UniZK: Accelerating Zero-Knowledge Proof with Unified Hardware and Flexible Kernel Mapping

Cheng Wang*
wangcheng@stu.xjtu.edu.cn
Xi'an Jiaotong University
Xi'an, Shaanxi, China
Institute for Interdisciplinary Information Core Technology
Xi'an, Shaanxi, China

Mingyu Gao
gaomy@tsinghua.edu.cn
Tsinghua University
Beijing, China
Shanghai Qi Zhi Institute
Shanghai, China
Institute for Interdisciplinary Information Core Technology
Xi'an, Shaanxi, China

Abstract

Zero-knowledge proof (ZKP) is an important cryptographic tool that sees wide applications in real-world scenarios where privacy must be protected, including privacy-preserving blockchains and zero-knowledge machine learning. Existing ZKP acceleration approaches using GPUs, FPGAs, and ASICs focus only on classic protocols that rely on expensive elliptic curve arithmetics. Emerging ZKP protocols based on hash functions can greatly reduce the algorithmic complexity, but they also introduce much more diverse computation kernels that cannot be efficiently handled by a single accelerator chip if dedicated units for each kernel are used. Our approach is to leverage a unified hardware architecture that is able to efficiently support the common primitives in ZKP, and then use smart mapping strategies to flexibly map various kernels to such hardware while ensuring high resource utilization. We design UniZK as such a ZKP accelerator, with a systolic-array-based hardware architecture enhanced with extra local links and a new vector processing mode. We propose novel mapping strategies to support diverse kernels including number theoretic transforms, hash functions, and general polynomial computations. UniZK provides 97× and 46× speedups on average compared to the CPU and GPU implementations of the same protocols, and is also 840× faster than previous ZKP accelerators using different protocols.

CCS Concepts: • **Hardware → Hardware accelerators**; • **Theory of computation → Cryptographic protocols**.

Keywords: domain-specific acceleration, zero-knowledge proof, mapping

*Work done during the internship at the Institute for Interdisciplinary Information Core Technology, Xi'an.

ASPLOS '25, March 30–April 3, 2025, Rotterdam, Netherlands
© 2025 Copyright held by the owner/author(s).
ACM ISBN 979-8-4007-0698-1/25/03
https://doi.org/10.1145/3669940.3707228

ACM Reference Format:
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 (ASPLOS '25), March 30–April 3, 2025, Rotterdam, Netherlands.* ACM, New York, NY, USA, 17 pages. https://doi.org/10.1145/3669940.3707228

1 Introduction

Given a function $F(x, w)$ and a public input x, zero-knowledge proof (ZKP) protocols enable a prover to convince a verifier that she possesses a secret input w (a.k.a., witness) that satisfies $F(x, w) = 0$, without revealing any information about the value of w. The capability of ZKP to validate statements without revealing any potentially sensitive information is particularly important in a variety of fields, including blockchains, financial transactions, authentication protocols, electronic voting systems, and zero-knowledge machine learning (ZKML). Consequently, ZKP has recently become a powerful cryptographic tool with numerous applications when it is critical to protect privacy.

The protocol of ZKP involves two parties, the prover who generates a proof, and the verifier who verifies the proof. Because one generated proof may be potentially verified by many individual verifiers, most ZKP protocols are designed to have very fast and low-cost verification processes, but the proof generation is usually complicated and takes substantial time when proving real-world statements. For example, generating a consensus proof of the finalized header for the Ethereum blockchain using an AWS `r6a.8xlarge` machine takes about 300 seconds [18]. This makes proof generation an interesting target for specialized hardware and software acceleration in the computer architecture and system community [4, 19, 40, 41, 43, 56, 68, 69, 72], with various hardware choices ranging from GPUs, FPGAs, to ASICs.

Nevertheless, almost all existing ZKP acceleration systems focus on the classic protocols based on elliptic curves [27], which only contain two computation kernels in the proof generation process. Modern ZKP protocols [16, 17, 51, 62, 70, 71] now start to use hash functions instead of elliptic-curve operations, and offer many highly desired features including elimination of trusted setup, removal of expensive elliptic curve cryptography, as well as flexible tradeoff between proving time, verification time, and proof size. These new protocols are also being increasingly widely adopted in real-world applications like blockchains [15, 49, 50, 57] and ZKML [66]. However, these hash-based protocols have much more diverse computation kernels, not only including the conventional ones like number theoretic transforms (NTTs) and Merkle trees, but also incorporating emerging hash functions like Poseidon [26] as well as miscellaneous polynomial operations such as additions, multiplications, and partial products on their quotient polynomials. Designing separate and dedicated hardware units for each kernel and putting them on one chip would result in a large chip area and low resource utilization. If new kernels that cannot be supported by the current accelerator are introduced, we have to fall back to the host CPU with back-and-forth data transfers.

In this paper, we aim to design a general ZKP accelerator that is able to accelerate as many kernels as possible involved in modern hash-based protocols such as Plonky2 [51] and Starky [52]. Our key philosophy is to leverage a *unified hardware architecture* not overly customized to any specific kernel, but still able to efficiently support the common primitives of ZKP cryptography like modular arithmetic operations and typical data access patterns. Then, we propose *novel kernel mapping strategies* to flexibly map various computation kernels to such hardware, while ensuring high utilization of hardware resources. Actually, the widely successful neural network accelerators have followed exactly the same paradigm, using relatively general architectures to support fast evolving operators such as convolution, matrix multiplication, and self-attention [10, 23, 32–34, 39].

More specifically, we propose an accelerator named UniZK following the above philosophy. Our hardware architecture contains a global SRAM buffer and multiple tiles of processing element (PE) arrays, where each PE contains several modular add/multiply units and a register file. The inter-PE connection within each array follows the efficient systolic manner [33, 37], but is slightly enhanced with the addition of a few additional local links and the support of a vector processing mode. Our kernel mapping strategies can successfully map a diverse set of kernels. These include (1) variable-length NTTs with several variants like coset and bit-reverse orders, (2) hash functions with irregular computation patterns, e.g., Poseidon, (3) Merkle tree construction, and (4) polynomial additions, multiplications, and various other operations. Most of these kernels exhibit high utilization on both on-chip logic and off-chip memory bandwidth.

When evaluated against highly optimized CPU and GPU baselines using the Plonky2 protocol, UniZK is 97× and 46× faster on average, and up to 147× and 104×, respectively. The individual kernels of NTT, hash function, and polynomial computation see speedups ranging from 92× to 191×. When using the more efficient protocol of Starky-enhanced Plonky2, UniZK is up to 267× faster than the CPU. Compared to the previous accelerator PipeZK [72] for classic elliptic-curve-based protocols, UniZK achieves an 840× speedup with both algorithmic and architectural advantages.

2 Background

In this section, we first review the recent development of zero-knowledge proof (ZKP) techniques (Section 2.1), and introduce the specific protocols, Plonky2 and Starky, as the representatives for our accelerator design (Section 2.2). We then break down their execution time of proof generation, to motivate our architectural innovations (Section 2.3).

2.1 Zero-Knowledge Proof Protocols

ZKP is a fundamental primitive in modern cryptography and has evolved rapidly in recent years. Early implementations of ZKP were mainly based on elliptic curve (EC) cryptography. While they provide strong security, these early ZKP protocols require complex mathematical operations over large integer fields, resulting in significant computational overheads. For example, Groth16 [27] generates succinct proofs that are within hundreds of bytes and can be verified quickly. However, proof generation in Groth16 involves number theoretic transforms and EC-based multi-scalar multiplications on integers as wide as 256 to 768 bits. Both the wide integer arithmetic and the EC-based operations are highly expensive on modern hardware, limiting the performance of these protocols in many real-world applications.

Recent advancements have introduced *hash-based ZKP protocols* [16, 17, 51, 62, 70, 71] that operate over smaller integer fields. Hash functions are faster and consume fewer resources, making them a more cost-efficient alternative to EC operations. These changes greatly reduce the computational cost of proof generation. But on the other hand, these protocols may result in somewhat larger proof sizes and higher verification cost compared to EC-based ones.

While various modern ZKP protocols exhibit different tradeoffs among proof size, proof generation time, and verification time, usually the proof generation phase is always the most time-consuming part and thus the target for most performance optimizations. Proof generation in modern protocols is typically constructed in three generic steps [8, 11]: Arithmetization, Polynomial Interactive Oracle Proof (PIOP), and Polynomial Commitment Scheme (PCS). First, the statement to be proved is converted into an arithmetic circuit (Arithmetization). Then, the prover uses its own secret witness as well as some randomness from the verifier to build a set of polynomial equations (PIOP). Finally, the prover sends the cryptographic commitment of these polynomials to the verifier (PCS). The interactive parts in the above steps, such as the randomness sent from the verifier to the prover, could be all eliminated by the Fiat-Shamir transform [20], thus constructing a non-interactive protocol.

Plonky2 [51] is one of the most popular implementations of hash-based ZKPs. Its three components are Plonkish, Plonk [22], and Fast Reed-Solomon IOP of Proximity (FRI) [5], which together provide numerous beneficial features, including not requiring a trusted setup, high performance due to the use of small algebraic fields and not relying on expensive elliptic curves, as well as support for proof aggregation by recursive proving [7] to reduce storage and communication overheads. Plonky2 can be further enhanced with another protocol called *Starky* [52]. Starky is much less expensive than Plonky2, but it has large proof sizes (several MBs) and does not support zero knowledge. We can use Starky to prove non-recursive statements and then use Plonky2 to compress and aggregate the Starky proofs. This combination yields a quite efficient ZKP scheme. Plonky2 is now widely used in many application domains such as blockchains [15, 49, 50, 57], electronic voting [13], zero-knowledge virtual machines [38], and zero-knowledge machine learning [66]. Consequently, we use it as a representative to design our accelerator.

2.2 Example Protocols: Plonky2 and Starky

Now we describe the algorithm details of Plonky2 and Starky. Figure 1 illustrates the proof generation flow of Plonky2 [51], where a prover aims to prove that she knows a set of private values (x_0, x_1, x_2, x_3) that satisfy $(x_0 + x_1) * (x_2 * x_3) = 99$. We first follow this example to explain the key steps of Plonky2. We omit the Arithmetization part that converts a statement into an arithmetic circuit, because it does not depend on the prover's input secret data, and thus can be done fully offline. At the end, we discuss Starky, focusing on the key differences compared to Plonky2. We mainly describe the high-level algorithm flow of the protocols here, and defer the details of their computation kernels to Section 5.

Plonk for PIOP. The process of converting an arithmetic circuit to a set of polynomial equations involves two types of constraints: gate constraints and copy constraints. Gate constraints represent rules for computing arithmetic gates in the circuit, while copy constraints ensure that gates in the circuit are properly connected through wires. As Figure 1 left shows, the arithmetic circuit of the statement $(x_0 + x_1) * (x_2 * x_3) = 99$ is represented by the matrix Q. Each of its first three rows represents a gate in the circuit, and the fourth row explicitly indicates that the output is 99. The matrix W keeps the secret witness data filled by the prover, where each row is the input and output values of each gate. Each column of Q is the coefficients for linear, quadratic, or constant constraints with respect to the variables in W. W and Q should satisfy the gate constraints at the bottom when combined using element-wise multiplications ("$\cdot$"). For example, for gate $i = 2$, only the quadratic coefficient $q_{M,2}$ is effectual for the product of $w_{a,2} = x_2$ and $w_{b,2} = x_3$, which enforces $x_2 * x_3 = q_{O,2} \cdot w_{c,2} = x_5$, i.e., the bottom right gate in the circuit.

