over 70% to 5% for some modules. zkSpeed was intentionally designed (via the design space search in Figure 9) to allocate resources to cores to optimize high performance per area, i.e. the Pareto front. This can be seen in our analysis in two ways. First, the cores taking up most area, notably MSM at 64.6%, are the most used, and following the profiling data (Figure 13) require the most speedup. Second, though some units are used infrequently, they (i) take up little area and (ii) are essential to accelerate to achieve the speedups desired (i.e., 2-3 orders of magnitude). For example, the SHA-3 unit is rarely used, but provides a speedup of over 300× over the CPU and takes only 5888μm^2 area. Additionally, consider that MLE Combine makes up 3.3% of the CPU runtime, but without acceleration caps speedup to a mere 30.3×, thus justifying its 5.85% area allocation and relatively low utilization.

Table 5: Area and power of zkSpeed. Other includes the SHA3 unit and interconnect.

	Area (mm^2)	Average Power (W)
MSM (16 PEs)	105.64	76.19
SumCheck (2 PEs)	24.96	5.38
Construct N&D	1.35	0.19
FracMLE	1.92	0.25
MLE Combine	9.56	0.34
MLE Update	5.84	1.13
Multifunction Tree	12.28	4.16
Other	1.98	0.04
Total Compute	**163.53**	**87.68**
SRAM	143.73	19.60
HBM3 (2 PHYs)	59.20	63.60
Total Memory	**202.93**	**83.20**
Total	**366.46**	**170.88**

7.4 Workload Evaluation

We pick a fixed design and show the end-to-end speedups in Table 3. As mentioned in Section 6.2, we assume a pessimistic 10% sparse scalar statistics for each workload. Our fixed design has one MSM unit with 9-bit windows, 16 PEs, and 2048 points per PE, with 1 FractionMLE PE, 2 SumCheck PEs, 11 MLE Update PEs, and 4 modular multipliers per MLE Update PE. The area is provided in Table 5. At roughly iso-CPU-core compute cost, zkSpeed achieves a geometric mean speedup of 801× over the CPU, with total area of 366.46 mm^2, total average power of 170.88 W, and total power density of 0.46 W/mm^2, which is within that of our CPU [12].

8 Related Work

Much of the prior body of crytographic hardware and systems research has focused on Fully Homomorphic Encryption and Multi-Party Computation [19, 26–28, 37, 41, 47, 48, 54]. ZKP hardware research is relatively newer, and has focused primarily on accelerating NTTs and MSMs [9, 11, 23, 25, 31–33, 35, 46, 59, 60, 63, 66, 67]. A few recent works have accelerated SumChecks on GPU [34] and ASIC [49] as well as hashing alternatives to SHA-based hash functions [4, 53, 60]. Some systems accelerate end-to-end Groth16 proofs (using NTTs and MSMs) on GPU [36] and ASICs [12, 64].

SZKP is presently the only ASIC that accelerates Groth16 proofs entirely on-chip. NoCap [49] is an ASIC that accelerates the Spartan protocol, using Orion as the polynomial commitment scheme. NoCap focuses on SumCheck and NTTs used in Spartan. We compare zkSpeed with two ASICs that accelerate full proofs end-to-end.
SZKP is the state-of-the-art for accelerating Groth16 proofs, focusing on scalable MSM designs and (quasi)-deterministic scheduling for Pippenger's algorithm. It accelerates all MSMs, including Sparse G2 MSMs, achieving geomean speedups of 493× over a CPU. SZKP improves on PipeZK [64], the first hardware accelerator for Groth16 proofs. While Groth16 and HyperPlonk have similar application spaces, as mentioned in Section 1, the key advantage of using HyperPlonk is the universal setup, which means that the protocol parameters are application-agnostic. For Groth16, *every new application* that wants to use a ZKP needs its own trusted setup ceremony [3], which is impractical as the application space grows. Given this context and the recent shift away from Groth16 [42], the slightly larger proof sizes are considered a reasonable tradeoff.
NoCap is a vector-based processor for accelerating Spartan+Orion proofs, but its application space differs from zkSpeed's. NoCap thrives in applications where proof size is not critical or there are few verifiers. It achieves 41× geomean speedups over PipeZK. In contrast, zkSpeed is ideal for many verifiers and in consensus-based systems; this is where ZKPs are experiencing growing interest.

