Accelerating Zero-Knowledge Proofs Through Hardware-Algorithm Co-Design

Nikola Samardzic*
MIT CSAIL
nsamar@csail.mit.edu

Simon Langowski*
MIT CSAIL
slangows@mit.edu

Srinivas Devadas
MIT CSAIL
devadas@mit.edu

Daniel Sanchez
MIT CSAIL
sanchez@csail.mit.edu

***Abstract*—*Zero-Knowledge Proofs* (ZKPs) are a cryptographic tool that enables one party (a *prover*) to prove to another (a *verifier*) that a statement is true, without requiring the prover to disclose any data to the verifier. ZKPs have many use cases, such as letting clients delegate computation to servers with cryptographic correctness guarantees, while enabling the server to use secret data in these computations. ZKP applications span verifiable machine learning (ML) and databases, online auctions, electronic voting, and blockchains. While ZKPs are already widely used in blockchains, the prohibitive costs of proof generation limit them to proving very simple computations.**

We present a novel accelerator, NoCap, that leverages hardware-algorithm co-design to achieve transformative speedups. NoCap generates proofs 586× faster than a 32-core CPU, and 41× faster than PipeZK, a state-of-the-art ZKP accelerator. We leverage recent algorithmic developments to achieve these speedups: we identify and combine two recent hash-based ZKP algorithms, Orion and Spartan, which have similar performance on CPUs to the ZKPs targeted by prior accelerators, but are much more amenable to hardware acceleration. Though these algorithms result in larger proofs, we show that the end-to-end speedups (including prover time, proof transmission, and verification time) more than justify this size increase.

We contribute a novel hardware organization to exploit these acceleration opportunities: NoCap is a programmable vector processor with functional units tailored to the needs of hash-based ZKPs. We also contribute a co-designed implementation of the Spartan+Orion ZKP tailored to accelerators, with optimizations that improve parallelism and reduce memory traffic. As a result, NoCap achieves speedups that enable new use cases for ZKP.

***Index Terms*—Zero-Knowledge Proofs, hardware acceleration, verifiable computation**

I. Introduction

Zero-Knowledge Proofs (ZKPs) are an emerging family of cryptographic tools that enables one party (*prover*) to prove to other parties (*verifiers*) that a statement is true, without requiring the prover to disclose any data to verifiers. Fig. 1 illustrates how ZKPs work. The prover generates a *small* proof of a statement and publishes it. Any verifier can download the proof and verify the statement *cheaply*. The prover can also restrict the proof so that it is only verifiable by a single verifier.

ZKPs are a powerful tool with myriad applications. The prover can convince the verifier that an arbitrary computation has been performed correctly; for example, clients can delegate computation to a server (e.g., transactions on a shared database), and the server can prove that it executed the clients' computation correctly [84]. Further, the prover can convince the verifier that the prover has data that produces a specific output; for example, a server can use ZKPs to prove to clients that a (secret) machine learning model achieves a certain accuracy [90]. Other ZKP applications include electronic voting [94], online auctions [33], static program verification [30], blockchains [2, 4, 56, 62, 89], and cryptocurrencies.

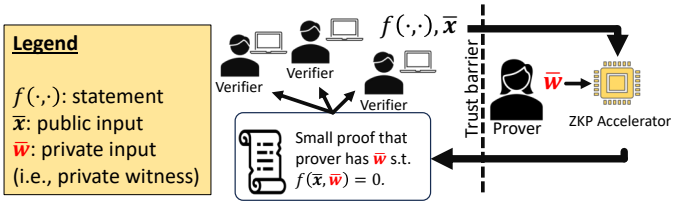

Fig. 1: ZKPs allow a prover to convince other parties, called verifiers, of a statement f, possibly requiring private input w from the prover.

ZKPs currently see limited use, e.g., for simple computations in blockchains, as their performance overheads greatly limit their applicability to the above areas. ZKPs suffer from extremely slow proof generation: in commonly used ZKPs, producing a proof for a given computation is 5–6 orders of magnitude slower than executing the computation itself when using a CPU, which limits ZKPs to verifying very simple computations. This has sparked efforts to accelerate proof generation, including using ASICs [92], FPGAs [5, 41, 79], and GPUs [52]. Recently, over $100M has been invested to develop and deploy ZKP accelerators [26, 77, 79, 88].

A full-system approach to accelerate ZKPs: In this paper, we present a novel ZKP accelerator that achieves transformative speedups, 586× over a 32-core CPU and 41× over prior ZKP accelerators, enabling new ZKP use cases.

To achieve these high speedups, we take an end-to-end systems design approach: instead of focusing on ZKP schemes accelerated by prior work, we identify a new family of ZKP schemes that make better tradeoffs between prover and verifier costs for many ZKP applications.

Concretely, prior work is focused on accelerating ZKPs based on *elliptic-curve cryptography*, like Groth16 [39]. These schemes produce tiny proofs (hundreds of bytes) that are cheap to verify, but proof generation is hard to accelerate. In particular, elliptic-curve operations are so computationally demanding that speedups are limited, even with hardware acceleration. For example, PipeZK [92], the state-of-the-art ASIC accelerator,

*The first two authors contributed equally to this work.

uses Groth16 and outperforms a CPU by 5–15×. These speedups are insufficient to counter the high overheads of ZKPs. We identify that combining Spartan [70] and Orion [86]—two recently proposed *hash-based* ZKP components—makes for proof generation that performs similarly on a CPU but is *much more amenable to acceleration*. However, this comes at the expense of larger proofs (megabytes). We show that for many applications, i.e., those where each proof is verified by one or a few clients, trading prover speed for proof size is the right tradeoff (Sec. III).

NoCap accelerator: Based on these insights, we design NoCap,[1] an accelerator for hash-based ZKPs. Proof generation comprises several complex algorithms that continue to evolve, so we build a programmable architecture that accelerates the primitive operations shared by these components.

NoCap is a vector processor with functional units tailored to the primitives used by ZKPs, including modular arithmetic on 64-bit integers, number-theoretic transforms (NTTs), cryptographic hashing, and shuffles. NoCap features high-bandwidth memory and an on-chip banked register file. We synthesize NoCap's building blocks using 14nm technology, and find that a chip with modest area, 46 mm^2, and power, 62 W, suffices to achieve high performance.

We evaluate NoCap on a wide range of applications. Our algorithm-accelerator co-design yields high speedups: NoCap is gmean 586× faster than a 32-core CPU that has significantly higher area and power budgets, and is 41× faster than PipeZK. These transformative speedups enable new applications, such as real-time verifiable databases.

Because all hash-based schemes build on the same primitives, NoCap is able to accelerate all hash-based ZKPs. To our knowledge, it is the first hash-based ZKP accelerator.

NoCap enables new use cases for ZKPs: NoCap's high speedups broaden the use cases for ZKPs. As a concrete example, consider secure photo modification. A camera can create a (hardware) signed image to verify its authenticity. Suppose a user wants to modify this image in some allowable way, e.g., cropping, with another photo editing program. The user can then prove that the cropped image is a descendant of the original, without revealing the original, and has not been modified further. For a 256KB image, this would take over 12 minutes to prove on a CPU, but with NoCap a proof takes just over a second, and verification takes only 0.2 seconds.

NoCap also enables real-time verified databases. The Litmus [84] cryptographically verified database achieves high throughput (thousands of transactions per second), but does so by batching over 80 thousand transactions at a time and using pipelined provers, which causes very high verification latencies, over 100 seconds. Though verification cost is small, a 100-second latency is untenable for many use cases, like financial transactions. Reducing batch size to achieve 1-second transaction latency (including computation, proof generation, and verification) reduces Litmus's throughput to only 2 transactions/second. NoCap can achieve 1-second transaction latency with a throughput of 1,142 transactions/second (Sec. VIII).

As NoCap allows for proofs about much larger programs, we can enable additional applications in privacy-preserving data analysis. For example, one can prove that differentially-private (DP) training was carried out correctly [72], reducing the required 100 hours of computation to less than 30 minutes. Other examples include showing that a machine learning model was applied to the data correctly [24, 50, 83]. As demand for even larger use cases grows, acceleration of ZKPs will become increasingly important.

In summary, our key contributions are:

- The observation that a combination of two recent ZKP primitives, Spartan and Orion, enable ZKPs with very high degrees of hardware acceleration (Sec. III).
- A novel programmable accelerator, NoCap, that accelerates the key primitives used by these ZKPs (Sec. IV).
- A high-performance implementation of Spartan+Orion targeting NoCap that achieves high data reuse, high arithmetic intensity, and scales to proofs for very large programs (Sec. V).
- A detailed synthesis and performance-based evaluation of NoCap, showing transformative speedups using an ASIC with modest cost (Sec. VI–Sec. VIII).

II. Background

As Fig. 1 shows, ZKP is a cryptographic tool that allows a prover to convince a verifier that for a given function $f(\cdot,\cdot)$ and *public input* $\bar{x}$, the prover knows a *private input* (a.k.a., *witness*) $\bar{w}$, such that $f(\bar{x},\bar{w})=0$ (this can be trivially generalized to prove that a function evaluates to an arbitrary value).

Functions f verified by ZKPs are *arithmetic circuits*. These are directed acyclic graphs where each node is an addition or multiplication (modulo some prime) and each edge (also called *wire value*) represents an intermediate value between operations. For example, the arithmetic circuit in Fig. 2 (left) implements the function $f(\bar{x},\bar{w}) = x_0 + w_0 + x_1 * w_1 + x_1 * w_1 * w_2$.

The most studied and widely used of ZKPs are zk-SNARKs [17] (zero-knowledge, succinct, non-interactive arguments of knowledge), because they have attractive properties: *(1)* zero-knowledge means the verifier does not learn anything about the private witness $\bar{w}$; *(2)* succinctness means that proof size and verifier compute costs are small (regardless of the complexity of f), *(3)* non-interactive means a verifier and prover need not communicate to check and generate the proof. For these reasons, we focus on accelerating zk-SNARKs.

A. zk-SNARK structure and design space

Fig. 2 shows the general structure of a common class of zk-SNARKs. zk-SNARKs generate proofs by reducing the task of proving $f(\bar{x},\bar{w})=0$ for an arbitrarily complex f (e.g., the circuit on the left of Fig. 2) into one of proving the correctness of one or a few polynomial evaluations.

As Fig. 2 shows, there are three main steps in doing this:

First, the prover constructs a system of equations whose correctness implies that $f(\bar{x},\bar{w})=0$ (1 in Fig. 2). This

[1] NoCap is so named because it erases the prover's performance cap in ZKPs, and after the slang phrase, as ZKPs prevent the server from lying.