To ensure a correct W, the elements in the same color in W must have the same value, meaning that the output of one gate is connected to the input of another gate. The protocol thus constructs an index matrix id, and a permutation matrix σ with the values in the same color in id being permuted. With the random numbers β and γ from the verifier, (W, id) and (W, σ) are used to form f and g, respectively, where they should be equal as required by the copy constraints in Figure 1 left bottom. This is because the permuted values in id and σ correspond to the same values in W.

It is worth noting that, in the actual implementation, the above computations on scalar values are transformed by interpolating the scalars into polynomials and then computing on the polynomials, which improves parallelization and efficiency. The polynomial computations involve not only element-wise additions and multiplications, but also a small number of number theoretic transforms (NTTs) and their variant coset-NTTs [6] for polynomial multiplications, as well as some accumulations on partial products.

FRI for PCS. FRI is a protocol for verifying the degree of a given polynomial is within a certain bound. The protocol has three main steps as illustrated in Figure 1 right.

Step ① applies the natural-input-natural-output inverse NTTs (iNTT$^{\text{NN}}$) on the polynomials to convert the value representation into the coefficient representation. For security reasons, in step ②, these polynomials must be evaluated by low degree extension (LDE), where the coefficient vectors are expanded to k times of their original length through zero padding. Here k is referred to as the blowup factor, which is at least 8 in Plonky2. Then, the natural-input-bit-reverse-output NTTs (NTT$^{\text{NR}}$) are performed in the LDE domain.

Step ③ is the construction of the Merkle tree, starting from the leaf nodes and moving upwards. The input data to each leaf node are formed by taking values from the same position of all the polynomials and concatenating them. A hash function is applied on them to determine the leaf node value. Then, the Merkle tree is built as a binary tree where the parent node value is the hash result of the concatenated value of the two children. In Plonky2, the Poseidon hash function [26] is used. Finally, the prover sends the root of the Merkle tree as a commitment to the verifier. The verifier may query a random leaf node, for which the prover provides to the verifier the corresponding authentication path from this leaf to the root for verification.

Starky. Starky [52] adopts different Arithmetization and PIOP components from Plonky2. It still uses FRI as its PCS, except that the blowup factor k is set to a different value of 2. In Starky, the computation is represented by an Algebraic Execution Trace (AET), which is essentially a table with each entry describing the state at a specific time step, and adjacent entries adhering to the transition constraints. The protocol also supports input and output constraints to enforce certain

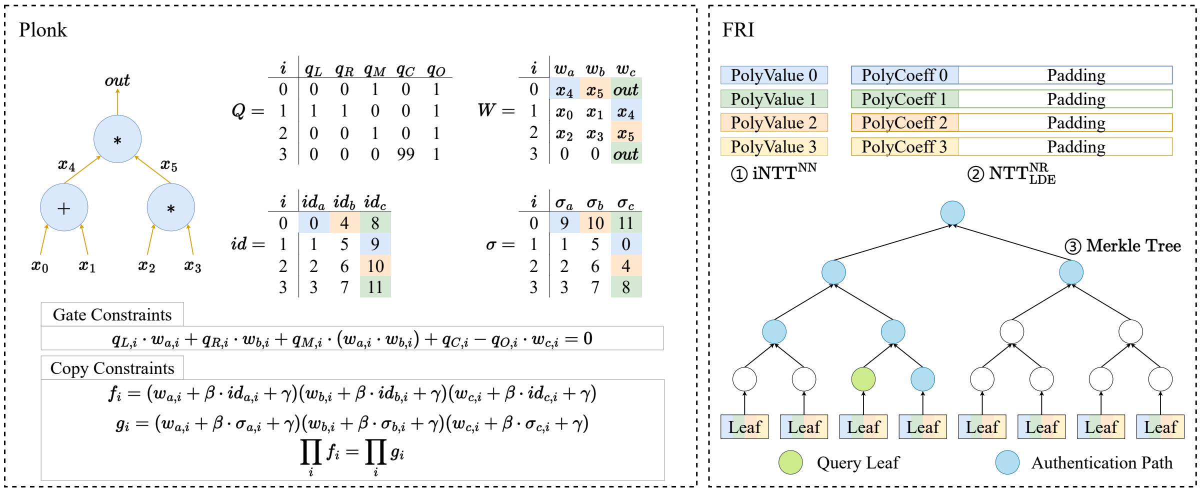

Plonk
out
*
x_4
x_5
+
*
x_0 x_1 x_2 x_3
$Q =$
i q_L q_R q_M q_C q_O
0 0 0 1 0 1
1 1 1 0 0 1
2 0 0 1 0 1
3 0 0 0 99 1
$W =$
i w_a w_b w_c
0 x_4 x_5 out
1 x_0 x_1 x_4
2 x_2 x_3 x_5
3 0 0 out
$id =$
i id_a id_b id_c
0 0 4 8
1 1 5 9
2 2 6 10
3 3 7 11
$\sigma =$
i σ_a σ_b σ_c
0 9 10 11
1 1 5 0
2 2 6 4
3 3 7 8
Gate Constraints
$q_{L,i} \cdot w_{a,i} + q_{R,i} \cdot w_{b,i} + q_{M,i} \cdot (w_{a,i} \cdot w_{b,i}) + q_{C,i} - q_{O,i} \cdot w_{c,i} = 0$
Copy Constraints
$f_i = (w_{a,i} + \beta \cdot id_{a,i} + \gamma)(w_{b,i} + \beta \cdot id_{b,i} + \gamma)(w_{c,i} + \beta \cdot id_{c,i} + \gamma)$
$g_i = (w_{a,i} + \beta \cdot \sigma_{a,i} + \gamma)(w_{b,i} + \beta \cdot \sigma_{b,i} + \gamma)(w_{c,i} + \beta \cdot \sigma_{c,i} + \gamma)$
$\prod_i f_i = \prod_i g_i$
FRI
PolyValue 0
PolyValue 1
PolyValue 2
PolyValue 3
PolyCoeff 0 Padding
PolyCoeff 1 Padding
PolyCoeff 2 Padding
PolyCoeff 3 Padding
① iNTT^{NN}
② NTT$^{NR}_{LDE}$
③ Merkle Tree
Leaf Leaf Leaf Leaf Leaf Leaf Leaf Leaf
Query Leaf
Authentication Path

Figure 1. Steps of Plonk and FRI during proof generation in Plonky2, with an example statement of $(x_0 + x_1) * (x_2 * x_3) = 99$.

$\overrightarrow{x_0}$	$\overrightarrow{x_1}$	$\overrightarrow{x_2}$	$\overrightarrow{x_3}$	$\overrightarrow{x_4}$
0	1	1	2	3
1	1	2	3	5

Transition Constraints

$$P_0(\overrightarrow{x_n}, \overrightarrow{x_{n+1}}) = x_n[1] - x_{n+1}[0]$$
$$P_1(\overrightarrow{x_n}, \overrightarrow{x_{n+1}}) = x_n[0] + x_n[1] - x_{n+1}[1]$$

Input and Output Constraints

$$P_3(\overrightarrow{x_0}) = x_0[0] - 0 \quad P_4(\overrightarrow{x_0}) = x_0[1] - 1$$
$$P_5(\overrightarrow{x_4}) = x_4[1] - 5$$

Figure 2. An example AET for proving the 5th Fibonacci sequence number is 5. The constraints should be equal to 0.

input and output values. Figure 2 shows an example AET for proving the Fibonacci sequence.

Despite the significant differences between Plonky2 and Starky in Arithmetization and PIOP, the underlying computational operations are highly similar, both involving element-wise polynomial computations, polynomial multiplications, etc. In addition, their FRI steps follow the same procedure with only different parameters. Thus, it is feasible to design a single accelerator to support both protocols.

2.3 Performance Breakdown Analysis

As we can see from Section 2.2, the proof generation processes in Plonky2 and Starky contain many intensive computations including polynomial operations during Plonk, NTTs, and Merkle tree construction during FRI, as well as various data layout transformations such as transpose. To further understand the main time consumers in Plonky2, we run several proof generation workloads with a single-threaded CPU implementation (see Section 6 for workload and hardware details). We use a single thread to simplify time breakdown; our final evaluation uses multi-threaded baselines. The measured time includes Plonk and FRI but excludes Arithmetization. Table 1 shows the results. The other hash operations are needed in Fiat-Shamir transforms and proof-of-work computations outside the Merkle tree phase. They also use the same Poseidon hash function.

Table 1. Plonky2 proof generation time breakdown.

	Time (s)	Poly-nomial	NTT	Merkle Tree	Other Hash	Layout Transform
Factorial	580	13.4%	21.8%	62.4%	0.0%	2.4%
Fibonacci	34	12.1%	20.0%	65.8%	0.1%	2.0%
ECDSA	101	24.9%	15.7%	57.2%	0.2%	2.0%
SHA-256	673	11.5%	19.0%	67.0%	0.0%	2.5%
Image Crop	333	11.5%	17.1%	68.8%	0.3%	2.3%
MVM	512	13.7%	15.9%	65.7%	0.1%	4.6%

From the breakdown results we can see that, usually the Merkle tree construction accounts for the majority (about 60%) of the execution time. This is mainly due to the large number of Poseidon hash computations involved. The NTT and various polynomial operations rank the second and third in terms of computation time, together contributing to about 35% of the overall time. The remaining hash operations and the data layout transformations exhibit small portions.

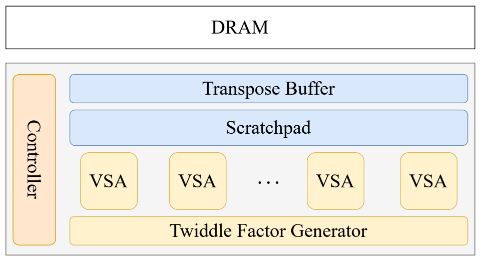
3 Design Philosophy

From the performance analysis in Section 2.3 we see that, modern ZKP protocols usually contain multiple diverse kernels that each contributes a non-negligible portion to the overall execution time. In particular, the polynomial operations, though shown as a single category in Table 1, actually include multiple diverse kernels. This observation puts several new challenges to the hardware acceleration. First, existing ZKP accelerators targeting EC-based ZKPs, e.g., PipeZK [72], extracted two dominant kernels from the protocol, and designed dedicated hardware modules for them separately. When it comes to our case like Plonky2, from Table 1 we can quickly see that only capturing the top-2 kernels (e.g., Merkle tree and NTT) will at most give us less than 7× speedup according to Amdahl's law. Second, because the time portions of different kernels could vary significantly across different proof generation workloads (e.g., 11% to 25% for polynomial operations), having multiple dedicated modules with rigid static resource provisioning would likely face the issue of unbalanced throughput when running different workloads. One kernel may run too long due to insufficient resources, while other kernels finish quickly and leave their modules idle. Third, some of the kernels use specific cryptographic primitives, e.g., Poseidon hash, which requires highly specialized dedicated units that may easily become out-dated if the next version switches to another hash choice. Finally, being able to accelerate all the kernels can also reduce data movement. If some kernels cannot run on the accelerator and are left to the host CPU, the intermediate data must be transferred to the CPU and then back over the slow PCIe connections, dwarfing the performance gains.