For easier comparison, Table 4 compares zkSpeed, NoCap, and SZKP's protocols and software and hardware costs. zkSpeed's parent Hyperplonk has the slowest software prover, reflecting the complexity of the protocol. Of note, Spartan's prover is slow; NoCap's authors explain this is due to inefficient implementation.

We compare NoCap's hardware implementation using the design point and numbers from their paper scaled to 7nm using scale factors from prior work [12, 37]. We then select a zkSpeed configuration with roughly similar prover time. At iso-prover time, zkSpeed incurs a nearly 10× area cost in return for a three orders-of-magnitude reduction in proof size. NoCap's lower costs come from eliminating MSMs, having simpler sumchecks, and using a 64-bit Goldilocks-64 prime field that yields smaller modmuls. In contrast, zkSpeed supports arbitrary 255-bit and 381-bit primes for MLEs and elliptic curves points, respectively. Consequently, NoCap runs all operations several times, including SumChecks 3 times, to obtain 128 bits of security. We further compare zkSpeed with an iso-area SZKP (Groth16) implementation, giving them the benefit of zkSpeed's improved MSMs, and optimistically scale up their design to use the BLS12-381 curve. This design, SZKP+, enjoys a 6× reduction in proving time compared to zkSpeed, largely because it has fewer MSMs on its critical path. These speedups come at the cost of circuit-specific setup, incurring large costs any time the application is updated. In sum, NoCap, SZKP, and zkSpeed each address different application domains, representing a range of trade-offs ranging from security and protocol properties to software/hardware costs.

Jellyfish: Jellyfish is a HyperPlonk variant supporting gates of arity (fan-in) higher than 2. Unlike R1CS, it supports higher degree constraints, e.g. $x^7 = y^5 + y^2 + 7$. The additional expressiveness means, iso-application, the total size of all MLE tables decreases (the number of tables increases with arity, but table size decreases super-proportionally). High-degree gates have utility in many applications[14]; this is especially pronounced when proving the correctness of cryptographic operations like encryption[62] or hash-functions[20]. zkSpeed could be extended to support Jellyfish, in which case the ratio of table count to table size may improve the runtime (with sufficient bandwidth). We leave this for future work.

9 Conclusion

This paper presents zkSpeed, the *first* work to accelerate HyperPlonk proofs in hardware, which offers $O(n)$ time complexity compared to prominent zkSNARKs that rely on computational primitives that have $O(n \log n)$ complexity (e.g. Groth16). zkSpeed constitutes accelerator units for all core HyperPlonk functions, with special attention paid to prominent kernels: SumCheck and MSM. zkSpeed is a modular architecture, and we leverage a performance model to conduct design space optimization, and analyze the pareto frontier to identify well performing designs. A zkSpeed accelerator with 366 mm^2 and 2 TB/s of bandwidth achieves gmean speedup of 801× over CPU baselines, demonstrating the promise zkSpeed offers to accelerate HyperPlonk.

Acknowledgments

This research was developed with funding in part from the NSF CAREER award #2340137, an NSF RINGS Award, from DARPA under the Data Protection in Virtual Environments (DPRIVE) program, grant number HR0011-21-9-0003, and the Hybrid Electromagnetic side-channel and Interactive-proof Methods to Detect and Amend LogicaL Rifts (HEIMDALLR) programs, grant number HR0011-25-C-0300, and from a gift award from Google.

