IACR Transactions on Cryptographic Hardware and Embedded Systems
ISSN 2569-292, Vol. 2025, No. 4, pp. 1–45. DOI:XXXXXXXX

Accelerating Hash-Based Polynomial Commitment Schemes with Linear Prover Time

Florian Hirner[†], Florian Krieger[†], Constantin Piber and Sujoy Sinha Roy

Institute of Information Security, Graz University of Technology, Austria
firstname.lastname@tugraz.at

Abstract. Zero-knowledge proofs (ZKPs) are cryptographic protocols that enable one party to prove the validity of a statement without revealing any information beyond its truth. Central building blocks in many ZKPs are polynomial commitment schemes (PCS) where constructions with *linear-time provers* are especially attractive. Two such examples are Brakedown and its extension Orion, which enable linear-time and quantum-resistant proving by leveraging linear-time encodable Spielman codes. However, these PCS operate over large datasets, creating significant computational bottlenecks. For example, committing to and proving a degree 2^{28} polynomial requires around 1.1 GB of data while taking 463 seconds on a high-end server CPU.

This work addresses the performance bottleneck in Orion-like PCS by optimizing their most critical operations: Spielman encoding and Merkle commitments. These operations involve Gigabytes of data and suffer from random off-chip memory access patterns that drastically reduce off-chip bandwidth. We resolve this issue and introduce *inverted expander graphs* to eliminate random writes and reduce off-chip memory accesses by over 50%. Additionally, we propose an on-the-fly graph sampling method that avoids streaming large auxiliary data by generating expander graphs dynamically on-chip. We also provide a formal security proof for our proposed graph transformation. Beyond encoding, we accelerate Merkle Tree construction over large data sets through a scalable multi-pass SHA3 pipeline. Finally, we reutilize existing hardware components used in commitment to accelerate the so-called proximity and consistency checks during proof generation.

Building upon these concepts, we present the first hardware architecture for PCS – with linear prover time – on an Xilinx Alveo U280 FPGA. In addition, we discuss the practical challenges of manually partitioning, placing, and routing our large-scale architecture to efficiently map it to the multi-SLR and HBM-equipped FPGA. The final implementation achieves a speedup of two orders of magnitude for full proof generation, covering commitment and proving steps. When combined with Virgo as an outer CP-SNARK protocol, our accelerator reduces end-to-end latency by up to 3.85× – close to the theoretical maximum of 3.9×.

Keywords: Zero-Knowledge Proof · Orion · Brakedown · Spielman Code · FPGA

1 Introduction

Zero-knowledge proofs (ZKP) enable one party, known as the *prover*, to demonstrate to another party, known as the *verifier*, that a given statement is true without revealing any information about why the statement is true or any private data that was used to prove it. As an illustration, the prover can convince the verifier that it knows a private witness w for a public input x such that $\mathcal{C}(x, w) = 0$ is satisfied for a circuit $\mathcal{C}$, all without revealing any

[†] These authors contributed equally to this work.

information about w. The concept of ZKP systems was first introduced by Goldwasser, Micali, and Rackoff in the 1980s in their seminal paper [GMR85].

ZKPs have seen growing use in recent years, with their real-life applications expected to expand further. One notable application area is verifiable computation, where a client delegates computation to a powerful but untrusted server. Subsequently, the client can easily verify (without re-doing the computation) whether the server executed the computation correctly or not. Concrete realizations of verifiable machine learning using ZKP are presented in [LXZ21, WYX$^+$21]. ZKPs are extensively used in blockchains and cryptocurrencies to achieve anonymity and privacy [BSCG$^+$14]. Cryptocurrency transactions can be fully encrypted on the blockchain, yet their legitimacy can still be verified using ZKP [Fou, Pro]. Other application areas include online auction [GY18], verifiable database query [LWX$^+$23], and classical authentication systems. Recently proposed zero-knowledge proof of training framework (zkPoT) [APPK24] enables a prover to prove the correct training of a deep neuronal network on a committed dataset without disclosing any details about the model or dataset. The prover trains the model iteratively, committing to the model parameters and providing a zkPoT at each step.

Driven by the rapid development of ZKP applications, many new schemes have been recently proposed [XZS22, GLS$^+$23, CBBZ23, DP23]. One very promising direction of scheme development involves polynomial commitment schemes (PCS) with linear prover time. In such schemes, the prover can prove to the verifier that a committed polynomial was genuinely evaluated. In addition, the prover benefits from an asymptotically linear runtime concerning the huge polynomials at hand. Usually, the polynomials have between Millions and Billions of coefficients thus necessitating an efficient prover. Brakedown [GLS$^+$23] and its extension work Orion [XZS22] are highly interesting PCS with linear-time commitment and proving. Their cryptographic security is based on the preimage resistance of a cryptographic hash function, making them post-quantum secure and avoiding trusted setups. The fast prover and post-quantum security set Orion and Brakedown apart from commonly used proof systems [KZG10, Gro16, WTS$^+$18, BBB$^+$18, Lee21], which are based on pairings or discrete logarithm assumptions. The advantages of Orion and Brakedown attract broad research, mostly focusing on their cryptographic and protocol-level properties. Recent works [dHS24] improve the soundness guarantees of the Orion scheme and present the closely related Scorpius scheme. Other efforts [CBBZ23] propose new schemes re-using the linear-time encoding initially proposed in Brakedown. This underlines the relevance of Orion and Brakedown-like schemes as a polynomial commitment scheme in contemporary research. In our paper, we take Orion as a case study for post-quantum secure linear-prover-time PCS due to Orion's small proof size. However, our concepts and contributions directly apply to Brakedown and other related works as well.

Motivation for hardware acceleration of Orion [XZS22]: Several works in the literature have accelerated pairing and discrete logarithm-based proof systems. However, hardware accelerating Orion (which was proposed in 2022) using FPGAs has not gained broad attention. Although Orion's commitment and proving mechanisms have linear asymptotic complexity, reaching concrete efficiency and practical performance is hard. This is due to the Gigabytes of data involved in the prover's complex computations, which reflects in low software performance [Su]. At the same time, Orion as a PCS is repeatedly used as a building block in larger ZKP applications (e.g., zkPoT [APPK24]). Hence, the commitment and proving steps of Orion are invoked multiple times within a single execution of the ZKP application. These frequent invocations require performant polynomial commitment and proving mechanisms.

A detailed timing analysis of Orion's software implementation [Su] (presented in Section 3.3) shows that the most demanding part of Orion is the polynomial commitment phase, involving linear error-correcting Spielman codes and large expander graphs. The

Spielman encoding operates over Gigabytes of data, necessitating off-chip memory storage. Even more critically, Spielman codes access the data in off-chip memory in a non-optimal random pattern, which degrades the memory bandwidth. This challenge was also identified by prior work [SLDS24], but no solution has been presented so far. Furthermore, the handling of large expander graphs within the encoding is challenging. In typical implementations, these graphs require considerable storage of several hundred Megabytes or even Gigabytes. Hence, the high demand for off-chip memory is a critical aspect of Orion and requires a memory-aware hardware design.

Another challenge in Orion's commitment is the generation of Merkle Trees. Constructing these trees quickly demands massive hashing throughput. In addition to the commitment step, the proving mechanism in Orion's software is also slow. Proving contains so-called proximity and consistency checks which compute random linear combinations of the large Spielman codewords. Subsequently, Orion Merkle-commits to the linear combinations causing a total latency of up to several seconds.

Given the performance bottlenecks in Orion, there is a strong need to explore optimization strategies that can accelerate the scheme. This paper addresses the three key operations within commitment and proving for optimization: (1) improving the performance of linear Spielman encoding by lowering the costly random off-chip memory accesses and by on-the-fly expander graph sampling, (2) optimizing the generation of Merkle Trees for faster commitment and proving in Orion, (3) accelerating the proximity and consistency checks within proving. By focusing on these core aspects, we significantly reduce the latency of the Orion PCS. Therefore, our methodology enhances the scalability and practicality of linear-prover-time PCS – such as Orion or Brakedown – for real-world applications.

1.1 Contributions

We present a series of novel memory-aware optimizations to enhance the performance of Orion and Orion-like PCS. Our contributions are:

- **Reduced off-chip memory accesses in Spielman encoding:** We propose *inverted expander graphs* which relax the critical demand on off-chip memory bandwidth during linear encoding. Compared to regular expander graphs, inverted expanders save roughly 50% of off-chip memory accesses, which allows a significant improvement of computational performance. Moreover, problematic random writes are avoided entirely, and overheads due to read-write-turnarounds are reduced.
- **On-the-fly graph sampling:** The baseline version of both regular and inverted expander graphs requires storing the graph structure. In case of $N = 2^{28}$, the graph structure consumes up to 1.1 Gigabytes in off-chip memory. The overall overhead is even higher when storing the graph structure of multiple N, i.e., a range of $N = 2^{16}$ to 2^{28}. Loading this large data would degrade the effective off-chip bandwidth during linear encoding. We hence present an on-the-fly graph sampling method and a novel postprocessing technique that avoids off-chip storage of the graph structure. In addition, we extend Brakedown's proof [GLS+21] to show that our on-the-fly sampled inverted expander graphs do not compromise code distance.
- **Efficient Merkle Tree generation:** We develop a pipelined method for data rearrangement and constructing Merkle Trees, which schedules interleaved hashing and Merkle Tree construction. This accelerates the computational overhead in the commitment and proving phases.
- **Accelerated proximity and consistency checks:** We accelerate the proximity and consistency checks within proving by reutilizing the linear encoding datapath. In addition, proving requires dedicated Merkle commitments, which we compute by reusing the Merkle Tree unit. Hence, the proving step in Orion leverages existing

hardware components to lower the latency of the overall scheme.

- **FPGA implementation for Orion:** We present the first FPGA architecture for the Orion PCS. Our heterogeneous design relies on High-Bandwidth-Memory (HBM) on an Alveo U280 datacenter FPGA [Xil19] and benefits from our memory-aware design strategy. In addition, we detail our architecture partitioning to address the physical placement challenges for the multi-SLR FPGA. Our large-scale FPGA design accelerates Orion's commitment and the proving phase by up to 264× and 65× respectively.

1.2 Organization

The remainder of the paper is structured as follows: In Section 2, we provide an overview of the necessary background, including expander graphs, Spielman codes, Merkle Tree commitments, and off-chip memory details. Section 3 contains a comprehensive study of proof systems, functional commitments, and the Orion scheme. In addition, we show the similarities and differences between Orion and Brakedown. Section 4 delves into the specific challenges and our proposed solutions for efficient Spielman encodings. In Section 5, we discuss the challenges associated with Merkle Tree generation and our optimized hardware-friendly approach. Section 6 presents our overall FPGA architecture and our implementation results. Section 7 compares our results to the Orion software. Section 8 discusses the impact of this work and how other PCSs can benefit from our design. Finally, Section 9 concludes the paper.

2 Basic Background

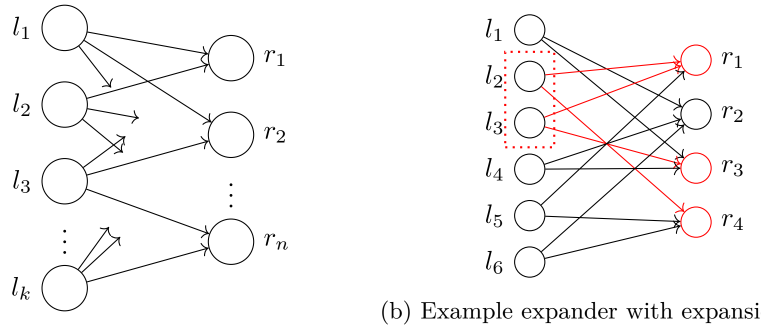
This section offers the essential background required to understand the contributions of this paper. To ensure the paper is self-contained, the background section is relatively extensive. For a more detailed explanation of Orion, including proofs and protocols, refer to the original Orion paper [XZS22].

2.1 Notation and Acronyms

Natural numbers and field elements are denoted using lowercase letters, e.g., d. Vectors are represented by $\vec{v}$, while matrices are indicated by bold capital letters, such as $\mathbf{M}$. The element from the i-th row and j-th column is $\mathbf{M}[i, j]$, and the i-th column vector is $\mathbf{M}[:, i]$. To denote the size of a set S, we use the notation $|S|$. The implementation of Orion uses an extension field $\mathbb{F}_{p^2}$, e.g., $GF((2^{61} - 1)^2)$ in the software developed by the authors [XZS22]. The elements of this extension field can be represented as a degree-1 polynomial $a + bz$ where a and b are from the base field $\mathbb{F}_p$. It is not always necessary to use an extension field if the base field is sufficiently large. We define the Hamming weight function $HW(\vec{v})$ to be the number of non-zero elements in the vector $\vec{v}$, i.e. $HW(\vec{v}) = ||\vec{v}||_0$. We use $\text{poly}(x)$ to refer to a function upper-bounded by a polynomial in variable x with a constant degree. We use the base-2 logarithm unless otherwise specified.

2.2 Graphs and expanders

Graphs in discrete mathematics are networks of points. Formally, a graph is a set of vertices and edges, where each edge is an unordered pair of vertices representing a connection. We denote this as $G = (V, E)$ with V as the vertex set and E as the edge set. Both sets are usually finite, though not necessarily always [Big02]. The *degree* of a vertex is the number of edges connected to it [Hei03].



(a) General expander structure.

(b) Example expander with expansion property highlighted.

Figure 1: Unbalanced bipartite expanders.

2.2.1 Expander graphs

In an expander graph, any small subset of vertices has a large proportion of its vertices connected to vertices outside the subset [SS94], i.e., the subset 'expands' to neighbors. Formally, a graph $G = (V, E)$ satisfies the expansion property if, for some constants $\delta > 0$ and $m > 0$, we have:

$$\forall S \subset V,\ |S| \leq m \Rightarrow |\{y \in V \setminus S : \exists x \in S \text{ such that } (x, y) \in E\}| \geq \delta|S| \qquad (1)$$

This means that for any subset S of at most m vertices, the number of distinct vertices outside S that are connected to at least one vertex in S is at least a δ-fraction of the size of S [Spi96]. Thus, the number of neighboring vertices grows proportionally with the size of the subset.

Bipartite expanders: The Orion proof system uses expander graphs that are also bipartite. In a bipartite graph, the set of vertices can be divided into two disjoint sets L and R such that there are no connections within L nor R. Thus, a vertex in L can only be connected to vertices in R and vice versa. We call L the left vertex set and R the right vertex set. We call such a bipartite graph $G = (L, R, E)$ as (c, d)-*regular* if all vertices in L have the degree c and all vertices in R have the degree d. In addition, we define the compression parameter α such that $|R| = \alpha|L|$.

Given parameters ε, δ with $0 \leq \varepsilon < 1$ and $\delta > 0$, a (c, d)-regular graph is a $(c, d, \varepsilon, \delta)$-*expander* if it upholds the expansion property mentioned in Equation 1 for an ε-fraction of the larger vertex set L or R. In Orion, we have $|L| > |R|$, thus we substitute V by L in Equation 1 and set $m = \varepsilon|L|$. The expansion property now tells us that for every subset of left vertices, there must be outgoing connections to R depending on δ. Figure 1a illustrates such an expander. It connects a left vertex set of $L = \{l_1, \dots, l_k\}$ to a right-vertex set $R = \{r_1, \dots, r_n\}$ with dense connections. In Figure 1b, an example is given with the expansion of vertices $\{l_2, l_3\}$ highlighted with $c = 2$ and $d = 3$.

2.2.2 Random Expanders

Constructing expander graphs is typically challenging. However, using a random construction, where edges are randomly placed between vertices, can yield good results with relatively low complexity [Spi96]. Thus, using randomized edges is a convenient way to construct a bipartite expander. Let L be the left vertex set and R be the right. Then, we can construct the edge set as follows. Set $E_i = \{(l_i, r_t) : r_t \text{ random vertex} \in R\}$ with $|E_i| = c$ for each vertex $l_i \in L$. That is, for vertex l_i of L, find c distinct random vertices in R to connect to. Then, the union $E = \bigcup_i E_i$ of all random edge sets allows us to

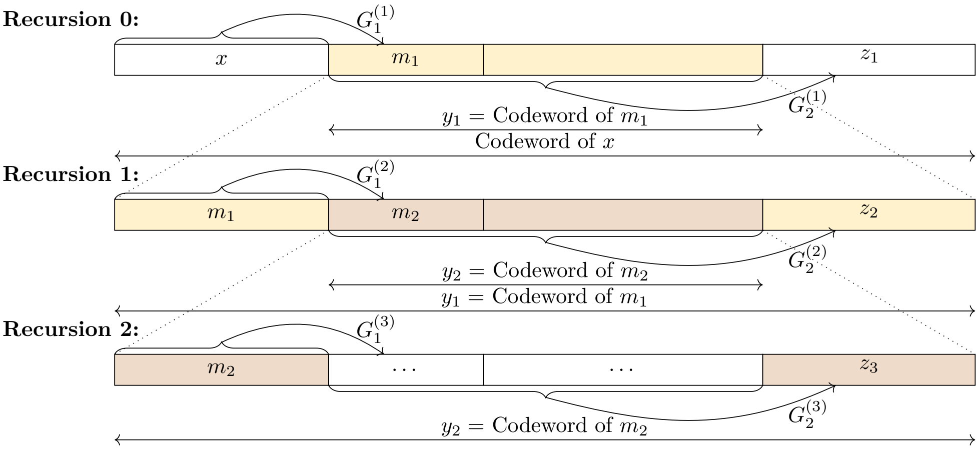


Figure 2: Illustration of the recursive Spielman encoding.

construct $G = (L \cup R, E)$. The chosen edges result in c neighbors for all vertices in L [Spi96] while the edges on R will follow a *binomial* distribution. Algorithm 2 shows the graph generation.

It is often desirable to prove that a particular graph G is a good expander. An efficient test for a good expander is provided in [XZS22]. They prove that a random construction as described above gives a good expander with probability $1 - O\left(\frac{1}{\mathrm{poly}(k)}\right)$, where $k = |L|$. They define the *Very Small Set Expansion* problem distinguishing two cases [XZS22]:

1. Non-expanding: $\exists S \subset L$ with $|S| \leq \log\log k$ and Equation 1 does not hold for S.
2. Expanding: $\forall S \subset L$ with $|S| \leq \log\log k$, Equation 1 holds.

To test for the non-expanding case, the authors introduce an algorithm given in Algorithm 3. If this algorithm outputs `NotFound`, then with overwhelming probability, the graph is a good expander.

2.3 Spielman Codes: Linear-Time Encodable Error-Correcting Codes

The goal of an error-correcting code is to detect and correct errors, e.g., transmission errors due to the unreliability of networks [HP10]. A code is defined over an alphabet, commonly consisting of binary digits. A message with length k is *encoded* into a *codeword*, which is represented using exactly n digits of the alphabet. Of the n digits, k digits are associated with the information, while the remaining $m = n - k$ digits are used for error detection and correction. Such a code is referred to as an $[n, k, d]$ code, with *distance* d. Therein, the distance d of the code is the minimal Hamming distance between any two valid codewords. The *relative distance* of a $[n, k, d]$ code is defined as $\delta = {}^{d}/_{n}$ and the *rate* is $\frac{1}{r}$ with $r = {}^{n}/_{k}$, which is the ratio of digits used by the code against the minimum number of digits necessary to contain the same information.

Linear codes are the most studied form of error-correcting codes. Their characteristic is that any linear combination of codewords is again a valid codeword [RL09]. The work in [Spi96] proposes the Spielman code, a linear error-reduction code with a *linear-time encoding* using expander graphs. Orion [XZS22], Brakedown [GLS$^+$23], and related schemes use this linear-time encoding to achieve a linear-time polynomial commitment scheme. We hence explain the generalization of Spielman codes, which work over finite fields.

The construction of a linear Spielman code, denoted as E_C, for a message x uses a recursive encoding, as shown in Figure 2. The recursion step or level is indicated in superscript. At every level, two types of expander graphs, G_1 and G_2, are used. In the first

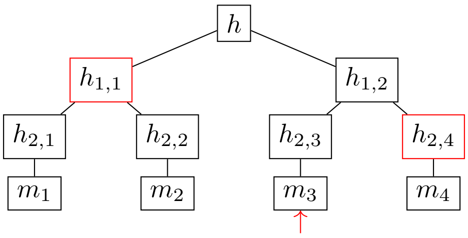


Figure 3: Merkle Tree example with $l = 2$. Hash values for opening m_3 are marked in red.

step, the k-field-element-wide message x is encoded by the expander $G_1^{(1)} = (L_1^{(1)}, R_1^{(1)}, E_1^{(1)})$ with $|L_1^{(1)}| = k$ and $|R_1^{(1)}| = \alpha_1 k$. Therein, $\alpha_1 = \alpha$ where $0 < \alpha < 1$ is a Spielman code-specific parameter. This encoding results in m_1 with $\alpha_1 k$ elements. Then, this procedure is applied recursively to encode m_1, as shown in Figure 2. As soon as the recursion computes the codeword for m_1 and returns to level-1, a different expander $G_2^{(1)} = (L_2^{(1)}, R_2^{(1)}, E_2^{(1)})$ is applied to the codeword of m_1 which results in the final z_1 component of the codeword of x. For $G_2^{(1)}$ we have $|R_2^{(1)}| = \alpha_2 |L_2^{(1)}|$ with $\alpha_2 = \frac{r-1-\alpha r}{\alpha r}$. The final Spielman codeword of x is the concatenation of x, the codeword of m_1, and z_1, as shown at the top (recursion 0) of Figure 2.