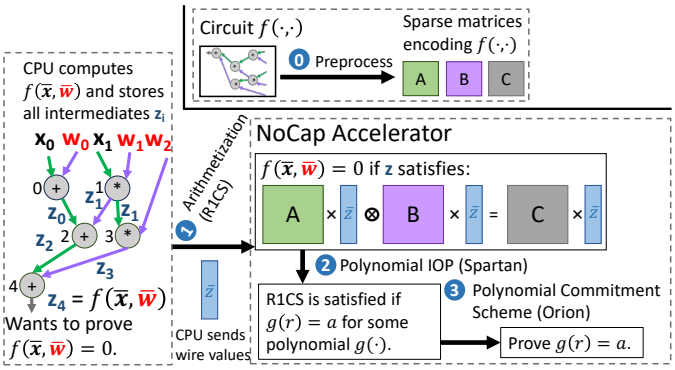

Fig. 2: Anatomy of a zk-SNARK.

process is called *arithmetization*. Arithmetization also involves a circuit-preprocessing step (0 in Fig. 2), which constructs matrices or polynomials (depending on the arithmetization) that only depend on the circuit being computed ($f(\cdot,\cdot)$), and not on the inputs ($\bar{x}$ and $\bar{w}$); therefore, the cost of this preprocessing is amortized by all runs of the same circuit, and is not accelerated by NoCap. In NoCap's design, the CPU is tasked with preprocessing and computing all intermediate wire values required to compute $f(\bar{x},\bar{w})$ (z_i's in Fig. 2). These wire values are then sent to NoCap ($\bar{z}$ in Fig. 2) and NoCap performs the rest of the proof.

Second, the prover constructs a polynomial (g in Fig. 2) with the property that the system of equations is satisfied if the evaluations of the polynomial at random points take on specific values (2 in Fig. 2). This is called the *polynomial Interactive Oracle Proof (IOP)*.

Third, the prover proves the correctness of these polynomial evaluations via a *polynomial commitment scheme*, an algorithm that enables the prover to *cheaply* convince the verifier of values a polynomial takes at any point (3 in Fig. 2).

Finding a zk-SNARK that is secure, performs well, and is amenable to hardware acceleration is challenging. This is because many zk-SNARKs can be constructed by combining any set of compatible components: an arithmetization, a polynomial IOP, and a polynomial commitment scheme. Moreover, any zk-SNARK can work with many fields (i.e., types of arithmetic operations), each of which has non-obvious tradeoffs between arithmetic performance and cryptographic security.

Given this complex design space, the cryptographic community has focused on ZKPs that are efficient on CPUs. But hardware acceleration has different tradeoffs than CPUs, so to leverage their potential, it is crucial to use accelerator-friendly ZKPs. To our knowledge, we are the first to co-design an accelerator and ZKP.

The seminal work of Groth16 [39] showed how to create the smallest possible proofs resulting in the fastest verification times. Subsequent work removed the strong trusted setup assumption from Groth16, generally resulting in longer verification times, proof sizes, and sometimes proving costs: Pinocchio [60], Hyrax [82], Libra [85], Plonk [32], and Hyperplonk [22], are ZKPs with new IOPs, but use the same polynomial commitment scheme as Groth16, called KZG [44].

From an acceleration perspective, the KZG polynomial commitment scheme has the largest computational cost of these ZKPs. Thus, any ZKP that uses KZG will suffer from the same computational bottlenecks. Hence, our focus on hash-based (as opposed to curve-based) ZKPs. Specifically, we consider works like Orion [86], Ligero [7], FRI [81], Orion+ [22], and Brakedown [36] that instead use error-correcting codes to construct the polynomial commitment scheme.

The three components that form the zk-SNARK we accelerate are *(1)* the R1CS arithmetization [13], *(2)* the Spartan IOP [70], but with *(3)* the Orion polynomial commitment scheme [86]. R1CS is common to many zk-SNARKs, but Spartan and Orion are recent, and we are the first to combine them. In Sec. III, we characterize this novel combination, showing how it enables high end-to-end speedups and comparing it with the zk-SNARK accelerated by prior work. We discuss all acceleration-relevant details of these components in Sec. V, when we discuss how to map them to NoCap, although we first briefly touch on the codesign with Spartan and Orion.

The Spartan algorithm shows how to prove a set of R1CS constraints using the sumcheck protocol and a polynomial commitment scheme (PCS). In order to build a zkSNARK, Spartan needs to be combined with a PCS. The Spartan authors evaluate several PCSs: Hyrax [82], Libra [85], vSQL [93], Virgo [91], Brakedown [36], and Shockwave [36]. Each of these compositions is correct and produces different computational tradeoffs; see [70] for details. We considered other alternative methods of constructing a zkSNARK, such as Plonk [32] or Bulletproofs [21]. We chose Spartan since the sumcheck protocol is highly amenable to acceleration, whereas the other protocols are more data-movement-bound.

The Orion PCS has been proposed as a replacement for Brakedown [86]; this makes the composition with the Spartan IOP straightforward, even though prior work has not evaluated Spartan+Orion. The key insight in Orion is a new proof composition technique, which uses a second (smaller) ZKP to show that the result of the first ZKP is valid. This composition reduces proof size and verifier time. Specifically, the fastest combination evaluated in Spartan, Spartan+Hyrax, is slower than Spartan+Orion in both proving and verification time.

Orion's original implementation uses linear-time error-correcting codes, which are typically based on expander graphs [20, 73]. But codes based on expander graphs are hard to accelerate: the expander graph can take several gigabytes, and encoding a message requires traversing many neighbors in this graph, resulting in serialized, data-dependent accesses to off-chip memory. These codes would make NoCap memory-bound and greatly limit speedups. To address this problem, we use Reed-Solomon codes [66], which are much more accelerator-friendly. Recent work has developed the transformation, called Shockwave [36], of substituting an expander-graph code by a Reed-Solomon code; we apply it to this problem.

Finally, the Spartan+Orion scheme has two advantages over elliptic-curve ZKPs beyond faster proofs. First, Spartan+Orion is post-quantum secure (specifically, it is "plausibly quantum-secure", i.e., secure against quantum computers given the

current understanding of their capabilities [36, 86]), whereas Groth16 is not. Second, Spartan and Orion are *transparent*, meaning they do not require a third party to perform trusted setup (in contrast, Groth16 does require trusted setup). This means that Spartan+Orion belongs to the subset of zkSNARKS known as zkSTARKs [12], reducing deployment complexity.

B. R1CS arithmetization

The first step in Fig. 2 is to convert a circuit into R1CS form. R1CS or the rank-1 constraint system is a set of quadratic equations that encode constraints of the circuit $f(\cdot,\cdot)$. The translation of a circuit to an R1CS system happens in steps ⓪ and ① in Fig. 2. The vector $\bar{z}$ is the vector of wire values of the circuit for a particular input $\bar{x}$ and $\bar{w}$ ($\bar{z} = (x_0, x_1, w_0, w_1, w_2, z_0, z_1, z_2, z_3, z_4)$ in Fig. 2).

The matrices A, B and C are computed during preprocessing and do not depend on the inputs $\bar{x}$ and $\bar{w}$. In particular, A is constructed so that the i-th element of the vector $A\bar{z}$ stores the left input to the i-th gate (for example, the zeroth element of $A\bar{z}$ is x_0). Therefore, roughly, A is a matrix representation of the permutation that, when applied to the wire values $\bar{z}$, yields a vector whose i-th element is the left input to the i-th gate. Similarly, B and C are constructed so that the i-th element of vectors $B\bar{z}$ and $C\bar{z}$ store the right input and the output of the i-th gate, respectively. Since A, B, and C usually just encode permutations, they are very sparse ($O(1)$ non-zeros per row).

R1CS is the most-commonly used arithmetization and it is used by the ZKPs of prior accelerators [52, 92].

III. ZKP Algorithmic Tradeoffs

With respect to performance, the three key characteristics of a zk-SNARK are *(1)* the time required for the prover to generate a proof (i.e., proof generation time), *(2)* the time required for the verifier to verify the proof (i.e., verification time), and *(3)* the size of the proof. zk-SNARKs make different tradeoffs between these metrics, and it's important to consider all these factors when analyzing performance.

We now analyze the end-to-end performance of the zk-SNARK targeted by prior work, Groth16 [39], and the zk-SNARK we accelerate, Spartan+Orion. We consider a scenario where prover and verifier communicate over a 10 MB/s connection, which allows us to analyze how proof size impacts end-to-end execution time.

Table I compares the execution times of the two zk-SNARKs on both CPUs and hardware accelerators for a R1CS circuit with 16 M constraints. We use a 32-core CPU for these experiments; we also report Groth16's performance on an NVIDIA V100 GPU using GZKP [52]. We compare the state-of-the-art PipeZK [92] accelerator for Groth16, and NoCap for Spartan+Orion. Execution time is broken down into the time taken by the prover, time to send the proof to the verifier, and time for the verifier to check the proof. Hardware acceleration affects only the prover time.

Since PipeZK was designed for an older technology node, for fairness, we scale it up so that it is iso-resource with NoCap: it has the same area in the same technology node, frequency, and memory bandwidth (Sec. VII). However, *PipeZK leaves part of the proof to the CPU, and this CPU portion bottlenecks PipeZK's performance*, so this scaling does not help end-to-end performance.

zkSNARK	Prover HW	Execution time (seconds) Prover	Send	Verifier	Total
Groth16	CPU	53.99	0.00	0.01	54.00
Groth16	GPU	37.44	0.00	0.01	37.45
Groth16	PipeZK	8.02	0.00	0.01	8.03
Spartan+Orion	CPU	94.20	0.81	0.13	95.14
Spartan+Orion	**NoCap**	**0.15**	**0.81**	**0.13**	**1.09**

TABLE I: Analysis of end-to-end performance for different zk-SNARKs and prover hardware combinations: time taken by the prover, verifier, and to send proof assuming a 10 MB/s link between prover and verifier. We compare the Groth16 and Spartan+Orion zk-SNARKs on a CPU and their respective accelerators, on a proof with 16 M R1CS constraints. Proof size for Groth16 is 0.2KB. Proof size for Spartan+Orion is 8.1MB.

Table I shows that Groth16 is completely dominated by proof generation, which takes *four orders of magnitude longer* than send and verification—54 s on the CPU vs 10 ms. And even with PipeZK, proof generation takes 8 s.

Spartan+Orion achieves a slightly longer proof generation time, 94.2 s on a CPU. Note that proof size and verification time are four and one order of magnitude larger than Groth16, respectively; this is due to the differences in verifier algorithms and large constant factors. Specifically, proof size and verifier time in Spartan+Orion are both $O(log^2 N)$ with N constraints [86], so for larger computations, verification costs become better amortized. But even for this modest case, proof generation cost still dominates on the CPU, and proof transmission and verification are a tiny fraction of overall time.