References

[1] 2018. libsnark: a C++ library for zkSNARK proofs. https://github.com/scipr-lab/libsnark
[2] 2023. *High Bandwidth Memory DRAM (HBM3)*. Technical Report JESD238A. JEDEC. [Online]. Available: https://www.jedec.org/standards-documents/docs/jesd238a.
[3] 2023. What is the ZCash Ceremony? The Complete Beginners Guide. https://coinbureau.com/education/zcash-ceremony/.
[4] Anees Ahmed, Nojan Sheybani, Davi Moreno, Nges Brian Njungle, Tengkai Gong, Michel Kinsy, and Farinaz Koushanfar. 2024. AMAZE: Accelerated MiMC Hardware Architecture for Zero-Knowledge Applications on the Edge. arXiv:2411.06350 [cs.CR] https://arxiv.org/abs/2411.06350 Accepted to ICCAD 2024.
[5] Product AMD. 2024. Server Processor Specifications. https://www.amd.com/en/products/specifications/server-processor.html
[6] Eli Ben-Sasson, Alessandro Chiesa, Christina Garman, Matthew Green, Ian Miers, Eran Tromer, and Madars Virza. 2014. Zerocash: Decentralized Anonymous Payments from Bitcoin. In *2014 IEEE Symposium on Security and Privacy, SP 2014, Berkeley, CA, USA, May 18-21, 2014*. IEEE Computer Society, 459–474. https://doi.org/10.1109/SP.2014.36
[7] Jean-Paul Berrut and Lloyd N Trefethen. 2004. Barycentric lagrange interpolation. *SIAM review* 46, 3 (2004), 501–517.
[8] Benedikt Bünz and Binyi Chen. 2023. Protostar: Generic Efficient Accumulation/Folding for Special-Sound Protocols. In *Advances in Cryptology - ASIACRYPT 2023 - 29th International Conference on the Theory and Application of Cryptology and Information Security, Guangzhou, China, December 4-8, 2023, Proceedings, Part II (Lecture Notes in Computer Science, Vol. 14439)*, Jian Guo and Ron Steinfeld (Eds.). Springer, 77–110. https://doi.org/10.1007/978-981-99-8724-5_3
[9] Shahzad Ahmad Butt, Benjamin Reynolds, Veeraraghavan Ramamurthy, Xiao Xiao, Pohrong Chu, Setareh Sharifian, Sergey Gribok, and Bogdan Pasca. 2024. if-ZKP: Intel FPGA-Based Acceleration of Zero Knowledge Proofs. arXiv:2412.12481 [cs.AR] https://arxiv.org/abs/2412.12481
[10] Binyi Chen, Benedikt Bünz, Dan Boneh, and Zhenfei Zhang. 2022. HyperPlonk: Plonk with Linear-Time Prover and High-Degree Custom Gates. Cryptology ePrint Archive, Paper 2022/1355. https://eprint.iacr.org/2022/1355
[11] Xiangren Chen, Bohan Yang, Wenping Zhu, Hanning Wang, Qichao Tao, Shuying Yin, Min Zhu, Shaojun Wei, and Leibo Liu. 2024. 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 2025, 1 (Dec. 2024), 275–313. https://doi.org/10.46586/tches.v2025.i1.275-313

[12] Alhad Daftardar, Brandon Reagen, and Siddharth Garg. 2024. SZKP: A Scalable Accelerator Architecture for Zero-Knowledge Proofs. In *Proceedings of the 2024 International Conference on Parallel Architectures and Compilation Techniques*. 271–283.

[13] Leo de Castro, Rashmi Agrawal, Rabia Yazicigil, Anantha Chandrakasan, Vinod Vaikuntanathan, Chiraag Juvekar, and Ajay Joshi. 2021. Does Fully Homomorphic Encryption Need Compute Acceleration? Cryptology ePrint Archive, Paper 2021/1636. https://eprint.iacr.org/2021/1636

[14] Michel Dellepere, Pratyush Mishra, and Alireza Shirzad. 2024. Garuda and Pari: Faster and Smaller SNARKs via Equifficient Polynomial Commitments. Cryptology ePrint Archive, Paper 2024/1245. https://eprint.iacr.org/2024/1245

[15] Benjamin E. Diamond and Jim Posen. 2023. Succinct Arguments over Towers of Binary Fields. Cryptology ePrint Archive, Paper 2023/1784. https://eprint.iacr.org/2023/1784

[16] Benjamin E. Diamond and Jim Posen. 2024. Polylogarithmic Proofs for Multilinears over Binary Towers. *IACR Cryptol. ePrint Arch.* (2024), 504. https://eprint.iacr.org/2024/504

[17] Austin Ebel and Brandon Reagen. 2024. Osiris: A Systolic Approach to Accelerating Fully Homomorphic Encryption. arXiv:2408.09593 [cs.CR] https://arxiv.org/abs/2408.09593

[18] Ariel Gabizon, Zachary J. Williamson, and Oana Ciobotaru. 2019. PLONK: Permutations over Lagrange-bases for Oecumenical Noninteractive arguments of Knowledge. Cryptology ePrint Archive, Paper 2019/953. https://eprint.iacr.org/2019/953

