High-Performance SIMD Software for Spielman Codes in Zero-Knowledge Proofs

Florian Krieger, Christian Dobrouschek, Florian Hirner and Sujoy Sinha Roy

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

Abstract. We present the first high-performance SIMD software implementation of Spielman codes for their use in polynomial commitment schemes and zero-knowledge proofs. Spielman codes, as used in the Brakedown framework, are attractive alternatives to Reed-Solomon codes and benefit from linear-time complexity and field agnosticism. However, the practical deployment of Spielman codes has been hindered by a lack of research on efficient implementations. The involved costly finite-field arithmetic and random memory accesses operate on large volumes of data, typically exceeding gigabytes; these pose significant challenges for performance gains. To address these challenges, we propose several computational and memory-related optimizations that together reach an order-of-magnitude performance improvement in software. On the computation side, we propose SIMD optimizations using the AVX-512-IFMA instruction set and introduce a lazy reduction method to minimize the modular arithmetic cost. On the memory side, we implement a cache-friendly memory layout and a slicing technique, which exploit the CPU memory hierarchy. Finally, we present our multithreading approach to improve throughput without saturating memory bandwidth. Compared to prior Spielman software, our optimizations achieve speedups of up to 26.7× and 20.6× for single- and multi-threaded execution, respectively. In addition, instantiating our software with 64 threads on a high-end CPU even outperforms a recent FPGA accelerator by up to 4.3× for small and mid-sized polynomials. Our improvements make Spielman codes competitive with well-optimized Reed-Solomon codes on software platforms.

Keywords: Zero-Knowledge Proof · Polynomial Commitment Scheme · Spielman Encoding · SIMD

1 Introduction

Zero-knowledge proof (ZKP) systems allow one party to prove the validity of a statement to another party without revealing any additional information. Recent advances in ZKP constructions, along with the growing demand for privacy, integrity, and verifiability, have significantly expanded their adoption. Concrete applications of ZKPs have been demonstrated in machine learning [APKP24, LXZ21, WYX$^+$21], where computations on private models or data can be publicly verified. In banking and decentralized finance, ZKPs ensure user eligibility or transaction validity without disclosing sensitive information, as exemplified by privacy-preserving cryptocurrencies such as Monero and Zcash [Pro, Fou]. Similarly, ZKPs enable verifiable yet anonymous voting [PR21], where voters can prove that their vote is valid without revealing their identity or choice. The wide range of ZKP applications continues to broaden, driving the need for efficient and scalable implementations.

Many modern, transparent ZKP systems are structured as interactive oracle proofs (IOP) combined with a polynomial commitment scheme (PCS) [Tha22]. The IOP is an information-theoretic protocol, while the PCS relies on cryptographic assumptions. Commonly used assumptions are pairing-based assumptions for KZG-like commitments [KZG10], discrete-log assumptions for group-based schemes, or the collision resistance of hash functions for Merkle/FRI-based polynomial commitments. Among PCS frameworks, Brakedown [GLS+23] has attracted significant attention for combining transparency, flexibility, and plausibly post-quantum security via hash-based commitments. In Brakedown, the prover performs linear error-correcting encodings of data to ensure soundness, and then commits to the encoded data by computing a Merkle tree. Compared to the Merkle tree construction covered in prior work [Off25, ZRW25], the linear encoding is less explored, although it dominates the runtime.

A distinct property of the Brakedown framework is the flexibility in encoding algorithms, since any linear code with sufficient distance can be used. The most popular code choices in the Brakedown framework are Reed-Solomon (RS) codes[1] [GLS+23, CCC+25, DP23] and Spielman codes [dHS24, YZRM24, XZS22]. RS codes involve the Number-Theoretic Transform (NTT), which is well researched and highly optimized thanks to the progress in fully homomorphic encryption [RRG+25, BKS+21, CYY+21] and post-quantum cryptography [XHL+24, OZZ+24]. Due to the NTT, RS codes have a $\mathcal{O}(n \log n)$ time complexity and either require an NTT-friendly field or special methods for polynomial evaluation [LCH14]. Next to the RS code, Brakedown is often instantiated using the Spielman code [Spi96], which evaluates multiple bipartite expander graphs to compute a codeword. Spielman codes offer interesting benefits such as field agnosticism and a superior asymptotic runtime of $\mathcal{O}(n)$, setting them apart from RS codes. However, despite their lower time complexity, Spielman codes have attracted substantially less attention than the NTT-based RS codes. This research gap leads to an insufficient understanding of the implementation aspects of Spielman codes, although crucial for evaluating their concrete performance within PCS.

Challenges of Spielman Codes. The linear encoding in PCSs is the major performance bottleneck, causing up to 80% of the commitment latency [HKPSR25]. At the same time, efficiently implementing Spielman codes for PCSs is challenging. Specifically, the huge polynomials consuming gigabytes of data degrade performance and increase encoding latency, unless carefully optimized. The key challenges are threefold:

Challenge 1: Brakedown operates over large finite fields with 128-bit-wide coefficients, or more to guarantee soundness. Hence, the Spielman encoding with repeated expander graph evaluations demands optimized finite field arithmetic to perform the numerous large-field operations efficiently.

Challenge 2: The large-degree polynomials in Brakedown easily exceed gigabytes, making the Spielman encoding highly data-centric. In addition, the expander graph evaluations in Spielman codes access the large data set in a non-uniform and randomized pattern. Without spatial locality of memory accesses, caching in CPUs is ineffective. Therefore, performance gains are limited without proper measures to resolve the memory bottleneck.