Furthermore, NoCap drastically accelerates Spartan+Orion's proof generation, by 586× in this case. Proof size and verification time remain unchanged, so proof generation now takes a modest 14% of total time. Even with Spartan+Orion's larger proofs and higher verification time, *this is a good tradeoff for most use cases*: end-to-end performance is 7.4× better than PipeZK's. As we will see later (Sec. VIII-F), end-to-end speedups grow with larger computations, as proof size and verification time grow much more slowly than prover time.

This shows that NoCap is far better for use cases where either the number of verifiers is relatively small, verifier costs are not relevant, or high proving throughput is critical. For use cases with many verifiers (e.g., public blockchains), Spartan+Orion may make verification cost too onerous. In this case, these overheads could be reduced by combining NoCap with an additional recursive proof, a technique used with some applications of blockchains to zero-knowledge virtual machines [63]. We leave such combination to future work.

Disentangling algorithmic efficiency, software efficiency, and acceleration potential: Table I shows that the CPU versions of Groth16 and Spartan+Orion take roughly the same proof time, so it may seem puzzling why NoCap achieves much higher speedups than prior work. There are three key factors that contribute to this: *(1)* Spartan+Orion is algorithmically

more efficient than Groth16, requiring far fewer computations; however *(2)* implementation inefficiencies and other operations that can be done cheaply in an accelerator squander these algorithmic gains on the CPU; and finally, *(3)* PipeZK is CPU-bound because it does not accelerate the entire proof:

1. Spartan+Orion is algorithmically more efficient: Groth16 and Spartan+Orion perform very different computations. For example, Groth16 operates on 381-bit modular integer values, while Spartan+Orion uses the Goldilocks-64 field, so comparing their efficiency directly is complicated. To do this, we focus on *critical operations*. In these accelerators, multipliers take most of the area, because multiply operations do most work. So a good proxy for computation is the number of 64-bit multiplies done in both algorithms.

By this metric, *Spartan+Orion performs 4.94× fewer 64-bit multiplies than Groth16*. This means that, if we had two accelerators that had the same multiply throughput and were multiplier-bound, the Spartan+Orion one would be 4.94× faster.

2. Spartan+Orion is less efficient than Groth16 on the CPU, squandering the above 4.94× efficiency. Note that we are using highly optimized CPU implementations that are vectorized and parallelized, and the Spartan+Orion CPU implementation includes our optimizations, which improve performance by over 2× vs. just combining existing codebases (Sec. VIII details these contributions).

Two factors contribute to this inefficiency. First, when run serially, the Spartan+Orion CPU version performs 4.66× fewer multiplies per second than Groth16. This difference stems from control overheads and other operations that, while cheap on an accelerator, take instructions on the CPU, leaving the critical resource (multipliers) underused. Second, the Spartan+Orion CPU version exploits less parallelism than Groth16: at 32 cores, Spartan+Orion achieves only 2.7× parallel speedup, while Groth16 achieves 5.0×. Combining these factors, Spartan+Orion proofs are 4.66/4.94/(2.7/5.0)=1.74× slower than Groth16.

This also explains why NoCap achieves such large speedups over the CPU: it has much higher multiplier throughput, it uses specialization to avoid inessential overheads that leave CPU multipliers underutilized, and it exploits parallelism better.

3. PipeZK is CPU-bound: Finally, NoCap is 53× faster than PipeZK in the previous example because PipeZK accelerates only part of the proof. For the part of the proof that PipeZK handles, PipeZK runs in 1.43s and achieves a 32× speedup over the CPU, but the non-accelerated part caps speedup to 6.7×. Note how, discounting the portion of Groth16 that PipeZK offloads to the CPU, the ratio of PipeZK's and NoCap's execution time is 9.5×. Much of this difference stems from the 4.94× difference in work among proof schemes; the remainder is due to architectural differences.

IV. NoCap Architecture

Fig. 3 shows an overview of NoCap. NoCap is a wide-vector processor with functional units (FUs) that accelerate the primitive operations used by hash-based ZKPs. NoCap has an explicitly managed memory hierarchy with decoupled data orchestration [61]: data is fetched ahead of its use into the 8 MB register file to hide memory latency. NoCap is a statically scheduled architecture, leveraging the regular structure of ZKPs to achieve high utilization with minimal control overheads.

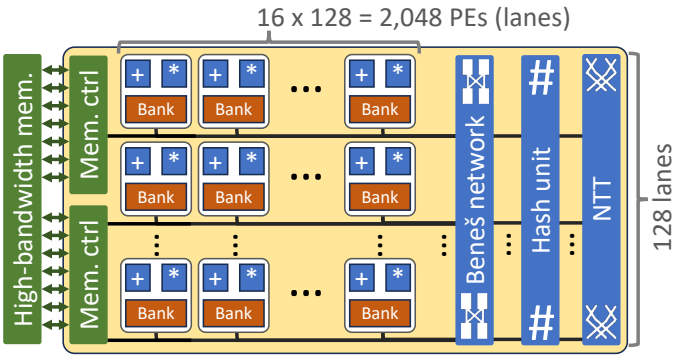

Fig. 3: NoCap architecture and physical organization overview.

We first describe NoCap's ISA and data types (Sec. IV-A), then its FUs and pipeline (Sec. IV-B) and memory system (Sec. IV-C). We then discuss how NoCap is integrated with a host processor (Sec. IV-D) and how NoCap can be leveraged to accelerate other ZKPs (Sec. IV-E). Sec. V details how to efficiently map Spartan+Orion to NoCap.

A. Vector ISA and data types

Data types: NoCap operates on k-element vectors, where k is a power of two between 2^7 (128) and 2^{16} (64K). Each element is a 64-bit integer modulo the Goldilocks64 prime [40], $p = 2^{64} - 2^{32} + 1$. This prime has an efficient modular reduction algorithm using only additions and bitshifts, which makes compute operations more efficient, especially multiplies.

Instruction set: NoCap supports the following vector compute operations: element-wise modular adds and multiplies, forward and backward NTTs, SHA3 hashing, and two types of permutations of vector elements: structured vector rotates and grouped interleavings.

NoCap supports vector loads and stores to move data between the register file and main memory, and a small number of control-flow instructions, described next.

Static scheduling: ZKPs are regular programs, and can be expressed as dataflow graphs, where all computations are known a priori. Each ZKP always performs the same computation for a given function f, regardless of its inputs ($\bar{x}$ and $\bar{w}$).

NoCap exploits this by adopting static scheduling [9, 18, 37, 53, 59], similar to VLIW architectures [31]: each instruction has a fixed latency, which is exposed to the compiler. The compiler schedules instructions at the appropriate cycles to respect data dependencies and avoid structural hazards. To cope with variable main-memory latency, NoCap assumes the worst-case latency, and buffers requests that are satisfied earlier.

Unlike VLIW, we adopt a distributed control approach, where each FU and component has its own instruction stream. This reduces code size over VLIW and avoids centralized instruction scheduling. Each instruction stream has two types of simple control instructions: delay instructions allow waiting for a specified number of cycles before issuing the next instruction

(so that components can be scheduled cycle-accurately), and simple branches that allow implementing loops with a fixed number of iterations. Each branch specifies a trip count and a (limited) negative offset; it is taken for that count, then not-taken. Since ZKPs have many regular loops, this helps keep code size small. Code is prefetched to small on-chip instruction buffers (similar to the CDC6600 [78]), then dispatched to FUs.

Static scheduling enables high throughput with little control overhead. NoCap's distributed control approach is similar to that of some recent accelerators in other domains, like fully homomorphic encryption [67, 68] and circuit simulation [29].

In terms of the hardware-software interface, distributed control is an implementation detail that can be abstracted away. We specify programs as in a VLIW machine, specifying the operations that start each cycle in a single instruction template. Then, a program transforms this into separate instruction streams (branches are replicated in every stream).

B. Pipeline and functional unit microrarchitecture

NoCap's FUs are fully pipelined, and can produce and consume a new chunk of vector elements every cycle. There are six types of FUs: the modular adder, modular multiplier, and hash FUs perform element-wise vector operations, while the NTT, and shuffle FUs are not element-wise.

ZKPs demand very different throughputs from these operations. To provide balanced performance, these FUs have a *heterogeneous number of lanes*: the register file and modular multipliers and adders have 2,048 lanes, but the hash and shuffle FUs have 128 lanes, and the NTT FU has 64 lanes. FUs with fewer than 2,048 lanes can still produce and consume wide vectors, but do so at lower throughputs.

The width of these units are sized based on two factors: the amount of reuse they achieve, and how frequently they are used. The hash and shuffle FUs sized to match HBM bandwidth (1 TB/s, i.e., 128 elements/cycle), since most of their input operands are loaded from memory, so more lanes would not help throughput. The multiply and add FUs are 2,048 elements wide because they are frequently used and can achieve significant reuse of operands during sumcheck recomputation (Sec. V-A). Conversely, the NTT FU has 64 lanes because it is used relatively infrequently, and it is a deep pipeline that consumes substantial area per lane.

Fig. 3 shows how NoCap's physical organization accomodates FUs with different lane widths. Main memory and the hash, shuffle and NTT FUs are laid vertically, over 128 lanes (the NTT FU can process one of the two input lanes). The 2,048-lane FUs are laid out in a 16×128 grid of PEs. Each PE contains one multiplier, adder, and register file bank. Vectors are stripped across PEs so that the i-th PE holds elements i, $i + 2048$, $i + 2 \cdot 2048$, etc. of each vector. Pipelined *row links* connect the 16 PEs in each row with a lane of the shuffle, hash, and NTT units, as well as the HBM memory.

NTT FU: We implement a pipelined NTT FU following the four-step algorithm [8], similar to F1's design [67]. The NTT FU works at a throughput of 64 elements per cycle, and performs forward and inverse NTTs on vectors of up to 64× 64 = 2^{12} elements. The building blocks of this FU are two 64-point NTT pipelines and a 64× 64 SRAM-based transpose unit. ZKPs need to perform NTTs on much larger vectors; this is done using repeated passes through the NTT unit via multiple applications of the four-step NTT algorithm (Sec. V).

Hash FU: The SHA3 [16] hash unit hashes at a throughput of 1 KB per cycle (i.e., 128 elements per cycle). The SHA3 hash algorithm takes two 256-bit values and outputs a 256-bit result. Thus, this FU reinterprets each group of four consecutive 64-bit elements as a 256-bit input. The NoCap hash instruction takes two equal-sized vectors and produces one output vector of the same size, where the first 256 bits of the output are the SHA3 hash of the concatenation of the first 256 bits of the two input vectors, and so on.

Shuffle FU: The shuffle FU is implemented with a 128-wide Beneš network [14], which supports arbitrary permutations. This network is used for two purposes. First, to support fast sparse matrix-dense vector multiplication (SpMV) in Spartan+Orion (Sec. V). Second, this network enables some structured permutations on vectors *wider* than 128 elements by leveraging NoCap's physical organization, as explained below.