All expanders $G_1^{(i)}$ and $G_2^{(i)}$ for recursion i use the constant parameter α_1 and α_2, respectively. Since $\alpha_1 < 1$, the encodings m_i shrink in size during the recursions, and at some point m_i will have fewer than n_0 elements left, where n_0 is a scheme-specific threshold for returning from the recursion. Once this threshold is reached, no further recursion occurs, and the codeword is formed by appending the input message with the result of $G_2^{(1)}$, as in Figure 2. The expanders used in Orion or Brakedown are public information and can be randomly generated as discussed in Section 2.2.2.

The actual evaluation of an expander graph $G = (L, R, E)$ with $|L| = k$ and $|R| = \alpha_j k$ over a k-element wide message can be expressed as matrix-vector multiplication. For that, the $k \times \alpha_j k$ adjacency matrix $\mathbf{A}$ of the graph G is used. This matrix has $\mathbf{A}[i][t] = 0$ if there is no edge connecting node l_i in L and node r_t in R. Conversely, if $\mathbf{A}[i][t] = \omega_{i,t} \neq 0$, there is such an edge, and a random weight $\omega_{i,t}$ is assigned to the edge. Based on that, the graph evaluation result m and the message x are interpreted as row vectors $\vec{m}$ and $\vec{x}$, whereby $\vec{m} = \vec{x}\mathbf{A}$.

2.4 Merkle Trees

A Merkle Tree is a binary tree, used to commit to a vector of 2^l messages [Gol01] efficiently using a single hash value h at the root of the tree. The 2^l leaf nodes store the cryptographic hashes of the 2^l messages. Each non-leaf node stores the hash of its two child nodes. The root of the tree, h, serves as the final commitment, known as the Merkle commitment.

To prove the inclusion of any individual message m_i in the committed message vector, a Merkle proof π_i is generated. This proof consists of l hash values in a path to the root: starting from the leaf hash(m_i) and the sibling node's hash at each level of the tree, all the way up to the root. A verifier combines these sibling hashes to recompute the root hash and accepts π_i if and only if the re-computed root hash is h. The proof generation does not reveal any other messages $m_j \forall j \neq i$.

Figure 3 shows a representation of the Merkle Tree for a vector of four messages $[m_1, \ldots, m_4]$. To prove that m_3 is part of the committed vector, the prover provides the verifier with the siblings that appear in the path to the root, i.e., $\pi_3 = \{h_{2,4}, h_{1,1}\}$. The verifier re-computes $h_{2,3}$ from m_3, then $h_{1,2}$ from $h_{2,3}$ and $h_{2,4}$, and finally the root-hash from $h_{1,1}$ and $h_{1,2}$. If the prover tries to cheat by claiming that a different m_3' is present in the committed vector, then the recomputed hash will not match with the commitment h with very high cryptographic probability. This proof size is logarithmic in the number of elements in the committed vector. Such logarithmic efficiency is particularly important for cryptographic proof systems.

2.5 Impact of Memory Access Patterns on Off-Chip Bandwidth

In data-intensive applications, accelerators need to rely on off-chip memory like DDR4 or High Bandwidth Memory (HBM), as on-chip storage capacity is limited. The latest generation DDR or HBM may offer a high peak bandwidth, e.g., HBM2 offering up to 460 GB/s; however, such peak performance is rarely achieved in practice, particularly when memory access patterns are irregular or random. This subsection highlights how memory access patterns critically impact the effective off-chip bandwidth in data-intensive hardware accelerators, motivating the need for access-efficient designs. Communication with off-chip memory typically follows the AXI4 (Advanced eXtensible Interface) protocol [FPG], which supports two primary modes: burst and random accesses. Burst mode transfers multiple contiguous memory words in a single transaction, reducing protocol overhead and enabling memory controllers to exploit row buffering and bank-level parallelism. This allows burst transactions to utilize up to 80–95% of the theoretical bandwidth. In contrast, random accesses involve scattered, non-contiguous memory locations. These require frequent row activations and pre-charges (in DDR4) or underutilized bank parallelism (in HBM), significantly degrading performance. Random accesses are often limited to just 20–40% of the peak bandwidth, even with AXI's decoupled channel design that separates address and data phases. Empirical results from Xilinx [Xil24] on the Alveo U280 FPGA provide quantitative insights. Each HBM channel offers a theoretical 14.4GB/s bandwidth. For read operations, burst accesses achieved 13.9GB/s (96.6% utilization), while random reads dropped to 4.3GB/s (29.9%). Similarly, linear writes reached 12.9GB/s (89.7%), but random writes fell to 4.6GB/s (32.0%). Across 32 HBM channels, the aggregate practical bandwidth under random access reaches only 137.6 – 147.2GB/s, far below the 460 GB/s theoretical peak. Hence, the design methodology needs to take care of memory access patterns and optimize them to achieve high compute acceleration.

Further performance implications: The AXI4 protocol is the de facto standard for memory communication in modern FPGA- and ASIC-based accelerators. It defines independent read and write channels, each of which is further divided into address and data phases. This decoupled design allows the memory controller to issue address requests far in advance and pipeline multiple outstanding memory accesses. However, this flexibility comes at a cost. Due to the deep pipelining of off-chip memory controllers, there is often a significant initial latency between issuing an address and receiving the corresponding data. This startup latency is especially pronounced when random accesses break the streaming pattern and prevent effective prefetching or pipelining. For example, issuing a burst read may still take tens of cycles before the first data word is returned, depending on memory controller arbitration and row activation delays. Furthermore, read and write channels are not truly independent at the memory controller level. When an accelerator interleaves reads and writes, the memory controller must drain and flush internal buffers, reconfigure internal paths, and stall ongoing transfers. This results in read/write turnaround penalties, which introduce idle cycles even if the AXI interface is otherwise fully utilized. These turnarounds are particularly harmful in scenarios like Orion's expander graph evaluations, which exhibit frequent switches between read and write phases due to multiply-accumulate steps.

3 Background on Proof Systems

Proof systems are protocols that allow a prover to convince a verifier of the correctness of a statement. For example, the prover can demonstrate knowledge of a private witness w for a public input x such that a circuit $\mathcal{C}(x, w) = 0$ holds, without disclosing w. With this hiding feature, the proof system is called 'zero-knowledge'.

Proof systems are commonly used to ensure the correctness of computations outsourced to untrusted entities. The verifier can verify the proof using a small amount of computation, hence efficiently, without performing the outsourced computation themselves. There are several important performance metrics for proof systems, such as proof generation time, proof size, and proof verification time. Usually, proof systems are optimized for succinct proofs and low verification costs. Having a low proof generation time is a plus and desired in applications that require scalability.

Modern succinct non-interactive proof systems are built by combining an interactive oracle proof (IOP) with a functional commitment scheme to form a succinct *interactive* argument. To render the proof system non-interactive, the Fiat-Shamir transform [FS86] can be used. In the following part, we describe functional commitment with a focus on the specific type of 'polynomial commitment'.

3.1 Functional Commitment (with a focus on polynomial commitment)

A functional commitment scheme is a commitment scheme that enables a party to commit to a function (computation procedure) they intend to evaluate. A prover can prove statements about the committed function, e.g., the evaluation of the function at a given point is correct. Functional commitment schemes have binding and optionally hiding properties. The binding property ensures that the committer cannot change the function or the input to the function, so that the commitment remains unaffected. The hiding property, which is required for *zero-knowledge* succinct non-interactive argument of knowledge or SNARK construction, ensures that no information about the function or the inputs to the function is revealed. A functional commitment scheme is a cryptographic protocol with an underlying cryptographic assumption, e.g., discrete logarithm assumption, pairing assumption, cryptographic hash assumption, etc.

The Merkle Tree presented in the previous subsection is a vector commitment scheme to prove membership in a committed vector. While vector commitments like Merkle Trees are useful for committing to a static set of values and proving membership, they lack the flexibility, efficiency, and built-in polynomial structure needed for polynomial-related operations. Polynomial commitments [KZG10] are functional commitments, specifically designed to address these limitations by enabling succinct proofs of evaluations, degree checks, and functional properties of polynomials, making them essential in many cryptographic applications. With a polynomial commitment scheme, a prover can commit to a polynomial, for example, $\phi(x) \in \mathbb{F}[x]$ of degree t with coefficients from the field $\mathbb{F}$. Simply committing to each coefficient $(\phi_0 \| \dots \| \phi_t)$ of the polynomial is ineffective, as verifying the commitment would require revealing the entire polynomial. Furthermore, many cryptographic applications require that only evaluations of $\phi(x)$ at specific x are revealed without revealing the entire polynomial. Polynomial commitments play a crucial role in modern proof systems, e.g., SNARKs.

A polynomial commitment scheme [KZG10] consists of four main steps. First, the 'Setup Phase' involves the generation of public parameters using the SETUP algorithm for a given function family, say $\mathcal{F}$. In the second step, the 'Commitment Phase', the prover P uses the COMMIT algorithm and the public parameters to generate a commitment com_ϕ for a polynomial $\phi \in \mathcal{F}$. Third, during the 'Evaluation Phase', the verifier V selects an evaluation point $x = a$ and requests the prover to evaluate ϕ at that point. The prover computes the value $b = \phi(a)$ and generates an evaluation proof π that asserts the correctness of this evaluation. Finally, in the 'Verification Phase', the verifier checks the validity of the proof π against the commitment com_ϕ, the evaluation point a, and the claimed output b, accepting or rejecting the claim based on the result.

Example of polynomial commitment: To explain how a polynomial commitment scheme works, we use the famous KZG scheme [KZG10] with some simplifications. Let $\phi(x) = \phi_0 + \phi_1 x + \cdots + \phi_t x^t$ be a polynomial over $\mathbb{Z}_p$ of degree t. The scheme uses the discrete logarithm and bilinear pairing cryptographic assumptions and requires a trusted setup. The setup generates public parameters $\mathrm{PP} = (g, g^s, g^{s^2}, \ldots, g^{s^t}, g_2, g_2^s)$ where $g \in G_1$ and $g_2 \in G_2$ are generators of two bilinear groups of prime order p, and s is a secret 'toxic waste' destroyed after the setup. The bilinear pairing $e : G_1 \times G_2 \to G_T$ is used for verification. To commit to $\phi(x)$, the prover computes $com_\phi = g^{\phi(s)} = \prod_{i=0}^{t} (g^{s^i})^{\phi_i}$ which is an element of G_1. To prove that the evaluation $y_a = \phi(x_a)$ at $x = x_a$ is correct, the prover computes the quotient polynomial $q(x) = \frac{\phi(x)-y_a}{x-x_a}$ and then the proof $\pi = g^{q(s)} \in G_1$. The verifier checks the evaluation proof π by using the bilinear pairing $e(com_\phi / g^{y_a}, g_2) \overset{?}{=} e(\pi, g_2^{s-x_a})$ where $g_2^{s-x_a}$ is precomputed from the public parameters.

The prover in the KZG polynomial commitment scheme is very slow, and due to the use of elliptic curve cryptography, the scheme is not quantum-resilient. Furthermore, the scheme requires a trusted setup. In the following, we present the basics of Orion [XZS22], which provides fast proving times, quantum resilience, and transparent setup.

3.2 Orion and Brakedown Proof Systems

Orion [XZS22] is a highly efficient zero-knowledge proof system that extends the Brakedown proof system [GLS$^+$23]. Both Orion and Brakedown utilize linear-time polynomial commitments and Merkle Trees to achieve succinct proofs with fast proving. Unlike most contemporary systems with superlinear prover times [KZG10, Gro16, CHM$^+$20], Orion's and Brakedown's provers run in linear time. Proving only relies on hash functions, making it resistant to quantum attacks, which is not the case in schemes based on discrete logarithm or pairing assumptions. Built on linear codes via expander graphs, Orion uses code-switching to achieve poly-logarithmic proof sizes. In contrast, Brakedown has a square-root proof size, leading to larger proofs. Nevertheless, Brakedown's encoding and parts of the proving mechanism are very similar to Orion, which allows extending our presented concepts to Brakedown as well. The following section explains the protocol with a focus on Orion and highlights the similarities and differences to Brakedown.

Orion and Brakedown operate on multilinear polynomials ϕ in $\log N$ variables, where each variable has a degree 0 or 1. There are N monomials and coefficients in ϕ. The coefficients of ϕ are in a field, say $\mathbb{F}_{p^2}$, and any evaluation of ϕ is also performed in $\mathbb{F}_{p^2}$. Authors in [GLS$^+$23] found that such a polynomial evaluation can be expressed as a tensor product. The evaluation of ϕ at $\vec{x} = [x_0, \ldots, x_{\log N-1}]$ can be written as

$$\phi(\vec{x}) = \sum_{i_0=0}^{1} \cdots \sum_{i_{\log N-1}=0}^{1} w_{i_0,\ldots,i_{\log N-1}} x_0^{i_0} x_1^{i_1} \ldots x_{\log N-1}^{i_{\log N-1}} \tag{2}$$

In the above expression, $i_0, \ldots, i_{\log N-1}$ represent the binary decomposition of monomial index $i \in \{0, 1, \ldots, N-1\}$, and the w_i terms correspond to the coefficients of the monomials $X_i = x_0^{i_0} x_1^{i_1} \ldots x_{\log N-1}^{i_{\log N-1}}$. Let $\vec{w}$ denote the vector of monomial coefficients with the i-th element $\vec{w}[i] = w_{i_0,\ldots,i_{\log N-1}}$. Assuming N has an integer square root (e.g., N a power of 2), let $k = \sqrt{N}$. Also, let $\vec{r}_0 = (X_0, X_1, \ldots, X_{k-1})$ and $\vec{r}_1 = (X_{0\cdot k}, X_{1\cdot k}, \ldots, X_{(k-1)\cdot k})$ be two vectors containing k distinct monomials. Then all the monomials in Equation 2 are obtained using the tensor product $\vec{r}_0 \otimes \vec{r}_1$. Finally, the evaluation of ϕ can be obtained using the inner product [GLS$^+$23] as $\phi(x_0, \ldots, x_{\log N-1}) = \langle \vec{w}, \vec{r}_0 \otimes \vec{r}_1 \rangle$. Protocol 1 gives

Protocol 1 Overview of Orion [XZS22] with simplifications

```
   Public input: Evaluation point $\vec{x}$ parsed as tensors $\vec{r}_0$ and $\vec{r}_1$;
   Private input: Polynomial $\phi$ with coefficients $\vec{w}$; Let $E_C$ be the encoding function of
   a $[n, k, d]$ linear code and $N = k^2$;

 1: function COMMIT($\phi$)                                         ▷ Similar to Brakedown
 2:     Parse the coefficient vector $\vec{w}$ of length $N$ as a $k \times k$-matrix $\mathbf{W}$;
 3:     Using $E_C$ encode each row of $\mathbf{W}$ to obtain code $\mathbf{C}$ which is a $k \times n$ matrix;
 4:     for $0 \leq i < n$ do
 5:         Compute Merkle root for each column $\texttt{Root}_i \leftarrow \texttt{Merkle.Commit}(\mathbf{C}[:, i])$;
 6:     Compute the Merkle root $\mathcal{R} \leftarrow \texttt{Merkle.Commit}([\texttt{Root}_0, \ldots, \texttt{Root}_{n-1}])$;
 7:     Output $\mathcal{R}$ as the commitment;

 8: function PROVE($\phi, \vec{x}, \mathcal{R}$)                      ▷ Similar to Brakedown until line 13
 9:     Prover receives from the verifier a random vector $\vec{\gamma}_0 \in \mathbb{F}^k_{p^2}$;
10:     $\vec{c}_{\gamma_0} \leftarrow \sum_{i=0}^{k-1} \vec{\gamma}_0[i]\mathbf{C}[i, :]$, $\vec{y}_{\gamma_0} \leftarrow \sum_{i=0}^{k-1} \vec{\gamma}_0[i]\mathbf{W}[i, :]$;          ▷ Proximity check
11:     $\vec{c}_1 \leftarrow \sum_{i=0}^{k-1} \vec{r}_0[i]\mathbf{C}[i, :]$, $\vec{y}_1 \leftarrow \sum_{i=0}^{k-1} \vec{r}_0[i]\mathbf{W}[i, :]$;          ▷ Consistency check
12:     Prover sends the evaluation of $\phi$ at $\vec{x}$ as $y \leftarrow \langle \vec{y}_1, \vec{r}_1 \rangle$ to verifier;
    /* Starting here, Brakedown differs from Orion */
13:     Compute Merkle root $\mathcal{R}_{\vec{c}_{\gamma_0}} \leftarrow \texttt{Merkle.Commit}(\vec{c}_{\gamma_0})$;
14:     Compute Merkle root $\mathcal{R}_{\vec{c}_1} \leftarrow \texttt{Merkle.Commit}(\vec{c}_1)$;
    /* The following message to the verifier forms Orion's proof string $\pi_{\vec{x}}$ */
15:     Prover sends $\mathcal{R}_{\vec{c}_{\gamma_0}}$, $\mathcal{R}_{\vec{c}_1}$ to the verifier;
16:     Verifier sends the set $\hat{I}$ of $t$ $(0 < t < n)$ randomly sampled column indexes;
17:     $\pi_i \leftarrow \texttt{ZK.Prove}()$: The prover executes a CP-SNARK $\texttt{ZK}$ to prove the genuine vali-
        dation of the proximity and consistency checks. [Su] uses Virgo [ZXZS20] for this;
18:     Send $\pi_i$ to the verifier.

19: function VERIFYEVAL($\mathcal{R}, \vec{x}, y = \phi(\vec{x}), \pi_{\vec{x}}, \pi_i$)
    /* Verifier parses the proof string $\pi_{\vec{x}}$ and $\pi_i$ and obtains the prover's messages */
20:     Verify the CP-SNARK $\texttt{ZK.Verify}(\pi_i)$;
21:     Check consistency to $\mathcal{R}_{\vec{c}_{\gamma_0}}$, $\mathcal{R}_{\vec{c}_1}$, and $\mathcal{R}$;
22:     Accept if all checks pass;
```

an overview of Orion's polynomial commitment scheme [XZS22] with some simplifications and shows the similarities and differences to Brakedown [GLS+23].

Commitment: In line 2 of Protocol 1, the coefficient-vector $\vec{w}$ of length $N = k^2$ is parsed as the matrix $\mathbf{W}$ of dimension $k \times k$. Note that the evaluation $y = \phi(\vec{x}) = \langle \vec{w}, \vec{r}_0 \otimes \vec{r}_1 \rangle$ can also be represented as a vector-matrix-vector multiplication as follows: the row-vector $\vec{r}_0$ is multiplied from the left with the matrix $\mathbf{W}$, and the resulting row-vector is multiplied with the column-vector $\vec{r}_1$ to produce y. Orion utilizes the tensor IOP protocol from [BCG20] to construct a polynomial commitment based on Brakedown [GLS+23].

Let E_C be the encoding procedure of an $[n, k, d]$ linear code (Section 2.3). In line 3, each row $\mathbf{W}[i, :]$ is encoded into a codeword of length n. After encoding all rows of $\mathbf{W}$, we obtain the code matrix $\mathbf{C}$ of dimension $k \times n$. In lines 4 and 5, the columns of $\mathbf{C}$ are Merkle-committed to leaf hashes $\texttt{Root}_i$. Finally, from the n leaf hashes, the Merkle Tree is calculated in line 6 to produce the root hash $\mathcal{R}$ as the final commitment to the tree. Orion and Brakedown use the linear-time Spielman code (described in the previous section) to achieve linear time complexity for the polynomial commitment. Moreover, Orion's and

Brakedown's commitment mechanisms are very similar, with only minor differences such as the underlying finite field. This allows applying our acceleration techniques for linear-time commitment to both schemes.

Proving mechanism: The prover and verifier engage in an interactive protocol starting from line 8 in Protocol 1. As a challenge, the verifier provides the prover with the random vector $\vec{\gamma}_0 \in \mathbb{F}_{p^2}^k$. Using $\vec{\gamma}_0$, in line 10 of the protocol, the prover computes random linear combinations of the rows of $\mathbf{C}$ and $\mathbf{W}$ via inner products. As the code is linear, any linear combination of codewords is also a codeword. Thus, if the calculations are performed correctly (i.e., non-cheating prover), the resulting codeword $\vec{c}_{\gamma_0}$ should be the encoding of $\vec{y}_{\gamma_0}$. Next, in line 11, a similar linear combination is performed using $\vec{r}_0$ of the tensor query $\vec{r}_0 \otimes \vec{r}_1$. If performed correctly, $\vec{c}_1$ will be the encoding of $\vec{y}_1$.