[19] Karthik Garimella, Zahra Ghodsi, Nandan Kumar Jha, Siddharth Garg, and Brandon Reagen. 2023. 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* (Vancouver, BC, Canada) *(ASPLOS 2023)*. Association for Computing Machinery, New York, NY, USA, 89–104. https://doi.org/10.1145/3582016.3582065

[20] Lorenzo Grassi, Dmitry Khovratovich, Christian Rechberger, Arnab Roy, and Markus Schofnegger. 2019. Poseidon: A New Hash Function for Zero-Knowledge Proof Systems. Cryptology ePrint Archive, Paper 2019/458. https://eprint.iacr.org/2019/458

[21] Jens Groth. 2016. On the Size of Pairing-based Non-interactive Arguments. Cryptology ePrint Archive, Paper 2016/260. https://eprint.iacr.org/2016/260 https://eprint.iacr.org/2016/260.

[22] Jens Groth, Markulf Kohlweiss, Mary Maller, Sarah Meiklejohn, and Ian Miers. 2018. Updatable and Universal Common Reference Strings with Applications to zk-SNARKs. In *Advances in Cryptology - CRYPTO 2018 - 38th Annual International Cryptology Conference, Santa Barbara, CA, USA, August 19-23, 2018, Proceedings, Part III (Lecture Notes in Computer Science, Vol. 10993)*, Hovav Shacham and Alexandra Boldyreva (Eds.). Springer, 698–728. https://doi.org/10.1007/978-3-319-96878-0_24

[23] Florian Hirner, Florian Krieger, and Sujoy Sinha Roy. 2025. Chiplet-Based Techniques for Scalable and Memory-Aware Multi-Scalar Multiplication. Cryptology ePrint Archive, Paper 2025/252. https://eprint.iacr.org/2025/252

[24] Rambus Inc. 2020. White Paper: HBM2E and GDDR6: Memory Solutions for AI. White Paper.

[25] Zhuoran Ji, Zhiyuan Zhang, Jiming Xu, and Lei Ju. 2024. 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* (La Jolla, CA, USA) *(ASPLOS '24)*. Association for Computing Machinery, New York, NY, USA, 57–70. https://doi.org/10.1145/3620666.3651364

[26] Jongmin Kim, Sangpyo Kim, Jaewan Choi, Jaiyoung Park, Donghwan Kim, and Jung Ho Ahn. 2023. SHARP: A Short-Word Hierarchical Accelerator for Robust and Practical Fully Homomorphic Encryption. In *Proceedings of the 50th Annual International Symposium on Computer Architecture* (Orlando, FL, USA) *(ISCA '23)*. Association for Computing Machinery, New York, NY, USA, Article 18, 15 pages. https://doi.org/10.1145/3579371.3589053

[27] Jongmin Kim, Gwangho Lee, Sangpyo Kim, Gina Sohn, Minsoo Rhu, John Kim, and Jung Ho Ahn. 2022. ARK: Fully Homomorphic Encryption Accelerator with Runtime Data Generation and Inter-Operation Key Reuse. In *2022 55th IEEE/ACM International Symposium on Microarchitecture (MICRO)*. 1237–1254. https://doi.org/10.1109/MICRO56248.2022.00086

[28] Sangpyo Kim, Jongmin Kim, Michael Jaemin Kim, Wonkyung Jung, John Kim, Minsoo Rhu, and Jung Ho Ahn. 2022. BTS: An Accelerator for Bootstrappable Fully Homomorphic Encryption. In *Proceedings of the 49th Annual International Symposium on Computer Architecture* (New York, New York) *(ISCA '22)*. Association for Computing Machinery, New York, NY, USA, 711–725. https://doi.org/10.1145/3470496.3527415

[29] Tohru Kohrita and Patrick Towa. 2024. Zeromorph: Zero-Knowledge Multilinear-Evaluation Proofs from Homomorphic Univariate Commitments. *J. Cryptol.* 37, 4 (2024), 38. https://doi.org/10.1007/S00145-024-09519-0

[30] Abhiram Kothapalli, Srinath T. V. Setty, and Ioanna Tzialla. 2022. Nova: Recursive Zero-Knowledge Arguments from Folding Schemes. In *Advances in Cryptology - CRYPTO 2022 - 42nd Annual International Cryptology Conference, CRYPTO 2022, Santa Barbara, CA, USA, August 15-18, 2022, Proceedings, Part IV (Lecture Notes in Computer Science, Vol. 13510)*, Yevgeniy Dodis and Thomas Shrimpton (Eds.). Springer, 359–388. https://doi.org/10.1007/978-3-031-15985-5_13

[31] Changxu Liu, Hao Zhou, Patrick Dai, Li Shang, and Fan Yang. 2024. PriorMSM: An Efficient Acceleration Architecture for Multi-Scalar Multiplication. *ACM Transactions on Design Automation of Electronic Systems* 29, 5 (2024), 1–26.