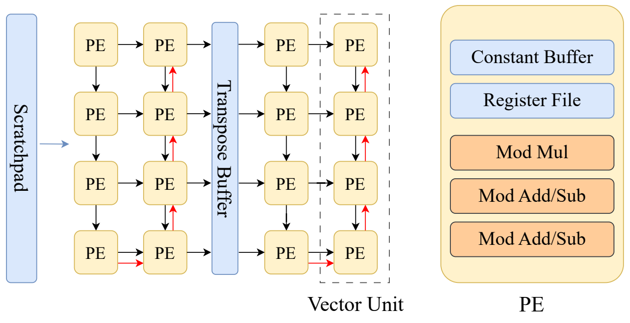
As a result, our goal is to design a general ZKP accelerator that is able to accelerate *as many kernels as possible* involved in modern ZKP protocols such as Plonky2 and Starky, by leveraging a *unified* hardware architecture that can *flexibly map* diverse kernels. Our design philosophy is to *keep the hardware architecture simple and general* (Section 4), not overly customized to any specific kernels, while still offering sufficient efficiency by capturing the common characteristics of ZKP algorithms, such as the modular arithmetic computations and the common data access patterns and layout transformations. On the other hand, we rely on *novel kernel mapping strategies* (Section 5), in order to realize the diverse kernels efficiently on the general hardware and ensure high computational utilization. This approach is akin to the philosophy of modern neural network accelerators that use the same hardware for various tensor operations like matrix multiplication, convolution, and self-attention [10, 21, 23, 32–34, 39, 48, 63]. In the next sections, we respectively introduce the hardware architecture and the kernel mapping strategies.

4 Hardware Architecture

Figure 3a illustrates the overall architecture of UniZK. The central compute engine is the multiple homogeneous vector-systolic arrays [44] (VSAs), each of which is an array of processing elements (PEs) interconnected with customized networks. Recall that UniZK aims to provide a unified architecture for diverse ZKP kernels. Therefore the VSAs are homogeneous and not specialized to any kernels. In addition, UniZK utilizes a double-buffered scratchpad between the VSAs and the off-chip DRAM, hiding memory access



(a) Overall architecture.



(b) Vector-systolic array (VSA) and processing element (PE).

Figure 3. Hardware architecture of UniZK.

latencies and maximizing on-chip data reuse. We also incorporate a global transpose buffer to perform data transpose operations on-chip, implicitly when fetching data from memory [59, 72]. Finally, a twiddle factor generator is implemented on-chip, which consists of several modular multipliers and a set of buffers to support on-the-fly twiddle factor generation during NTT computations [36, 65].

The structures of the VSA and the PE are further shown in Figure 3b. The VSA resembles a classic systolic array [37] widely used in neural network accelerators [10, 31–33, 46]. The input and output data are transferred to and from the VSA from its boundaries, and the data communication within the VSA is restricted to neighbor PEs. UniZK instantiates each VSA with 12 × 12 PEs. This size is decided to match the ZKP kernels, specifically the Poseidon hash, as explained in Section 5.2. Each PE is connected to its right and bottom neighbors in the standard systolic way, plus several additional links described shortly. Standard matrix multiplications can thus be mapped to the VSA. For example, a constant matrix can be loaded to the array and multiplied with the input matrix, in a weight stationary manner [25, 58].

Each PE in the VSA contains a 64-bit modular multiplier, two 64-bit modular adders/subtractors, and a register file of 64 × 64 bits. All operations in UniZK are performed on 64-bit data elements in the Goldilocks field [51]. The simplified Goldilocks field operations reduce the complexity of the hardware modular adders and modular multipliers. In Plonky2, there are also a small number of computations based on the extension field [12] for soundness [64]. Each extension field element consists of D elements from the base Goldilocks field, and thus requires $64D$ bits of storage. Usually a quadratic extension with $D = 2$ is employed. Nevertheless, the arithmetic operations can still execute on the base field, treating each 64 bits as a limb, albeit with more steps.

To efficiently support the diverse kernels in modern ZKP protocols, UniZK further enhances the VSA with an additional *vector mode* and a few *additional links* between PEs. In the vector mode, each column of the VSA can be considered as an individual vector unit. The entire chip thus contains many independent vector units for parallel data processing. The vector mode is particularly suitable for the large amount of polynomial operations in ZKP. Interestingly, the systolic links between neighbor PEs still provide some data reuse, such as directly forwarding a vector from one unit to another, and accumulating vector elements to a scalar. We will show concrete examples in Section 5 about how to make use of these links to efficiently map various ZKP kernels.

Nevertheless, for more complex and irregular kernels such as the Poseidon hash, the rigid systolic links are insufficient. Hence we add a limited amount of new links to the VSA. Specifically, we add reverse links that go bottom up in certain columns in the array, shown as the red arrows in Figure 3b. Their specific usage will be explained in Section 5. These new links still connect only neighbor PEs and avoid long global wires, in order to minimize the impact on the placement and routing complexity and thus the overall area cost.

5 Kernel Mapping

In this section, we describe how UniZK maps each kernel in the ZKP protocol (Sections 5.1 to 5.4) onto the general hardware architecture in the last section. UniZK follows a static scheduling approach. Given the specific arithmetic circuit to prove, the kernels to execute are all known before execution. Thus we employ a compiler to map each kernel (Section 5.5), following the schemes described below.

5.1 NTT

NTT is a widely used primitive in cryptography, and has been extensively studied in previous accelerators for fully homomorphic encryption and EC-based ZKP algorithms [35, 36, 59, 60, 65, 72]. An NTT applied to the N coefficients of a polynomial consists of $\log N$ stages, where the ith stage conducts $N/2$ butterfly operations between all pairs of coefficients with a strided distance of $N/2^i$. Each butterfly operation also needs a constant twiddle factor that is a certain power of the Nth root of unity in the finite integer field. This pattern is similar to the well-known fast Fourier transform (FFT), but conducted on the integer field rather than the complex numbers. The result of an NTT is also a size-N vector representing the N values of the polynomial evaluated at the roots of unity. It can be converted back to the coefficients using an inverse NTT (iNTT), which follows the same computation pattern but uses different twiddle factors.

Nevertheless, the NTT operations in our ZKP protocols pose several new challenges. First, the NTT size, i.e., the polynomial length, depends on the size of the arithmetic circuit to prove. Also, the LDE step in FRI would change the polynomial length. Thus we must efficiently support variable-length NTTs. Second, instead of using specialized datapaths in previous NTT designs, we need to map the irregular NTT dataflow onto our general hardware architecture. This is challenging because we need to realize the variable-strided butterfly operations only with the limited neighbor-connected links in our PE array. Third, NTTs could happen at various phases during the protocol, in which the input and output polynomials could have varying layouts in memory, such as polynomial-major (i.e., all elements in a polynomial are stored continuously) and index-major (i.e., the elements in the same position of all polynomials are stored continuously). Our NTT mapping should support efficient memory accesses to both layouts. Fourth, several variants of NTTs, such as coset-(i)NTTs and (i)NTTs with both natural and bit-reverse data orders (NN, NR, etc.), are needed, and should be handled seamlessly in the hardware.

Mapping variable-length NTTs to unified hardware. To efficiently support variable-length NTTs with fixed hardware, we adopt the multi-dimensional NTT decomposition approach in SAM [65], which decomposes an NTT kernel of *variable* size N into multiple smaller NTTs of *fixed* size n that matches the hardware size. The decomposed fixed-length small NTTs can then fully utilize the hardware. Element-wise twiddle factor multiplications are also needed between the groups of small NTTs when switching between the decomposed dimensions. We next describe how to map these two sub-kernels, i.e., size-n NTTs and inter-dimension twiddle multiplications, onto our VSA structure.

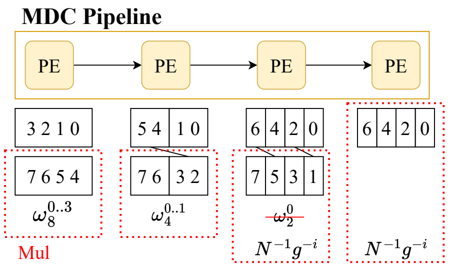
First, for the small fixed-size NTTs, we follow the MDC (multi-path delay commutator) pipelined structure [24]. Figure 4a shows how to map a DIF NTT to a linear sequence of PEs using the MDC pipeline. Each pipeline stage has a throughput of 2 elements per cycle. We use the modular multiplier/adder/subtractor in one PE to implement a stage. The corresponding twiddle factors are stored in the register file. Each stage also needs to shuffle the elements to realize the desired strides for butterfly operations. For example, in the first stage, we pair 0 and 4, 1 and 5, ..., for a stride of 4. Their outputs are buffered, and element 0 needs to wait for element 2 for the next stage. We use the register file of each PE to realize such data buffering. Specifically, the results of 0, 1 in the first stage are buffered locally, and sent to the next stage along with the results of 2, 3 generated later. The required register capacity is bound by the fixed NTT size n. Although such data buffering delays the processing of the next stages, the overall throughput does not decrease, and the pipeline delays can be tolerated when there are sufficient data waiting for NTT processing. Note that UniZK supports both DIT and DIF dataflows to support different NTT variants. Compared with DIF, DIT just has reversed element orders and different twiddle factors across all of the stages. We just need to change the corresponding PE register file contents and how they are accessed.

Second, the inter-dimension twiddle multiplications are simple element-wise operations, so they can be easily mapped to a PE and chained with the above NTT pipeline either at the beginning or at the end. The twiddle factors needed in this phase are generated on-the-fly [36, 65] using the twiddle factor generator in Figure 3a rather than stored on-chip. However, if we want to maintain the same throughput of 2 elements per cycle, we now need two multipliers, one per each element, instead of one multiplier per each pair in the butterfly operations. Fortunately, we notice one of the NTT stages (e.g., the last stage of DIF in Figure 4a) always multiplies with a twiddle factor of $1 = \omega_n^0$. Thus we reuse this PE, and only one extra PE is added to the pipeline. In Figure 4a, the last two PEs multiply with $N^{-1}g^{-i}$, which is the twiddle factor in the last round of a coset-iNTT (more details below).

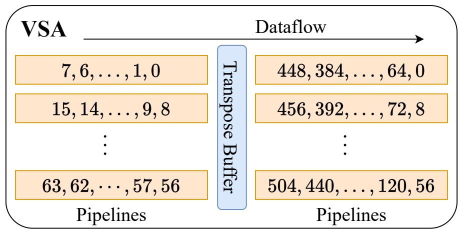
In summary, we map a size-n NTT to a sequence of $\log n+1$ PEs in a pipelined manner. Recall that each VSA row/column has 12 PEs. Using the entire row/column as a single pipeline would set $n = 2^{11}$, requiring too much register space in each PE for data buffering and twiddle storage. We thus split a row of PEs into two pipelines with 6 PEs each for $n = 2^5$. The register usage is now bound to 2^5 64-bit elements per PE. With such a split, we can further pipeline the NTTs along two decomposed dimensions on the two half-arrays, with a transpose buffer in between, as in Figure 4b. This reduces the bandwidth requirement on the global scratchpad by 2×.