Beneš network routing is complicated, but because all dependencies in ZKP are known at compile time, we determine the network's routing control bits at compile time, and embed them in the instruction (Sec. IV-A). The Beneš network requires $\sim N \log_2 N$ bits of control state for an N-element network. This means that instructions for setting the Beneš network control state occupy 7 bits per 64-bit element.

Implementing wide permutations: NoCap must support two kinds of permutations on wide vectors: cyclic shifts and grouped interleavings. Grouped interleavings include putting even-indexed elements in the first half of the vector and odd-indexed elements in the second half; more generally, grouped interleavings entail grouping even-indexed groups of 2^G-element contiguous chunks to the first half, with odd-indexed groups going to the second half. Cyclic rotations are used for folding in sumcheck, a key kernel in many hash-based ZKPs including Spartan+Orion (Sec. V). Grouped interleavings are used to compact hashes into adjacent vector lanes.

These permutations are not performance-critical, so we implement them using several passes through the shuffle network combined with reads and writes to different register banks in each PE row. For example, a rotation by 520=8+512 is implemented as a rotation by 8 in the 128-wide shuffle FU, combined with writing the result 4 PEs ahead from the source in each row, which implements a rotation by 128×4=512.

C. Memory system

NoCap has an 8 MB heavily banked on-chip register file. Because computation is entirely known at compile time, we leverage decoupled data orchestration [61]: loads are scheduled independently of compute operations, fetching data explicitly well ahead of its use. This allows us to hide memory latency.

NoCap uses 1 TB/s high-bandwidth memory (HBM) and compute is sized to fully use memory bandwidth on common ZKP tasks (Sec. VIII). HBM bandwidth is 48× smaller than

register file bandwidth. Despite our new contributions to reduce memory traffic (Sec. V), many of Spartan+Orion's operands are too large to keep on-chip and reuse in several algorithms is limited. In Sec. VIII-D, we show that decreasing memory bandwidth drastically limits performance.

D. System integration

NoCap is meant to be connected to one or multiple host CPUs, e.g., over PCIe. The host CPU first loads the program for the function $f(\cdot,\cdot)$ that should be verified into NoCap's memory, along with the precomputed circuit-dependent sparse matrices A, B, C. (Multiple such programs, corresponding to different functions, can be preloaded.) Then, the host CPU sends the wire values ($\bar{z}$; Fig. 2) for the function, which initiates proof generation. PCIe 5.0 supports 64 GB/s bandwidth [3], more than enough to keep NoCap busy.

E. Generality

NoCap can support any hash- or LWE-based SNARK protocol, as they all build on the same primitives: hashing, NTTs, and modular multiplies and adds. These include Brakedown [36], STARKs [62], and lattice-based SNARK schemes [6, 43]. By being programmable, NoCap should be applicable to future algorithmic advances in hash-based zk-SNARKs.

V. Mapping Spartan+Orion to NoCap

The Spartan+Orion ZKP relies on several complex cryptographic algorithms. However, we observe that all these algorithms are built on a small number of low-level computations that we call *tasks*. In turn, these tasks can be implemented with the few *primitive operations* that NoCap FUs implement. Tasks are large, so we execute them serially (one at a time) without meaningful performance degradation. Our implementation exploits abundant parallelism within each task.

Fig. 4 shows the relationships between high-level algorithms, low-level tasks, and primitive operations: each arrow $A \rightarrow B$ denotes that A uses B. Fig. 4 also shows the percentage of execution time (in a CPU) that each task spends.

This taxonomy makes it easy to understand Spartan+Orion's performance and to reveal optimization opportunities: in this section, we *focus on the tasks*, presenting each task and discussing how we map it to NoCap. While understanding the cryptographic algorithms in Fig. 4 is crucial to implement Spartan+Orion, they are not needed to understand its performance, so we do not consider them further. Interested readers can refer to [7, 11, 35, 91] for details.

Before diving into these tasks, note that the execution time percentages in Fig. 4 show that achieving high speedups requires accelerating all tasks. For example, the CPU spends 2% of time on SpMV; this may seem low, but if we delegated SpMV to the CPU, this would limit speedups to 50×, far below NoCap's 586× speedup.

A. Key tasks

We now describe five families of tasks that are used in Spartan+Orion (Fig. 4). Hash-based zk-SNARKs make heavy use of these tasks. By using Reed-Solomon codes, the verifier can be fast as it only needs to check a fraction of the values. By using Merkle trees, the proof can be succinct. Finally, the sumcheck protocol enables the verifier to check statements about polynomials (note ZKPs encode arithmetic circuits using polynomials) in logarithmic time and space.

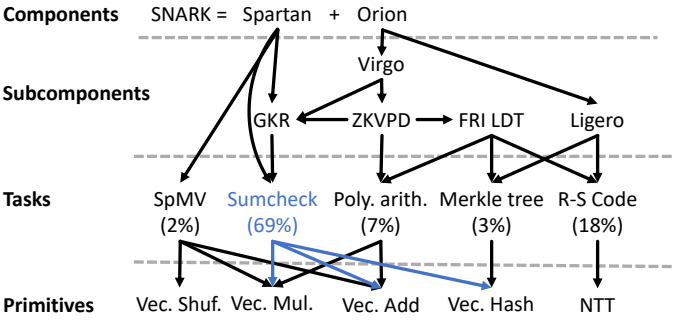

Fig. 4: Cryptographic components of Spartan+Orion, along with tasks (w/ CPU runtime percentages) and computational primitives that implement them. Each arrow A→B denotes "A uses B".

Reed-Solomon codes: An error-correcting code transforms a message m into a (slightly longer) codeword c. If a few elements of the codeword become corrupted, it is still possible to recover the original message. In ZKPs, this property is instead used to check that the message is well-formed by verifying the codeword.

Following the best parameters in Shockwave [36], a message vector of n finite field elements is encoded into a codeword of $4n$ elements. The codeword is verified by checking only 189 of the elements; this is independent of n, which is important for achieving fast verification.

We implement the Reed-Solomon code using the NTT primitive. First, the n-element message is extended with zeros to length $4n$. Then, a NTT (on $4n$ elements) is applied.

The NTTs used in Reed-Solomon codes are much larger than the maximum supported by NoCap's NTT FU (i.e., 2^{12}). But this NTT FU can still be leveraged to implement larger NTTs. Specifically, n-element NTTs (for $n > 2^{12}$) are done by viewing the input to the NTT as a matrix of elements with dimensions $2^{12} \times \frac{n}{2^{12}}$. The NTT is performed row-wise and column-wise on this matrix. In practice, this means the NTT is applied to blocks of 2^{12} consecutive elements, and then the matrix is transposed. Applying this recursively enables NTTs of arbitrary length. Transposes are done on-chip when possible, then through main memory when the vector does not fit in the register file (which holds 2^{20} elements). One transpose involving off-chip memory is sufficient for an input R1CS size of up to 2^{36}, well above our maximum target.

Finally, Reed-Solomon codes are *linear* functions: they have the property that the codeword of $m_1 + m_2$ will be $c_1 + c_2$, where c_i is the codeword for m_i and the addition is in the finite field. To create a succinct proof, codewords are combined together, utilizing the vector arithmetic units.

Merkle tree: A cryptographic hash function $h(x)$ produces a small (fixed-size) output for any input. Cryptographic hash

functions (e.g., SHA [15]) involve multiple steps of computation to make them computationally intractable to reverse.

We use dedicated hash functional units to implement hash functions. These hash functions are used to create a cryptographic structure known as a Merkle tree [54]. First, the finite field elements are packed and hashed into the leaves of a binary tree. Then, each node is updated to the hash of the values of its two children. We implement the tree with our vector hashing unit: hashing the nodes in the largest layers in parallel, and by applying the grouped interleavings to rearrange data when computing the layers smaller than our vector size.

SpMV: Sparse-matrix dense-vector multiplication (SpMV) computes $y = Ax$ where A is a sparse matrix and x is a dense vector. Spartan performs SpMV on very large matrices and vectors that do not fit on-chip (roughly as many elements as there are R1CS constraints, or 8 GB at 2^{30} constraints).

Specifically, Spartan applies SpMV to the same vector using three matrices. These sparse matrices are A, B, and C from Fig. 2, i.e., the ones computed during the preprocessing of the circuit $f(\cdot, \cdot)$. These matrices have two helpful properties: *(1)* they are limited-bandwidth matrices, i.e., most of their non-zeros are in a narrow band around their diagonal, *(2)* they are known at compile-time. We use the first property to achieve good reuse of the vector, and the second to achieve this reuse without a cache, by finding a good schedule at compile-time.

Prior work has shown how to efficiently map SpMV to vector processors [10, 64]; we leverage these techniques in NoCap. We compute SpMV with an output-stationary dataflow [75], producing the output vector y sequentially. To do this, we load the chunks of the input vector x that contribute to each chunk of y, use the flexible Beneš network to align x elements to the locations of y they contribute to, stream the sparse matrices and multiply them with the aligned x elements to produce partial products, and finally accumulate these partial products to produce a chunk of y. Because the sparsity pattern of the matrices is known, their non-zeros are stored in the right order to produce the partial products. This avoids storing or loading matrix coordinates, as is common in compressed sparse formats.

This approach enables NoCap to saturate memory bandwidth, while at the same time minimizing the amount of off-chip data fetched: the input vector achieves good reuse, while each sparse matrix is read exactly once.

Sumcheck DP algorithm: The sumcheck algorithm is used to check the prover's claims about the evaluation of polynomials. The input to the first sumcheck will be three sparse-matrix vector products: $A\bar{z}$, $B\bar{z}$, and $C\bar{z}$, where $\bar{z}$ stores the wire values from the circuit evaluation. These vectors are "mapped" to polynomials by interpreting each entry as the evaluation on a corresponding point. Specifically, for an 2^L length vector, we view the element in index i as the evaluation of a L-variable polynomial where the L variable correspond to the bit pattern of i. The sumcheck protocol produces a succinct L-size proof of a 2^L size polynomial statement in this form. Listing 1 shows the dynamic programming algorithm for implementing sumcheck for a L-variable polynomial [76, 85]. This algorithm is what NoCap spends most time on, so we will explain our optimizations and architecture mapping in detail.

```
def sumcheckDP(A[0:(1<<L)]) \
  -> result[0:L][0:2], rx[0:L]:
  for i in range(1, L+1):
    let s = (1<<(L-i))
    # sum of evaluations
    let y0, y1 = 0
    for b in range(s):
      if i > 1:
        # Update DP arrays
        A[b]   = A[b]  *(1-rx[i-1]) + A[b+2*s]*rx[i-1]
        A[b+s] = A[b+s]*(1-rx[i-1]) + A[b+3*s]*rx[i-1]
      y0 += A[b]
      y1 += A[b+s]
    result[i][0:2] = [y0, y1]
    # Create random challenge
    rx[i] = HASH(result[i])
  return result, rx
```

Listing 1: The sumcheck dynamic programming algorithm for proving the value of $\sum_{b\in\{0,1\}^L} A(b)$.