[32] Changxu Liu, Hao Zhou, Lan Yang, Zheng Wu, Patrick Dai, Yinlong Li, Shiyong Wu, and Fan Yang. 2024. Myosotis: An Efficiently Pipelined and Parameterized Multi-Scalar Multiplication Architecture via Data Sharing. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems* (2024), 1–1. https://doi.org/10.1109/TCAD.2024.3524364

[33] Changxu Liu, Hao Zhou, Lan Yang, Jiamin Xu, Patrick Dai, and Fan Yang. 2024. Gypsophila: A Scalable and Bandwidth-Optimized Multi-Scalar Multiplication Architecture. In *Proceedings of the 61st ACM/IEEE Design Automation Conference* (San Francisco, CA, USA) *(DAC '24)*. Association for Computing Machinery, New York, NY, USA, Article 94, 6 pages. https://doi.org/10.1145/3649329.3658259

[34] Tao Lu, Yuxun Chen, Zonghui Wang, Xiaohang Wang, Wenzhi Chen, and Jiaheng Zhang. 2024. BatchZK: A Fully Pipelined GPU-Accelerated System for Batch Generation of Zero-Knowledge Proofs. Cryptology ePrint Archive, Paper 2024/1862. https://eprint.iacr.org/2024/1862

[35] Tao Lu, Chengkun Wei, Ruijing Yu, Chaochao Chen, Wenjing Fang, Lei Wang, Zeke Wang, and Wenzhi Chen. 2022. cuZK: Accelerating Zero-Knowledge Proof with A Faster Parallel Multi-Scalar Multiplication Algorithm on GPUs. Cryptology ePrint Archive, Paper 2022/1321. https://eprint.iacr.org/2022/1321 https://eprint.iacr.org/2022/1321.

[36] Weiliang Ma, Qian Xiong, Xuanhua Shi, Xiaosong Ma, Hai Jin, Haozhao Kuang, Mingyu Gao, Ye Zhang, Haichen Shen, and Weifang Hu. 2023. 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* (Vancouver, BC, Canada) *(ASPLOS 2023)*. Association for Computing Machinery, New York, NY, USA, 340–353. https://doi.org/10.1145/3575693.3575711

[37] Jianqiao Mo, Jayanth Gopinath, and Brandon Reagen. 2023. HAAC: A Hardware-Software Co-Design to Accelerate Garbled Circuits. In *Proceedings of the 50th Annual International Symposium on Computer Architecture* (Orlando, FL, USA) *(ISCA '23)*. Association for Computing Machinery, New York, NY, USA, Article 10, 13 pages. https://doi.org/10.1145/3579371.3589045

[38] Peter L. Montgomery. 1987. Speeding the Pollard and elliptic curve methods of factorization. *Math. Comp.* 48, 177 (1987), 243–264. https://doi.org/10.1090/S0025-5718-1987-0866113-7

[39] Timothy Prickett Morgan. 2019. AMD Doubles Down – And Up – With Rome Epyc Server Chips. https://www.nextplatform.com/2019/08/07/amd-doubles-down-and-up-with-rome-epyc-server-chips/

[40] Timothy Prickett Morgan. 2019. *A Deep Dive Into AMD's Rome EPYC Architecture.* https://www.nextplatform.com/2019/08/15/a-deep-dive-into-amds-rome-epyc-architecture/ Accessed: [Insert date of access here].

[41] Negar Neda, Austin Ebel, Benedict Reynwar, and Brandon Reagen. 2024. CiFlow: Dataflow Analysis and Optimization of Key Switching for Homomorphic Encryption. In *2024 IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS)*. 61–72. https://doi.org/10.1109/ISPASS61541.2024.00016

[42] Jason Nelson. 2022. Zcash Nixes Trusted Setup, Enters New Era With Major Network Update. https://decrypt.co/101762/zcash-nixes-trusted-setup-enters-new-era-with-major-network-update

[43] OpenCores. 2013. SHA-3 IP Core. https://opencores.org/projects/sha3. Accessed: November 16, 2024.