Data layouts. Our NTT pipeline design described above accepts/generates data elements consecutively from/to memory if data are in the polynomial-major layout. To also support the index-major layout, we conduct batched NTTs and transpose the data layout using the global transpose buffer. Specifically, we process b polynomials in parallel as a batch, where the transpose buffer can store $b \times b$ elements in its capacity. When data are in index-major, we fetch b consecutive elements each time from memory, which are the elements at the same position of the polynomials in the batch. These elements are transposed in the buffer to polynomial-major, and then fed into the VSAs. Writing data back in the index-major order is performed similarly. In practice, we use $b = 16$, so the memory accesses are sufficiently consecutive while the transpose buffer capacity is still acceptable.

NTT variants. Recall that we need to support both NTT and iNTT, as well as their coset variants [6]. Also the input and output data orders could be either natural or bit-reverse (NN, NR, etc.). For the variants that require modular multiplying with a constant *after* the standard NTT, e.g., N^{-1} for iNTT and $N^{-1}g^{-i}$ for coset-iNTT, we use the PEs reserved



(a) A size-8 DIF NTT mapped to the MDC pipeline. Numbers indicate element indices. The last two stages conduct inter-dimension twiddle multiplications.



(b) Mapping NTT onto VSA, with data transpose between two pipelines.

Figure 4. NTT mapping strategies. We show the example of using an 8 × 8 PE array for a size-512 coset-iNTT, which is decomposed into 3D size-8 DIF NTTs.

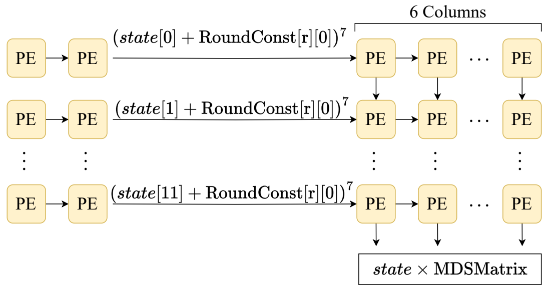
for inter-dimension twiddle multiplications in the last round, which are idle otherwise. This is because for k dimensions of decomposed NTTs, inter-dimension twiddle multiplications are only performed *between* dimensions for $k - 1$ times. In these situations, the DIF dataflow is used, so the pipeline begins with the NTT and ends with inter-dimension twiddle multiplications. Similarly, the constant multiplications *before* the standard NTT, e.g., g^i for coset-NTT, reuse the idle PEs in the DIT pipeline of the first round.

To efficiently perform NTT$^{\text{NR}}$, the output elements should be written back to memory in the bit-reverse order, while still keeping sufficient continuity in these memory accesses. We leverage the multi-dimensional NTT decomposition inherently supported by our hardware to naturally realize this requirement. Using the example in Figure 4b, a size-512 NTT is decomposed into 8 × 8 × 8. The input side accesses data consecutively along the innermost (first) dimension (of stride 1), while the output side finishes the outermost (last) dimension (of stride 64). From the element index perspective, the first/last dimensions correspond to the lowest/highest 3 bits in the index; e.g., $0, 64, \ldots, 384, 448$ have their highest 3 bits as $000_2, 001_2, \ldots, 110_2, 111_2$, and their rest lower bits are all 0. When they are bit-reversed, the highest bits become the lowest ones, so these indices become 0, 4, 2, 6, 1, 5, 3, 7. After a local shuffle among this short list of elements in the on-chip global scratchpad buffer, off-chip memory access continuity could be realized.

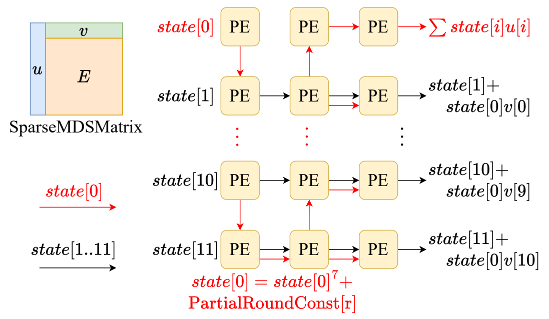
5.2 Poseidon Hash

Algorithm 1: Poseidon Permutation in Plonky2

```
Input/output: $state[12]$
1  function FullRound($state, r$):
2      for $i \leftarrow 0$ to 11 do
3          $state[i] \leftarrow state[i] + \text{RoundConst}[r][i]$;
4          $state[i] \leftarrow state[i]^7$;
5      $state \leftarrow state \times \text{MDSMatrix}$;
6      return $state$;
7  function PartialRound($state, r$):
8      $state[0] \leftarrow state[0]^7$;
9      $state[0] \leftarrow state[0] + \text{PartialRoundConst}[r]$;
10     $state \leftarrow state \times \text{SparseMDSMatrix}$;
11     return $state$;
12 function PrePartialRound($state$):
13     $state \leftarrow state + \text{PrePartialRoundConst}$;
14     $state \leftarrow state \times \text{PreMDSMatrix}$;
15     return $state$;
16 for $r \leftarrow 0$ to 3 do $state \leftarrow$ FullRound($state, r$) ;
17 $state \leftarrow$ PrePartialRound($state$)
18 for $r \leftarrow 0$ to 21 do $state \leftarrow$ PartialRound($state, r$) ;
19 for $r \leftarrow 4$ to 7 do $state \leftarrow$ FullRound($state, r$) ;
```


6 Columns
PE
$(state[0] + \text{RoundConst}[r][0])^7$
$(state[1] + \text{RoundConst}[r][0])^7$
$(state[11] + \text{RoundConst}[r][0])^7$
$state \times \text{MDSMatrix}$

(a) Full round of Poseidon hash, using 12×8 PEs.



(b) Partial round of Poseidon hash, using 12×3 PEs.

Figure 5. Poseidon hash mapping strategies.

In Plonky2 and Starky, the Poseidon hash function is used in many places, such as Merkle tree construction, generating verifier randomness with Fiat-Shamir transforms, and proof-of-work computations. As shown in Algorithm 1, this hash function processes 12 64-bit Goldilocks elements as the *state*, and uses x^7 as the S-box. It consists of 8 full rounds and 22 partial rounds to ensure 128-bit security.

Despite of the various irregular processing rounds in Poseidon hash, we are able to map it onto our general hardware structure. We design three schemes for FullRound, PartialRound, and PrePartialRound, respectively. The key is to efficiently use only the limited neighbor PE links in the VSA.

Full round. The first step in a full round is to add a constant value to each element in *state* and raise to the 7th power (Lines 2 to 4 in Algorithm 1). We use a row of 4 PEs to compute this step, where data are transmitted horizontally. Then, the *state* vector is multiplied with a matrix MDSMatrix (Line 5). When there are many individual Poseidon hashes to compute, this becomes a matrix multiplication, and is naturally mapped to a systolic array of 12×12. In total the full round needs 12 rows $\times$ 16 columns. We fold the computation by performing two consecutive operations on one PE, and map it to a 12×8 region of the VSA, as in Figure 5a.

Pre-processing of partial round. Similar to the full round, the pre-processing of a partial round consists of adding a constant vector to *state* and performing a matrix multiplication. Since each PE in the VSA has a modular adder besides the multiplier, we can merge the constant addition to the first column of the matrix multiplication, and the whole round is mapped to the 12×12 systolic array, without folding.

Partial round. The partial round is the most complex one. First, we need to operate on a single element $state[0]$ (Lines 8 to 9 in Algorithm 1). Then, the multiplication is with a sparse matrix (Line 10), with non-zero values only in the first row, the first column, and the diagonal, i.e., SparseMDSMatrix is decomposed into u, v, and E as in Figure 5b.

We first use the first column of PEs to perform the scalar operations on $state[0]$ in a pipelined way, with data flowing from top to bottom vertically. Then, we use the newly added reverse links of the second PE column to distribute the result to all rows. At the same time, we transmit each element of *state* to each of the 12 rows, and perform the dot product between the two vectors *state* and u in the second PE column. The result of this dot product, as a scalar that is added to the result of $state[0]$, is accumulated vertically from bottom to top also using the reversed links, and received at the top PE. We use the third column to compute the scalar-vector multiply-add, i.e., $state[0] \times v + state$, where the scalar $state[0]$ has been distributed to all rows as described above. Register file space is used to buffer data and match their arrival timing at each PE throughout the process. The final result comes out at the right boundary. In summary, we use 12×3 PEs to compute one partial round, and the entire VSA can process four consecutive partial rounds together. The total latency of four partial rounds is 145 cycles, but with systolic pipelining, the array can accept one 12-element *state* every cycle.

5.3 Merkle Tree

A Merkle tree [45] is usually a binary tree. The data blocks are assigned to the leaf nodes, each of which stores the result of applying the specific hash function to its own block. Then, each parent node in the tree stores the hash result of the concatenated content of both its child nodes. In this way, the root hash becomes a summary of all the data blocks. Updating or verifying any single block only involves computations along the path from this block to the root.

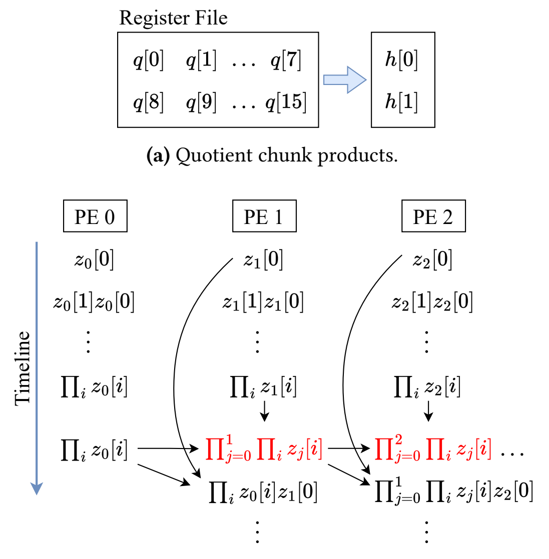
To construct the Merkle tree, each time we load a subtree whose size fits in the scratchpad, and fully process the subtree on-chip. The data of the lower tree levels are read, and our hardware computes the data of the higher levels in the bottom-up order. The hash computations at the same tree level are independent, and performed either in parallel on different VSAs, or in a pipelined manner to one VSA (Section 5.2). The memory layout of the Merkle tree nodes follows the level order, which ensures long sequential memory accesses as well as efficient data caching on-chip when going from one level to the next.

Since the length of the leaf nodes (e.g., 135) may be longer than the length of the Poseidon *state*, Plonky2 uses the absorb method [51]. At the leaf level, we pop the first 8 elements of the leaf and use them as $state[0:8]$, one at a time, until the leaf is used up. For the other nodes, a Poseidon *state* is created by combining 4 elements from each of its left and right children, and padding with 4 zeros to get 12 elements.