The protocol operates in L rounds, condensing a polynomial of size 2^i to 2^{i-1} in round i. In a round, we first do some vector arithmetic, which we implement accordingly. Then, we sum the values in the array, which we do with the standard reduction technique of summing the values in parallel with a binary tree structure, and using cyclic shifts to sum the values within a vector. Then, the sum(s) are hashed into the hash unit. The output of this hash is used in the next iteration.

Unfortunately, the array `A` is too large to store on-chip. The main bottleneck in this protocol becomes loading of the DP array on each iteration. However, for the sumchecks we run in Spartan+Orion, we noticed the form of `A` can often be compressible, in the sense that the evaluations that form the elements of `A` can be derived from a more compressed format.

Specifically, the initial circuit for a ZKP can be considered to have N gates with approximately N intermediate wires (one intermediate wire corresponding to the output of each gate). This circuit description is translated into an R1CS form with approximately N nonzero entries in each of the matrices A, B, and C, where A lists the left input wires, B the right input wires, and C the output. These steps are ordinarily performed once, and the resulting output A, B, C (now of size $3N$) reused throughout the computation. Recomputing the sparse matrix vector products $A\bar{z}$, $B\bar{z}$, and $C\bar{z}$ on demand requires loading $2N$ values—the circuit and $\bar{z}$—instead of $3N$. We use the values of `rx[1]`, `rx[2]`, ..., `rx[i-1]` to fast-forward to the needed values of `A` for iteration `i` directly, without requiring additional memory accesses. This allows us to use NoCap's large computational throughput to lower memory bandwidth. Because of the sequential data pattern in sumcheck, we can stream the three outputs in parallel (for $A\bar{z}$, $B\bar{z}$, and $C\bar{z}$) by streaming the circuit and $\bar{z}$ from memory. The key insight is that for even larger sumchecks (which can be up to size $18N$), we can always recompute everything from the smaller N-size circuit and witness. This circuit is streamed in 61-bit elements: for each operation, we keep track of the operation type (add or multiply) as well as the address of the operand node. By storing the address relative to the current node, we

Building block	Area [mm^2]
NTT FU	1.80
Multiply FU	6.34
Add FU	0.96
Hash FU	0.84
Total Compute	**9.95**
Reg. file (2,048× 4 KB banks)	6.01
Beneš network	0.11
Memory interface (2× PHY)	29.80
Total memory system	**35.92**
Total NoCap	**45.87**

TABLE II: Area of NoCap, and breakdown by building block.

can compress this representation to 61 bits per node.

While on a CPU it is more logical to compute these values once and reuse them, on the accelerator, as the bottleneck is data movement, recomputation makes sense. By recomputing the inputs, we are able to reduce the memory usage across all sumcheck by 31%. This recomputation uses many intermediates, which is why NoCap requires an 8 MB scratchpad.

Sumcheck algorithms for more complicated statements simply add more vector operations to each iteration. The data access pattern remains the critical bottleneck despite the increase in operations, and our key optimization of recomputing `A` from smaller data applies.

Polynomial arithmetic: Finally, Spartan+Orion has some operations that rely on polynomial addition and multiplication. Polynomial addition is trivial: polynomial coefficients are stored in vectors and added element-wise with vector units.

Polynomial multiplication is more complicated, because it requires convolving the coefficients of the input polynomials. This is efficiently done using NTTs: coefficients are transformed to the NTT domain, multiplied element-wise, and an inverse NTT transforms this result back to the coefficient domain [8] (this is analogous to how FFTs enable efficient convolutions). We use NoCap's NTT units for this purpose.

VI. NoCap Implementation

We have implemented NoCap's building blocks in RTL, and synthesized them in a commercial 14nm process. We target a 1 GHz frequency. We use a commercial SRAM compiler for register file banks.

Table II shows a breakdown of area by component. 13% of area goes to the register file, 21% to FUs, and 64% to two HBM PHYs. We assume 512 GB/s of bandwidth per PHY; this is similar to the NVIDIA A100 GPU. We use prior work to estimate the HBM2E PHY area [28, 65] and power [65].

VII. Methodology

Modeled system: We evaluate NoCap as configured in Table II. We hand-schedule all tasks from Sec. V except SpMV; these tasks are simple and regular, so this takes modest effort. We automate the scheduling of SpMV, by producing a stream of instructions specific to each benchmark as described in Sec. V. We then implement a simple linker program that composes these parameterized tasks to implement the Spartan+Orion prover. Tasks are executed one at a time, following program order. A simulator executes this program, keeping track of the FU and memory bandwidth usage of each task. This cycle-level simulator models the timing of each task by using timing models for the functional units and main memory. The simulator also collects activity factors for all units, which we then combine with per-event energies from RTL synthesis (Sec. VI) to compute power.

Baseline: Our software baseline is a parallelized Orion implementation from the original authors [74]; we use the multicore branch of Spartan's codebase for the Spartan IOP [55]. We enhance this baseline by using Goldilocks64 and Reed-Solomon codes, as presented in Sec. V. This matches NoCap's implementation and improves CPU performance by over 2×. For Groth16's CPU version, we use the libsnark parallel implementation [69]. We use a 3.5GHz AMD Ryzen Threadripper 3975WX CPU, which has 32 cores and 64 threads.

Finally, we optimistically scale the area and frequency of PipeZK to our 14nm technology and have it match NoCap's area; we also use the same memory bandwidth as NoCap. PipeZK [92] uses the MNT4-753 curve for their headline results. We use the BLS12-381 curve for PipeZK instead, because it is 4–10× faster while still achieving 128 bits of security. PipeZK's execution time is bottlenecked by the MSM G2 phase, which is offloaded to the CPU; therefore, our scaling of area and frequency does not bring any improvements to PipeZK's end-to-end performance. We measure PipeZK's MSM G2 phase using the same CPU that we use for our CPU baseline.

A. Parameters

We target 128 bits of security for both Groth16 and Spartan+Orion and statements with up to 2^{30} R1CS constraints.

As explained in Sec. IV, Spartan+Orion uses fast 64-bit modular arithmetic through the Goldilocks64 field. This requires repeating several parts of the protocol to achieve 128-bit soundness. First, we run all sumchecks 3 times. For the multiset hash function in Spartan, we run 4 separate instantiations (i.e., different γ values). Finally, in Orion's polynomial commitment, we use 4 random vectors to do the proximity test. We follow the observation made by prior work that these can reuse the same columns resulting in a smaller proof [36].

We use Reed-Solomon codes instead of expander graphs in Orion. The Reed-Solomon code blowup factor is 4, so the number of Orion column queries is 189. This also results in smaller proofs and verifier times, as the expander graph required 1,222 column queries. We set the number of rows in the Orion matrix to 128, as in the original implementation [74].

For zero knowledge, we compute and commit to a random masking polynomial to hide the witness $\bar{w}$ [71], as in protocol 5 of Orion [86]. However, as the time is dominated by sumcheck, achieving zero knowledge in this way is very cheap.

B. Benchmarks

For benchmarks, we use scaled-up versions of ZKP circuits used by PipeZK [92] and include one additional database application [84]. Table III lists these benchmarks and their

Benchmark	R1CS Size	Proof [MB]	V time [ms]
AES [92]	16.0M	8.1	134.0
SHA [92]	32.0M	8.7	153.7
RSA [92]	98.0M	10.1	198.0
Litmus [84]	268.4M	10.9	222.4
Auction [33]	550.0M	12.5	276.1

TABLE III: Number of R1CS constraints, proof size, and CPU verification time for our benchmarks.

characteristics, including circuit size (in number of R1CS constraints), proof size, and verifier time.

We use scaled-up ZKP benchmarks that perform 100–1,000× more work than the original versions in order to show NoCap's performance improvements. Prior systems use small circuits due to prover overheads, but since NoCap supports higher throughput, very small circuits would be dominated by fixed overheads, like sending wire values.

AES is a cryptographic primitive with several applications, mainly in encryption. A zero-knowledge proof of the AES circuit can be used to prove that a ciphertext is well-formed (useful for malicious security applications), or to show a third party that a ciphertext decrypts to a particular message (without revealing the decryption key). For example, it could be used to selectively decrypt an end-to-end encrypted message without revealing the contents of any other messages encrypted with the same key. The zero knowledge is important because simply revealing the key would allow any message to be decrypted. The original benchmark proves the AES evaluation for a 128-bit block; instead, we use a circuit for 1,000 blocks, corresponding to the encryption of a 16 KB message.

SHA is a hash function that can also be used to prove an entity has data (e.g., code) that corresponds to a specific SHA. In a ZKP, this can be used to show ownership of a digital object, without revealing the digital object. The zero knowledge is important because someone could make a copy of the digital object if they see it. For example, someone can prove one has the source code to a proprietary binary without revealing anything about the source. Instead of proving the SHA hash of 512 bits of data (as in PipeZK), we use a circuit for 1,000 512-bit hash blocks, equivalent to hashing a 64KB file.

RSA performs many of the same functions as AES. Since RSA operates on large prime fields, typically primes of 2,048 bits, the size of message per evaluation is larger. For one instance, the function can correspond to a 256-byte message, and for 1,000× to a 256 KB message.

Auction implements a verifiable sealed-bid auction [33]. This circuit enables trustless private auctions. In particular, this circuit enables an auctioneer to prove to all auction participants that the rules for determining the auction winner have been used correctly, without revealing any information about the losing bids. This benchmarks an auction with 100× the bids and is thus 100× larger than in prior work [92].

Litmus is a database management system that provides cryptographic proofs of transactional correctness, including atomicity and serializability [84]. We benchmark YCSB transactions [42]. In the original paper, the circuit is divided into separate subcircuits to allow parallel proving. Instead of proving one large circuit, the computation in each subcircuit is proved in parallel. This does not reduce prover work, but it does improve parallelism. However, the verifier must then check that the outputs of each subcircuit are consistent with the input to the next. Overall, this parallelism improves the prover's wall-clock time, but imposes 100× overhead on the verifier. Thus, we do not apply this optimization; NoCap achieves high parallelism for a single proof. The benchmark consists of 10,000 transactions that access two random rows, reading or writing to them with equal probability.

VIII. Evaluation

A. Performance

Table IV compares the performance of NoCap and the CPU on our benchmarks; both run the Spartan+Orion ZKP. It reports execution time for each (lower is better) and NoCap's speedups over the CPU (higher is better). NoCap achieves very large speedups, gmean 586×. Speedups are consistent across benchmarks, from 560× to 622×.

These speedups open new use cases of ZKPs. For example, assuming a 1s transaction latency target, proving database transactions in software achieves a measly throughput of 2 transactions/second. NoCap increases throughput to 1,142 transactions/second, making real-time-verifiable databases practical.