Up to line 12, Orion and Brakedown follow very similar steps. Yet, from line 13 on, the two schemes differ. Brakedown directly sends $\vec{c}_1$, $\vec{y}_1$, $\vec{c}_{\gamma_0}$, and $\vec{y}_{\gamma_0}$ to the verifier, which locally performs the so-called proximity check and consistency check. During these checks, the verifier encodes $\vec{y}_{\gamma_0}$ and $\vec{y}_1$ and ensures $E_C(\vec{y}_{\gamma_0}) == \vec{c}_{\gamma_0}$ (proximity check) and $E_C(\vec{y}_1) == \vec{c}_1$ (consistency check). In addition, the verifier randomly samples the set $\hat{I}$ containing t column indices of $\mathbf{C}$. The prover sends the specified columns along with their Merkle proofs under the commitment $\mathcal{R}$. Finally, the verifier checks the obtained columns against $\vec{c}_{\gamma_0}$ and $\vec{c}_1$ and validates the Merkle proofs against $\mathcal{R}$. The verifier accepts if all checks succeed. The prover time complexity of Brakedown is $\mathcal{O}(N)$. However, the proof size is $\mathcal{O}(\sqrt{N})$, which is quite large compared to commonly used pairing-based proof systems.

Orion significantly reduces the proof sizes to $\mathcal{O}(\log^2 N)$ by a so-called 'proof composition' technique [RZR24]. Orion's proof composition technique does not perform the consistency and proximity checks on the verifier side but on the prover side. Moreover, the prover ensures a genuine execution of these checks via a CP-SNARK ZK. This technique requires the prover to Merkle-commit to $\vec{c}_{\gamma_0}$ and $\vec{c}_1$, as shown in lines 13 and 14 in Protocol 1. After the commitment has been sent to the verifier, the prover and the verifier execute the ZK protocol. The verifier finally checks the consistency of the Merkle commitments (line 21) and accepts if the checks succeed. Orion treats the CP-SNARK ZK as a black box; thus, any proper scheme can be used. The software implementation of Orion [Su] uses the Virgo [ZXZS20] protocol. Note that Brakedown does not involve any CP-SNARK at the cost of larger proof sizes.

Generalization to $N = k_1 \cdot k_2$: For simplicity, the above-mentioned description of encoding and Orion used $N = k^2$, resulting in $\mathbf{W}$ of dimension $k \times k$. Assuming N is a power-of-two, it is easy to see that the same can be generalized to asymmetric decomposition $N = k_1 \cdot k_2$. The software implementation of Orion [Su] by the authors uses the fixed $k_1 = 128$ for all large N. Depending on the value of N, the number of columns k_2 in $\mathbf{W}$ is adjusted. This gives flexibility. Following the authors of Orion, we use fixed $k_1 = 128$ in our hardware implementation.

3.3 Latency of Operations in Orion

In this section, we present a detailed latency analysis of Orion's core operations, particularly how the system's performance scales as N increases. Note that in contrast to other cryptographic schemes, such as the one-time key generation process in digital signatures, the commitment and proving phase in Orion is repeatedly invoked as part of a recursive proving system. For example, when creating proofs of training for Deep Neural Networks as in [APKP24], dedicated commitments are required for each layer. Each time the recursive prover processes a new layer or polynomial, it calls Orion to commit to the new data and

Table 1: Timing results in milliseconds (ms) of different operations in the reference implementation of Orion [Su]. Results collected in software on AMD EPYC 9754 @2.25GHz.

Size	**Prover's Operations in Orion**			**Virgo**	**Total**	**Verifier**
log(N)	Commit	Prove	Commit+Prove			
16	40	4	44	37	81	43
18	155	10	165	102	267	48
20	701	35	736	483	1,219	56
22	3,175	135	3,310	1,370	4,680	74
24	14,756	593	15,349	5,849	21,198	131
26	60,011	2,495	62,506	26,457	88,963	218
28	334,250	9,093	343,343	118,554	461,897	211

Table 2: Timing breakdown for Commit suboperations in milliseconds (ms). Results collected in software [Su] on an AMD EPYC 9754 @2.25GHz.

Size	**Latencies of Sub-operations in Commit**				**Total**
log(N)	Initialize	Encode($\mathbf{W}$)	Hash($\mathbf{C}$)	Merkle Tree($\mathbf{H}$)	**Commit**
16	4	25	10	1	40
18	10	104	40	1	155
20	73	463	163	2	701
22	279	2,233	656	7	3,175
24	1,998	10,080	2,651	27	14,756
26	3,448	45,855	10,600	108	60,011
28	13,137	278,100	42,580	433	334,250

Table 3: Timing breakdown for Prove suboperations in milliseconds (ms). Results collected in software [Su] on an AMD EPYC 9754 @2.25GHz.

Size	**Latencies of Sub-operations in Prove**				**Total**
log(N)	Inner Product	Hash($\vec{c}$)	Merkle Tree($\mathbf{H}$)	Other	**Prove**
16	2	1	1	0	4
18	8	1	1	0	10
20	31	2	2	0	35
22	120	7	7	1	135
24	534	27	27	5	593
26	2,262	106	108	19	2,495
28	8,160	430	433	70	9,093

perform the proof steps. As a result, the commitment and proving phases must efficiently handle repeated invocations, making it computationally intensive.

The computations in Orion can be broken down into three main phases: Commitment (lines 1-7 in Protocol 1), proving (lines 8-15), and the Virgo CP-SNARK (lines 16-21). Note that commitment and proving are executed solely by the prover, whereas Virgo, as the outer proof protocol, involves the prover and verifier. Table 1 presents a timing breakdown of the three phases collected using Orion's software implementation [Su]. While all phases contribute to overall latency, the commitment phase stands out due to its computational complexity, particularly in managing the polynomial commitments across recursive calls. Compared to commitment, the proving phase adds a smaller latency of up to 9 seconds for $N = 2^{28}$. These two phases - commit and prove - define the latency of Orion, excluding the outer Virgo protocol, and cause the major share of 74% of the overall runtime ($N = 2^{28}$). The Virgo column in Table 1 contains the latency for executing Virgo's prover protocol, including circuit generation, circuit evaluation, sum-checks, etc. Hence, the Virgo prover causes around 26% of the overall latency. Finally, Table 1 shows the end-to-end latency of a full Orion execution including commit, prove, and the Virgo protocol, and the comparably low verifier latency.

The commitment phase stands out as the most time-intensive operation in Orion. This phase consists of several computationally demanding sub-operations, including initialization, linear encoding, hashing, and Merkle Tree generation. Table 2 presents the software latencies of these sub-operations. Therein, the initialization phase plays a crucial role in constructing expander graphs, which are essential for establishing the system's foundational structure. The most prominent operation, however, is linear encoding, which takes the input and encodes it using expander graphs. The encoding process, as well as expander graphs, are described in detail in Section 2.3. Following the encoding, the hashing of the columns of the encoded matrix $\mathbf{C}$ for Merkle Tree construction represents the second largest time cost. Note that Orion's software implementation supports SHA2 and SHA3 algorithms, where we choose the more recent SHA3 setting for all benchmarks.

Orion's proving phase (lines 8-14 in Protocol 1) consists of the inner product computation, hashing, and Merkle Tree generation. In addition, smaller operations such as tensorization, evaluation computation, etc., are grouped into an 'Other' category. The latencies of these sub-operations are reported in Table 3. We observe that the inner product computation is the prevailing operation within proving, whereas hashing and Merkle Tree generation are more lightweight.

Operations we target for hardware acceleration: This work targets the most demanding computations within the Orion scheme and presents FPGA acceleration techniques for Orion and Brakedown-like PCS. Our design offers support for on-the-fly graph generation, thereby omitting the initialization sub-operation and avoiding storing the large graph in memory. Moreover, we address the challenging linear encoding using Spielman codes and present hardware-oriented optimizations. We also include units for hashing and Merkle Tree computations needed in the commitment and proving steps. Finally, we reuse the datapath for linear encoding to accelerate the inner product computation during proving. These measures effectively improve the runtime of the commitment and proving steps.

While our paper presents a deep design exploration and hardware optimization of Orion- and Brakedown-like commitment and proof generation components, it excludes the acceleration of the higher-level Virgo protocol from its scope. The Virgo protocol integrates multiple complex mechanisms, including sum-check protocols [LFKN92], GKR protocols [GKR15], and arithmetic circuit transformations. Each of these components introduces distinct computational patterns and design challenges. Consequently, accelerating the full Virgo protocol requires substantial additional research beyond the scope of this work. Nevertheless, extending hardware acceleration to encompass an entire CP-SNARK system such as Virgo remains an exciting direction for future investigation.

4 Accelerating Expander Graph based Linear Encoding

The linear-time commitment and proof generation in Orion and Brakedown stems from their expander-based linear encoding as covered in Section 2.2. These graphs are carefully designed to be sparse, with a limited number of edges per node, while maintaining high connectivity to preserve the security and succinctness required for cryptographic protocols. This dual property is essential because the sparsity ensures efficient computations, while high connectivity guarantees the soundness of encoding.

The straightforward process of constructing and using expander graphs involves Gigabytes of storage and random accesses to off-chip memory. Random accesses substantially lower the off-chip memory bandwidth [Xil24], as discussed in Section 2.5. In addition,

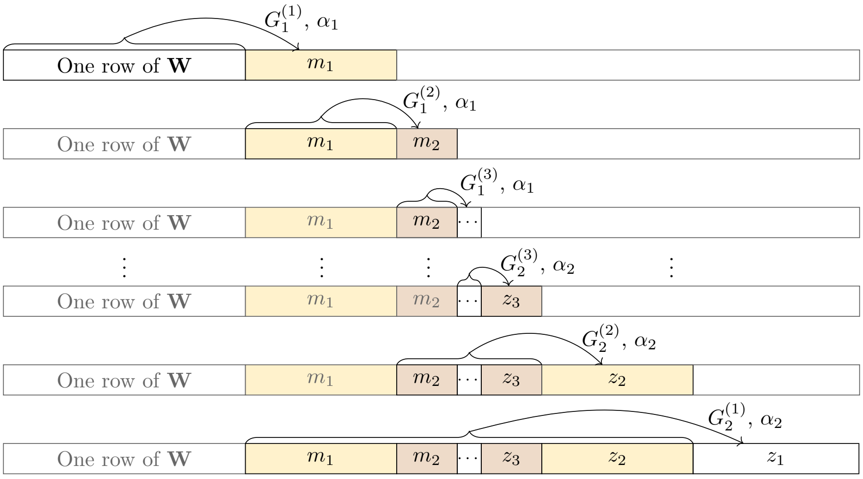


Figure 4: Our iterative approach for computing Spielman codes. The example has 3 levels.

frequent read and write interleavings during graph evaluations, as used in the reference software implementation [Su], cause significant turnaround overheads in the DRAM-based off-chip memory systems.

Our hardware accelerator tackles these off-chip data movement challenges through a holistic design approach. The remainder of this section presents our suite of novel optimizations for efficient and hardware-friendly linear encoding using expander graphs. We begin by discussing the advantages of on-the-fly graph generation and our custom iterative encoding procedure. Next, we introduce *inverted expander graphs*, which significantly reduce the random off-chip memory accesses and read-write turnarounds. We also detail the non-trivial integration of on-the-fly graph generation with the graph inversion technique. Finally, we describe our memory-bandwidth-aware accelerator architecture tailored for high-throughput linear encoding.

4.1 On-the-Fly Graph Sampling

In the baseline software implementation of Orion [Su], the entire expander graph is first randomly sampled according to Algorithm 2 and then stored in memory before its usage. Storing the sampled graph, however, causes a substantial memory consumption. For example, a $N = 2^{28}$ polynomial, which needs 8 recursions and hence 16 different graphs, consumes 1.1 GiB. Storing this data requires additional off-chip memory resources such as DDR RAM. Also, although reading the graph from memory follows a linear pattern, scarce off-chip bandwidth is consumed.

However, the used expander graphs are public and randomly sampled (see Section 2.2.2). This allows for resampling the graphs on the fly during each encoding. Therefore, we use a pseudo-random number generator (PRNG) to expand the needed graphs based on a public seed. Note that using a PRNG with a sufficiently high period for graph sampling does not compromise security; it is just used to deterministically sample public data. In our implementation, we use the Trivium [Bou20] PRNG with a period $> 2^{90}$. This allows for the dynamic construction of large parts of the expander graphs during the encoding process.

4.2 Iterative Linear Encoding

Orion encodes the coefficient matrix $\mathbf{W}$ of a polynomial into a code matrix $\mathbf{C}$ in a row-wise manner. In essence, each row in $\mathbf{W}$ is encoded into one row of $\mathbf{C}$ as detailed in Section 3.2. In the software implementation of Orion, the linear encoding algorithm for one row in $\mathbf{W}$ is recursive, as discussed in Section 2.3. Yet, a recursive approach is not ideal for hardware designs due to increased control overhead. Thus, we implement linear encoding iteratively, as shown in Figure 4.

The iterative linear encoding procedure starts with applying an expander graph $G_1^{(1)}$ with compression parameter $\alpha_1 < 1$ to the input message, which is one row of $\mathbf{W}$ (Figure 4 top). The result of this operation is m_1, which is smaller in size than the input message. In the next step, another expander graph $G_1^{(2)}$ with α_1 is applied on m_1 yielding m_2. Thereby, m_2 is again smaller than m_1. This procedure is applied until the size of m_i is below a certain threshold n_0.

Thereafter, different expander graphs $G_2^{(j)}$ with compression parameter $\alpha_2 < 1$ are applied iteratively on the previous codewords, as shown in Figure 4. The results of the $G_2^{(j)}$ expanders are denoted with z_j. The $G_2^{(j)}$ graph evaluations are repeated until all previous intermediate codewords are consumed. According to Section 2.3, α_1 and α_2 are given by the code parameter α which is $\alpha = 0.238$ in Orion [Su]. We obey this configuration in our work. The final expander graph evaluation $G_2^{(1)}$ yields z_1, which completes the linear encoding.

4.3 Optimizing Memory Bandwidth with Inverted Expander Graphs

Traversing expander graphs involves irregular and random memory access patterns, creating performance bottlenecks when working with large datasets stored in external memory such as DDR or HBM. These random accesses can degrade memory bandwidth by up to 3× (see Section 2.5), and frequent read-write turnarounds further increase latency. The reference implementation [Su] does not mitigate these overheads.

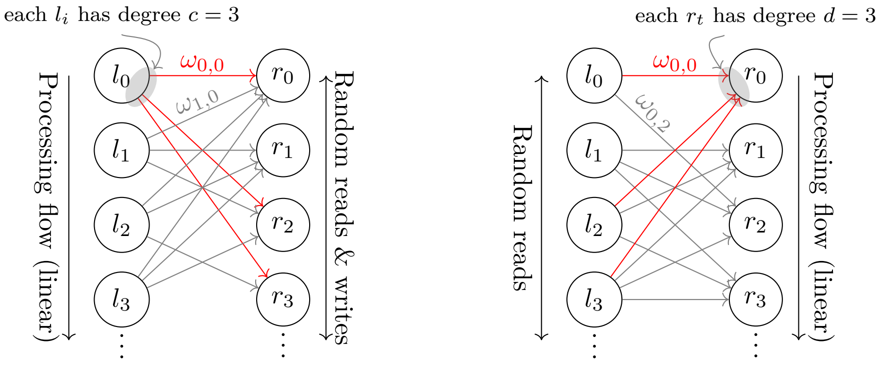
To overcome these limitations, we propose an *inverted expander graph* that reduces random memory accesses and minimizes read-write turnarounds, thereby improving overall performance. This section begins with a review of the baseline implementation [Su], highlighting its inefficiencies for hardware. We then introduce our inverted approach, analyze its impact on memory bandwidth, and discuss its security implications.

4.3.1 The Baseline Expander Graph Evaluation and their Disadvantages

The baseline expander graph evaluation (e.g., as implemented in [Su]) is illustrated in Figure 5a. The expander graph $G = (L, R, E)$ is represented by two node arrays $L = \{l_0, l_1, \ldots\}$ and $R = \{r_0, r_1, \ldots\}$, where the values of nodes $l_i, r_t \in \mathbb{F}_{p^2}$. Each left-side node l_i has a fixed degree c ($c = 3$ in Figure 5a, $c = 10$ in [Su]), and a connecting edge between l_i and r_t carrying a random weight $\omega_{i,t} \in \mathbb{F}_{p^2}$. The degree of right-side nodes r_t varies due to the randomized left-to-right node assignments, leading to a binomial distribution of edge counts across R (Figures 6a–6c). All r_t are initialized to zero. During evaluation, each l_i is processed sequentially: it is multiplied by its c associated edge weights and accumulated into the corresponding r_t nodes. For example, as highlighted in red in Figure 5a, l_0 updates r_0, r_2, and r_3 as follows:

$$r_0 = r_0 + l_0 \cdot \omega_{0,0} \qquad r_2 = r_2 + l_0 \cdot \omega_{0,2} \qquad r_3 = r_3 + l_0 \cdot \omega_{0,3} \tag{3}$$

Importantly, the left-side nodes are accessed in a linear order, while the right-side nodes are accessed in a randomized order. This approach of graph evaluation requires random read and write operations, which lowers the effective memory bandwidth. For each input



(a) Baseline expander graph.

(b) Inverted expander graph.

Figure 5: The two different approaches of expander graph implementation. Only a few weights $\omega_{i,t}$ are explicitly shown, but every edge connecting l_i to r_t has a weight $\omega_{i,t}$.

node l_i, c random right-nodes r_{t_1} to r_{t_c} must be read from off-chip memory, as storing the large data structure on the chip is infeasible. Then, the c accumulations are performed (as in Equation 3), and the c results are stored back in scattered memory locations. In this way, per input node l_i, c random reads and c random writes are required. For a whole graph evaluation of $G = (L, R, E)$, $|L|$ linear reads, $|E|$ random reads, and $|E|$ random writes must be performed.

Table 4 reports the off-chip memory access count for the full encoding process with multiple graph evaluations. For the baseline expander graphs, up to 104 million read and write operations are required to load and store the actual codeword data. Moreover, up to 50 million additional read operations would be required to load the graph structure from memory. This high number of non-uniform memory accesses is challenging even for HBM. While HBM offers a high theoretical peak bandwidth, the effective bandwidth is significantly lower due to random access patterns as explained and detailed in Section 2.5.

In addition to random memory accesses, the graph evaluation approach introduces read-after-write hazards in the multiplication and accumulation pipeline. Consider two consecutive input nodes l_i and l_{i+1} mapping to the same output node r_t, which is likely to occur in expanders with few right nodes $|R|$. In such cases, the write operation updating r_t from l_i must complete before the subsequent read by l_{i+1} can proceed. While these dependencies can be efficiently managed in fully on-chip designs using techniques like data forwarding or buffering, they become significantly more problematic when using off-chip memory, due to higher latency and the lack of immediate write visibility. As a result, the pipeline must stall to ensure correctness, leading to increased control logic complexity and reduced throughput. These stalls, combined with inefficient memory access patterns, substantially degrade the performance of the baseline expander graph evaluation. The next section presents our optimized solution to address these bottlenecks.

4.3.2 Inverted Expander Graph Evaluation for Minimizing Memory Accesses

We propose an inverted expander graph evaluation that reverses the baseline approach from Section 4.3.1, which processes the *left* nodes sequentially to distribute values to the right nodes. Instead, our inverted method iterates over the *right* nodes, aggregating contributions from connected left nodes, as shown in Figure 5b. For example, to compute r_0, the evaluation loads the $d = 3$ connected left nodes l_0, l_2, and l_3, then multiplies each by its respective weight and accumulates the results. Once r_0 is computed, the process moves to r_1, repeating the same steps.