5.4 Polynomial Operations

With NTTs, both polynomial additions and multiplications can be transformed into element-wise vector additions and multiplications. These element-wise operations are naturally supported by the vector mode of our VSAs. The operand vectors are fetched from the global scratchpad to the PE register files, and maximally reused inside the PE. Given the multiple functional units (modular adder/subtractor/multiplier) in one PE, we also support chained operations to reduce register access pressure [60].

Element-wise operations. Element-wise vector operations are known for their low computational intensity, and can be easily bound by off-chip memory accesses. We apply several techniques to improve their on-chip data reuse. First, we use the standard LRU cache replacement policy as our basic approach. Second, we adopt *vector tiling*, which splits each vector into many tiles, and fully processes the same tiles of many operand vectors that fit on-chip, before moving to the next tile. Our compiler statically analyzes the vector computation graph and determines the number of vectors to buffer as well as their proper tile sizes. Note that compared to the batch parallelization in the CPU implementation of Plonky2 [51], our tiling is more aggressive and can use much larger batch sizes. Finally, in addition to the LRU policy, our compiler also incorporates hand-crafted replacement policies for certain critical code regions to further maximize data reuse. For example, the evaluation of gate constraints in Plonky2 processes the polynomials of all gates in the circuits on the same wire data. To better reuse the wire data, we prioritize them on-chip and try to replace other data.



(a) Quotient chunk products.

(b) Partial products.

Figure 6. Mapping strategies for partial products of quotient polynomial chunks.

Partial products. Besides element-wise operations, there is another critical vector computation during the verification of copy constraints in the Plonk phase. Specifically, the polynomials $f(x)$ and $g(x)$ in the copy constraints (Figure 1) are divided to obtain a quotient polynomial $q(x)$. Then we use the quotient chunk product PP to reduce the degree of the quotient polynomial. Specifically,

$$h[i] = \prod_{j=8i}^{8i+7} q[j] = \prod_{j=8i}^{8i+7} f[j]/g[j] \tag{1}$$

$$PP[i] = \prod_{j=0}^{i} h[j] = PP[i-1] \times h[i] \tag{2}$$

where h is the product of each 8-element chunk of $q(x)$, and the partial products PP are obtained by accumulating these quotient chunk products. While Equation (1) can be easily processed in parallel, Equation (2) exhibits a long sequential dependency chain between $PP[i]$ and $PP[i-1]$.

To efficiently map the above computations, we first make each PE compute 16 quotient polynomials $q[i]$ and accumulate them into 2 chunks $h[i]$, as in Figure 6a. This is constrained by the available register file capacity. Then we reorganize $h[i]$ into groups through the global scratchpad, where the kth group z_k contains $n = 32$ chunks of $z_k[i] = h[kn + i]$, $i = 0, 1, \ldots, n-1$, and is stored to one PE's register file. The partial product then proceeds in three steps as in Figure 6b. First, each PE locally computes the partial products within its group z_k, i.e., $Z_k[j] = \prod_{i=0}^{j} z_k[i]$. Then, the PEs propagate their last products, i.e., $Z_k[n-1]$, to the next neighbors and compute their partial products. Finally, each PE uses the received partial product from the previous neighbor, and multiplies it with each of the previous calculated $Z_k[j]$, to obtain the final results.

5.5 Implementation

The above mapping strategies of the various ZKP kernels are implemented with a customized compiler in our design. Currently, the frontend of our compiler is done manually, i.e., converting functions in standard ZKP libraries into specially-defined computation graphs. Figure 7 illustrates an example. The execution starts with the *Wires Commitment* function, which is parsed into three steps: iNTT, NTT, and Merkle tree. The *Get Challenges* function is part of the Fiat-Shamir transform, and only requires hash computations. The computation graph also executes many other functions, and eventually ends with the *Prove Openings* function that involves hash and polynomial kernels.

Our backend implementation takes such computation graphs as input, and maps each kernel to our hardware in a fully automated way, incorporating the proposed novel techniques. The backend outputs detailed schedules that describe how the kernels execute on the hardware, including how to fetch the data from memory, parallelize the computations on multiple PEs in the VSAs, and dictate the on-chip data communication between PEs.

6 Methodology

We have developed comprehensive RTL implementations for the key components in UniZK, including the VSA, the transpose unit, and the twiddle factor generator. We have extensively verified their functionality correctness. We synthesize these modules using the ASAP 7 nm technology. Additionally, we use FN-CACTI [55] to model the scratchpad and transpose buffers. Our default configuration has 32 VSAs and an 8 MB scratchpad. We equip the chip with two HBM2e PHYs [29, 47], achieving peak bandwidth of approximately 1 TB/s. The overall chip operates at 1 GHz, with power consumption of 64 W to the logic and 32 W to the HBM PHYs. Table 2 shows the detailed area and power breakdown per component in UniZK.

We also build a cycle-accurate simulator for UniZK to evaluate its performance on complex workloads. We use Ramulator2 [42] to model the off-chip DRAM access timing. The simulator is validated against our RTL implementations. It is open-sourced at https://github.com/tsinghua-ideal/UniZK.

Table 2. Area and power breakdown of UniZK.

Component	Area (mm^2)	Power (W)
32 VSAs	21.3	58.0
8 MB scratchpad	5.0	1.0
Twiddle factor generator	0.8	2.6
Transpose buffer	0.9	3.1
2 HBM PHYs	29.8	31.7
Total	57.8	96.4

Baselines. We mainly compare UniZK with CPU and GPU baselines for hash-based ZKPs, as well as a previous accelerator, PipeZK [72], which targets EC-based ZKPs. For CPU solutions, we use a server with two 20-core Intel Xeon Gold 5218R processors operating at 2.1 GHz, with 8 channels of DDR4 memory of 32 GB capacity and approximately 200 GB/s bandwidth. We use all the 80 threads to run parallel Plonky2 programs.

The GPU baseline runs a Plonky2 CUDA implementation [54] on an NVIDIA A100 GPU. The A100 GPU features 80 GB memory and 2 TB/s bandwidth. The GPU code primarily focuses on accelerating NTT, Merkle tree, and element-wise polynomial computations. The other kernels are still executed on the host CPU.

Applications. We use a variety of typical ZKP applications to measure the performance improvement of UniZK. (1) **Factorial** [51]. The prover demonstrates the correct calculation of the factorial of 2^{20} using ZKP. (2) **Fibonacci** [51]. The prover generates a proof of knowing the 2^{20}-th number in the Fibonacci sequence. (3) **ECDSA** [1] is a digital signature algorithm based on elliptic curve cryptography. The authenticity of the prover's signature is ensured using ZKP. We test it on a task of signing a 256-bit random file hash. (4) **SHA-256** [53] is a cryptographic hash function. The prover uses ZKP to demonstrate the possession of a message corresponding to a specific hash. We evaluate SHA-256 on a 8000 B message with 126 blocks. (5) **Image Crop** [2]. The prover generates a proof of cropping a 512×512 block from a 1024×1024 RGBA PNG image at the left-top corner. (6) **MVM** [67] (matrix-vector multiplication) is extensively used in scientific computing and neural networks. We test with a 3000×3000 16-bit matrix. In all workloads, we use typical configurations of Plonky2 and Starky aimed at achieving approximately 100 bits of conjectured security [51, 64].

7 Evaluation

7.1 Performance Comparison

Table 3 shows the overall performance comparison between UniZK and the CPU and GPU baselines. For each application, we report the end-to-end proof generation time and the speedup over the CPU. The GPU solution cannot generate an entire proof on itself but must rely on the CPU for certain kernels, which causes back-and-forth data transfers.

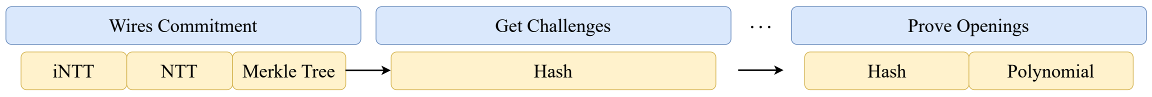


Figure 7. An example of the static computation graph in our compiler implementation.

Table 3. Overall performance comparison between UniZK and the CPU and GPU baselines for Plonky2.

Application	CPU	GPU		UniZK	
	Time (s)	Time (s)	Speedup	Time (s)	Speedup
Factorial	57.561	26.673	2.2×	0.828	70×
Fibonacci	3.373	0.736	4.6×	0.023	147×
ECDSA	7.463	2.063	3.6×	0.065	115×
SHA-256	55.445	26.845	2.1×	0.908	61×
Image Crop	23.765	16.182	1.5×	0.373	64×
MVM	39.669	33.383	1.2×	0.320	124×

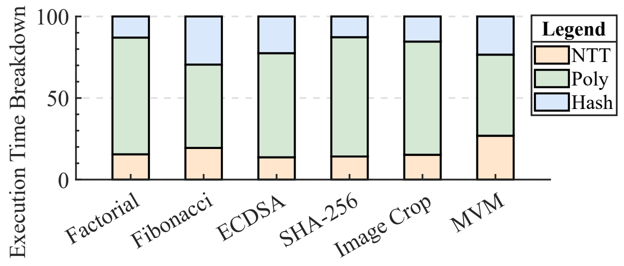


Figure 8. Performance breakdown by kernel types in UniZK.

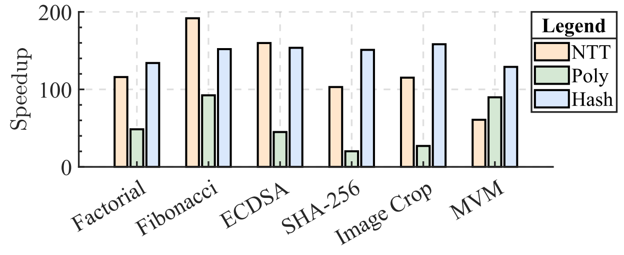


Figure 9. Speedups by kernel types in UniZK.

Even for the GPU running kernels, operations such as NTTs require irregular memory accesses that are not friendly to GPUs. Overall, the GPU speedups are limited to up to 4.6×. In contrast, UniZK achieves much more significant improvements, from 61× to 147×, and on average 97× over the CPU. Compared to the GPU, UniZK is still 46× faster on average.

In Figure 8, we further break down the execution time of UniZK by kernel types, into three major components: NTTs, element-wise polynomial computations, and hash computations (including Merkle tree and other hashes). Note that unlike the CPU, in UniZK the global transpose buffer manages data transpose and other layout transformations implicitly and in parallel with one of the computation kernels running, so this cost is eliminated in UniZK and not shown in the breakdown. We see that after efficient acceleration of the NTT and hash functions, the miscellaneous polynomial operations now account for the majority of the execution time, and become the bottleneck. Figure 9 also confirms this observation, where the speedup of the polynomial kernels is relatively lower than the other two kernel types. The NTT speedup is also generally lower than that of hashes, because NTT is mainly memory-bound.