Table IV also reports PipeZK's execution times, and NoCap's speedups over PipeZK. NoCap again achieves large speedups, gmean 41×, ranging from 25× to 53×.

As discussed in Sec. III, these speedups come from accelerating all parts of the protocol (unlike PipeZK, which leaves part of the proof to the CPU), and from using a zk-SNARK that is algorithmically more efficient (even though this efficiency does not translate into speedup on the CPU).

B. Performance breakdown

Fig. 5 shows NoCap's power breakdown per hardware component. We show the average for AES, but the breakdown and total power are essentially identical across benchmarks. Although runtime numbers change with the number of R1CS constraints (i.e., the size of each benchmark's circuit), all components scale linearly with the number of R1CS constraints for the relevant range (2^{20} to 2^{30} R1CS constraints).

We see that 13% of power goes to FUs, 44% to the register file, and 42% to high-bandwidth memory. Total power is 62 W.

Fig. 6(a) shows the runtime breakdown across tasks (Sec. V-A) of the Spartan+Orion SNARK for CPU and NoCap. Like for power, we do not show per-benchmark breakdowns because the breakdowns look identical in this range of circuit sizes. We see that both CPU and NoCap spend the majority of runtime (∼70%) in the sumcheck protocol. The remaining time is spent on Reed-Solomon codes (9% for NoCap, 19% for CPU), polynomial arithmetic (12% for NoCap, 6% for CPU), building Merkle trees (5% for NoCap, 3% for CPU), and running sparse matrix vector multiply (0.5% for NoCap, 2% for CPU). The CPU breakdown shows that all tasks must be

Proving time on	NoCap	CPU	vs. CPU	PipeZK	vs. PipeZK
AES	151.3 ms	94.2 s	622×	8.0 s	53×
SHA	311.0 ms	188.4 s	605×	16.0 s	51×
RSA	1.3 s	753.6 s	578×	49.1 s	37×
Litmus	2.6 s	1,507.2 s	571×	134.6 s	50×
Auction	10.8 s	1.7 h	560×	275.8 s	25×
gmean			**586×**		**41×**

TABLE IV: Proof generation time for NoCap, CPU (running Spartan+Orion) and PipeZK, and NoCap's speedups over CPU and PipeZK.

Fig. 5: Power breakdown for NoCap across FUs, reg. file, and memory (HBM) for a 16M-constraint statement.

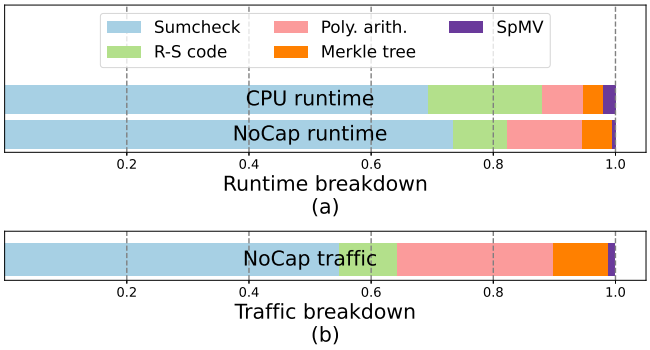

Fig. 6: Breakdown of (a) CPU and NoCap runtime, and (b) NoCap memory traffic across tasks.

accelerated to achieve high speedups. The NoCap breakdown shows that NoCap accelerates these tasks well.

Fig. 6(b) shows the breakdown of memory traffic by task for NoCap. While we recompute sumcheck values multiple times to reduce memory traffic (Sec. V), Fig. 6 shows that sumcheck's traffic is still dominant (55%), while 25% goes to polynomial arithmetic, 9% to building Merkle trees and Reed-Solomon codes each, and 1% to sparse matrix vector multiply. All tasks achieve limited reuse, which leads to traffic breakdown roughly matching that of runtime. Overall utilization of compute resources is 60%.

C. Effect of protocol optimizations

So far, we have evaluated a combination of Spartan+Orion that includes three new optimizations over the original implementations: using the narrower Goldilocks64 field, using Reed-Solomon codes, and using recomputation in sumcheck (Sec. V). Our software version uses the first two optimizations, while NoCap uses all three.

In the CPU version, switching to the narrower field improves performance by 1.7×, and using Reed-Solomon codes instead of an expander graph improves CPU performance by a further 1.2×. Recomputing sumcheck inputs improves NoCap's performance by 1.1× (and sumcheck performance by 31%), but it slightly hurts performance on the CPU (by 1%), because the CPU is not memory-bound. This is why we leave recomputation

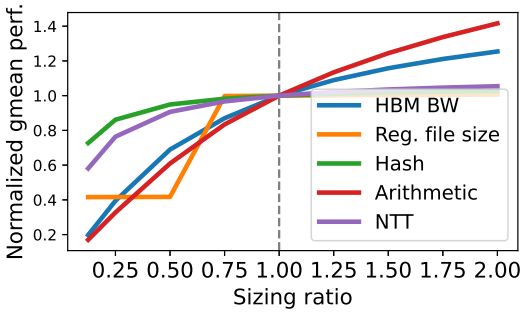

Fig. 7: NoCap parameter sensitivity study.

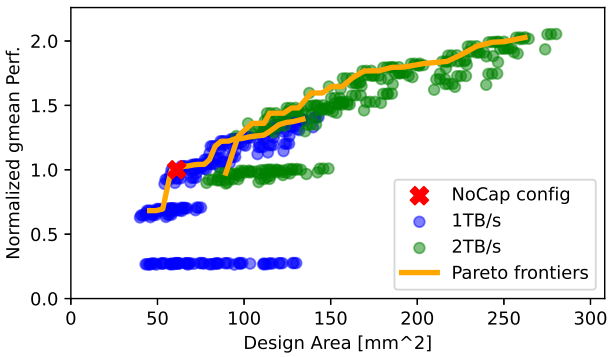

Fig. 8: NoCap's design space along with the Pareto frontier in orange and the chosen NoCap configuration.

off in the CPU version. Overall, these improvements yield a 2.1× speedup on the CPU.

D. Sensitivity study

Fig. 7 illustrates the relative importance of each hardware building block on NoCap's gmean performance on our benchmarks. Specifically, Fig. 7 sweeps the compute throughput of each FU individually: hash FU, arithmetic (i.e., add and multiply FU), and NTT FU. We also scale the HBM memory bandwidth and register file size.

We see that performance is most sensitive to raw arithmetic throughput. Other FUs contribute significantly to performance *up to* our chosen parameters for NoCap.

Increasing register file size has negligible impact on performance, but decreasing register file size leads intermediates in the sumcheck recomputations described in Sec. V to spill, drastically degrading performance.

Fig. 7 also shows that we balanced the choice of hardware parameters well: scaling any one building block brings small benefits, while reducing performance of any one of them quickly leads to performance degradation.

E. Design space

Fig. 8 shows how NoCap's performance changes with area: we sweep on-chip storage size, and throughputs of all the FUs independently. We see two different scatter plots: one for 1 TB/s HBM (blue) and one for 2 TB/s (green). Fig. 8 also shows the performance vs. area Pareto frontier for both the 1 TB/s and 2 TB/s HBM. We see that our chosen configuration makes a good tradeoff between performance and area, because the Pareto curve flattens out for areas larger than that we chose.

	Prover	Send	Verifier	Total	vs. PipeZK
AES	0.1	0.8	0.1	1.1	7.4×
SHA	0.3	0.9	0.2	1.3	12.1×
RSA	1.3	1.0	0.2	2.5	19.6×
Litmus	2.6	1.1	0.2	4.0	34.1×
Auction	10.8	1.2	0.3	12.3	22.4×
gmean					**16.8×**

TABLE V: Per-benchmark end-to-end runtime in seconds for NoCap along with end-to-end speedups vs. PipeZK.

F. End-to-end performance

Table V shows end-to-end runtime for NoCap, with speedups versus PipeZK. Table V is similar to Table I: It shows proof generation (Prover), time to download the proof assuming a 10 MB/s link (Send) and verification time (Verifier); the end-to-end total runtime is set to the sum of these three. Speedups versus PipeZK refer to the end-to-end runtime speedups.

While Spartan+Orion's proofs and verification times are larger than Groth16 (Sec. III), we see that NoCap's fast proof generation more than makes up for the difference. Speedups are high across applications, and grow the larger the circuit is, as proof generation time becomes more dominant.

IX. Additional Related Work

A. Prior work on ZKP acceleration

PipeZK [92] is a ZKP accelerator ASIC, and GZKP [52] is a GPU ZKP implementation. Both target the Groth16 SNARK [39], which, as we have shown in Sec. III and Sec. VIII has limited acceleration potential. Specifically, Sec. VIII shows that PipeZK is 41× slower than NoCap.

B. Alternative hardware platforms

As we saw, prior work has also implemented key building blocks of hash-based ZKPs on GPUs and FPGAs [1, 51, 57], but neither platform can approach NoCap's performance, due to bottlenecks that we explain below.

GPUs do not offer sufficient modular arithmetic throughput [23, 27, 34, 38, 47]: Prior work shows that GPUs can implement a large 2^{24}-element Goldilocks 64 NTT at a throughput of at most 125 GB/s [58]. This implementation saturates the GPU's compute resources, indicating it can perform about 200 Goldilocks 64 multiply-adds per cycle. This is 10× off of NoCap's multiply-add bandwidth. Looking at end-to-end benchmarks, assuming linear scaling (which is generous), GZKP [52] would run the Auction benchmark in 513s, or 47.5× slower than NoCap.

FPGAs do not have sufficient on-chip logic to implement all of Spartan+Orion's components at sufficient throughput. For example, a high-end Xilinx Alveo U55C FPGA Card [87] would have 1,000 multipliers exhaust its hardware resources, while running at a frequency that is at least 3× off NoCap's [80].

C. Related accelerators

Recent accelerators for fully homomorphic encryption (FHE) [45, 46, 48, 67, 68] achieve high speedups over CPUs by leveraging similar principles to NoCap: modular vector arithmetic, NTT, static dataflow scheduling, and a focus on reducing data movement.

While NoCap has similarities with these designs, it also has significant differences. It requires new FUs (like the Benes network); it lacks FUs that are common in FHE accelerators (e.g., change-of-RNS-base and automorphisms); it uses heterogeneously sized FUs; it uses wider modular integers with a fixed field (Goldilocks64, whereas several of these accelerators use 28–36-bit modular integers); and it is a much smaller chip due to the memory-bound nature and reuse opportunities in hash-based ZKPs, which are quite different from FHE algorithms.

Ultimately, NoCap shares similarities with FHE accelerators because FHE and hash-based ZKPs both consist of computations on large polynomials with modular integer coefficients. This suggests that a future architecture could integrate these mechanisms to efficiently accelerate ZKPs, FHE, and other cryptographic protocols that rely on the same primitives.