Table 4: Off-chip read and write operations for linear encoding using baseline expander graphs and our inverted expander graphs with approach 1 and approach 2. GRd refers to off-chip reads to obtain the stored graph structure.

log (N)	**Baseline Expander Graphs**				**Inverted Expander Graphs**							
	Left to Right Evaluation				Right to Left Evaluation							
	Stored Graph				Approach 1: Stored Graph				Approach 2: Sampled Graph			
	#Rd	#Wr	#Total	#GRd	#Rd	#Wr	#Total	#GRd	#Rd	#Wr	#Total	#GRd
16	12,773	11,850	24,623	11,850	12,048	368	12,416	11,850	13,151	1,196	14,347	0
18	52,253	48,490	100,743	48,490	48,484	1,474	49,958	48,490	53,251	5,070	58,321	0
20	210,267	195,140	405,407	195,140	194,212	5,898	200,110	195,140	213,754	20,752	234,506	0
22	842,154	781,580	1,623,734	781,580	777,024	23,592	800,616	781,580	855,308	83,028	938,336	0
24	3.37M	3.13M	6.50M	3.13M	3.11M	0.09M	3.20M	3.13M	3.42M	0.33M	3.75M	0
26	13.48M	12.51M	25.99M	12.51M	12.43M	0.38M	12.81M	12.51M	13.68M	1.33M	15.01M	0
28	53.93M	50.05M	103.98M	50.05M	49.74M	1.51M	51.25M	50.05M	54.74M	5.31M	60.05M	0

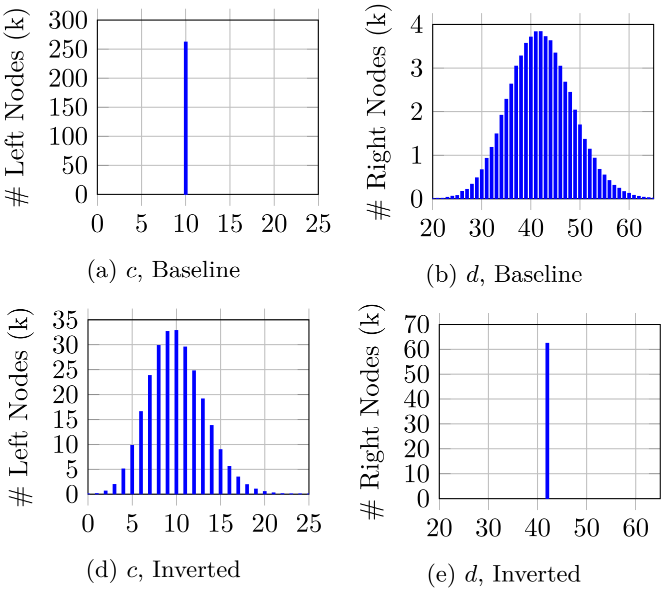
Our inverted expander graphs remove the read-after-write hazards and reduce the penalty of memory accesses in a $G = (L, R, E)$ graph evaluation. In particular, just $|R|$ *linear* write operations are needed whereas baseline graphs require $|E|$ *random* writes (note that $|E| \gg |R|$). In addition, $|L|$ linear reads are saved, and frequent read-write turnarounds are avoided.

How to deploy inverted expander graphs: Although the described inverted expander graphs significantly lower the random memory accesses, they cannot be directly combined with on-the-fly graph sampling. This is because the binomially distributed right node degree d and the corresponding edges to left nodes are not computable via a PRNG such that each left node has a constant degree c (as it is the case in baseline graphs, see Figures 6a to 6c). Instead, two different approaches can be applied:

1. **Storing the graph structure in memory:** In this approach, a baseline graph is generated using Algorithm 2. This graph is then reordered from left-node major order to right-node major order, where all connections of one right node are stored consecutively. The resulting graph is kept in off-chip memory and streamed to the hardware architecture, which computes the inverted graph evaluation.
2. **Relaxing the left-node degree c**: This approach samples the inverted expander graph structure on-the-fly using a PRNG and directly evaluates the inverted graph. Specifically, we fix the right node degree d and iterate linearly over the right nodes. For each right node, the PRNG randomly selects d left nodes, accumulating the right node. This leads to a constant degree of right nodes and a binomially distributed degree of left nodes c. Figures 6d to 6f show examples of the corresponding distributions.

The encoding in approach 1 yields the identical result as Orion's or Brakedown's baseline implementations, but also has the drawback of high off-chip memory consumption as both the graph and data must be accessed. Table 4 shows the resulting off-chip memory accesses. Compared to baseline graphs, our inverted graphs with approach 1 clearly reduce the data reads and writes by about 50% (column #Total). Nevertheless, the data movements still take place in the range of Gigabytes. In addition to the encoding data, the graph structure must also be loaded from off-chip memory in approach 1. Loading the graph structure causes almost as many reads as the actual data streaming (see column #GRd in Table 4). Hence, storing the graph in HBM would degrade the effective memory bandwidth during linear encoding since reading the graph structure blocks the data reads and writes. Using supplementary DDR memory for storing the graph structure is also problematic since FPGAs like Alveo U55C [Xil23] or [Xil21] do not have DDR memory. Thus, it is desirable not to store the graph structure but to generate it on the fly.

On the other hand, approach 2 changes the structure of the Spielman code, yielding a different encoding result compared to the baseline implementation. This difference, however, is *transparent* to the verifier due to Orion's proof composition technique (see



(a) c, Baseline

(b) d, Baseline

	Min	Max	Mean	Variance
c	10	10	10.0	0.0
d	17	71	42.0	42.2

(c) Distribution characteristics for baseline graphs.

(d) c, Inverted

(e) d, Inverted

	Min	Max	Mean	Variance
c	2	25	10.0	9.9
d	42	42	42.0	0.0

(f) Distribution characteristics for inverted graphs using on-the-fly graph generation.

Figure 6: Distributions of left and right node degree c and d for baseline graphs and inverted expander graphs using on-the-fly graph sampling. Parameters for Orion's G_1 graph.

Section 3.2). In particular, the verifier never performs the encoding themself but only verifies the outer Virgo proof and Merkle commitments. Hence, the verifier remains unaffected by our proposed approach 2. In addition, approach 2 combines the benefits of on-the-fly graph sampling (i.e., avoiding graph storage in off-chip memory and not relying on DDR in linear encoding) with inverted expander graph evaluation (reduced HBM accesses). This makes approach 2 more interesting for high-performance hardware architectures and inspires us to design our hardware using approach 2. Yet, changing our design to approach 1 is possible given that the target FPGA provides sufficient off-chip memory and bandwidth. The PRNG queries must be replaced by off-chip memory reads, which requires data routing across the FPGA. This data routing requires engineering effort to place and route the additional logic within the dense design. Hence, we will present estimated timing benchmarks for approach 1 in the result section (Section 7.4) and actual latencies for our FPGA design using approach 2. In the remainder of this section, we will solely focus on our inverted expander graphs using approach 2.

How to mitigate low-degree left nodes in approach 2: The inverted expander graph with on-the-fly sampling connects a right node with d left nodes randomly, as described in RunExpander of Algorithm 1. While the right nodes have the fixed degree d, this random sampling causes the left node degree c to follow a binomial distribution (Figure 6d), unlike the baseline graph. Hence, there are left nodes with fewer connections (i.e., degree lower than c), and this can compromise the minimal distance guarantees of the linear code. We address this issue using a postprocessing step (function PostProcess in Algorithm 1). Postprocessing is executed after the inverted expander graph evaluation and adds additional connections from low-degree left nodes to right nodes of the public graph. These additional connections leverage the left node's degree to meet a certain lower bound. Satisfying this lower bound on the left node degree leads to sufficient connectivity and code distance, as will be shown in Section 4.3.3. Yet, introducing additional connections to left nodes causes a small amount of additional computations. This is reflected in the number of data reads

Algorithm 1 Our on-the-fly inverted expander graph evaluation with postprocessing.

```
function RunExpander($L, k, \alpha_i, d$, seed,$W$)    ▷ $L = [l_0, \dots, l_{k-1}]$ left nodes
                                                     ▷ $d$ is degree of right-side nodes
    PRNG.Init(seed)
    $q \leftarrow \alpha_i k$                                        ▷ Number of right-side nodes
    for $t$ from $0$ to $q-1$ do                     ▷ Iterate over all right-side nodes $r_t$
        $r_t \leftarrow 0$
        for $j$ from $0$ to $d-1$ do             ▷ Find $d$ many random left-side nodes $l_i$
            $i \leftarrow$ PRNG.Random()  mod $k$                  ▷ Random node index for $L$
            $\omega_{i,t} \leftarrow$ PRNG.FieldRandom()                    ▷ Random field element
            $r_t \leftarrow r_t + l_i \cdot \omega_{i,t}$   ▷ Field multiplication with the value of left node
    $R \leftarrow [r_0, \dots, r_{q-1}]$
    return PostProcess($L$,$R$,$W$)

function PostProcess($L, R, W$)                    ▷ $W$: list of low-degree left nodes
    for each $(i, n_{rs}) \in W$ do              ▷ Node $l_i$ requires $n_{rs}$ additional connections
        for $j$ from $0$ to $n_{rs}-1$ do       ▷ Find $n_{rs}$ many random right-side nodes $r_t$
            $t \leftarrow$ PRNG.Random()  mod $\alpha_i k$
            $\omega_{i,t} \leftarrow$ PRNG.FieldRandom()
            $l_i \leftarrow L[i]$, $r_t \leftarrow R[t]$
            $r_t \leftarrow r_t + l_i \cdot \omega_{i,t}$                  ▷ Add connection from $l_i$ to $r_t$
            $R[t] \leftarrow r_t$
    return $R$
```

and writes in Table 4. Compared to inverted graphs with approach 1, about 17% more read and write operations are required, but graph reads from off-chip memory are entirely avoided.

We precompute and store the addresses of the low-degree left nodes for the selected PRNG seed and store them in memory. Unlike storing the whole graph consuming Gigabytes, this postprocessing information is small enough to fit into on-chip memory. This eliminates the need for external DDR memory and enables linear encoding on DDR-less FPGAs. Overall, our inverted expander graphs effectively reduce the memory accesses to HBM and allow for performant linear encoding. In our design, the inverted graph evaluation causes about 82.5% of the linear encoding runtime, whereas postprocessing accounts for about 17.5% of the runtime, as discussed in Section 6.2. Importantly, our on-the-fly graph sampling and postprocessing do not require off-chip memory to store the large graph.

4.3.3 Security Analysis of Inverted Expander Graphs

The expander graph evaluation in Orion and Brakedown proof systems implements a linear error-correcting encoding that detects malicious alterations in the coefficient matrix $\mathbf{W}$ by a cheating prover. To ensure security, the underlying $[n, k, d]$ linear code must have a sufficiently high relative distance δ, which intuitively guarantees strong diffusion, meaning small changes in the input significantly impact the encoded output. In this section, we demonstrate that our proposed encoding strategy based on inverted expander graphs and postprocessing maintains this crucial relative distance property. Specifically, by enforcing a minimal left-node degree greater than or equal to the fixed degree of baseline Orion's codes, our encoding achieves, with overwhelming probability, at least the same relative distance as the original Orion encoding [XZS22]. This result similarly applies to the parameter sets used in the Brakedown proof system [GLS⁺23]. While a detailed formal proof is provided later, this intuitive explanation highlights the rationale behind ensuring a minimum left-node degree to preserve encoding security.

For this analysis, we recall the functionality of Spielman codes as explained in Section 2.3 and Figure 2. To compute the $[n, k, d]$ Spielman code $E_C(\vec{x})$ of message $\vec{x} \in \mathbb{F}_q^{k}$[†], the expander graph G_1 with compression factor $\alpha_1 = \alpha$ is applied to $\vec{x}$ yielding $\vec{m}_1 = G_1(\vec{x}) \in \mathbb{F}_q^{\alpha k}$. Then, a recursive $[\alpha n, \alpha k, \alpha d]$ Spielman encoding $\vec{y}_1 = E_C^{\text{rec}}(\vec{m}_1)$ with relative distance δ is computed using the same strategy. The intermediate result $\vec{y}_1$ is the input to the second graph evaluation $\vec{z}_1 = G_2(\vec{y}_1)$, having compression factor of α_2. The overall Spielman codeword $E_C(\vec{x}) = (\vec{x} \parallel \vec{y}_1 \parallel \vec{z}_1)$ is the concatenation of $\vec{x}$, $\vec{y}_1$, and $\vec{z}_1$.

Using this construction, Brakedown in [GLS$^+$21] shows that a $[n, k, d]$ Spielman encoding $E_C(\vec{x})$ with parameter α and rate $r = n/k$ reaches a relative distance $\delta = d/n$ with overwhelming probability if all left nodes in G_1 and G_2 have a constant degree of $c_{1,\text{BD}}$ and $c_{2,\text{BD}}$ as computed in Equation 4 and Equation 5, respectively. Therein, $H(\cdot)$ denotes the binary entropy function, $\mu = r - 1 - r\alpha$, and $\nu = \delta r + \alpha\delta r + 0.03$.

$$c_{1,\text{BD}} = \left\lceil \min\left(\max(1.28\delta n, \delta n + 4), \frac{1}{\delta r \log \frac{\alpha}{1.28\delta r}} \left(\frac{110}{k} + H(\delta r) + \alpha H(\frac{1.28\delta r}{\alpha}) \right) \right) \right\rceil \quad (4)$$

$$c_{2,\text{BD}} = \left\lceil \min\left(2\delta n + \frac{n - k + 110}{\log p^2}, D \right) \right\rceil \quad (5)$$

$$D = \max\left(\frac{r\alpha H(\delta) + \mu H(\nu/\mu) + 110/k}{\alpha\delta r \log \frac{\mu}{\nu}}, \frac{r\alpha H(\delta/\alpha) + \mu H((2\delta r + 0.03)/\mu) + 110/k}{\delta r \log \frac{\mu}{2\delta r + 0.03}}, \right.$$
$$\left. (2\delta r + 0.03)\left(\frac{1}{(\alpha - \delta) r} + \frac{1}{\alpha\delta r} + \frac{1}{\mu - 2\delta r - 0.03} \right) + 1 \right) \quad (6)$$

In contrast to Brakedown where G_1 and G_2 graphs have constant left node degrees of $c_{1,\text{BD}}$ and $c_{2,\text{BD}}$, our inverted expander graphs are randomly generated on the fly which causes *variable* left node degrees following a binomial distribution (see Figure 6d). After this inverted graph evaluation, our postprocessing adds edges to the low-degree left nodes such that every left node in the public graphs G_1 and G_2 has a degree of at least $c_{1,\text{BD}}$ and $c_{2,\text{BD}}$, respectively. Other central code parameters, such as α or rate r, remain the same as in the baseline graphs. We show the security of our construction by proving Lemma 1, which is a generalization of Brakedown's [GLS$^+$21] security proof.

Lemma 1. *A $[n, k, d]$ Spielman code using expander graphs G_1 and G_2 with parameter α, rate r, and variable left-node degrees c_1 and c_2 has a relative distance of at least δ with overwhelming probability if $c_1 \geq c_{1,BD}$ and $c_2 \geq c_{2,BD}$.*

Proof. This proof significantly overlaps with [GLS$^+$21] with minor extensions. Due to the linearity of $[n, k, d]$ Spielman codes, their relative distance δ is given as

$$\delta = \frac{1}{n} \min\left(HW(\vec{x} \parallel \vec{y}_1 \parallel \vec{z}_1) \; \forall \vec{x} \in \mathbb{F}_q^k \backslash \{0\} \right). \quad (7)$$

Therein, $HW(\vec{v})$ denotes the Hamming weight of the vector $\vec{v} \in \mathbb{F}_q^k$, which is the number of non-zero elements in $\vec{v}$. From Equation 7 follows that for all messages $\vec{x}$ with Hamming weight $HW(\vec{x}) \geq \delta n$, the codeword $(\vec{x} \parallel \vec{y}_1 \parallel \vec{z}_1)$ has a Hamming weight of at least δn since $\vec{x}$ is part of the codeword. Contrarily, if $\vec{x}$ has low Hamming weight of $0 < HW(\vec{x}) < \delta n$, then $\vec{y}_1$ might have $HW(\vec{y}_1) \geq \delta n$ again leading to a codeword with Hamming weight of at least δn. Finally, if $0 < HW(\vec{x}) < \delta n$ and $HW(\vec{y}_1) < \delta n$, the Brakedown paper [GLS$^+$21] gives sufficient criteria (i) and (ii) such that $\vec{z}_1$ has $HW(\vec{z}_1) \geq \delta n$. Thus, in any case of

[†] Orion uses an extension field, hence $q = p^2$.

$\vec{x} \neq 0$, $HW(\vec{x} \parallel \vec{y}_1 \parallel \vec{z}_1) \geq \delta n$ and the code has relative distance of at least δ. The criteria from Brakedown are:

(i) $\forall \vec{x}$ with $0 < HW(\vec{x}) < \delta n : \vec{m}_1 \neq 0$
(ii) $\forall \vec{m}_1 \neq 0$ and $HW(\vec{y}_1) < \delta n : HW(\vec{z}_1) \geq \delta n$
(Note that if $\vec{m}_1 \neq 0$ then $HW(\vec{y}_1) \geq \delta \alpha n$ since E_C^{rec} has a relative distance of δ)[†]

Brakedown shows that if (i) and (ii) hold, the Spielman code E_C has a relative distance of at least δ. This is a sufficient but not necessary criterion since there might exist codewords where $HW(\vec{x}) < \delta n$, $HW(\vec{y}_1) < \delta n$, and $HW(\vec{z}_1) < \delta n$ but $HW(\vec{x} \parallel \vec{y}_1 \parallel \vec{z}_1) \geq \delta n$. Such a case is not considered in Brakedown's proof, leading to conservative code parameters with higher security margins.

Brakedown presents four claims that show that (i) and (ii) hold with overwhelming probability if the graphs G_1 and G_2 have constant left node degrees of $c_{1,\text{BD}}$ and $c_{2,\text{BD}}$ as computed in Equations 4 and 5. We extend these four claims in the following for our variable left node degrees c_1 and c_2, lower bound by $c_{1,\text{BD}}$ and $c_{2,\text{BD}}$, respectively. By extending the four claims and proving them individually, we prove Lemma 1.

Brakedown's claim 1: Brakedown [GLS+21] defines $E^{(1,\text{BD})}$ to be the event that the graph G_1 with constant left node degree of $c_{1,\text{BD}}$ contains a set of j left nodes that connect to less than $\max(j+4, 1.28j)$ right nodes. Given the constraint for $c_{1,\text{BD}}$ in Equation 4 and for $\delta < \alpha/(1.28r)$ and $0 < j < \delta n$, claim 1 of Brakedown bounds the probability $\Pr(E^{(1,\text{BD})})$ of event $E^{(1,\text{BD})}$ by

$$\begin{aligned}\Pr(E^{(1,\text{BD})}) \leq \max\Bigg(&2^{-110}, 2^{kH(\frac{15}{k})+\alpha k H(\frac{19.2}{\alpha k})-15c_{1,\text{BD}}\log(\frac{\alpha k}{19.2})},\\ &\max_{c_{1,\text{BD}}-3\leq i\leq\min(14,\delta n)} \frac{\binom{k}{i}\binom{\alpha k}{i+3}\binom{i+3}{c_{1,\text{BD}}}^i}{\binom{\alpha k}{c_{1,\text{BD}}}^i}\Bigg) \ll 2^{-100}.\end{aligned} \tag{8}$$

Extension of claim 1: Under the same assumptions as in Brakedown, we define $E^{(1)}$ as the event that the graph G_1 with variable left node degree $c_1 \geq c_{1,\text{BD}}$ contains a set of j left nodes that connect to less than $\max(j+4, 1.28j)$ right nodes. Using Brakedown's constraints of $\delta < \alpha/(1.28r)$, $0 < j < \delta n$, and $c_{1,\text{BD}} \leq c_1$, the following holds:

$$\begin{aligned}\Pr(E^{(1)}) &\leq \max\Bigg(2^{-110}, 2^{kH(\frac{15}{k})+\alpha k H(\frac{19.2}{\alpha k})-15c_1\log(\frac{\alpha k}{19.2})}, \max_{\substack{c_1-3\leq i\\ i\leq\min(14,\delta n)}} \frac{\binom{k}{i}\binom{\alpha k}{i+3}\binom{i+3}{c_1}^i}{\binom{\alpha k}{c_1}^i}\Bigg)\\ &\leq \max\Bigg(2^{-110}, 2^{kH(\frac{15}{k})+\alpha k H(\frac{19.2}{\alpha k})-15c_{1,\text{BD}}\log(\frac{\alpha k}{19.2})}, \max_{\substack{c_{1,\text{BD}}-3\leq i\\ i\leq\min(14,\delta n)}} \frac{\binom{k}{i}\binom{\alpha k}{i+3}\binom{i+3}{c_{1,\text{BD}}}^i}{\binom{\alpha k}{c_{1,\text{BD}}}^i}\Bigg)\end{aligned} \tag{9}$$

In particular, Equation 9 means that our G_1 graphs with variable left node degree lower bounded by $c_{1,\text{BD}}$ are not more likely to produce a low-connectivity graph than Brakedown's fixed left-node-degree graphs.

Proof of extended claim 1. First, $\log\frac{\alpha k}{19.2} > 0$ for all $k \geq 5$. Moreover, k is always larger than 5 due to the base case n_0 (see Section 2.3). This means that the term $2^{kH(\frac{15}{k})+\alpha k H(\frac{19.2}{\alpha k})-15c_1\log(\frac{\alpha k}{19.2})}$ decreases with increasing c_1, which leads to a more restrictive probability bound for $c_1 \geq c_{1,\text{BD}}$.