While the above results demonstrate the effectiveness of our NTT and hash mapping strategies with up to 191× speedups, we also want to investigate why the polynomial kernels are not so efficient, with only moderate speedups of 20× to 92×. We find that during gate evaluation, we usually have pseudo-randomly data accesses where the accessed indices are determined by, e.g., bit-reversed operations. The access data size is constrained by the width and some other parameters of the arithmetic circuit. Usually the circuit width is not very large, e.g., 135, and the other parameters could be as low as 2. Therefore these random memory accesses have limited size, and underutilize the memory bandwidth. An exception is MVM, whose circuit width is as high as 400, thereby improving bandwidth utilization and offering a higher speedup for polynomial computations in Figure 9.

Table 4. Memory and VSA utilization breakdown in UniZK.

Application	NTT		Poly		Hash	
	Memory	VSA	Memory	VSA	Memory	VSA
Factorial	47.6%	4.3%	15.7%	2.0%	20.6%	96.9%
Fibonacci	55.5%	5.0%	17.9%	5.8%	20.6%	96.7%
ECDSA	56.4%	5.0%	15.4%	9.2%	20.6%	96.1%
SHA-256	47.4%	4.3%	13.6%	1.9%	20.7%	97.2%
Image Crop	54.0%	4.8%	13.5%	2.2%	20.7%	97.1%
MVM	53.0%	4.8%	24.5%	5.9%	21.7%	95.3%

7.2 Resource Utilization

Table 4 summarizes the utilization of memory bandwidth and VSAs, respectively, for each application. NTTs exhibit the highest bandwidth utilization but low VSA utilization, due to their memory-bound nature. In contrast, hash computations demonstrate the highest VSA utilization, with moderate bandwidth usage. The Poseidon hash function involves several on-chip matrix-vector multiplications after fetching a single input state, making it mostly compute-bound. Polynomial computations, on the other hand, show relatively low utilization on both memory bandwidth and VSA logic. This matches the previous results of their lower speedups.

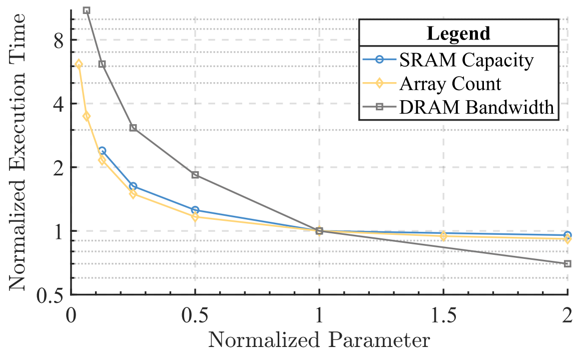


Figure 10. Performance sensitivity of UniZK when scaling different hardware configurations. With MVM.

Table 5. Overall performance comparison between UniZK and the CPU baseline for Starky + Plonky2.

Application	Stage	CPU Time (s)	UniZK Time (ms)	Speedup	Size (kB)
Factorial	Base	2.8	42	67×	261
	Recursive	1.7	12	142×	155
Fibonacci	Base	2.3	26	88×	259
	Recursive	1.9	12	158×	155
SHA-256	Base	0.8	3	267×	778
	Recursive	2.0	12	167×	187

7.3 Design Space Exploration

Figure 10 illustrates the normalized performance of UniZK on the MVM task, when various hardware resources are scaled up and down. We individually adjust the SRAM scratchpad size, the number of VSAs, and the memory bandwidth, to investigate their impact on UniZK. Reducing the scratchpad size or memory bandwidth affects the NTT and polynomial operations due to their memory bottlenecks. Additionally, the Merkle tree's performance is primarily dependent on the number of VSAs, which facilitates high parallelization.

7.4 Performance of Starky

To demonstrate the generality of supporting different ZKP protocols, we combine Starky and Plonky2 in the way described in Section 2.1, i.e., using Starky for the base proofs, followed by Plonky2 to aggregate the proofs to compress the proof size. The total execution time is the sum of the two stages. We select three applications that have existing Starky implementations [3, 52], using the same parameter set as those on Plonky2. Table 5 shows that, compared to the CPU, UniZK achieves the maximum speedup of 267×. Compared with running the same application with Plonky2 on UniZK (Table 3), running with Starky could achieve about 61× performance improvements.

Table 6. Performance comparison on the CPU and the ASIC accelerators (UniZK and PipeZK [72]) with two ZKP protocols of Starky + Plonky2 and Groth16.

	CPU Time (s)		ASIC Time (ms)		Speedup	
	Groth16	Starky + Plonky2	PipeZK	UniZK	PipeZK	UniZK
SHA-256	1.5	2.0	102	12.6	15×	159×
AES-128	1.1	3.4	97	27.7	12×	123×

7.5 Comparison with Previous ZKP Accelerators

PipeZK [72] is a state-of-the-art ZKP accelerator that targets the Groth16 protocol. We conduct a preliminary comparison between UniZK and PipeZK. Since the hardware architecture and the protocol algorithm are both different, a direct fair comparison is difficult. We thus evaluate both the CPU and the ASIC accelerator to gain better insights. We use two applications, SHA-256 and AES-128, both processing one data block, as in the original PipeZK paper [72]. This allows us to directly compare with their reported performance numbers.

We can see from Table 6 that, when running on the CPU, Starky + Plonky2 is slower than Groth16. This is because Starky + Plonky2 is dominated by the time of recursively compressing the proof, while the base proof generation part is short due to the small number of (a single) block being processed. If we use the corresponding ASIC designs to accelerate the two protocols, respectively, UniZK's speedup over the CPU baseline is 10.6× higher than PipeZK's. PipeZK leaves some work to the CPU, while UniZK supports end-to-end proof generation thanks to its general unified hardware design. But even if we look at the ASIC-only portion of PipeZK, which is about 1/4 to 1/3 of the end-to-end time, UniZK's speedup is still higher. This result implies that the hardware accelerator design of UniZK is more efficient than PipeZK. If we directly compare the two accelerators, we see 3.5× to 8.1× speedups.

Actually, the above applications of processing a single data block are not the best way to use Starky + Plonky2. If we can generate the proofs for multiple blocks at once, as in Table 5, only the base proof time increases, while the cost of the recursive compression can be amortized. UniZK can achieve over 8400 blocks per second for SHA-256. Compared to PipeZK, which processes 10 blocks per second for SHA-256, UniZK achieves a 840× speedup.

8 Related Work and Further Discussion

PipeZK [72] is an ASIC designed for the Groth16 protocol [27]. It incorporates two modules to accelerate the two most expensive kernels, NTT and MSM. However, it requires collaboration with the host CPU and thus cannot achieve full end-to-end acceleration. The nature of the EC-based algorithm also constrains its performance. SZKP [14], on the other hand, is a ZKP accelerator framework for end-to-end Groth16 proof generation. It adds support for sparse MSM which is handled by the CPU in PipeZK. SZKP uses high-level synthesis to generate accelerators in a scalable way, offering options from large, highly parallel designs to small cores. Finally, a concurrent work, NoCap [61], proposes an accelerator designed for hash-based ZKP protocols, specifically Spartan [62] and Orion [70]. It integrates a variety of functional units targeting different kernels, such as NTT, hashing, sparse matrix-vector multiplications, etc. Compared to NoCap, UniZK has a different design philosophy, mapping variable kernels onto unified hardware, resulting in more area-efficient architectural designs.

On GPUs, cuZK [40], an efficient GPU implementation to accelerate ZKP, achieves significant speedups by parallelizing operations and reducing CPU-GPU data transfers. GZKP [43] leverages GPU acceleration for ZKP, with special focuses on optimizing memory access patterns and parallelizing computational tasks. Ji et al. propose DistMSM [30], a highly optimized algorithm for the MSM task on multi-GPU systems. They leverage GPU shared memories to alleviate register pressures and utilize tensor cores to accelerate large integer multiplications on elliptic curves. On the other hand, Huang et al. [28] propose a CPU-GPU collaborative acceleration approach for the bulletproofs protocol [9].

Several works have utilized FPGAs to accelerate EC-based ZKPs, with a particular focus on the MSM kernel [4, 41, 56, 69]. The state-of-the-art solution for FPGA-based MSM is BSTMSM [73], which introduces a dedicated RAM to track the state of buckets in the Pippenger algorithm, thereby eliminating bucket collisions and improving hardware utilization.

On CPUs and distributed clusters, DIZK [68] utilizes a computer cluster to enable proof generation for large arithmetic circuits with up to billions of gates, which overcomes the resource limitation on a single server. ZENO [19] focuses on optimizing the proof generation for neural network inferences in the ZKML domain. It integrates a framework that maintains the semantics of neural networks, and enables more parallelization opportunities through techniques such as data encoding and tensor-level and layer-level scheduling.

8.1 Generality to Other Protocols

The novelty of UniZK lies in its ability to map various kernels onto the VSAs. This adaptability makes the design particularly suitable for the emerging hash-based ZKP protocols that have diverse kernels. The concrete design of UniZK presented in this paper particularly focuses on Plonky2, and has several specialized design choices like the 64-bit Goldilocks field, the Poseidon hash function, etc. Nevertheless, we believe that the same design philosophy can be adapted to other ZKP protocols, by making minor hardware changes in a similar architecture, and proposing new kernel mapping strategies. This is because, in modern cryptography, a key data structure is polynomials, often organized as vectors or matrices. Such matrix and tensor computations are inherently suitable for systolic arrays [33]. Thus it is promising to unify the diverse kernels on the common data structure to the same architecture.

To see how UniZK may generalize, we examine several recent ZKP protocols including Spartan [62], Binius [16, 17], and Basefold [71]. Most of their components match well with what we have already considered when designing UniZK, such as sparse matrix-vector multiplications and polynomial computations. Nevertheless, they also introduce a few new primitives. A challenging one is the sum-check function shown in Algorithm 2, following a dynamic programming computation flow. Its main loop body contains two parts: summing up the updated vector elements, and updating the vector itself. An accelerator similar to UniZK can efficiently execute the vector update operation similarly to element-wise polynomial operations. The data paths between PEs can be utilized to compute the vector sum, similar to how partial sums are accumulated in matrix multiplications. A more detailed evaluation is beyond the scope of this paper.

Algorithm 2: Sum-Check

```
Input  : An initial vector $A[2^n]$, random numbers $r[n]$.
Output : Results $y[n][2]$.
1 for $i \leftarrow 1$ to $n$ do
2     $y[i] \leftarrow [0, 0]$;
3     for $b \leftarrow 0$ to $2^{n-i} - 1$ do
          /* Sum up vector elements */
4         $y[i][0] \leftarrow y[i][0] + A[b]$;
5         $y[i][1] \leftarrow y[i][1] + A[b + 2^{n-i}]$;
          /* Update vector (element-wise polynomial operations) */
6         $A[b] \leftarrow (1 - r[i]) \cdot A[b] + r[i] \cdot A[b + 2^{n-i}]$;
```

9 Conclusions

In this paper, we propose UniZK, a general ZKP architecture to accelerate emerging hash-based ZKP protocols beyond classic EC-based ZKP schemes. UniZK uses a unified hardware architecture with systolic arrays of modular arithmetic units to offer hardware efficiency. It then maps diverse ZKP kernels onto such a generic template using novel mapping strategies, while ensuring high resource utilization. UniZK is significantly faster than the CPU and GPU baselines when executing the Plonky2 and Starky protocols, and also substantially outperforms previous ZKP accelerators.