[44] Nicholas Pippenger. 1976. On the evaluation of powers and related problems. In *17th Annual Symposium on Foundations of Computer Science (sfcs 1976)*. 258–263. https://doi.org/10.1109/SFCS.1976.21

[45] Thomas Pornin. 2020. Optimized Binary GCD for Modular Inversion. 2020/972 (2020). https://eprint.iacr.org/2020/972 Publication info: Preprint. MINOR revision..

[46] Pengcheng Qiu, Guiming Wu, Tingqiang Chu, Changzheng Wei, Runzhou Luo, Ying Yan, Wei Wang, and Hui Zhang. 2024. MSMAC: Accelerating Multi-Scalar Multiplication for Zero-Knowledge Proof. In *Proceedings of the 61st ACM/IEEE Design Automation Conference* (San Francisco, CA, USA) *(DAC '24)*. Association for Computing Machinery, New York, NY, USA, Article 66, 6 pages. https://doi.org/10.1145/3649329.3655672

[47] Nikola Samardzic, Axel Feldmann, Aleksandar Krastev, Srinivas Devadas, Ronald Dreslinski, Christopher Peikert, and Daniel Sanchez. 2021. F1: A Fast and Programmable Accelerator for Fully Homomorphic Encryption. In *MICRO-54: 54th Annual IEEE/ACM International Symposium on Microarchitecture* (Virtual Event, Greece) *(MICRO '21)*. Association for Computing Machinery, New York, NY, USA, 238–252. https://doi.org/10.1145/3466752.3480070

[48] Nikola Samardzic, Axel Feldmann, Aleksandar Krastev, Nathan Manohar, Nicholas Genise, Srinivas Devadas, Karim Eldefrawy, Chris Peikert, and Daniel

Sanchez. 2022. CraterLake: A Hardware Accelerator for Efficient Unbounded Computation on Encrypted Data. In *Proceedings of the 49th Annual International Symposium on Computer Architecture* (New York, New York) *(ISCA '22)*. Association for Computing Machinery, New York, NY, USA, 173–187. https://doi.org/10.1145/3470496.3527393

[49] Nikola Samardzic, Simon Langowski, Srinivas Devadas, and Daniel Sanchez. 2024. Accelerating Zero-Knowledge Proofs Through Hardware-Algorithm Co-Design. Preprint. https://people.csail.mit.edu/devadas/pubs/micro24_nocap.pdf.

[50] Scott Schlachter and Brian Drake. 2019. Introducing Micron® DDR5 SDRAM: More than a generational update. *XP055844818* 31 (2019), 6.

[51] Srinath Setty. 2020. Spartan: Efficient and General-Purpose zkSNARKs Without Trusted Setup. In *Advances in Cryptology – CRYPTO 2020*, Daniele Micciancio and Thomas Ristenpart (Eds.). Springer International Publishing, Cham, 704–737.

[52] Srinath T. V. Setty, Justin Thaler, and Riad S. Wahby. 2024. Unlocking the Lookup Singularity with Lasso. In *Advances in Cryptology - EUROCRYPT 2024 - 43rd Annual International Conference on the Theory and Applications of Cryptographic Techniques, Zurich, Switzerland, May 26-30, 2024, Proceedings, Part VI (Lecture Notes in Computer Science, Vol. 14656)*, Marc Joye and Gregor Leander (Eds.). Springer, 180–209. https://doi.org/10.1007/978-3-031-58751-1_7

[53] Nojan Sheybani, Tengkai Gong, Anees Ahmed, Nges Brian Njungle, Michel Kinsy, and Farinaz Koushanfar. 2025. Gotta Hash 'Em All! Speeding Up Hash Functions for Zero-Knowledge Proof Applications. arXiv:2501.18780 [cs.CR] https://arxiv.org/abs/2501.18780