Second, in our graphs, $c_1 < \alpha k$ always holds since no left node connects to all right nodes of a graph. In addition, i is bound to $c_1 - 3 \leq i \leq \min(14, \delta n)$ due to the max

[†] This is guaranteed by applying this proof recursively to the Spielman encoding E_C^{rec}.

operation, hence $i \leq \alpha k - 3$ for relevant parameter sets. Using this,

$$\frac{\binom{k}{i}\binom{\alpha k}{i+3}\binom{i+3}{c_1}^i}{\binom{\alpha k}{c_1}^i} \geq \frac{\binom{k}{i}\binom{\alpha k}{i+3}\binom{i+3}{c_1+1}^i}{\binom{\alpha k}{c_1+1}^i} \tag{10}$$

holds since

$$\frac{\binom{i+3}{c_1}}{\binom{\alpha k}{c_1}} = \frac{(i+3)!(\alpha k - c_1)!}{(i+3-c_1)!(\alpha k)!} \geq \frac{(i+3)!(\alpha k - c_1)!}{(i+3-c_1)!(\alpha k)!} \cdot \underbrace{\frac{i+3-c_1}{\alpha k - c_1}}_{\in[0,1]} = \frac{\binom{i+3}{c_1+1}}{\binom{\alpha k}{c_1+1}}. \tag{11}$$

Thus, if c_1 is increased, $\max\limits_{c_1-3 \leq i \leq \min(14,\delta n)} \frac{\binom{k}{i}\binom{\alpha k}{i+3}\binom{i+3}{c_1}^i}{\binom{\alpha k}{c_1}^i}$ does not increase since the individual terms do not increase (according to Equation 10) and no additional terms are added to the max operation.

Given that the individual probabilities in Equation 10 do not increase with $c_1 \geq c_{1,\mathrm{BD}}$, Equation 9 holds. This proves extended claim 1.

□

Brakedown's claim 2: In the second claim in Brakedown, $E^{(2,\mathrm{BD})}$ is defined to be the event that there exists a vector $\vec{x}$ with $0 < HW(\vec{x}) < \delta n$ such that a graph G_1 not fulfilling event $E^{(1,\mathrm{BD})}$ evaluates $G_1(\vec{x}) = 0$ to zero. Brakedown shows that the probability $\Pr(E^{(2,\mathrm{BD})})$ is overwhelmingly small. Brakedown's claim 2 is not affected by our inverted expander graphs with variable left node degrees and still holds as presented in Brakedown.

□

Brakedown's claim 3: Brakedown [GLS+21] defines $E^{(3,\mathrm{BD})}$ to be the event that the graph G_2 with constant left node degree of $c_{2,\mathrm{BD}}$ contains a set of j left nodes that connect to less than $(\delta n + j + \frac{110+k(r-1)}{\log q})$ right nodes, where $\delta \alpha n \leq j < \delta n$. Given the constraint for $c_{2,\mathrm{BD}}$ in Equation 5 , let $\gamma = j/k$, $\mu = r - 1 - r\alpha$, and let $\nu' = \delta r + \gamma + 0.03$. If $\frac{r-1+110/k}{\log q} \leq 0.03$ and $2\delta r + 0.03 \leq \mu = r - 1 - r\alpha$ and $\delta \leq \alpha$ then claim 3 of [GLS+21] bounds the probability $\Pr(E^{(3,\mathrm{BD})})$ of event $E^{(3,\mathrm{BD})}$ by

$$\Pr(E^{(3,\mathrm{BD})}) \leq 2^{\alpha r k H(\frac{\gamma}{r\alpha}) + \mu k H(\frac{\nu'}{\mu}) - c_{2,\mathrm{BD}} \gamma k \log \frac{\mu}{\nu'}} \ll 2^{-110} \quad \forall k, \gamma : \alpha\delta r \leq \gamma < \delta r. \tag{12}$$

Extension of claim 3: Under the same assumptions of Brakedown, we define $E^{(3)}$ to be the event that the graph G_2 with variable left node degree of $c_2 \geq c_{2,\mathrm{BD}}$ contains a set of j left nodes that connect to less than $(\delta n + j + \frac{110+k(r-1)}{\log q})$ right nodes, where $\delta \alpha n \leq j < \delta n$. Let $\gamma = j/k$, $\mu = r - 1 - r\alpha$, and let $\nu' = \delta r + \gamma + 0.03$. If $\frac{r-1+110/k}{\log q} \leq 0.03$ and $2\delta r + 0.03 \leq r - 1 - r\alpha$ and $\delta \leq \alpha$ then

$$\begin{aligned}\Pr(E^{(3)}) &\leq 2^{\alpha r k H(\frac{\gamma}{r\alpha}) + \mu k H(\frac{\nu'}{\mu}) - c_2 \gamma k \log \frac{\mu}{\nu'}} \\ &\leq 2^{\alpha r k H(\frac{\gamma}{r\alpha}) + \mu k H(\frac{\nu'}{\mu}) - c_{2,\mathrm{BD}} \gamma k \log \frac{\mu}{\nu'}} \quad \forall k, \gamma : \alpha\delta r \leq \gamma < \delta r. \end{aligned} \tag{13}$$

Similar to the extended claim 1, Equation 13 states that our graphs G_2 with variable but lower-bound left node degrees are not more likely to yield a low-connectivity graph compared to Brakedown's graphs.

Proof of extended claim 3. First, we recall $j < \delta n = \delta k r$ hence $\gamma = j/k < \delta r$. With Brakedown's assumption $2\delta r + 0.03 \leq \mu$, we get $\nu' = \delta r + \gamma + 0.03 < 2\delta r + 0.03 \leq \mu$. Thus, the term $\gamma k \log \frac{\mu}{\nu'} \geq 0$. Using this and since $c_2 \geq c_{2,\mathrm{BD}}$, $\forall k, \gamma : \alpha\delta r \leq \gamma < \delta r$,

$$2^{\alpha r k H(\frac{\gamma}{r\alpha}) + \mu k H(\frac{\nu'}{\mu}) - c_2 \gamma k \log \frac{\mu}{\nu'}} \leq 2^{\alpha r k H(\frac{\gamma}{r\alpha}) + \mu k H(\frac{\nu'}{\mu}) - c_{2,\mathrm{BD}} \gamma k \log \frac{\mu}{\nu'}} \tag{14}$$

holds and Equation 13 is proven.

□

Brakedown's claim 4: In the fourth claim in Brakedown, $E^{(4,\mathrm{BD})}$ is defined to be the event that there exists a vector $\vec{y}$ with $\delta\alpha n \leq HW(\vec{y}) < \delta n$ such that a graph G_2 not fulfilling event $E^{(3,\mathrm{BD})}$ leads to $HW(G_2(\vec{y})) < \delta n$. Brakedown shows that the probability $\Pr(E^{(4,\mathrm{BD})})$ is overwhelmingly small. Brakedown's claim 4 is not affected by our inverted expander graphs with variable left node degrees and still holds as presented in Brakedown.

□

At this stage, the four claims are extended to graphs with variable left node degrees, and their validity is proven. Thus, Lemma 1 is proven, and our inverted expander graphs with variable, lower-bound left node degrees are secure.

□

4.4 Hardware Architecture of Linear Encoding Unit (LEU)

We implement our optimized linear encoding unit for a hardware platform with HBM as the off-chip memory. The architecture is primarily designed for the Xilinx Alveo U280 FPGA, but can also be deployed on similar FPGAs or ASICs. An overview of our Linear Encoding Unit design is presented in Figure 7. The left side in Figure 7 shows the U280 FPGA with one DDR and two HBM stacks, whereas each HBM stack provides 16 pseudo-channels. The 32 pseudo-channels are connected via DMA controllers to a total of 32 linear encoding units (LEU). It is important to note that our linear encoding only uses the HBM but does not require the DDR memory. The DDR is solely used during the proving mechanism and Merkle Tree commitments.

Inverted graph evaluation: The overall linear encoding takes the coefficient matrix $\mathbf{W}$ as input and computes the code matrix $\mathbf{C}$ in a row-wise manner. Since 32 pseudo-channels are available and $\mathbf{W}$ has $k = 128$ rows, we store four rows of $\mathbf{W}$ in one HBM pseudo-channel (PC) and dedicate one linear encoding unit (LEU) to each PC. Each LEU operates on one HBM pseudo-channel (PC) and performs the inverted expander graph evaluation (i.e., function RunExpander in Algorithm 1) for the four rows. A detailed view of a single LEU is provided on the right side of Figure 7. Each LEU features shift registers for input and output buffering. The MAC unit is responsible for computing the right nodes r_t by multiplying the randomly read coefficients $\mathbf{W}[i, j] = w_{i,j}$ (corresponding to the left nodes of the graphs) with the random weights $\omega_{i,t}$. Our field multiplier for $\mathbb{F}_{p^2}$ uses three sub-multipliers for $\mathbb{F}_p$ according to [Fam88]. The sub-multipliers map to DSP according to standard tiling and benefit from the shift-add-based reduction due to $p = 2^{61} - 1$. Note that only the first iteration of graph evaluation $G_1^{(1)}$ takes all the coefficients from a row of $\mathbf{W}$ as input. Thereafter, the output of the previous graph evaluations is used iteratively as input. The random weights and the random read addresses required in the computations are generated by the Trivium PRNG [Can06] modules shown at the bottom of Figure 7.

In the inverted graph computation, data is read concurrently from the 32 HBM channels, enabling parallel processing of independent rows. The HBM memory accesses are performed on 256 bits or 512 bits, depending on the clock frequency of the FPGA. This size is larger than the 128-bit size of our extension field elements, hence we pack, e.g. $512/128 = 4$ extension field elements into one memory word, as shown in Figure 7. When loading one 512-bit word, the input shift register serializes the four elements, which are then processed sequentially. Finally, the four computation results are again packed into one memory word, which is stored to the HBM.

This approach effectively uses the available memory capabilities and balances the resource consumption of the LEU units. The shift registers compensate for the latency

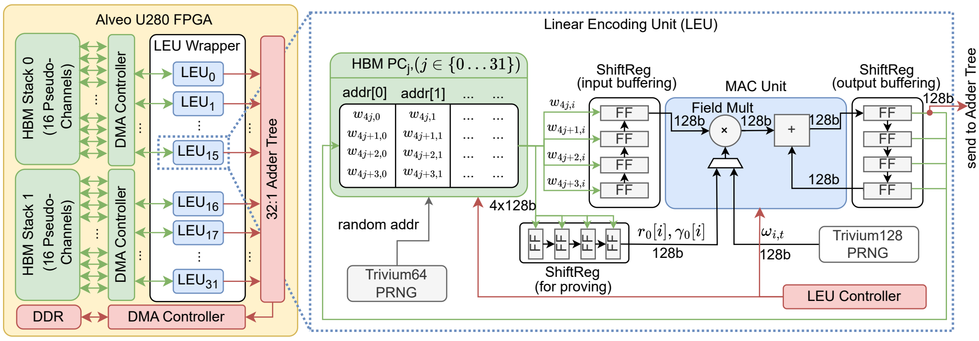


Figure 7: Architecture for parallel linear encoding and inner product computation for HBM-based FPGAs.

of random memory reads, which typically take around 3 to 4 cycles. This means that a 512-bit input arrives every 4 cycles, whereas it contains 4 field elements. Hence, the LEU is optimally utilized through our data scheduling approach.

Postprocessing: As discussed in Section 4.3, the straightforward inverted expander graph evaluation (i.e., function RunExpander in Algorithm 1) may result in an insufficient left node degree. Hence, we perform a postprocessing step (function PostProcess in Algorithm 1) which adds connections from low-degree left nodes to random right nodes. This postprocessing uses the same LEU datapath as the inverted graph evaluation, shown in Figure 7. Thereby, the address of each weak left node and its number of missing connections are stored in on-chip memory and fetched by the LEU Controller. After loading a weak left node from HBM into the LEU, it is multiplied by random weights obtained from the PRNG and added to random right nodes. Finally, the updated right nodes are stored to HBM again. This computation pattern in postprocessing is susceptible to read-after-write hazards. We avoid these hazards by carefully scheduling the postprocessing steps during design time (i.e., while computing the on-chip-memory content for weak left nodes).

Unlike in the inverted graph evaluation, all read and write operations in postprocessing follow a random pattern, which lowers the memory bandwidth (see Section 2.5). Yet, only a few left nodes require postprocessing, which causes a latency share of 17.5% within linear encoding. The benefit of our inverted graph evaluation and postprocessing technique is the reduced memory consumption: We trade a small latency overhead for not storing the whole graph, consuming Gigabytes in off-chip memory. This allows deploying our linear encoding accelerator on FPGAs without DDR memory [Xil23, Xil21] while not introducing additional HBM memory accesses.

Inner product (Proving): The proving mechanism in Orion-like schemes (discussed in Section 3.2) involves inner product computations between the $k \times n$ matrix stored in HBM and the verifier's $k = 128$ element input vector. The similarity of the inner product computation to the linear encoding allows reusing the MAC unit in the LEU. Thereby, the verifier's input vector gets split into 32 chunks with 4 field elements each. Every chunk is stored in the correct HBM channel. In the next step, the LEU persistently loads the verifier's input chunk into the shift register for proving, shown in Figure 7. Then, the 4-element vector chunk is multiplied by the 4 rows of the matrix and accumulated, yielding one output element for each column per LEU. These partial outputs per column of the 32 LEUs are fed to the Adder Tree module shown in red in Figure 7, which aggregates the results. The Adder Tree performs field addition and lazy reduction to output the n-element wide matrix-vector multiplication result. We store the result to DDR memory

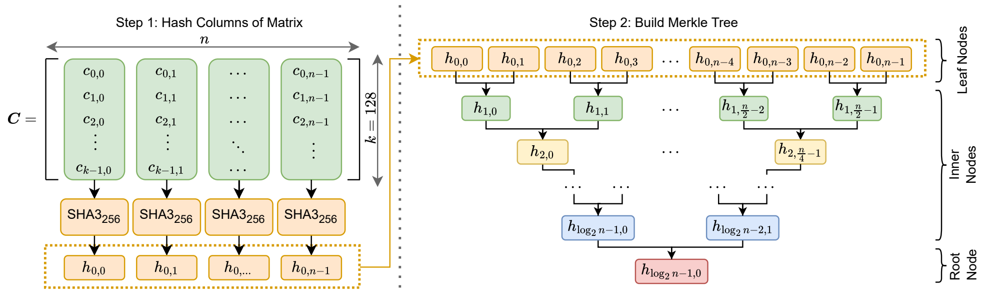

Step 1: Hash Columns of Matrix
n
$C =$
$c_{0,0}$
$c_{1,0}$
$c_{2,0}$
$c_{k-1,0}$
$c_{0,1}$
$c_{1,1}$
$c_{2,1}$
$c_{k-1,1}$
$c_{0,n-1}$
$c_{1,n-1}$
$c_{2,n-1}$
$k = 128$
SHA3$_{256}$
$h_{0,0}$
$h_{0,1}$
$h_{0,\dots}$
$h_{0,n-1}$
Step 2: Build Merkle Tree
$h_{0,0}$ $h_{0,1}$ $h_{0,2}$ $h_{0,3}$ $h_{0,n-4}$ $h_{0,n-3}$ $h_{0,n-2}$ $h_{0,n-1}$
$h_{1,0}$ $h_{1,1}$ $h_{1,\frac{n}{2}-2}$ $h_{1,\frac{n}{2}-1}$
$h_{2,0}$ $h_{2,\frac{n}{4}-1}$
$h_{\log_2 n-1,0}$ $h_{\log_2 n-2,1}$
$h_{\log_2 n-1,0}$
Leaf Nodes
Inner Nodes
Root Node

Figure 8: Computation flow of column hashing and Merkle Tree generation.

to ease routing in our design. Yet, storing the resulting vector to HBM would also be possible. Unlike in linear encoding, the off-chip memory bandwidth in proving is not as limiting since just linear reads and writes are performed, and matrix-vector multiplication has a data-compressing nature. Hence, storing the result to HBM instead of DDR would be feasible without substantially compromising latency.

Note that during inner product computation, our DMA Controller synchronizes the 32 HBM read channels. This is required as read pressure from the HBM may delay the execution of individual LEUs, whereas the Adder Tree must consume all 32 inputs of one column simultaneously. Therefore, by synchronizing the read channels, we omit expensive compensation buffering between the LEUs and the Adder Tree.

5 Accelerating Merkle Tree Generation

The generation of the Merkle Tree is the second critical operation in Orion's commitment phase. This operation takes the encoded matrix $\mathbf{C}$ from linear encoding as input and requires two steps. In step 1, the columns $\mathbf{C}[:\, i] = [c_{0,i} \;\dots\; c_{k-1,i}]^T$ are hashed to leaf nodes $h_{l,i}$, where $l = 0$ indicates the leaf level within the Merkle Tree. This step is shown on the left side in Figure 8 and requires data rearrangement. Building upon these leaf nodes, step 2 constructs a Merkle Tree (right in Figure 8). We describe our hardware design for this dual-step computation in the remainder of this section by starting with the column hashing and then advancing to the Merkle Tree computation. Although conceptually simple, implementing a Merkle Tree becomes challenging when handling large datasets stored in off-chip memory.

5.1 Column Hashing

The hashing of columns operates on the linear encoded matrix $\mathbf{C}$ with its $k = 128$ rows. Since the matrix $\mathbf{C}$ is the result of the row-wise linear encoding, the data resides in HBM memory according to the memory layout for linear encoding (explained in Section 4). Specifically, four rows of $\mathbf{C}$ are stored in one HBM pseudo-channel (PC), which means that the data of individual matrix columns is distributed over all 32 HBM PCs. This memory layout is shown in Figure 9. In the i-th PC, the rows $\mathbf{C}[4i\, :]$ to $\mathbf{C}[4i+3\, :]$ are stored. Moreover, address j in the i-th PC holds 4 field elements $\mathbf{C}[4i : j]$ to $\mathbf{C}[4i+3 : j]$, which are are packed in a 512-bit memory word. The linear encoding process efficiently utilizes this memory layout as each LEU operates on just one PC. Yet, in column hashing, the data of one column is scattered across all PCs. This requires special synchronized handling during data fetches from HBM. Ensuring a high-throughput data rearrangement for column hashing is challenging since the high HBM bandwidth must be optimally exploited to boost performance. Next to the non-trivial synchronization of HBM PCs, the

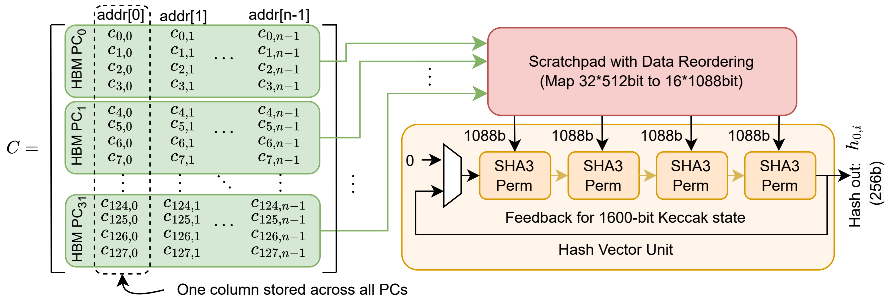

$C =$
addr[0]
addr[1]
addr[n-1]
HBM PC$_0$
HBM PC$_1$
HBM PC$_{31}$
$c_{0,0}$ $c_{1,0}$ $c_{2,0}$ $c_{3,0}$
$c_{0,1}$ $c_{1,1}$ $c_{2,1}$ $c_{3,1}$
$c_{0,n-1}$ $c_{1,n-1}$ $c_{2,n-1}$ $c_{3,n-1}$
$c_{4,0}$ $c_{5,0}$ $c_{6,0}$ $c_{7,0}$
$c_{4,1}$ $c_{5,1}$ $c_{6,1}$ $c_{7,1}$
$c_{4,n-1}$ $c_{5,n-1}$ $c_{6,n-1}$ $c_{7,n-1}$
$c_{124,0}$ $c_{125,0}$ $c_{126,0}$ $c_{127,0}$
$c_{124,1}$ $c_{125,1}$ $c_{126,1}$ $c_{127,1}$
$c_{124,n-1}$ $c_{125,n-1}$ $c_{126,n-1}$ $c_{127,n-1}$
One column stored across all PCs
Scratchpad with Data Reordering
(Map 32*512bit to 16*1088bit)
1088b
1088b
1088b
1088b
0
SHA3 Perm
SHA3 Perm
SHA3 Perm
SHA3 Perm
Feedback for 1600-bit Keccak state
Hash Vector Unit
Hash out: $h_{0,i}$
(256b)

Figure 9: Computation flow and schematic architecture of the Hash Vector Unit.

hashing must be performant to serve the high HBM bandwidth. We hence instantiate multiple SHA3-256 permutation units (SHA3 Perm), which form our Hash Vector unit shown in Figure 9. However, running multiple SHA3 Perm units in parallel introduces two challenges: (1) efficiently synchronizing the HBM PCs and rearranging the columns to allow for parallel hashing of the leaf nodes, and (2) parallelizing the hashing of multiple columns simultaneously to optimally utilize the high throughput of HBM. We address these two challenges in the next subsections.