Acknowledgments

The authors thank the anonymous reviewers and our shepherd David Kaeli for their valuable suggestions, and the Tsinghua IDEAL group members for constructive discussion. Mingyu Gao is the corresponding author.

A Artifact Appendix

A.1 Abstract

The source code for the simulator used in this work is available at https://github.com/tsinghua-ideal/UniZK. All the experiments described in this paper can be reproduced using the provided running scripts. The implementations of the evaluated applications are in the `examples` directory.

We have included necessary third-party libraries in the repository. The installation and build instructions are provided in the `README.md` file in the repository.

A.2 Artifact check-list (meta-information)

- **Compilation:** Rust 1.8, g++ 11.4.
- **Run-time environment:** Ubuntu 22.04.
- **Hardware:** Intel Xeon Gold 5218R.
- **Output:** The number of cycles in the log files.
- **How much disk space required (approximately)?:** 1 TB for temporary trace file.
- **How much time is needed to prepare workflow (approximately)?:** 30 minutes.
- **How much time is needed to complete experiments (approximately)?:** < 1 day for each experiment.
- **Publicly available?:** Yes.
- **Code licenses (if publicly available)?:** BSD-3.

A.3 Description

A.3.1 How to access.

- Source code: https://github.com/tsinghua-ideal/UniZK.

A.3.2 Hardware dependencies.

- The code is tested on a server with two Intel Xeon Gold 5218R CPUs, 256 GB memory, and 4 TB disk space.
- Most experiments can run on a computer with an Intel or AMD CPU, 32 GB memory, and 1 TB disk space.

A.3.3 Software dependencies.

- A recent Linux distribution.
- Rust 1.8.
- g++ 11.4.
- CMake 3.22.

A.4 Installation

- Use a nightly toolchain for Plonky2.

```
rustup override set nightly
```

- Build RamSim.

```
cd thirdparty/ramsim
mkdir build
cd build
cmake ..
make -j
```

RamSim is an enhanced version of Ramulator2 [42]. We add support for simulating the computation latency and properly handling the data dependencies between computations and memory accesses.

A.5 Experiment workflow

- To reproduce the performance results of the CPU baseline in Table 3, run the script:

```
./run_cpu_test.sh
```

- To reproduce UniZK's simulation results in Table 3, run the tests for applications using Plonky2:

```
./run_plonky2_test.sh
```

Run the tests for Starky and recursive proofs in Table 5 and Table 6:

```
./run_starky_test.sh
```

These scripts will set up the required third-party libraries and execute the experiments in the `examples` directory.

A.6 Evaluation and expected results

CPU baseline. When running the applications in Table 3 with Plonky2 on the CPU baseline, performance results are logged to files prefixed with `cpu_`, e.g., `cpu_ecdsa.log`:

```
Number of CPUs: 80
...
prove_with_partition_witness took: 7 463 553 us
```

This output indicates that the proof was generated in 7.46 s using 80 threads.

Similarly, when running the applications in Table 5 and Table 6 with Starky + Plonky2 on the CPU baseline, performance results are logged to files prefixed with `cpu_` and suffixed with `_starky`, e.g., `cpu_factorial_starky.log`:

```
Number of CPUs: 80
...
prove took: 2 792 781 us
Proof size: 260 688 bytes
...
prove_with_partition_witness took: 1 684 995 us
Recursive proof size: 155 380 bytes
```

This shows that the base proof generation took 2.8 s and the recursive proof generation took 1.7 s. The corresponding proof sizes are 261 kB and 155 kB, respectively.

UniZK. The experiment for each application in Table 3, Table 5, and Table 6 creates a log file named after the application (e.g., `sha256.log` or `sha256_starky.log`).

```
total_num_write_requests: 11704733
total_num_read_requests: 11946930
memory_system_cycles: 4207818
```

This indicates that UniZK performed 11704733 write requests and 11946930 read requests, each involving 64 bytes of data. The reported cycles of 4207818, at a clock speed of 1 GHz, translate to an execution time of 4.2 ms.

A.7 Experiment customization

For each application in Table 3, the provided command in `run_plonky2_test.sh` could be customized in several ways using command-line arguments, as shown in the following example:

```
cargo run --release --example ecdsa \
    -- -r 8 -t 32 -e 0
```

These arguments have the following meanings:

- `-r`: the scratchpad capacity in MB.
- `-t`: the number of VSAs.
- `-e`: the target kernel for simulation during proof generation, `0` for NTTs only, `1` for hash computations only. Omitting `-e` simulates the entire proof generation.

These arguments can be used to reproduce the results presented in Figure 8 and Figure 10.

A.8 Note

- The current simulator is built entirely on a single-threaded implementation, which results in relatively long simulation time. This long simulation duration is not related to UniZK's performance.
- The GPU performance in Table 3 can be evaluated using the GPU implementation of Plonky2 [54] with our CPU-based application code located in the `examples` directory.

References

[1] 2023. plonky2-ecdsa. https://github.com/qope/plonky2-examples.

[2] 2023. plonky2-zkedit. https://github.com/ChickenLover/plonky2-zkedit.

[3] 2023. sha256-starky. https://github.com/tumberger/plonky2/tree/sha256-starky.

[4] Kaveh Aasaraai, Don Beaver, Emanuele Cesena, Rahul Maganti, Nicolas Stalder, and Javier Varela. 2022. FPGA Acceleration of Multi-Scalar Multiplication: CycloneMSM. Cryptology ePrint Archive, Paper 2022/1396.

[5] Eli Ben-Sasson, Iddo Bentov, Yinon Horesh, and Michael Riabzev. 2018. Fast Reed-Solomon Interactive Oracle Proofs of Proximity. In *45th International Colloquium on Automata, Languages, and Programming (ICALP)*.

[6] Eli Ben-Sasson, Lior Goldberg, Swastik Kopparty, and Shubhangi Saraf. 2019. DEEP-FRI: Sampling Outside the Box Improves Soundness. Cryptology ePrint Archive, Paper 2019/336.

[7] Sean Bowe, Jack Grigg, and Daira Hopwood. 2019. Recursive Proof Composition without a Trusted Setup. Cryptology ePrint Archive, Paper 2019/1021.

[8] Benedikt Bünz, Ben Fisch, and Alan Szepieniec. 2020. Transparent SNARKs from DARK Compilers. In *Advances in Cryptology – EUROCRYPT*.

[9] Benedikt Bünz, Jonathan Bootle, Dan Boneh, Andrew Poelstra, Pieter Wuille, and Greg Maxwell. 2018. Bulletproofs: Short Proofs for Confidential Transactions and More. In *2018 IEEE Symposium on Security and Privacy (S&P)*.

[10] Yu-Hsin Chen, Joel S. Emer, and Vivienne Sze. 2016. Eyeriss: A Spatial Architecture for Energy-Efficient Dataflow for Convolutional Neural Networks. In *43rd Annual International Symposium on Computer Architecture (ISCA)*.

[11] Alessandro Chiesa, Yuncong Hu, Mary Maller, Pratyush Mishra, Noah Vesely, and Nicholas Ward. 2020. Marlin: Preprocessing zkSNARKs with Universal and Updatable SRS. In *Advances in Cryptology – EUROCRYPT*.

[12] Henri Cohen, Gerhard Frey, Roberto Avanzi, Christophe Doche, Tanja Lange, Kim Nguyen, and Frederik Vercauteren. 2012. *Handbook of Elliptic and Hyperelliptic Curve Cryptography* (2nd ed.). Chapman & Hall/CRC.

[13] Véronique Cortier, Alexandre Debant, Anselme Goetschmann, and Lucca Hirschi. 2024. Election Eligibility with OpenID: Turning Authentication into Transferable Proof of Eligibility. Cryptology ePrint Archive, Paper 2024/261.

[14] Alhad Daftardar, Brandon Reagen, and Siddharth Garg. 2024. SZKP: A Scalable Accelerator Architecture for Zero-Knowledge Proofs. In *International Conference on Parallel Architectures and Compilation Techniques (PACT)*.

[15] Sai Deng and Bo Du. 2023. zkTree: A Zero-Knowledge Recursion Tree with ZKP Membership Proofs. Cryptology ePrint Archive, Paper 2023/208.

[16] Benjamin E. Diamond and Jim Posen. 2023. Succinct Arguments over Towers of Binary Fields. Cryptology ePrint Archive, Paper 2023/1784.

[17] Benjamin E. Diamond and Jim Posen. 2024. Polylogarithmic Proofs for Multilinears over Binary Towers. Cryptology ePrint Archive, Paper 2024/504.

[18] Electron-Labs. 2024. eth-lc-plonky2. https://github.com/Electron-Labs/eth-lc-plonky2.

[19] Boyuan Feng, Zheng Wang, Yuke Wang, Shu Yang, and Yufei Ding. 2024. ZENO: A Type-Based Optimization Framework for Zero Knowledge Neural Network Inference. In *29th ACM International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*.

[20] Amos Fiat and Adi Shamir. 1986. How To Prove Yourself: Practical Solutions to Identification and Signature Problems. In *Advances in Cryptology — CRYPTO*.

[21] Jeremy Fowers, Kalin Ovtcharov, Michael Papamichael, Todd Massengill, Ming Liu, Daniel Lo, Shlomi Alkalay, Michael Haselman, Logan Adams, Mahdi Ghandi, Stephen Heil, Prerak Patel, Adam Sapek, Gabriel Weisz, Lisa Woods, Sitaram Lanka, Steven K. Reinhardt, Adrian M. Caulfield, Eric S. Chung, and Doug Burger. 2018. A Configurable Cloud-Scale DNN Processor for Real-Time AI. In *45th Annual International Symposium on Computer Architecture (ISCA)*.

[22] 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.

[23] Mingyu Gao, Xuan Yang, Jing Pu, Mark Horowitz, and Christos Kozyrakis. 2019. TANGRAM: Optimized Coarse-Grained Dataflow for Scalable NN Accelerators. In *24th ACM International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*.

[24] Mario Garrido. 2022. A Survey on Pipelined FFT Hardware Architectures. *Journal of Signal Processing Systems* (2022).

[25] Hasan Genc, Seah Kim, Alon Amid, Ameer Haj-Ali, Vighnesh Iyer, Pranav Prakash, Jerry Zhao, Daniel Grubb, Harrison Liew, Howard Mao, Albert Ou, Colin Schmidt, Samuel Steffl, John Wright, Ion Stoica, Jonathan Ragan-Kelley, Krste Asanovic, Borivoje Nikolic, and Yakun Sophia Shao. 2021. Gemmini: Enabling Systematic Deep-Learning Architecture Evaluation via Full-Stack Integration. In *58th ACM/IEEE Design Automation Conference (DAC)*.

[26] Lorenzo Grassi, Dmitry Khovratovich, Christian Rechberger, Arnab Roy, and Markus Schofnegger. 2021. Poseidon: A New Hash Function for Zero-Knowledge Proof Systems. In *30th USENIX Security Symposium (USENIX Security)*.

[27] Jens Groth. 2016. On the Size of Pairing-Based Non-Interactive Arguments. In *Advances in Cryptology – EUROCRYPT*. 305–326.