X. Conclusion and Future Work

ZKPs enable cryptographically verifying both the correctness of computation and the knowledge of data that has arbitrary properties. But the large compute overheads of ZKP proof generation constrain its applicability to narrow domains. NoCap addresses this challenge by accelerating ZKP by three orders of magnitude. Our key contribution is to leverage hardware-algorithm co-design by exploring a class of ZKPs that perform similarly to conventional ones on a CPU, but are much more amenable to acceleration. These speedups enable new use cases, such as real-time verifiable databases.

Even with NoCap, ZKPs are still costly, and additional acceleration will unlock more use cases. We believe a combination of hardware architecture and algorithmic techniques will be needed to achieve substantial speedups. Specifically, recursive [19], incremental [25], or folding-based [49] zero-knowledge proofs are exciting recent developments that can enable large performance and scalability gains. A simplified view is that these schemes break a computation into several parts, and create a small individual proof for each. Then, a final proof shows that all of the individual proofs are correct, thus verifying the whole computation. These techniques would allow designing accelerators that are less memory-bound because they target small individual proofs, achieving higher throughput than NoCap cheaply, while still scaling to very large programs. Moreover, large proofs could be parallelized across many accelerators, with little communication among them, which would enable rack-scale ZKP accelerator systems.

Acknowledgments

We thank Elaine Shi for suggesting that we look into Orion and Spartan. We thank Aleksandar Krastev, Axel Feldmann, Courtney Golden, Fares Elsabbagh, Hyun Ryong Lee, Maggie Du, Yifan Yang, Joel Emer, and our anonymous reviewers for their helpful feedback. This work was partially supported by NSF grant 2330065 and by a Wistron research grant.

REFERENCES

[1] "FPGA SNARK Prover," https://github.com/bsdevlin/fpga_snark_prover, 2020.

[2] "Aztec network," https://aztec.network/, 2023.

[3] "PCI-SIG Specifications," https://pcisig.com/specifications, 2023.

[4] "StarkWare," https://starkware.co/, 2023.

[5] K. Aasaraai, E. Cesena, R. Maganti, N. Stalder, J. Varela, and K. Bowers, "CycloneNTT: An NTT/FFT architecture using quasi-streaming of large datasets on DDR-and HBM-based FPGA platforms," *Cryptology ePrint Archive*, 2022.

[6] M. R. Albrecht, V. Cini, R. W. Lai, G. Malavolta, and S. A. Thyagarajan, "Lattice-based SNARKs: Publicly verifiable, preprocessing, and recursively composable," in *Proc. of the Annual International Cryptology Conference (CRYPTO)*, 2022.

[7] S. Ames, C. Hazay, Y. Ishai, and M. Venkitasubramaniam, "Ligero: Lightweight sublinear arguments without a trusted setup," in *Proc. of the 2017 ACM SIGSAC conf. on Computer and Communications Security (CCS)*, 2017.

[8] D. H. Bailey, "FFTs in external of hierarchical memory," in *Proc. of the 1989 ACM/IEEE conference on Supercomputing*, 1989.

[9] G. Barany, "Register reuse scheduling," in *9th Workshop on Optimizations for DSP and Embedded Systems (ODES-9)*, 2011.

[10] N. Bell and M. Garland, "Implementing sparse matrix-vector multiplication on throughput-oriented processors," in *Proc. of the conf. on high performance computing networking, storage and analysis (SC)*, 2009.

[11] E. Ben-Sasson, I. Bentov, Y. Horesh, and M. Riabzev, "Fast reed-solomon interactive oracle proofs of proximity," in *Proc. of the 45th intl. colloquium on automata, languages, and programming (ICALP)*, 2018.

[12] E. Ben-Sasson, I. Bentov, Y. Horesh, and M. Riabzev, "Scalable, transparent, and post-quantum secure computational integrity," *Cryptology ePrint Archive*, 2018.

[13] E. Ben-Sasson, A. Chiesa, D. Genkin, E. Tromer, and M. Virza, "SNARKs for C: Verifying program executions succinctly and in zero knowledge," in *Proc. of the 33rd Annual Cryptology Conference (CRYPTO)*, 2013.

[14] V. E. Beneš, *Mathematical theory of connecting networks and telephone traffic*. Academic Press, 1965.

[15] G. Bertoni, J. Daemen, M. Peeters, and G. Van Assche, "Keccak specifications," *Submission to NIST (round 2)*, vol. 3, no. 30, 2009.

[16] G. Bertoni, J. Daemen, M. Peeters, G. Van Assche, R. Van Keer, and B. Viguier, "Kangaroo twelve: Fast hashing based on keccak," in *Proc. of the 16th intl. conf. on Applied Cryptography and Network Security (ACNS)*, 2018.

[17] N. Bitansky, R. Canetti, A. Chiesa, and E. Tromer, "From extractable collision resistance to succinct non-interactive arguments of knowledge, and back again," in *Proc. of the 3rd Innovations in Theoretical Computer Science Conference*, 2012.

[18] G. E. Blelloch, P. B. Gibbons, and Y. Matias, "Provably efficient scheduling for languages with fine-grained parallelism," *Journal of the ACM (JACM)*, vol. 46, no. 2, 1999.

[19] D. Boneh, J. Drake, B. Fisch, and A. Gabizon, "Halo infinite: Recursive zk-SNARKs from any additive polynomial commitment scheme," *Cryptology ePrint Archive*, 2020.

[20] J. Bootle, A. Chiesa, and J. Groth, "Linear-time arguments with sublinear verification from tensor codes," in *Proc. of the 18th intl. conf. on the Theory of Cryptography (TCC)*, 2020.

[21] B. Bünz, J. Bootle, D. Boneh, A. Poelstra, P. Wuille, and G. Maxwell, "Bulletproofs: Short proofs for confidential transactions and more," in *Proc. of the 2018 IEEE symp. on Security and Privacy (SP)*, 2018.

[22] B. Chen, B. Bünz, D. Boneh, and Z. Zhang, "Hyperplonk: Plonk with linear-time prover and high-degree custom gates," in *Proc. of the Annual International Conference on the Theory and Applications of Cryptographic Techniques (EUROCRYPT)*, 2023.

[23] L. Chen, S. Covanov, D. Mohajerani, and M. Moreno Maza, "Big prime field FFT on the GPU," in *Proc. of the 2017 ACM International Symposium on Symbolic and Algebraic Computation (ISSAC)*, 2017.

[24] S. Chen, J. H. Cheon, D. Kim, and D. Park, "Verifiable computing for approximate computation," *Cryptology ePrint Archive*, 2019.

[25] W. Chen, A. Chiesa, E. Dauterman, and N. P. Ward, "Reducing participation costs via incremental verification for ledger systems," *Cryptology ePrint Archive*, 2020.

[26] CoinDesk, "Crypto startup fabric systems raises $13m seed round to provide blockchain hardware," https://www.coindesk.com/business/2022/10/13/crypto-startup-fabric-systems-raises-13m-seed-round-to-provide-blockchain-hardware/, 2022.

[27] W. Dai and B. Sunar, "cuHE: A homomorphic encryption accelerator library," in *Proc. of the 2nd intl. conf. on Cryptography and Information Security in the Balkans (BalkanCryptSec)*, 2016.

[28] S. Dasgupta, T. Singh, A. Jain, S. Naffziger, D. John, C. Bisht, and P. Jayaraman, "Radeon RX 5700 Series: The AMD 7nm Energy-Efficient High-Performance GPUs," in *Proc. of the IEEE Intl. Solid-State Circuits Conf. (ISSCC)*, 2020.

[29] M. Emami, S. Kashani, K. Kamahori, M. S. Pourghannad, R. Raj, and J. R. Larus, "Manticore: Hardware-accelerated RTL simulation with static bulk-synchronous parallelism," 2023.

[30] Z. Fang, D. Darais, J. P. Near, and Y. Zhang, "Zero knowledge static program analysis," in *Proc. of the 2021 ACM SIGSAC Conference on Computer and Communications Security (CCS)*, 2021.

[31] J. A. Fisher, "Very long instruction word architectures and the ELI-512," in *Proc. of the 10th annual Intl. Symp. on Computer Architecture (ISCA-10)*, 1983.

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

[33] H. S. Galal and A. M. Youssef, "Verifiable sealed-bid auction on the ethereum blockchain," in *International Workshops on Financial Cryptography and Data Security (FC)*, 2018.

[34] J.-Z. Goey, W.-K. Lee, B.-M. Goi, and W.-S. Yap, "Accelerating number theoretic transform in gpu platform for fully homomorphic encryption," *The Journal of Supercomputing*, vol. 77, 2021.

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

[36] A. Golovnev, J. Lee, S. Setty, J. Thaler, and R. S. Wahby, "Brakedown: Linear-time and field-agnostic SNARKs for R1CS," in *Proc. of the Annual International Cryptology Conference (CRYPTO)*, 2023.

[37] J. R. Goodman and W.-C. Hsu, "Code scheduling and register allocation in large basic blocks," in *Proc. of the Intl. Conf. on Supercomputing (ICS'88)*, 1988.

[38] N. K. Govindaraju, B. Lloyd, Y. Dotsenko, B. Smith, and J. Manferdelli, "High performance discrete fourier transforms on graphics processors," in *SC'08: Proceedings of the 2008 ACM/IEEE conference on Supercomputing*, 2008.

[39] J. Groth, "On the size of pairing-based non-interactive arguments," in *Proc. of the 35th Annual International Conference on the Theory and Applications of Cryptographic Techniques (EUROCRYPT)*, 2016.

[40] M. Hamburg, "Ed448-goldilocks, a new elliptic curve," *Cryptology ePrint Archive*, 2015.

[41] F. Hirner, A. C. Mert, and S. S. Roy, "PROTEUS: A tool to generate pipelined number theoretic transform architectures for FHE and ZKP applications," *Cryptology ePrint Archive*, 2023.

[42] C.-C. Huang, A. Cidon, I. Cetindil, I. Zhang, J. Li, R. Agarwal, and C. Kozyrakis, "YCSB++: Benchmarking and performance debugging advanced features in scalable table stores," in *Proc. of the 2nd ACM Symposium on Cloud Computing (SoCC)*, 2012.

[43] Y. Ishai, H. Su, and D. J. Wu, "Shorter and faster post-quantum designated-verifier zkSNARKs from lattices," in *Proc. of the 2021 ACM SIGSAC Conference on Computer and Communications Security (CCS)*, 2021.

[44] A. Kate, G. M. Zaverucha, and I. Goldberg, "Polynomial commitments," University of Waterloo, Tech. Rep., 2010.

[45] J. Kim, S. Kim, J. Choi, J. Park, D. Kim, and J. H. Ahn, "SHARP: A short-word hierarchical accelerator for robust and practical fully homomorphic encryption," in *Proc. of the 50th annual Intl. Symp. on Computer Architecture (ISCA-50)*, 2023.