5.1.1 Pipelined Feeding of Hash Unit

The matrix **C** is the result of the linear encoding. Since linear encoding operates row-wise on the matrix, each row vector of **C** resides in one PC. Yet, unlike the linear encoding, column hashing operates *column-wise* on the matrix **C**. This fact requires a data rearrangement to emulate a transpose operation. A straightforward possibility to perform this step is to explicitly transpose the matrix **C** after encoding and before hashing by reading the data from HBM, performing the transpose operation on-chip, and storing the result back to HBM. This would make each column reside in a single PC and simplify the memory layout for pipelined hashing: Just a single linear read operation from one PC would be sufficient to obtain a whole column. However, while this approach simplifies memory access, it introduces an additional latency overhead caused by the explicit transposing operation.

We instead propose a more suitable approach, which involves synchronized reading operations across the 32 PCs and a streamlined data rearrangement of the encoded matrix **C**. We use an on-chip Scratchpad buffer that dynamically maps the 32×512-bit read interface to a variable number of 1088-bit SHA3 inputs. In our design, each HBM PC delivers parts of a column that are buffered and rearranged in the Scratchpad memory. This buffering also compensates for asynchronous reads due to randomly occurring memory pressure. As soon as a whole column is present in the Scratchpad, the hashing operation of this column is issued. Figure 9 shows the integration of our Scratchpad buffer into the hashing unit. Through the on-chip buffering, we establish a synchronized reading operation over the 32 PCs. In addition, our method allows for linear and pipelined reading from HBM to enhance throughput. Our design does not need an explicit transpose step and compensates for random read pressure, thereby providing pipelined input for a variable number of parallel hashing units.

5.1.2 Scalable Multi-Pass Column Hashing

The scratchpad unit described in the previous section enables the parallelization of column hashing by reshaping the 32×512-bit HBM output to a variable number of 1088-bit SHA3-256 inputs. Now that the data is in the right format and ready for absorption into

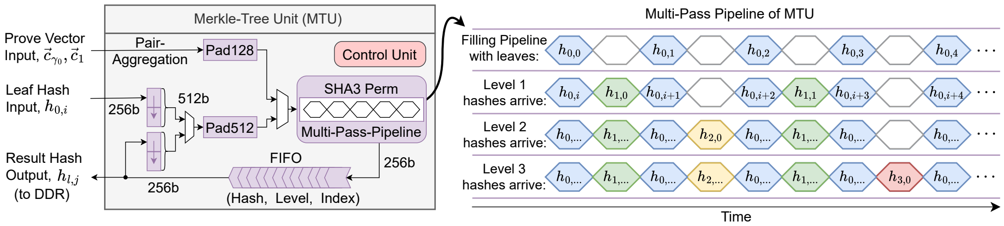

Merkle-Tree Unit (MTU)
Prove Vector Input, $\vec{c}_{\gamma_0}, \vec{c}_1$
Pair-Aggregation
Pad128
Control Unit
Leaf Hash Input, $h_{0,i}$
256b
512b
SHA3 Perm
Pad512
Multi-Pass-Pipeline
Result Hash Output, $h_{l,j}$ (to DDR)
FIFO
256b
256b
(Hash, Level, Index)
Multi-Pass Pipeline of MTU
Filling Pipeline with leaves:
$h_{0,0}$ $h_{0,1}$ $h_{0,2}$ $h_{0,3}$ $h_{0,4}$ ⋯
Level 1 hashes arrive:
$h_{0,i}$ $h_{1,0}$ $h_{0,i+1}$ $h_{0,i+2}$ $h_{1,1}$ $h_{0,i+3}$ $h_{0,i+4}$ ⋯
Level 2 hashes arrive:
$h_{0,...}$ $h_{1,...}$ $h_{0,...}$ $h_{2,0}$ $h_{0,...}$ $h_{1,...}$ $h_{0,...}$ $h_{0,...}$ ⋯
Level 3 hashes arrive:
$h_{0,...}$ $h_{1,...}$ $h_{0,...}$ $h_{2,...}$ $h_{0,...}$ $h_{1,...}$ $h_{0,...}$ $h_{3,0}$ $h_{0,...}$ ⋯
Time

Figure 10: Architecture (left) and timing (right) of our multi-pass pipeline for Merkle Tree generation.

the SHA3-256 state, we can instantiate multiple fully unrolled and pipelined SHA3-256 permutation units, as shown in the bottom right corner of Figure 9. Each permutation unit performs one SHA3 permutation operation together with the data absorption at a rate of 1088 bits. For the column hashing, we need to hash $k = 128$ field elements $c_{i,j}$, each having 128 bits. Therefore, we need to evaluate $\lceil 128 \cdot 128/1088 \rceil = 16$ SHA3 permutations.

A simple approach for hashing the 16 absorption stages is to instantiate 16 unrolled SHA3 permutation units. This allows the highest throughput but also introduces a substantial area consumption. This high area consumption results from the large, fully unrolled SHA3 Perm units. Moreover, the high area consumption does not justify the performance gain since hashing in Orion accounts for clearly less runtime compared to linear encoding. Alternatively, our hardware offers customized tradeoffs by instantiating a power-of-two SHA3 Perm units. In essence, the hardware design supports 1, 2, 4, 8, or 16 SHA3 Perm units. For example, Figure 9 shows an architecture with just four SHA3 Perm units. To still hash a full column of **C**, multiple passes through the pipeline are required. The according feedback datapath is also shown in Figure 9. In our experiments, a Hash Vector with up to 8 SHA3 Perm units can be instantiated on the U280 FPGA. Using this configuration, the Hash Vector unit consumes 429k LUTs, which is significantly larger than the linear encoding units with 125k LUTs. This configuration allows for the best performance on the target FPGA.

5.2 Merkle Tree Generation

After hashing the columns into leaf nodes $h_{0,i}$, the Merkle Tree is constructed. Building the tree involves iteratively hashing pairs of nodes to form parent nodes, as shown in Figure 8. In contrast to the leaf node hashing, the pairs of hashes only have 512 bits and hence can be processed by a single SHA3 permutation unit with its 1088-bit absorption. In addition, at most one leaf node per clock cycle is provided by our pipelined Hash Vector unit as described above. Based on these observations, we propose a fully pipelined multi-pass architecture called the Merkle Tree Unit (MTU). An overview of the MTU is shown on the left side of Figure 10.

The MTU has one input for the leaf node hashes $h_{0,i}$ from the Hash Vector unit and one output for the hashes of the Merkle Tree nodes $h_{l,j}$, where l is the level of the hash and j the index within the level. The unit operates as follows: it receives a stream of leaf node hashes (256-bit hashes) from the Hash Vector unit. Thereby, at most one hash arrives in each clock cycle, which requires a single-cycle buffer to aggregate the pair of hashes. After both input hashes are available, they are passed to a single fully unrolled, pipelined SHA3-256 permutation unit for hashing. Due to this, the SHA3 Perm pipeline is utilized every second cycle for processing leaf node hashes. This is shown in the top row of the timing diagram in Figure 10, right. To avoid the shown pipeline bubbles and ensure continuous operation, our architecture issues the level 1 hashing within the bubbles of the leaf node hashing, as soon as level 1 nodes $(h_{1,j})$ arrive in the feedback path shown in

Figure 10. Similar to the leaf node hashing, two hashes are combined in level 1. Therefore, one level 1 hashing operation is issued every four cycles. This is shown in the second row of the timing diagram in Figure 10. We also apply this strategy also for the remaining Merkle Tree levels, which allows us to fully fill the pipeline of the single SHA3 Perm unit. The shown feedback FIFO consists of one 256-bit wide buffer for each Merkle Tree level to properly arrange the data for hashing. Using our approach, the instantiated hardware units are optimally utilized, whereby just a negligible control and buffering overhead is introduced.

5.3 Hashing and Merkle Tree for Proving Mechanism

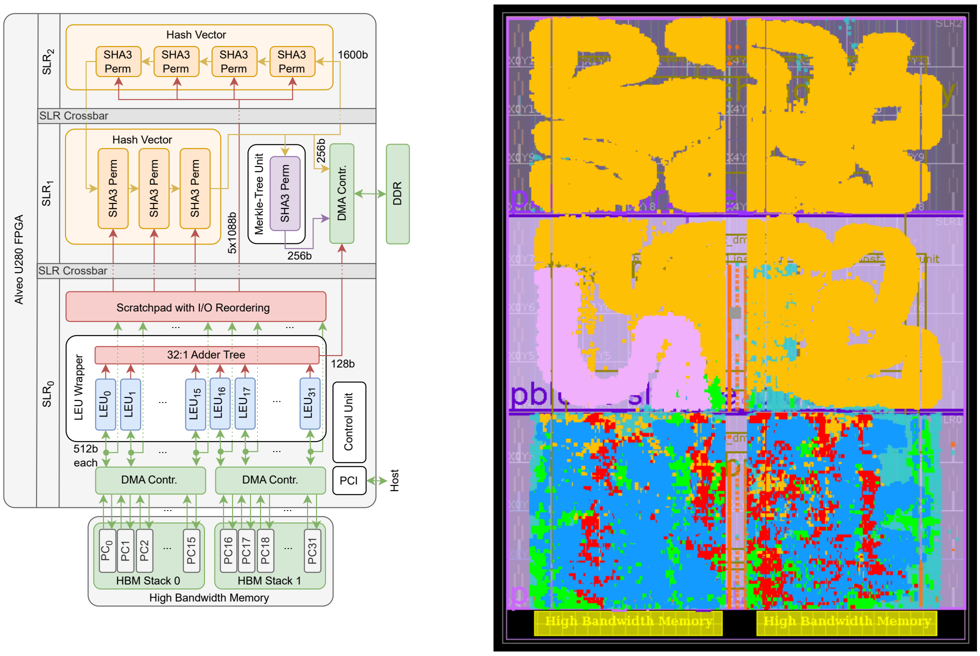
Orion's proving mechanism – discussed in Section 3.2 – performs a Merkle commit to the n-element wide vectors $\vec{c}_{\gamma_0}$ and $\vec{c}_1$. The Merkle commitment consists of two steps: First, each of the n elements is hashed using SHA3 (we again refer to this step as leaf node hashing). Second, the resulting hashes serve as leaf nodes for the Merkle Tree computation. Our leaf node hashing re-uses the Merkle Tree Unit from Figure 10. The inputs of the MTU are the 128-bit wide elements of $\vec{c}_{\gamma_0}$ and $\vec{c}_1$, which are padded to the SHA3 absorb rate of 1088 bits. The corresponding unit is shown in green in Figure 10. Thereafter, a single pass through the SHA3 Perm pipeline yields the leaf node hash, which is stored to DDR memory. After leaf node hashing is completed, the Merkle Tree is built upon the leaf hashes. The Merkle Tree computation follows the same approach as explained in Section 5.2 and re-utilizes the existing hardware components. This effectively accelerates the overall proving mechanism in Orion. Note that the Brakedown scheme does not involve Merkle commitments of $\vec{c}_{\gamma_0}$ or $\vec{c}_1$ since Brakedown directly sends these vectors to the verifier. Hence, our accelerator can skip the discussed steps when targeting Brakedown.

6 Overall Architecture and Results

Our architecture is designed to accelerate polynomial commitment and proof generation, which causes the majority of the latency in a proof system. As an implementation platform, we use the Xilinx Alveo U280 card, which contains a large FPGA coupled with two HBM2 devices using 32 pseudo-channels. In addition, we used the Vivado 2022.2 toolchain with the 'performance explore' strategy for implementation.

Physical implementation challenges: Large high-end FPGAs, such as the Xilinx Alveo U280, utilize Stacked Silicon Interconnect (SSI) technology [Xil12], which integrates multiple FPGA dies, referred to as Super Logic Regions (SLRs), onto a passive silicon interposer. Each SLR operates as an independent FPGA die, and the SLRs are connected through the silicon interposer. While this multi-SLR architecture enables high resource capacity, it introduces significant physical design constraints, as analyzed in LEAPS [DTM+24]. For example, Inter-SLR routing is limited by a fixed number of Super Long Lines (SLLs) that add extra latency between logic that is separated into different SLRs. Furthermore, excessive cross-SLR connections can lead to routing congestion and timing violations. As a result, an RTL design that appears resource-feasible when synthesized by an EDA tool may still be unplaceable if it does not account for proper partitioning across SLRs. Hence, achieving timing closure and implementation feasibility on such multi-SLR FPGAs requires explicit partitioning on the RTL level and careful floorplanning across SLRs, particularly in memory-intensive and highly parallel designs like ours, since the full data of HBM can not cross SLRs simultaneously.

Our SLR- and HBM-aware design layout: As illustrated in Figure 11a, the design spans all three SLRs and includes dedicated units for linear encoding, Merkle hashing, Merkle



(a) Overall architecture. SLR and unit sizes are not to scale with actual resource utilization.

(b) Real layout on the Alveo U280 FPGA.

Figure 11: Overall architecture for polynomial commitment and proving on U280 FPGA platform with 3 SLRs and HBM. The sub-units are identically colored in both figures to illustrate the placement on real FPGA.

Tree construction, memory controllers, and communication logic for interfacing with HBM and DDR memory. The linear encoder is implemented entirely within SLR0, colocated with the HBM memory controller to ensure minimal latency and localized routing. It consists of 32 parallel Linear Encoding Units (LEUs), each mapped to one HBM pseudo-channel. Each LEU processes four rows of the input coefficient matrix $\mathbf{W}$ and performs encoding using our inverted expander graph technique. Encoded data is then stored in HBM and later retrieved for the Merkle hashing phase.

For Merkle hashing and tree generation, encoded data is passed from HBM to a Scratchpad buffer in SLR0. The Scratchpad reorders the data layout before feeding it to the SHA3 Perm hashing cores distributed across SLR1 and SLR2. This distribution alleviates routing congestion that would arise from localizing hashing cores in a single SLR, while also balancing logic usage across SLRs. Afterwards, the hash outputs are fed into a pipelined Merkle Tree Unit (MTU) that performs the tree construction using its components placed in SLR1. The MTU uses feedback-aware scheduling to construct the Merkle Tree in a throughput-optimized manner while respecting interconnect constraints.

Our design achieves a 200 MHz clock frequency (after place-and-route) through carefully distributing logic across SLRs. The physical floorplan of the implementation is shown in Figure 11b with a color matching to illustrate the logic distribution. The carefully planned distribution minimizes cross-region communication and therefore places co-locating memory-intensive units with their associated controllers into specific region. Achieving such a frequency for a heterogeneous design of this scale and complexity, especially given the stringent placement and interconnect constraints imposed by multi-SLR architectures, is quite impressive.

Table 5: Resource utilization on Alveo U280 FPGA.

Modules	**#Units**	**LUTs**	**REGs**	**DSPs**	**BRAMs**	**#URAMs**
Total Alveo U280 FPGA	1	1,303k	2,607k	9,024	2,160	960
Processor	1	775,378	824,278	1,162	607.5	256
- Platform	1	118,102	156,188	10	71.0	-
- Cryptoprocessor	1	657,276	668,090	1,152	536.5	256
Hierachy of Cryptoprocessor						
DMA HBM Controller	1	22,728	26,078	-	514.0	-
- Read Interface	32	391	367	-	7.5	-
- Write Interface	32	222	232	-	7.5	-
DMA DDR Controller	1	848	922	-	18.5	-
- Read Interface	1	347	343	-	7.5	-
- Write Interface	1	240	253	-	7.5	-
Linear Encoding Wrapper	1	124,595	171,075	1,152	-	-
- Linear Encoding Unit	32	3,721	4,580	36	-	-
Scratchpad	1	21,523	19,293	-	-	256
Hash Vector Unit	1	429,112	384,146	-	-	-
- SHA3 Perm Unit	8	52,456	42,896	-	-	-
Merkle Tree Unit	1	53,351	42,331	-	-	-
- SHA3 Perm Unit	1	49,908	37,572	-	-	-

Side-channel security: Side-channel resistance is an important aspect in cryptographic implementations. To avoid timing side-channels, we carefully designed our hardware accelerator to have a secret-independent execution time. Specifically, the control flow and memory access pattern in linear encoding only depend on the *public* expander graph information. Hashing, Merkle Tree construction, and proving follow a linear execution flow and uniform memory access pattern. Hence, the execution time in our hardware design – although varying due to memory pressure – *does not* correlate with any secret information.

6.1 Resource Utilization Results

Table 5 provides a detailed breakdown of the resource utilization for the key hardware components in our implementation. The overall processor design, described in the previous section and illustrated in Figure 11, consumes 775k LUTs (60% of device capacity), 824k registers (32%), 1,162 DSPs (13%), 607.5 BRAMs (28%), and 256 URAMs (27%). The processor consists of two main components: the *Platform* and the *Cryptoprocessor*. The Platform includes the block design and all interfacing logic between the RTL-based Cryptoprocessor and hardware components such as HBM, DDR, and other peripherals.

The majority of the area is consumed by our *Cryptoprocessor*, which includes DMAs for HBM and DDR, Linear Encoding, Adder Tree, Scratchpad, Hashing, and Merkle Tree units. One of the major challenges when working with HBM is that the DMA subsystem must support extremely high data throughput. In the case of the U280, the DMA subsystem has to handle up to 32×512 bits per cycle for both the read and write channels. Due to this, the subsystem has a noticeable BRAM footprint to buffer data, which is often overlooked in theoretical ASIC-based hardware accelerator proposals. The HBM DMA controller alone uses 514 BRAMs, as each channel consumes about 2 × 7.5 BRAMs. We opted for a BRAM-based buffering strategy to minimize stalling during read and write turnarounds.

While the DMA subsystem consumes large on-chip memory, the Linear Encoding Wrapper causes high logic consumption. It uses 1,152 DSPs, 125k LUTs, and 171k registers

Table 6: Our latency in microseconds (μs) of commitment sub-operations. Linear encoding includes inverted graph evaluation (IGE) and post-processing (PP). Column hashing supports different numbers of SHA3 Perm units. MT refers to Merkle Tree generation.

Size	**Linear Encoding**			**Column Hashing**					**MT**	**Total**‡
lg(N)	IGE	PP	Total	$1^{\dagger}$	$2^{\dagger}$	$4^{\dagger}$	$8^{\dagger}$	$16^{\dagger *}$		**Commit**
16	241	54	295	110	55	28	14	7	9	318
18	969	207	1,176	358	179	90	45	22	24	1,245
20	3,884	808	4,692	1,341	671	335	168	84	86	4,946
22	15,540	3,263	18,803	5,274	2,637	1,318	659	330	332	19,794
24	62,171	13,065	75,236	21,002	10,501	5,251	2,625	1,313	1,315	79,176
26	248,692	51,908	300,600	83,917	41,958	20,979	10,490	5,245	5,247	316,337
28	994,777	208,311	1,203,088	335,575	167,788	83,894	41,947	20,973	20,975	1,266,010

† Number of SHA3 Perm units within the Hash Vector; *Results for 16 SHA3 Perm units are extrapolated and not implemented on U280 due to the high resource utilization; ‡ using 8 SHA3 Perm;

– primarily for modular multiplication – making it the second-largest subsystem in our design. Each of the 32 Linear Encoding Units (LEUs) is tightly coupled to one HBM PC.

The Scratchpad module instantiates 256 URAMs to buffer, reorder, and feed the SHA3 hashing cores. Using URAMs in the Scratchpad is ideal for this purpose as they reduce LUTs and REGs consumption. On the other hand, using BRAMs for the Scratchpad is less optimal: Many of the BRAMs in SLR0 are dedicated to the HBM DMA, and further increasing BRAM usage would lead to routing and congestion issues.

The hashing pipeline consists of eight fully unrolled SHA3 permutation cores used for column-wise hashing. Note that only up to eight units fit on the Alveo U280, and 16 units would require a larger FPGA platform, Additionally, the Merkle Tree Unit includes another independent unrolled SHA3 core to allow parallel construction of the Merkle Tree. The hashing components account for the majority of LUT and REG consumption in our architecture. This is due to the fully unrolled and deeply pipelined SHA3 Perm units, which allow high hashing capability. The Hash Vector and Merkle Tree units together consume around 73% of the LUTs in the Cryptoprocessor, while the Linear Encoding Wrapper causes 19% of LUT consumption. This shows that hashing is limited by logic resources and benefits from parallelism, whereas linear encoding is memory-bound. In particular, adding more computational units to the Linear Encoder Wrapper will not improve performance due to the limited off-chip memory bandwidth. This highlights the importance of our inverted expander graphs to reduce memory pressure, thereby allowing higher performance.

6.2 Timing Results for Commitment

Table 6 presents the latency results of our hardware-accelerated commitment phase of Orion. The commitment consists of linear encoding (inverted expander graph evaluation + post-processing), column hashing, and Merkle Tree construction. The timing is collected for the Alveo U280 FPGA running at 200MHz and covers a range of $N = 2^{16}$ to $N = 2^{28}$ degree polynomials.