[28] Ying Huang, Xiaoying Zheng, Yongxin Zhu, Xiangcong Kong, and Xinru Jing. 2021. CPU-GPU Collaborative Acceleration of Bulletproofs - A Zero-Knowledge Proof Algorithm. In *2021 IEEE Intl Conf on Parallel & Distributed Processing with Applications, Big Data & Cloud Computing, Sustainable Computing & Communications, Social Computing & Networking (ISPA/BDCloud/SocialCom/SustainCom)*.

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

[30] Zhuoran Ji, Zhiyuan Zhang, Jiming Xu, and Lei Ju. 2024. Accelerating Multi-Scalar Multiplication for Efficient Zero Knowledge Proofs with Multi-GPU Systems. In *29th ACM International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*.

[31] Norm Jouppi, George Kurian, Sheng Li, Peter Ma, Rahul Nagarajan, Lifeng Nai, Nishant Patil, Suvinay Subramanian, Andy Swing, Brian Towles, Clifford Young, Xiang Zhou, Zongwei Zhou, and David A Patterson. 2023. TPU v4: An Optically Reconfigurable Supercomputer for Machine Learning with Hardware Support for Embeddings. In *50th Annual International Symposium on Computer Architecture (ISCA)*.

[32] Norman P. Jouppi, Doe Hyun Yoon, Matthew Ashcraft, Mark Gottscho, Thomas B. Jablin, George Kurian, James Laudon, Sheng Li, Peter Ma, Xiaoyu Ma, Thomas Norrie, Nishant Patil, Sushma Prasad, Cliff Young, Zongwei Zhou, and David Patterson. 2021. Ten Lessons From Three Generations Shaped Google's TPUv4i : Industrial Product. In *48th Annual International Symposium on Computer Architecture (ISCA)*.

[33] Norman P. Jouppi, Cliff Young, Nishant Patil, David Patterson, Gaurav Agrawal, Raminder Bajwa, Sarah Bates, Suresh Bhatia, Nan Boden, Al Borchers, Rick Boyle, Pierre-luc Cantin, Clifford Chao, Chris Clark, Jeremy Coriell, Mike Daley, Matt Dau, Jeffrey Dean, Ben Gelb, Tara Vazir Ghaemmaghami, Rajendra Gottipati, William Gulland, Robert Hagmann, C. Richard Ho, Doug Hogberg, John Hu, Robert Hundt, Dan Hurt, Julian Ibarz, Aaron Jaffey, Alek Jaworski, Alexander Kaplan, Harshit Khaitan, Daniel Killebrew, Andy Koch, Naveen Kumar, Steve Lacy, James Laudon, James Law, Diemthu Le, Chris Leary, Zhuyuan Liu, Kyle Lucke, Alan Lundin, Gordon MacKean, Adriana Maggiore, Maire Mahony, Kieran Miller, Rahul Nagarajan, Ravi Narayanaswami, Ray Ni, Kathy Nix, Thomas Norrie, Mark Omernick, Narayana Penukonda, Andy Phelps, Jonathan Ross, Matt Ross, Amir Salek, Emad Samadiani, Chris Severn, Gregory Sizikov, Matthew Snelham, Jed Souter, Dan Steinberg, Andy Swing, Mercedes Tan, Gregory Thorson, Bo Tian, Horia Toma, Erick Tuttle, Vijay Vasudevan, Richard Walter, Walter Wang, Eric Wilcox, and Doe Hyun Yoon. 2017. In-Datacenter Performance Analysis of a Tensor Processing Unit. In *44th Annual International Symposium on Computer Architecture (ISCA)*.

[34] Sheng-Chun Kao, Suvinay Subramanian, Gaurav Agrawal, Amir Yazdanbakhsh, and Tushar Krishna. 2023. FLAT: An Optimized Dataflow for Mitigating Attention Bottlenecks. In *28th ACM International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*.

[35] 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 *55th Annual International Symposium on Microarchitecture (MICRO)*.

[36] 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 *49th Annual International Symposium on Computer Architecture (ISCA)*.

[37] Hsiang Tsung Kung and Charles E Leiserson. 1979. Systolic arrays (for VLSI). In *Sparse Matrix Proceedings*.

[38] Sin7y Labs. 2022. OlaVM. https://github.com/Sin7Y/olavm.

[39] Liqiang Lu, Yicheng Jin, Hangrui Bi, Zizhang Luo, Peng Li, Tao Wang, and Yun Liang. 2021. Sanger: A Co-Design Framework for Enabling Sparse Attention using Reconfigurable Architecture. In *54th Annual International Symposium on Microarchitecture (MICRO)*.

[40] 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.

[41] Guiwen Luo, Shihui Fu, and Guang Gong. 2023. Speeding Up Multi-Scalar Multiplication over Fixed Points Towards Efficient zkSNARKs. *IACR Transactions on Cryptographic Hardware and Embedded Systems* (2023).

[42] Haocong Luo, Yahya Can Tuğrul, F. Nisa Bostancı, Ataberk Olgun, A. Giray Yağlıkçı, and Onur Mutlu. 2024. Ramulator 2.0: A Modern, Modular, and Extensible DRAM Simulator. *IEEE Computer Architecture Letters* (2024).

[43] 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 *28th ACM International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*.

[44] Mateo Vázquez Maceiras, Muhammad Waqar Azhar, and Pedro Trancoso. 2022. VSA: A Hybrid Vector-Systolic Architecture. In *40th International Conference on Computer Design (ICCD)*.

[45] Ralph C. Merkle. 1988. A Digital Signature Based on a Conventional Encryption Function. In *Advances in Cryptology — CRYPTO*.

[46] Thomas Norrie, Nishant Patil, Doe Hyun Yoon, George Kurian, Sheng Li, James Laudon, Cliff Young, Norman P. Jouppi, and David Patterson. 2020. Google's Training Chips Revealed: TPUv2 and TPUv3. In *IEEE Hot Chips 32 Symposium (HCS)*.

[47] Mike O'Connor, Niladrish Chatterjee, Donghyuk Lee, John Wilson, Aditya Agrawal, Stephen W. Keckler, and William J. Dally. 2017. Fine-Grained DRAM: Energy-Efficient DRAM for Extreme Bandwidth Systems. In *50th Annual International Symposium on Microarchitecture (MICRO)*.

[48] Angshuman Parashar, Minsoo Rhu, Anurag Mukkara, Antonio Puglielli, Rangharajan Venkatesan, Brucek Khailany, Joel Emer, Stephen W. Keckler, and William J. Dally. 2017. SCNN: An Accelerator for Compressed-Sparse Convolutional Neural Networks. In *44th Annual International Symposium on Computer Architecture (ISCA)*.

[49] Polygon. 2022. Introducing Plonky2. https://polygon.technology/blog/introducing-plonky2.

[50] Polygon. 2024. zkEVM. https://github.com/0xPolygonZero/zk_evm.

[51] PolygonZero. 2022. Plonky2: Fast Recursive Arguments with PLONK and FRI. https://github.com/mir-protocol/plonky2/blob/main/plonky2/plonky2.pdf.

[52] PolygonZero. 2024. Starky. https://github.com/0xPolygonZero/plonky2/tree/main/starky.

[53] PolymerDAO. 2022. plonky2-sha256. https://github.com/polymerdao/plonky2-sha256.

[54] Side Protocol. 2024. plonky2-gpu. https://github.com/sideprotocol/plonky2-gpu.

[55] Divya Praneetha Ravipati, Rajesh Kedia, Victor M. Van Santen, Jörg Henkel, Preeti Ranjan Panda, and Hussam Amrouch. 2022. FN-CACTI: Advanced CACTI for FinFET and NC-FinFET Technologies. *IEEE Transactions on Very Large Scale Integration (VLSI) Systems* (2022).

[56] Andy Ray, Benjamin Devlin, Fu Yong Quah, and Rahul Yesantharao. 2024. Hardcaml MSM: A High-Performance Split CPU-FPGA Multi-Scalar Multiplication Engine. In *ACM/SIGDA International Symposium on Field Programmable Gate Arrays (FPGA)*.

[57] Erik Rybakken, Leona Hioki, and Mario Yaksetig. 2023. Intmax2: A ZK-Rollup with Minimal Onchain Data and Computation Costs

Featuring Decentralized Aggregators. Cryptology ePrint Archive, Paper 2023/1082.

[58] Ananda Samajdar, Yuhao Zhu, Paul N. Whatmough, Matthew Mattina, and Tushar Krishna. 2018. SCALE-Sim: Systolic CNN Accelerator. *CoRR* (2018). arXiv:1811.02883

[59] 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 *54th Annual International Symposium on Microarchitecture (MICRO)*.

[60] 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 *49th Annual International Symposium on Computer Architecture (ISCA)*.

[61] Nikola Samardzic, Simon Langowski, Srinivas Devadas, and Daniel Sanchez. 2024. Accelerating Zero-Knowledge Proofs Through Hardware-Algorithm Co-Design. In *57th Annual International Symposium on Microarchitecture (MICRO)*.

[62] Srinath Setty. 2020. Spartan: Efficient and General-Purpose zkSNARKs without Trusted Setup. In *Advances in Cryptology – CRYPTO*.

[63] Guan Shen, Jieru Zhao, Quan Chen, Jingwen Leng, Chao Li, and Minyi Guo. 2022. SALO: An Efficient Spatial Accelerator Enabling Hybrid Sparse Attention Mechanisms for Long Sequences. In *59th ACM/IEEE Design Automation Conference (DAC)*.

[64] StarkWare. 2021. ethSTARK Documentation. Cryptology ePrint Archive, Paper 2021/582.

[65] Cheng Wang and Mingyu Gao. 2023. SAM: A Scalable Accelerator for Number Theoretic Transform Using Multi-Dimensional Decomposition. In *IEEE/ACM International Conference on Computer Aided Design (ICCAD)*.

[66] Worldcoin. 2023. awesome-zkml. https://github.com/worldcoin/awesome-zkml.

[67] Worldcoin. 2023. proto-neural-zkp. https://github.com/worldcoin/proto-neural-zkp.

[68] Howard Wu, Wenting Zheng, Alessandro Chiesa, Raluca Ada Popa, and Ion Stoica. 2018. DIZK: A Distributed Zero Knowledge Proof System. In *27th USENIX Security Symposium (USENIX Security)*.

[69] Charles. F. Xavier. 2022. PipeMSM: Hardware Acceleration for Multi-Scalar Multiplication. Cryptology ePrint Archive, Paper 2022/999.

[70] Tiancheng Xie, Yupeng Zhang, and Dawn Song. 2022. Orion: Zero Knowledge Proof with Linear Prover Time. In *Advances in Cryptology – CRYPTO*.

[71] Hadas Zeilberger, Binyi Chen, and Ben Fisch. 2023. BaseFold: Efficient Field-Agnostic Polynomial Commitment Schemes from Foldable Codes. Cryptology ePrint Archive, Paper 2023/1705.

[72] 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 *48th ACM/IEEE Annual International Symposium on Computer Architecture (ISCA)*.

[73] Baoze Zhao, Wenjin Huang, Tianrui Li, and Yihua Huang. 2023. BSTMSM: A High-Performance FPGA-Based Multi-Scalar Multiplication Hardware Accelerator. In *International Conference on Field Programmable Technology (FPT)*.