[54] Deepraj Soni, Negar Neda, Naifeng Zhang, Benedict Reynwar, Homer Gamil, Benjamin Heyman, Mohammed Nabeel, Ahmad Al Badawi, Yuriy Polyakov, Kellie Canida, Massoud Pedram, Michail Maniatakos, David Bruce Cousins, Franz Franchetti, Matthew French, Andrew Schmidt, and Brandon Reagen. 2023. RPU: The Ring Processing Unit. In *2023 IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS)*. 272–282. https://doi.org/10.1109/ISPASS57527.2023.00034

[55] powerup tech. 2024. AMD EPYC 7502 Specs. https://www.techpowerup.com/cpu-specs/epyc-7502.c2250

[56] Justin Thaler. 2013. Time-Optimal Interactive Proofs for Circuit Evaluation. Cryptology ePrint Archive, Paper 2013/351. https://eprint.iacr.org/2013/351

[57] Justin Thaler. 2022. *Proofs, Arguments, and Zero-Knowledge*. https://people.cs.georgetown.edu/jthaler/ProofsArgsAndZK.pdf

[58] Paul Valiant. 2008. Incrementally Verifiable Computation or Proofs of Knowledge Imply Time/Space Efficiency. In *Theory of Cryptography, Fifth Theory of Cryptography Conference, TCC 2008, New York, USA, March 19-21, 2008 (Lecture Notes in Computer Science, Vol. 4948)*, Ran Canetti (Ed.). Springer, 1–18. https://doi.org/10.1007/978-3-540-78524-8_1

[59] Cheng Wang and Mingyu Gao. 2023. SAM: A Scalable Accelerator for Number Theoretic Transform Using Multi-Dimensional Decomposition. In *2023 IEEE/ACM International Conference on Computer Aided Design (ICCAD)*. 1–9. https://doi.org/10.1109/ICCAD57390.2023.10323744

[60] Cheng Wang and Mingyu Gao. 2025. 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* (Rotterdam, Netherlands) *(ASPLOS '25)*. Association for Computing Machinery, New York, NY, USA, 1101–1117. https://doi.org/10.1145/3669940.3707228

[61] Tiancheng Xie, Yupeng Zhang, and Dawn Song. 2022. 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 (Lecture Notes in Computer Science, Vol. 13510)*, Yevgeniy Dodis and Thomas Shrimpton (Eds.). Springer, 299–328. https://doi.org/10.1007/978-3-031-15985-5_11

[62] Alex Luoyuan Xiong, Binyi Chen, Zhenfei Zhang, Benedikt Bünz, Ben Fisch, Fernando Krell, and Philippe Camacho. 2023. VeriZexe: Decentralized Private Computation with Universal Setup. In *32nd USENIX Security Symposium, USENIX Security 2023, Anaheim, CA, USA, August 9-11, 2023*, Joseph A. Calandrino and Carmela Troncoso (Eds.). USENIX Association, 4445–4462. https://www.usenix.org/conference/usenixsecurity23/presentation/xiong

[63] Zhengbang Yang, Lutan Zhao, Peinan Li, Han Liu, Kai Li, Boyan Zhao, Dan Meng, and Rui Hou. 2025. LegoZK: a Dynamically Reconfigurable Accelerator for Zero Knowledge Proof. (2025).

[64] Ye Zhang, Shuo Wang, Xian Zhang, Jiangbin Dong, Xingzhong Mao, Fan Long, Cong Wang, Dong Zhou, Mingyu Gao, , and Guangyu Sun. 2021. PipeZK: Accelerating Zero-Knowledge Proof with a Pipelined Architecture. In *2021 ACM/IEEE 48th Annual International Symposium on Computer Architecture (ISCA)*.

[65] Zhenfei Zhang, Binyi Chen, Benedikt Bünz, and Alex Xiong. [n. d.]. *Hyperplonk Implementation*.

[66] Hao Zhou, Changxu Liu, Lan Yang, Li Shang, and Fan Yang. 2024. ReZK: A Highly Reconfigurable Accelerator for Zero-Knowledge Proof. *IEEE Transactions on Circuits and Systems I: Regular Papers* (2024).

[67] Xudong Zhu, Haoqi He, Zhengbang Yang, Yi Deng, Lutan Zhao, and Rui Hou. 2024. Elastic MSM: A Fast, Elastic and Modular Preprocessing Technique for Multi-Scalar Multiplication Algorithm on GPUs. Cryptology ePrint Archive, Paper 2024/057. https://eprint.iacr.org/2024/057