The latency of linear encoding consists of inverted expander graph evaluations and post-processing steps. In our design, inverted graph evaluation lasts between 241μs and 995ms for the reported range of N. Post-processing is more lightweight and takes between 54μs and 208ms, which is about 17% of the overall linear encoding runtime. The whole linear encoding latency in our architecture for $N = 2^{16}$ is 295μs and reaches up to 1.2s for $N = 2^{28}$. When N is doubled, the linear encoding latency also roughly doubles, which is expected in the linear-prover-time Orion scheme.

Considering the column hashing latency, Table 6 provides benchmarks for different numbers of SHA3 Perm units in the Hash Vector. The slowest and most lightweight configuration with 1 SHA3 Perm has a latency between 110μs and 336ms, while the largest

Table 7: Our latency in microseconds (μs) of proving sub-operations: Inner product, vector hashing, and Merkle Tree construction.

Size	**Proving Mechanism**				**Total**
log(N)	Inner Product	Vector Hashing	Merkle Tree	Other[†]	**Prove**
16	23	8	9	35	75
18	83	22	24	117	246
20	293	58	86	388	825
22	1,034	158	332	1,429	2,953
24	3,654	428	1,315	5,429	10,826
26	12,916	1,160	5,247	19,467	38,790
28	45,649	3,141	20,975	69,983	139,748

[†] Not accelerated on hardware.

configuration with 16 SHA3 Perm units has a latency between 7μs and 21ms for N between 2^{16} and 2^{28}. Note that at most 8 SHA3 Perm units fit on our targeted FPGA. In this configuration, column hashing takes between 14μs and 42ms for the reported N. Increasing the number of SHA3 Perm units leads to a linear decrease in column hashing latency, thereby improving performance through higher resource consumption. Furthermore, the hashing latency is linear in the polynomial size. The Merkle Tree computation takes up to 21ms for $N = 2^{28}$, as reported in Table 6.

Observing the timing in Table 6 reveals the two distinct characteristics of linear encoding and hashing. While the latency of column hashing can be reduced by using more SHA3 Perm units, the latency of linear encoding is memory-bound. This makes further acceleration of linear encoding highly challenging, given the limited off-chip memory bandwidth. As a result, the latency of linear encoding is up to 19× higher than the latency for column hashing (using 8 SHA3 Perm) and Merkle Tree generation.

6.3 Timing Results for Proof Generation

The proof generation phase in Orion follows the commitment phase and consists of Inner Product, Vector Hashing, and Merkle Tree generation. The main challenge during proof generation is that all LEU units must be synchronized to enable pipelining through the Adder Tree that forms each component of the vector-vector product. Our DMA controller orchestrates synchronized data access across all memory channels to ensure aligned and efficient aggregation.

Table 7 shows that in our design, the inner product computation lasts between 23μs and 46ms for the reported range of N. Vector hashing is more lightweight and takes between 8μs and 3ms, while Merkle Tree generation takes between 9μs and 21ms. In addition to these results, we also have *Other* operations that include the tensorization and evaluation computation, which we leave in software. The whole prove latency in our architecture for $N = 2^{16}$ is 75μs and reaches up to 140ms for $N = 2^{28}$.

Overall, we do not introduce any new hardware modules for proving, except for comparably small control logic and an adder tree. This implies that we reuse existing units developed for commitment during the proof generation. The MAC units used for linear encoding are repurposed to perform the inner product operations in proof generation, enabling the parallelization of matrix-vector multiplications across 32 HBM pseudo-channels. In addition, the Merkle Tree generation is handled by the same unit used during commitment, ensuring minimal resource overhead.

Table 8: Timing comparison in microseconds (μs) of commitment and proving between this work and Orion's software (SW) [Su].

Size	**Commit**			**Prove**		
log(N)	SW	Our	**Speedup**	SW	Our	**Speedup**
16	40,000	318	**126**×	4,000	75	**53**×
18	155,000	1,245	**124**×	10,000	246	**41**×
20	701,000	4,946	**142**×	35,000	825	**42**×
22	3,175,000	19,794	**160**×	135,000	2,953	**46**×
24	14,756,000	79,176	**186**×	593,000	10,826	**55**×
26	60,011,000	316,337	**190**×	2,495,000	38,790	**64**×
28	334,250,000	1,266,010	**264**×	9,093,000	139,748	**65**×

7 Comparison with Baseline Orion

We now compare the performance of our hardware implementation against the reference software implementation of Orion [XZS22, Su]. All software benchmarks (as presented in Tables 1 to 3) used for comparison are collected on an AMD EPYC 9754 @2.25GHz server CPU, while hardware benchmarks are from a Xilinx Alveo U280 FPGA operating at 200MHz. We used Vivado 2022.2 for synthesis, place, and route. Our architecture employs 32 Linear Encoder Units and 8 parallel SHA3 Perm units. We first consider the commitment and proving procedures individually. Then, we compare an end-to-end Orion execution including all steps. While all our concrete benchmarks stem from our on-the-fly sampled inverted expander graphs (Approach 2 of Section 4.3.2), we also present estimated benchmarks for our *stored* inverted expander graphs as presented in Approach 1 of Section 4.3.2.

7.1 Comparison of Commitment Phase

The commitment phase is the most performance-critical part of Orion. Commitment includes the linear encoding of the coefficient matrix **W** in a row-wise manner. Subsequently, the columns of the encoded matrix are hashed, and a Merkle Tree is built upon the column hashes. The sub-operations of commitment and their individual latencies on our hardware are presented in Section 6.2.

Table 8 compares the commitment latency of our hardware accelerator with the Orion software baseline. This shows that our accelerator reduces the commitment latency by factors ranging from 124× to 264× for $N = 2^{16}$ to $N = 2^{28}$. The speedup grows with increasing N. In particular, when N quadruples, the latency in software increases by about 4.5× whereas our latency only increases by about 4×. The super-linear increase in software can be explained by CPU-specific overheads, such as high memory and cache pressure due to the large datasets and the random access pattern during linear encoding. Considering a commitment for a $N = 2^{28}$ polynomial, Orion software requires more than 5.5 minutes, whereas our FPGA accelerator only needs 1.3 seconds. This significant performance improvement stems from efficient off-chip memory management and memory-aware compute acceleration techniques.

7.2 Comparison of Proof Generation

After Orion's commitment phase, the proof generation is done. Proving consists of a matrix-vector multiplication (inner product) and a Merkle Tree generation over the output vector. The detailed timing of the sub-operations in proving obtained from our hardware design is presented in Table 7. Therein, the category 'Other' summarizes operations like

Table 9: Overall timing comparison in microseconds (μs) between this work and Orion's software (SW) [Su].

Size	**Our Orion Prover**			**SW Orion**	**Speedup**	Virgo-Prove	**Speedup**
lg(N)	Commit	Prove	Total	**Prover**	**Orion Prover**	Protocol†	**End-2-End**
16	318	75	393	44,000	**112**×	37,000	2.17×
18	1,245	246	1,491	165,000	**111**×	102,000	2.58×
20	4,946	825	5,771	736,000	**128**×	483,000	2.49×
22	19,794	2,953	22,747	3,310,000	**146**×	1,370,000	3.36×
24	79,176	10,826	90,002	15,349,000	**171**×	5,849,000	3.57×
26	316,337	38,790	355,127	62,506,000	**176**×	26,457,000	3.32×
28	1,266,010	139,748	1,405,758	343,343,000	**244**×	118,554,000	3.85×

† Not accelerated on hardware.

tensorization and evaluation computation, which are not accelerated in our design.

The overall speedup achieved by our design for proving is presented in Table 8. For smaller sizes, such as $N = 2^{16}$, we reduce the proof generation time from 4ms to just 75μs. For the largest evaluated input size, $N = 2^{28}$, our architecture reduces the latency from 9.1s to 140ms. This results in a speedup between 41× and 65×, which is lower than the speedup in commitment. The reason for the lower proving speedup at larger N is that the tensorization and evaluation computation are still performed in software. Yet, the proving mechanism in our hardware design only causes a small area overhead since it reuses the Linear Encoder units and the hashing units from commitment.

7.3 Comparison of End-to-End Latency

We evaluate the overall performance impact of our hardware accelerator by measuring the execution time of Orion, including both the commitment and proof generation phases. Table 9 reports these latencies of our work and compares them to the software baseline. For the largest evaluated size, $N = 2^{28}$, the execution time of commit+prove is reduced from 343s to just 1.4s, resulting in a 244× speedup. Throughout all reported polynomial sizes N, we achieve speedups of two orders of magnitude for Orion's core operations.

Table 9 also reports the end-to-end speedup, which includes hardware-accelerated commitment and proving alongside the Virgo CP-SNARK composition protocol running in software (excluding communication overhead). In the measured scenario, where each component (commitment, proving, and Virgo composition) is executed once, the software baseline shows that commitment and proving account for approximately 74% of the total runtime, while Virgo accounts for the remaining 26%. After hardware acceleration, the Virgo component becomes dominant, constituting 98.8% of the total end-to-end runtime. Consequently, the overall system achieves an end-to-end speedup of up to 3.85× for $N = 2^{16}$ to $N = 2^{28}$ when each component (commitment, proving, and Virgo composition) is executed only once.

This measured speedup aligns closely with the theoretical upper bound of 3.9× given by Amdahl's law [Gus88], confirming the effectiveness of our optimization strategy. However, it is important to note that this evaluation reflects a conservative, one-time commitment, proving, and Virgo composition usage model. In applications where multiple commitment and proving operations are performed (e.g., per layer of a neural network), the CP-SNARK composition can be invoked only once to aggregate all proofs [APPK24]. In such a case, the performance benefits of our accelerator would be significantly amplified. Thus, while our focus is on accelerating Orion's most computationally intensive phases, further research is needed to reduce the latency of the CP-SNARK protocol itself, which remains outside the scope of this work.

Table 10: Estimated timing results in microseconds (μs) using our *stored* inverted expander graphs (Approach 1) and comparison to the software baseline.

Size	**Our Orion Prover** (Approach 1)			**SW Orion**	**Speedup**	Virgo-Prove	**Speedup**
lg(N)	Commit	Prove*	Total	**Prover***	**Orion Prover**	Protocol†*	**End-2-End**
16	268	75	343	44,000	128×	37,000	2.17×
18	1,097	246	1,343	165,000	123×	102,000	2.58×
20	4,416	825	5,241	736,000	140×	483,000	2.50×
22	17,685	2,953	20,638	3,310,000	160×	1,370,000	3.37×
24	70,776	10,826	81,602	15,349,000	188×	5,849,000	3.57×
26	283,137	38,790	321,927	62,506,000	194×	26,457,000	3.32×
28	1,132,584	139,748	1,272,332	343,343,000	270×	118,554,000	3.85×

† Not accelerated on hardware; *same as in Table 9

7.4 Estimated Benchmarks for Stored Inverted Expander Graphs

As explained in Section 4.3.2, we designed our FPGA architecture based on *on-the-fly* inverted expander graph sampling (approach 2). This section gives estimated results for *stored* inverted expander graphs (approach 1). Our hardware architecture can be extended for approach 1 by replacing PRNG queries with off-chip memory reads to obtain the large graph structure. The graph structure consumes up to 1.1 Gigabytes. We assume to store the graph in DDR memory of the Alveo U280 to not compromise HBM bandwidth. Based on these differences, we estimate the results for approach 1. Our estimation cycle accurately considers the low-level details of our linear encoding units, such as pipeline depth and HBM read latency, and adds a 5% safety margin for non-idealities and control overhead. The estimation results are shown in Table 10.

Considering the commitment latency of approaches 1 and 2, the results in Table 9 and Table 10 show that approach 1 reduces commitment latency by 10% to 12%. For small $N = 2^{16}$, the latency reduction is even 16%. This is within the expected range detailed throughout Section 4.3: The latency reduction is due to the absence of postprocessing in approach 1. However, this comes at the cost of increased off-chip memory usage that ranges from several hundred Megabytes to Gigabytes to store the graph structure. In contrast, linear encoding with approach 2 avoids off-chip graph storage and is compatible with DDR-less hardware platforms where only HBM is available [Xil23, Xil21]. Despite its slightly higher latency, our linear encoding with approach 2 is more flexible as it does not require DDR memory.

Table 10 also presents the estimated overall speedup of Orion commit+prove and the end-to-end latency of approach 1. As expected, the speedup of the Orion prover using approach 1 is higher than in approach 2 and reaches up to 270× (approach 2 reaches 244×). For the end-to-end latency, we estimate a speedup of up to 3.85× for approach 1, which is very similar to approach 2. In summary, approach 1 offers a modest latency advantage due to avoiding postprocessing, while approach 2 provides a more portable and off-chip memory-efficient solution with reduced bandwidth requirements. Both approaches result in speedups of two orders of magnitude for Orion's prover, compared to the reference software.

8 Discussion and Applicability to Related Proof Systems

Orion has inspired further research in the field, resulting in several follow-up publications. This section briefly introduces these works and discusses the applicability of our hardware acceleration concepts.

Brakedown: As highlighted throughout this paper, Brakedown [GLS^{+}23] is conceptually very similar to Orion [XZS22], which allows to extend our implementation techniques to

Brakedown as well. For example, our optimizations to reduce off-chip memory accesses in Spielman encoding or our Merkle Tree unit can also be applied. Using our methodology, the end-to-end prover speedup in Brakedown will be significantly higher – likely around two orders of magnitude – than in Orion. This is because Orion's prover is limited by the outer Virgo protocol. In Orion-based proof systems, proximity and consistency checks are performed on the prover side due to the composition with CP-SNARKs. In contrast, Brakedown requires the verifier to perform these checks, which means the verifier must execute the *same* linear encoding procedure as the prover. Since the graph generation process is deterministic and public, both the prover and verifier can independently generate (or store) identical graphs. These graphs can either use inverted or baseline generation as long as the same graph is used on both sides. The Brakedown verifier will likely benefit from our inverted expander graphs since they reduce the number of memory accesses. This may lead to faster verifiers in software and hardware. We consider concrete verifier-side implementations as an interesting direction for future research.

Orion+: The authors of [CBBZ23] extend the baseline Orion scheme [XZS22] and present Orion+. Orion+ replaces the Merkle Tree commitments in Orion with multilinear polynomial commitment schemes, allowing batch openings. In particular, Orion+ uses KZG [KZG10] as PCS together with HyperPlonk [CBBZ23] as outer proof, thereby shrinking the proof size compared to baseline Orion. This modification, however, introduces a reliance on bilinear pairings, which breaks post-quantum security and requires a trusted setup. Due to this matter, our focus is on the base version of Orion that maintains post-quantum security without any trusted setup. Nevertheless, Orion+ uses the same linear encoding approach as baseline Orion and thus also benefits from our presented inverted expander graph techniques. Especially our linear encoder architecture, presented in Section 4, applies to Orion+ as well. Combining our linear encoder architecture with Orion+-specific PCS functionality on hardware is an interesting direction for future work.

Scorpius: The authors of [dHS24] analyze the soundness of the proof composition technique in Orion and present Scorpius. Scorpius is mostly similar to Orion and uses linear-time Spielman codes and Merkle Tree commitments. The improvements proposed in Scorpius target the outer proof, the opening indices, and the random challenges sent by the verifier. Using these contributions, the proof size and verifier runtime are reduced. However, on the prover side, the Spielman codes, Merkle Trees, and inner products remain similar to Orion. Thus, our hardware accelerator can also extend to the Scorpius PCS.

8.1 Comparison to [SLDS24]

The recent work [SLDS24] explores high-level co-design aspects for a complete proof system, including R1CS, Spartan, Orion, and Virgo protocols. However, it abstracts away from low-level hardware implementation challenges. In contrast, our work provides a detailed, hardware-centric analysis of Orion and Orion-like PCS schemes, introducing memory-aware architecture design and optimizations tailored for FPGA implementation. The following discussion highlights the fundamental differences and orthogonal research directions between [SLDS24] and our work while underscoring our complementary and novel contributions.

Different encoding approach: The authors of [SLDS24] replace Spielman encoding with Reed-Solomon (RS) encoding to reduce memory access overheads. The Reed-Solomon encoding introduces *non-linear-time* complexity for the prover due to the required NTT-based operations. Furthermore, the RS codes in [SLDS24] have a rate $r = 4$, resulting in 2.3× longer codewords than in our Spielman codes with rate $r = 1.72$. This prohibitively increases memory requirements for the large codewords and necessitates additional hashing effort. In contrast, we continue to utilize the linear-time Spielman encoding, albeit in an optimized form, as proposed by Brakedown's and Orion's authors. Using Spielman codes is particularly crucial for preserving the central property of linear-prover-time complexity.

Small field size: The authors of [SLDS24] choose the Goldilocks-64 prime field for their work. This allows an efficient hardware implementation of the arithmetic units, such as finite-field multipliers. However, the Goldilocks-64 field only has fewer than 2^{64} field elements, which compromises the soundness of Orion [GLS+21, Theorem 1]. To mitigate the soundness issues, [SLDS24] repeats several operations up to 4×, which is a performance drawback. In contrast, our baseline version of Orion reaches high soundness by using a sufficiently large extension field $|\mathbb{F}_{p^2}| \approx 2^{122}$, thereby avoiding repeated operations.

Off-chip memory bandwidth: The ASIC architecture in [SLDS24] assumes idealized peak HBM bandwidth, overlooking practical overheads in off-chip data movement. For example, [SLDS24, Section IV.B] reports a per-cycle demand of 128 elements, translating to a total bandwidth requirement of 953 GB/s to 1 TB/s at 1 GHz. To meet this, [SLDS24, Section VI] assumes access to two HBM PHYs delivering 512 GB/s each. However, this exceeds the peak bandwidths of HBM2E technologies from Micron [Mic25], Samsung [Sam25], and SK Hynix [Hyn25] as they offer maximum bandwidths of 410- 460 GB/s per stack under ideal conditions. Under practical conditions, achieving the maximum theoretical bandwidth of HBM is very challenging, as found in [Hub19]. Especially non-uniform or random access patterns degrade the bandwidth by up to 3–4× [Hub19].

In contrast, our work presents a memory-aware design explicitly optimized for realistic HBM behavior. By minimizing random memory accesses through techniques such as on-the-fly expander graph sampling and bandwidth-efficient encoding, our architecture significantly reduces off-chip pressure and improves sustained throughput. In addition, data layout in off-chip memory is critical for performance, particularly as linear encoding and Merkle Tree hashing access data row-wise and column-wise, respectively – necessitating non-trivial data rearrangement or transposition. The paper [SLDS24] does not detail the memory layout of the large coefficient matrix in HBM. In contrast, our work presents a comprehensive treatment of memory access optimizations, including pipelined data rearrangement via scratchpad memory, scalable multi-pass column hashing adaptable to different bandwidth configurations, and a flexible Merkle Tree generation architecture. These contributions, together with detailed implementation insights, are intended to enable reproducibility and support future research in hardware-accelerated proof systems.

Physical placement: The data-centric operations in a proof system require a careful design strategy to distribute the data across the chip. Feeding the computational units is not feasible without proper placement and routing, especially if high clock frequencies must be met. Our work discusses these challenges with a focus on multi-SLR FPGAs and presents a suitable partitioning strategy to utilize all three SLRs. Yet, the paper in [SLDS24] does not give details on addressing the low-level physical properties of their used ASIC architecture. Although low-level physical placement may be considered an "engineering effort", we give insights into the arising challenges and highlight the necessity of a platform-aware architecture design.

Performance comparison: Due to fundamental differences in design choices, such as the use of Reed-Solomon codes and a smaller finite field in [SLDS24], a direct performance comparison with our work is not meaningful. In addition, the idealized off-chip bandwidth assumption for their ASIC architecture contrasts with our practical and resource-aware FPGA implementation. Furthermore, [SLDS24] reports only end-to-end runtimes for benchmarks involving multiple protocols (e.g., Spartan, Orion, Virgo) without isolating the performance of the Orion commitment and proving steps. For these reasons, we do not include a quantitative comparison between the two works.

Complementary contributions: Both our work and [SLDS24] make significant contributions to advancing hardware acceleration for proof systems, yet with distinct focuses. The techniques developed in our work can complement those of [SLDS24], enhancing its practicality and efficiency for hardware implementations by addressing real-world constraints that theoretical models often overlook. Together, these contributions represent steps forward in designing hardware accelerators for proof systems.

8.2 Comparison to Pairing-Based Proof Systems

Implementation aspects of pairing-based proof systems have gained broader attention, for example, in hardware acceleration [ZWZ+21, PdRB+25, ABC+22, HKR25]. Pairing-based proof systems mainly rely on the Number-Theoretic Transformation (NTT) and Multi-Scalar Multiplication (MSM). While NTT operates over large prime fields, MSM uses Elliptic Curve arithmetic. In both cases, multiple hundred-bit wide coefficients are involved. The fundamental difference in field arithmetic and the distinct high-level NTT and MSM algorithms make a comparison between hardware accelerators for pairing-based proof systems and our design hard and unfair. In addition, MSM accelerators leverage a more uniform off-chip memory access pattern than Spielman codes involved in our design, which makes these works less memory-bound than ours.