[46] J. Kim, G. Lee, S. Kim, G. Sohn, M. Rhu, J. Kim, and J. H. Ahn, "ARK: Fully homomorphic encryption accelerator with runtime data generation and inter-operation key reuse," in *Proc. of the 55th annual IEEE/ACM intl. symp. on Microarchitecture (MICRO-55)*, 2022.

[47] S. Kim, W. Jung, J. Park, and J. H. Ahn, "Accelerating number theoretic transformations for bootstrappable homomorphic encryption on GPUs," in *Proc. of the IEEE Intl. Symp. on Workload Characterization (IISWC)*, 2020.

[48] S. Kim, J. Kim, M. J. Kim, W. Jung, J. Kim, M. Rhu, and J. H. Ahn, "BTS: An accelerator for bootstrappable fully homomorphic encryption," in *Proc. of the 49th annual Intl. Symp. on Computer Architecture (ISCA-49)*, 2022.

[49] A. Kothapalli, S. Setty, and I. Tzialla, "Nova: Recursive zero-knowledge arguments from folding schemes," in *Proc. of the Annual International Cryptology Conference (CRYPTO)*, 2022.

[50] T. Liu, X. Xie, and Y. Zhang, "zkCNN: Zero knowledge proofs for convolutional neural network predictions and accuracy," in *Proc. of the 2021 ACM SIGSAC Conference on Computer and Communications Security (CCS)*, 2021.

[51] T. Lu, C. Wei, R. Yu, C. Chen, W. Fang, L. Wang, Z. Wang, and W. Chen, "cuZK: Accelerating zero-knowledge proof with a faster parallel multi-scalar multiplication algorithm on GPUs," *Cryptology ePrint Archive*, 2022.

[52] W. Ma, Q. Xiong, X. Shi, X. Ma, H. Jin, H. Kuang, M. Gao, Y. Zhang, H. Shen, and W. Hu, "GZKP: A GPU accelerated zero-knowledge proof system," in *Proc. of the 28th intl. conf. on Architectural Support for Programming Languages and Operating Systems (ASPLOS-XXVIII)*, 2023.

[53] L. Marchal, B. Simon, and F. Vivien, "Limiting the memory footprint when dynamically scheduling dags on shared-memory platforms," *Journal of Parallel and Distributed Computing*, vol. 128, 2019.

[54] R. C. Merkle, "A digital signature based on a conventional encryption function," in *Conference on the theory and application of cryptographic techniques*, 1987.

[55] Microsoft, "Spartan: High-speed zkSNARKs without trusted setup," https://github.com/microsoft/Spartan, 2022.

[56] Mina Protocol, "Decentralized, scalable and secure blockchain," https://minaprotocol.com/, 2023.

[57] N. Ni and Y. Zhu, "Enabling zero knowledge proof by accelerating zk-SNARK kernels on GPU," *Journal of Parallel and Distributed Computing*, vol. 173, 2023.

[58] A. Ş. Özcan and E. Savaş, "Two algorithms for fast gpu implementation of ntt," *Cryptology ePrint Archive*, 2023.

[59] E. Ozer, S. Banerjia, and T. M. Conte, "Unified assign and schedule: A new approach to scheduling for clustered register file microarchitectures," in *Proc. of the 31st annual IEEE/ACM intl. symp. on Microarchitecture (MICRO-31)*, 1998.

[60] B. Parno, J. Howell, C. Gentry, and M. Raykova, "Pinocchio: Nearly practical verifiable computation," *Communications of the ACM*, vol. 59, no. 2, 2016.

[61] M. Pellauer, Y. S. Shao, J. Clemons, N. Crago, K. Hegde, R. Venkatesan, S. W. Keckler, C. W. Fletcher, and J. Emer, "Buffets: An efficient and composable storage idiom for explicit decoupled data orchestration," in *Proc. of the 24th intl. conf. on Architectural Support for Programming Languages and Operating Systems (ASPLOS-XXIV)*, 2019.

[62] Polygon Hermez, "Scalable payments: Decentralised by design, open for everyone," https://hermez.io/.

[63] Polygon Labs, "The go fast machine: Adding recursion to Polygon zkEVM," https://polygon.technology/blog/the-go-fast-machine-adding-recursion-to-polygon-zkevm.

[64] E. Qin, A. Samajdar, H. Kwon, V. Nadella, S. Srinivasan, D. Das, B. Kaul, and T. Krishna, "SIGMA: A sparse and irregular GEMM accelerator with flexible interconnects for DNN training," in *Proc. of the 26th IEEE intl. symp. on High Performance Computer Architecture (HPCA-26)*, 2020.

[65] Rambus Inc., "White paper: HBM2E and GDDR6: Memory solutions for AI," 2020.

[66] I. S. Reed and G. Solomon, "Polynomial codes over certain finite fields," *Journal of the society for industrial and applied mathematics*, vol. 8, no. 2, 1960.

[67] N. Samardzic, A. Feldmann, A. Krastev, S. Devadas, R. Dreslinski, C. Peikert, and D. Sanchez, "F1: A fast and programmable accelerator for fully homomorphic encryption," in *Proc. of the 54th annual IEEE/ACM intl. symp. on Microarchitecture (MICRO-54)*, 2021.

[68] N. Samardzic, A. Feldmann, A. Krastev, N. Manohar, N. Genise, S. Devadas, K. Eldefrawy, C. Peikert, and D. Sanchez, "CraterLake: a hardware accelerator for efficient unbounded computation on encrypted data." in *Proc. of the 49th annual Intl. Symp. on Computer Architecture (ISCA-49)*, 2022.

[69] SCIPR Lab, "libsnark: a c++ library for zksnark proofs," https://github.com/scipr-lab/libsnark, 2017.

[70] S. Setty, "Spartan: Efficient and general-purpose zkSNARKs without trusted setup," in *Proc. of the Annual International Cryptology Conference (CRYPTO)*, 2020.

[71] S. Setty and J. Lee, "Quarks: Quadruple-efficient transparent zkSNARKs," *Cryptology ePrint Archive*, 2020.

[72] A. S. Shamsabadi, G. Tan, T. I. Cebere, A. Bellet, H. Haddadi, N. Papernot, X. Wang, and A. Weller, "Confidential-DPproof: Confidential proof of differentially private training," in *Proc. of the 12th International Conference on Learning Representations (ICLR)*, 2024.

[73] D. A. Spielman, "Linear-time encodable and decodable error-correcting codes," in *Proceedings of the twenty-seventh annual ACM symposium on Theory of computing*, 1995.

[74] Sunblaze-ucb, "Orion: Security via side-channel assisted reverse engineering," https://github.com/sunblaze-ucb/Orion, 2022.

[75] V. Sze, Y.-H. Chen, T.-J. Yang, and J. S. Emer, "Efficient processing of deep neural networks: A tutorial and survey," *Proceedings of the IEEE*, vol. 105, no. 12, 2017.

[76] J. Thaler, "Time-optimal interactive proofs for circuit evaluation," in *Proc. of the 33rd Annual Cryptology Conference (CRYPTO)*, 2013.

[77] The Block, "Ethereum scaling startup scroll closes $50 million funding round at $1.8 billion valuation," https://www.theblock.co/post/217340/ethereum-scaling-scroll-50-million-funding-round-1-8-billion-valuation, 2022.

[78] J. E. Thornton, "The CDC 6600 project," *Annals of the History of Computing*, vol. 2, no. 4, 1980.

[79] Ulvetanna, "Accelerating the zero-knowledge revolution," 2023. [Online]. Available: https://www.ulvetanna.io/

[80] Ulvetanna, "FPGA architecture for Goldilocks NTT," 2023. [Online]. Available: https://www.ulvetanna.io/news/fpga-architecture-for-goldilocks-ntt

[81] A. Vlasov and K. Panarin, "Transparent polynomial commitment scheme with polylogarithmic communication complexity," *Cryptology ePrint Archive*, 2019.

[82] R. S. Wahby, I. Tzialla, A. Shelat, J. Thaler, and M. Walfish, "Doubly-efficient zkSNARKs without trusted setup," in *Proc. of the 2018 IEEE symp. on Security and Privacy (SP)*, 2018.

[83] C. Weng, K. Yang, X. Xie, J. Katz, and X. Wang, "Mystique: Efficient conversions for Zero-Knowledge proofs with applications to machine learning," in *Proc. of the 30th USENIX Security Symposium (USENIX Security 21)*, 2021.

[84] Y. Xia, X. Yu, M. Butrovich, A. Pavlo, and S. Devadas, "Litmus: Towards a practical database management system with verifiable acid properties and transaction correctness," in *Proc. of the 2022 intl. conf. on Management of Data (SIGMOD)*, 2022.

[85] T. Xie, J. Zhang, Y. Zhang, C. Papamanthou, and D. Song, "Libra: Succinct zero-knowledge proofs with optimal prover computation," in *Proc. of the Annual International Cryptology Conference (CRYPTO)*, 2019.

[86] T. Xie, Y. Zhang, and D. Song, "Orion: Zero knowledge proof with linear prover time," in *Proc. of the Annual International Cryptology Conference (CRYPTO)*, 2022.

[87] Xilinx, "Alveo U55C data center accelerator card," 2023. [Online]. Available: https://www.xilinx.com/products/boards-and-kits/alveo/u55c.html

[88] Yahoo Finance, "Alan howard-backed cryptography investor raises new fund," https://finance.yahoo.com/news/alan-howard-backed-cryptography-investor-110000056.html, 2022.

[89] Zcash, "Zcash is cash for the new age," https://z.cash/, 2023.

[90] J. Zhang, Z. Fang, Y. Zhang, and D. Song, "Zero knowledge proofs for decision tree predictions and accuracy," in *Proc. of the 2020 ACM SIGSAC Conference on Computer and Communications Security (CCS)*, 2020.

[91] J. Zhang, T. Xie, Y. Zhang, and D. Song, "Transparent polynomial delegation and its applications to zero knowledge proof," in *Proc. of the 2020 IEEE symp. on Security and Privacy (SP)*, 2020.

[92] Y. Zhang, S. Wang, X. Zhang, J. Dong, X. Mao, F. Long, C. Wang, D. Zhou, M. Gao, and G. Sun, "PipeZK: Accelerating zero-knowledge proof with a pipelined architecture," in *Proc. of the 48th annual Intl. Symp. on Computer Architecture (ISCA-48)*, 2021.

[93] Y. Zhang, D. Genkin, J. Katz, D. Papadopoulos, and C. Papamanthou, "vSQL: Verifying arbitrary SQL queries over dynamic outsourced databases," in *Proc. of the 2017 IEEE symp. on Security and Privacy (SP)*, 2017.

[94] Z. Zhao and T.-H. H. Chan, "How to vote privately using bitcoin," in *Proc. of the 17th Intl. Conf. on Information and Communications Security (ICICS)*, 2016.