To give an impression of the different implementation aspects of pairing-based proof systems, we consider PipeZK [ZWZ+21]. PipeZK is an ASIC hardware accelerator for MSM and NTT, required in pairing-based proof systems. The 28nm ASIC dedicates 15mm^2 and 35mm^2 chip area to NTT and MSM, respectively, for the BN128 curve. Compared to software implementation with size 2^{16} to 2^{20}, PipeZK reaches a speedup of 8× to 12× for MSM and 29× to 107× for NTT. In comparison, our FPGA design reaches speedups between 111× and 128× for Orion's commit+prove operations with respect to the Orion software. A pure MSM accelerator is presented in OptiMSM [PdRB+25]. OptiMSM targets an Alveo U55C FPGA, which is comparable to our Alveo U280 FPGA, and reports a resource utilization of 721k LUTs, 885k REGs, and 2,147 DSPs. In comparison, our implementation uses 775k LUTs, 824k REGs, and 1,162 DSPs, as reported in Table 5. Our design supports operations from both the commitment and proving phases, whereas OptiMSM accelerates only the MSM from the commitment phase. To reach end-to-end functionality, OptiMSM would require additional components such as an NTT unit, thereby exceeding the FPGA's resources. In terms of performance, OptiMSM achieves a runtime of 914ms and 231ms for a size-2^{24} MSM on the BLS12-381 curve using 1 and 4 FPGAs, respectively. In contrast, our hash-based architecture completes the prover-side computations for $N = 2^{24}$ in 90ms, with 79ms for commitment and 11ms for proving.

We note that this section should not directly compare our work with PipeZK [ZWZ+21] or OptiMSM [PdRB+25]. Instead, it should give an impression of the difficult nature of comparisons of implementations based on mathematically distinct proof systems.

9 Conclusion

This work introduced several algorithmic and architectural optimization techniques for accelerating linear-prove-time PCS on hardware. We specifically targeted the challenging random access patterns to off-chip memory during data-centric Spielman encoding and proposed *inverted expander graphs*. Inverted expander graphs reduce the HBM memory accesses by about 50% and reduce the pressure on critical off-chip bandwidth. Since off-chip bandwidth is the limiting factor in linear encoding, our improvements lead to a speedup of up to 264× in linear encoding.

Next to linear encoding, Orion-like PCS involve hashing and Merkle Tree construction during the commitment and proving phases. We present hardware-specific optimization strategies to ensure high hashing throughput and a memory layout-aware data reordering. Our pipelined reordering consumes the high data rate from HBM and serves a variable number of unrolled SHA3 permutation units. This data distribution is challenging for high-end FPGAs with multiple SLRs. Thus, we partition our overall design and compensate for SLR-crossing bottlenecks in our Scratchpad memory. Finally, we accelerate the inner product computation, which is the prevailing operation in Orion's proving mechanism. Inner product computation reuses our linear encoder datapath and hence has a small additional area footprint.

We combined all concepts into one hardware architecture and mapped it to an Alveo U280 FPGA. This mapping involved floor planning and manual placement to enable routing of the complex and heterogeneous design. Overall, our results show a speedup of two orders of magnitude for the commitment and proving in the Orion PCS. Considering the end-to-end latency, including the outer Virgo protocol in software, we reach up to 3.85× performance improvement. The presented methodology, especially the concept of inverted expander graphs and our hashing architecture, can be applied to other PCS such as Brakedown, Orion+, or Scorpius as well. Therefore, our concepts and our FPGA accelerator allow a broad range of applications in the field of linear-prover-time polynomial commitment schemes.

9.1 Future Directions

There are several avenues for future research, building on the findings of this work. For example, adapting our optimizations for ASIC platforms could further enhance performance and energy efficiency, particularly for large-scale ZKP deployments. Due to the higher clock frequency in ASIC, the off-chip memory bandwidth becomes even more crucial to fully utilize the computational units. Extending our memory-aware inverted expander graph technique to an ASIC setting is an interesting field of future efforts. In addition to this, software implementations may also benefit from our inverted expander graphs since the reduction in memory accesses applies as well. This work focuses on hardware acceleration of polynomial commitment and proof generation, which are the most computationally intensive components of the system. Extending the scope to the full CP-SNARK construction, including SNARK composition and recursion, would introduce significant additional complexity beyond the focus of a single paper. Specifically for the Orion scheme, a dedicated research effort is required to accelerate the outer Virgo protocol functioning as CP-SNARK. The Virgo protocol covers a 26% share within the base execution of Orion for $N = 2^{28}$. Using our hardware accelerator, this share increases to 98.8% due to the significance of our optimization in the commitment and proof generation, making the Virgo protocol the limiting factor in Orion.

Finally, our developed methodology can be applied to various PCS such as Brakedown, Scorpius, or Orion+, which involve Spielman encodings or Merkle Trees. Implementing dedicated hardware accelerators that benefit from the concepts introduced in this paper would lead to broad support of linear-prover-time polynomial commitment schemes.

Acknowledgement

Florian Hirner and Sujoy Sinha Roy were partially supported by the State Government of Styria, Austria (Zukunftsfonds Steiermark). This work was driven by the authors' independent interest in hardware acceleration of zero-knowledge proofs and was not supported by any ZKP-specific funded project. We thank Anisha Mukherjee and Aikata Aikata for their feedback on our expander-graph adaptations and their security implications. We are grateful to Paul Gollob for carefully reviewing the paper for consistency and to Mathias Oberhuber for his helpful feedback. We would also like to extend our gratitude to the anonymous reviewers of TCHES 2025 for their constructive suggestions and valuable comments that helped improve the quality of this paper.

References

[ABC+22] Kaveh Aasaraai, Don Beaver, Emanuele Cesena, Rahul Maganti, Nicolas Stalder, and Javier Varela. FPGA acceleration of multi-scalar multiplication: CycloneMSM. Cryptology ePrint Archive, Paper 2022/1396, 2022.

[APKP24] Kasra Abbaszadeh, Christodoulos Pappas, Jonathan Katz, and Dimitrios Papadopoulos. Zero-knowledge proofs of training for deep neural networks. Cryptology ePrint Archive, Paper 2024/162, 2024.

[APPK24] Kasra Abbaszadeh, Christodoulos Pappas, Dimitrios Papadopoulos, and Jonathan Katz. Zero-knowledge proofs of training for deep neural networks. *Accepted in ACM CCS 2024*, page 162, 2024.

[BBB+18] Benedikt Bünz, Jonathan Bootle, Dan Boneh, Andrew Poelstra, Pieter Wuille, and Gregory Maxwell. Bulletproofs: Short Proofs for Confidential Transactions and More. In *2018 IEEE Symposium on Security and Privacy, SP 2018, Proceedings, 21-23 May 2018, San Francisco, California, USA*, pages 315–334. IEEE Computer Society, 2018.

[BCG20] Jonathan Bootle, Alessandro Chiesa, and Jens Groth. Linear-time arguments with sublinear verification from tensor codes. In *Theory of Cryptography: 18th International Conference, TCC 2020, Durham, NC, USA, November 16–19, 2020, Proceedings, Part II 18*, pages 19–46. Springer, 2020.

[Big02] Norman Biggs. *Discrete Mathematics*. OUP Oxford, 12 2002.

[Bou20] Charles Bouillaguet. Trivium. https://github.com/cbouilla/trivium, December 2020.

[BSCG+14] Eli Ben Sasson, Alessandro Chiesa, Christina Garman, Matthew Green, Ian Miers, Eran Tromer, and Madars Virza. Zerocash: Decentralized Anonymous Payments from Bitcoin. In *2014 IEEE Symposium on Security and Privacy*, pages 459–474, 2014.

[Can06] Christophe De Cannière. Trivium: A stream cipher construction inspired by block cipher design principles. *Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics)*, 4176 LNCS:171–186, 2006.

[CBBZ23] Binyi Chen, Benedikt Bünz, Dan Boneh, and Zhenfei Zhang. Hyperplonk: Plonk with linear-time prover and high-degree custom gates. In Carmit Hazay and Martijn Stam, editors, *Advances in Cryptology – EUROCRYPT 2023*, pages 499–530, Cham, 2023. Springer Nature Switzerland.

[CHM+20] Alessandro Chiesa, Yuncong Hu, Mary Maller, Pratyush Mishra, Psi Vesely, and Nicholas P. Ward. Marlin: Preprocessing zksnarks with universal and updatable SRS. In Anne Canteaut and Yuval Ishai, editors, *Advances in Cryptology - EUROCRYPT 2020 - 39th Annual International Conference on* *the Theory and Applications of Cryptographic Techniques, Zagreb, Croatia, May 10-14, 2020, Proceedings, Part I*, volume 12105 of *Lecture Notes in Computer Science*, pages 738–768. Springer, 2020.

[dHS24] Thomas den Hollander and Daniel Slamanig. A crack in the firmament: Restoring soundness of the orion proof system and more. Cryptology ePrint Archive, Paper 2024/1164, 2024.

[DP23] Benjamin E. Diamond and Jim Posen. Succinct arguments over towers of binary fields. Cryptology ePrint Archive, Paper 2023/1784, 2023.

[DTM$^+$24] Zhixiong Di, Runzhe Tao, Jing Mai, Lin Chen, and Yibo Lin. Leaps: Topological-layout-adaptable multi-die fpga placement for super long line minimization. *IEEE Transactions on Circuits and Systems I: Regular Papers*, 71(3):1259–1272, 2024.

[Fam88] A.T. Fam. Efficient complex matrix multiplication. *IEEE Transactions on Computers*, 37(7):877–879, 1988.

[Fou] Zcash Foundation. Zcash digital currency. https://z.cash, Accessed on October 2024.

[FPG] FPGAEmu. Axi protocol overview. https://fpgaemu.readthedocs.io/en/latest/axi.html.

[FS86] Amos Fiat and Adi Shamir. How to Prove Yourself: Practical Solutions to Identification and Signature Problems. In Andrew M. Odlyzko, editor, *Advances in Cryptology - CRYPTO 1986*, volume 263 of *LNCS*, pages 186–194. Springer, 1986.

[GKR15] Shafi Goldwasser, Yael Tauman Kalai, and Guy N. Rothblum. Delegating computation: Interactive proofs for muggles. *J. ACM*, 62(4), September 2015.

[GLS$^+$21] Alexander Golovnev, Jonathan Lee, Srinath Setty, Justin Thaler, and Riad S. Wahby. Brakedown: Linear-time and field-agnostic snarks for r1cs. *Cryptology ePrint Archive*, 2021.

[GLS$^+$23] Alexander Golovnev, Jonathan Lee, Srinath T. V. Setty, Justin Thaler, and Riad S. Wahby. Brakedown: Linear-time and field-agnostic snarks for R1CS. In Helena Handschuh and Anna Lysyanskaya, editors, *Advances in Cryptology - CRYPTO 2023 - 43rd Annual International Cryptology Conference, CRYPTO 2023, Santa Barbara, CA, USA, August 20-24, 2023, Proceedings, Part II*, volume 14082 of *Lecture Notes in Computer Science*, pages 193–226. Springer, 2023.

[GMR85] Shafi Goldwasser, Silvio Micali, and Charles Rackoff. The knowledge complexity of interactive proof-systems (extended abstract). In Robert Sedgewick, editor, *Proceedings of the 17th Annual ACM Symposium on Theory of Computing, May 6-8, 1985, Providence, Rhode Island, USA*, pages 291–304. ACM, 1985.

[Gol01] Oded Goldreich. Foundations of cryptography. *Foundations of Cryptography*, 8 2001.

[Gro16] Jens Groth. On the size of pairing-based non-interactive arguments. In Marc Fischlin and Jean-Sébastien Coron, editors, *Advances in Cryptology – EUROCRYPT 2016*, pages 305–326, Berlin, Heidelberg, 2016. Springer Berlin Heidelberg.

[Gus88] John L. Gustafson. Reevaluating amdahl's law. *Commun. ACM*, 31(5):532–533, May 1988.

[GY18] Hisham S. Galal and Amr M. Youssef. Verifiable sealed-bid auction on the ethereum blockchain. In *Financial Cryptography and Data Security - FC 2018*, volume 10958 of *LNCS*, pages 265–278. Springer, 2018.

[Hei03] James L. Hein. *Discrete Mathematics.* Jones and Bartlett Learning, 2003.

[HKR25] Florian Hirner, Florian Krieger, and Sujoy Sinha Roy. Chiplet-based techniques for scalable and memory-aware multi-scalar multiplication. Cryptology ePrint Archive, Paper 2025/252, 2025.

[HP10] W. Cary Huffman and Vera Pless. *Fundamentals of Error-Correcting Codes.* Cambridge University Press, 2 2010.

[Hub19] Xilinx Developer Hub. Basic tutorial for maximizing memory bandwidth with vitis and xilinx ultrascale+ hbm devices, 11 2019.

[Hyn25] SK Hynix. Hbm2e, fastest dram with enhanced heat dissipation, accessed: April 2025.

[KZG10] Aniket Kate, Gregory M. Zaverucha, and Ian Goldberg. Constant-size commitments to polynomials and their applications. *Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics)*, 6477 LNCS:177–194, 2010.

[Lee21] Jonathan Lee. Dory: Efficient, Transparent Arguments for Generalised Inner Products and Polynomial Commitments. In Kobbi Nissim and Brent Waters, editors, *Theory of Cryptography*, pages 1–34, Cham, 2021. Springer International Publishing.

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

[LWX$^+$23] Xiling Li, Chenkai Weng, Yongxin Xu, Xiao Wang, and Jennie Rogers. ZKSQL: verifiable and efficient query evaluation with zero-knowledge proofs. *Proc. VLDB Endow.*, 16(8):1804–1816, 2023.

[LXZ21] Tianyi Liu, Xiang Xie, and Yupeng Zhang. zkCNN: Zero Knowledge Proofs for Convolutional Neural Network Predictions and Accuracy. In Yongdae Kim, Jong Kim, Giovanni Vigna, and Elaine Shi, editors, *CCS '21: 2021 ACM SIGSAC Conference on Computer and Communications Security, Virtual Event, Republic of Korea, November 15 - 19, 2021*, pages 2968–2985. ACM, 2021.

[Mic25] Micron. Hbm2e: Our fastest memory for the modern data center, micron technical brief, accessed: April 2025.

[PdRB$^+$25] Xander Pottier, Thomas de Ruijter, Jonas Bertels, Wouter Legiest, Michiel Van Beirendonck, and Ingrid Verbauwhede. Optimsm: Fpga hardware accelerator for zero-knowledge msm. *IACR Transactions on Cryptographic Hardware and Embedded Systems*, 2025(2):489–510, Mar. 2025.

[Pro] The Monero Project. Monero - secure, private, untraceable. https://www.getmonero.org/, Accessed on October 2024.

[RL09] William Ryan and Shu Lin. *Channel Codes: Classical and Modern*. Cambridge University Press, 9 2009.

[RZR24] Noga Ron-Zewi and Ron Rothblum. Local proofs approaching the witness length. *J. ACM*, 71(3), June 2024.

[Sam25] Samsung. HBM: Hbm2e flashbolt, accessed: April 2025.

[SLDS24] Nikola Samardzic, Simon Langowski, Srinivas Devadas, and Daniel Sanchez. Accelerating zero-knowledge proofs through hardware-algorithm co-design. In *2024 57th IEEE/ACM International Symposium on Microarchitecture (MICRO)*, pages 366–379, Los Alamitos, CA, USA, Nov 2024. IEEE Computer Society.

[Spi96] Daniel A. Spielman. Linear-time encodable and decodable error-correcting codes. *IEEE Transactions on Information Theory*, 42:1723–1731, 1996.

[SS94] M. Sipser and D.A. Spielman. Expander codes. In *Proceedings 35th Annual Symposium on Foundations of Computer Science*, pages 566–576. Institute of Electrical and Electronics Engineers (IEEE), 12 1994.

[Su] Sunblaze-ucb. sunblaze-ucb/orion. https://github.com/sunblaze-ucb/Orion. Accessed: 2024-09-30.

[WTS+18] R. S. Wahby, I. Tzialla, A. Shelat, J. Thaler, and M. Walfish. Doubly-Efficient zkSNARKs Without Trusted Setup. In *2018 IEEE Symposium on Security and Privacy (SP)*, pages 926–943, Los Alamitos, CA, USA, may 2018. IEEE Computer Society.

[WYX+21] Chenkai Weng, Kang Yang, Xiang Xie, Jonathan Katz, and Xiao Wang. Mystique: Efficient Conversions for Zero-Knowledge Proofs with Applications to Machine Learning. In Michael Bailey and Rachel Greenstadt, editors, *30th USENIX Security Symposium, USENIX Security 2021, August 11-13, 2021*, pages 501–518. USENIX Association, 2021.

[Xil12] Xilinx. Xilinx stacked silicon interconnect technology delivers breakthrough fpga capacity, bandwidth, and power efficiency, 2012. Revision 1.2. https://docs.amd.com/v/u/en-US/wp380_Stacked_Silicon_Interconnect_Technology.

[Xil19] Xilinx. Alveo u280 data center accelerator card user guide, 11 2019. https://www.mouser.com/pdfDocs/u280userguide.pdf.

[Xil21] Xilinx. Varium c1100 compute adaptor data sheet, 2021. https://docs.amd.com/v/u/en-US/ds1003-varium-c1100.

[Xil23] Xilinx. Alveo u55c data center accelerator cards data sheet, 2023. https://docs.amd.com/r/en-US/ds978-u55c.

[Xil24] Xilinx. Hbm system considerations, 2024. https://docs.amd.com/r/en-US/pg313-network-on-chip/HBM2E-Access-Latencies, accessed on October 2024.

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

[ZWZ+21] Ye Zhang, Shuo Wang, Xian Zhang, Jiangbin Dong, Xingzhong Mao, Fan Long, Cong Wang, Dong Zhou, Mingyu Gao, and Guangyu Sun. Pipezk: Accelerating zero-knowledge proof with a pipelined architecture. In *2021 ACM/IEEE 48th Annual International Symposium on Computer Architecture (ISCA)*, pages 416–428, 2021.

[ZXZS20] Jiaheng Zhang, Tiancheng Xie, Yupeng Zhang, and Dawn Song. Transparent polynomial delegation and its applications to zero knowledge proof. In *2020 IEEE Symposium on Security and Privacy (SP)*, pages 859–876. IEEE, 5 2020.

A Algorithms

Baseline expander graph generation: The baseline expander graph $G = (L, R, E)$ used in Orion [XZS22] or Brakedown [GLS+23] is generated and stored in memory. The generation algorithm is shown in Algorithm 2. It can be seen that the graph is stored as a set E of tuples of the form $(l_i, r_t, \omega_{i,t})$. Each tuple describes one edge in the graph staring at the left node $l_i \in L$ and connecting the right node $r_t \in R$ with a weight $\omega_{i,t}$. The bit size of a single tuple is $2\log(n) + \log(|\mathbb{F}_{p^2}|)$ bits with n denoting the codeword length. Unlike in our inverted expander graphs, the degree of left nodes c is constant in Algorithm 2.

Algorithm 2 Baseline Expander Graph Generation Algorithm from [XZS22].

```
function GraphGen($L, R, c$)        ▷ $L$, $R$ set of left & right nodes, $c$ is left nodes degree
    $E \leftarrow \{\}$, $k \leftarrow |L|$, $q \leftarrow \alpha k$        ▷ $q = |R| = \alpha|L|$
    for $i$ from 0 to $k-1$ do
        for $j$ from 0 to $c-1$ do
            $t \leftarrow$ rand() mod $q$        ▷ A random vertex index in right subset
            $\omega_{i,t} \leftarrow$ field :: random()        ▷ A random field element
            Add edge $(l_i, r_t, \omega_{i,t})$ from left node $l_i$ to right node $r_t$ with weight $\omega_{i,t}$ to $E$
    return $E$
```

Searching non-expanding set: Algorithm 3 from [XZS22] tests randomly generated expander graphs for sufficient connectivity. If Algorithm 3 outputs `NotFound`, then with all but negligible probability, the randomly generated graph is a good expander.

Algorithm 3 Searching Non-Expanding Set [XZS22].

```
Let $G = (L \cup R, E)$ with the left and right vertex sets of a random bipartite graph.
If $\exists v \in R$ with degree $d \geq \frac{c}{\alpha} + 10\ln k$, abort.        ▷ $\alpha = \frac{|R|}{|L|}$
for each $v \in L$ do
    find set $D \subseteq L$ such that:
      - $\forall u \in D$ the minimum distance between $u$ and $v$ is $\leq 2\log\log k$
      - $\forall u \in L \setminus D$ the minimum distance between $u$ and $v$ is $> 2\log\log k$
    for all $S \subseteq D$ and $|S| \leq \log\log k$ do
        if Equation 1 does not hold for $S$ then
            return Found
return NotFound
```