Challenge 3: Reaching high performance of Spielman encoding in software targeting modern CPUs is a hard problem. It necessitates a careful design, which accelerates the computations on one hand, while relaxing the memory bandwidth requirements on the other hand. Yet, these two design aspects counteract each other, and an efficient balance must be found. In addition, CPU-specific fine-tuning is essential to optimally exploit caching and pipelining.

Prior Work. Inspired by these challenges, prior work has proposed implementations on hardware [HKPSR25] and software [XZS22, Wah21, CHL+24] to accelerate the heavyweight

[1]The RS-variant of Brakedown is also called Shockwave [GLS+23].

Spielman encoding on the prover side. However, hardware accelerators are expensive and have limited flexibility. In addition, their limited off-chip memory bandwidth restricts the potential performance gains [HKPSR25], making hardware acceleration of prover-side encoding less effective. Moreover, Brakedown also requires the *verifier* to perform Spielman encoding during verification, albeit more lightweight than the prover's linear encoding. Unlike the prover, the verifier often operates in a computationally less powerful setting and thus cannot leverage high-end FPGA, ASIC, or GPU accelerators for linear encoding.

On the other hand, software implementations are more flexible and apply to the constrained verifier setting. For example, even mobile CPUs [Arm20] but also high-end server CPUs [Int25] offer instruction-level SIMD support, allowing cost-effective and performant PCS operations. This SIMD support in CPUs has already resulted in SIMD implementations of other heavyweight cryptographic algorithms like Multi-Scalar Multiplication [JPL+25], Fully Homomorphic Encryption [BKS+21], or digital signature schemes [ZRW25] and demonstrates the demand for optimized SIMD software implementations of Spielman codes. Nevertheless, existing software implementations for Spielman codes [Wah21, CHL+24] mainly focus on providing a baseline implementation and do not leverage SIMD functionality. In addition, these works do not explore optimizations regarding the memory bottleneck and thus leave significant room for performance improvement.

Our Contributions. To address this research gap, we investigate the implementation aspects of Spielman codes in software with a focus on performance-enhancing methods on modern CPUs. Specifically, we present the first SIMD implementation of Spielman codes using the AVX-512 paradigm. Yet, SIMD computation alone is insufficient without proper measures to avoid memory access bottlenecks in the data-centric PCS. We therefore explore measures to reduce memory pressure for single- and multi-threaded deployment. Our contributions can be summarized as follows.

- We present the first SIMD implementation of Spielman codes in the Brakedown PCS. In our software implementation, we use AVX-512 and the IFMA instruction set extension to improve the computational efficiency of the critical expander graph evaluation. Our implementation elaborates on the left-to-right and right-to-left expander graph evaluation [HKPSR25] and reveals the right-to-left approach to be more suitable (Section 3.1).
- We propose several performance-enhancing techniques for Spielman codes, including a dedicated lazy reduction (Section 3.2) and a cache-friendly memory layout (Section 3.3). Moreover, we give a detailed discussion on memory bandwidth-related issues of Spielman codes. This analysis shows a high correlation between cache efficiency and encoding performance. We therefore propose our *slicing* technique to enhance the last-level cache (LLC) efficiency (Section 3.4).
- Building upon these optimizations, we introduce a method for multithreaded Spielman encoding in Section 3.5. Yet, reaching speedups via multithreading in Spielman codes is not trivial, as the worker threads must be supplied with sufficient data. To reduce the data bottleneck, we introduce *expander-level parallelism* to leverage multithreading while keeping memory bandwidth requirements low. We further enhance throughput via *encoding-level parallelism* on high-end CPUs.
- We map our implementation to desktop and high-end CPUs and individually benchmark each of our optimizations (Section 4). Compared to other software implementations, we reach up to 26.7× and 20.6× lower latency for single and multi-threaded execution, respectively. Our highly parallel software using 64 threads reaches a notable throughput increase over the FPGA design in [HKPSR25]. Finally, we compare our Spielman performance to AVX-based Reed-Solomon encoding. This

comparison confirms a competitive runtime of our Spielman codes with at most $1.44\times$ overhead over optimized RS codes (Section 4.2.3).

- We publish our source code in [KDHR26].

2 Preliminaries

This section introduces the notation and the background on expander graphs and Spielman codes. It also gives an overview of Brakedown and related PCS relying on Spielman codes.

2.1 Notation

We denote the finite field with q elements as $\mathbb{F}_q$. We write $a \xleftarrow{\$} S$ if a is chosen from set S uniformly at random. We denote vectors $\mathbf{u} \in \mathbb{F}_q^k$ and matrices $\mathbf{U} \in \mathbb{F}_q^{k\times n}$ with bold lowercase and uppercase letters, respectively. The i-th vector element of $\mathbf{u}$ is u_i and the matrix element at the i-th row and j-th column is indexed as $\mathbf{U}[i,j] = u_{i,j}$. We index the i-th row vector of a matrix as $\mathbf{U}[i:]$ and the j-th column vector as $\mathbf{U}[:j]$. The inner product between matrices or vectors is denoted as $\langle\cdot,\cdot\rangle$, and $\otimes$ is the tensor product operator.

The set of univariate polynomials $\phi(X) : \mathbb{F}_q \to \mathbb{F}_q$ with degree less than N and coefficients from $\mathbb{F}_q$ is denoted as $\mathbb{F}_q^{<N}[X]$. The set of multilinear polynomials $\phi(X_0,\ldots,X_{l-1}) : \mathbb{F}_q^l \to \mathbb{F}_q$ with at most $N = 2^l$ coefficients from $\mathbb{F}_q$ is denoted as $\mathbb{F}_q^{<N}[X_0,\ldots,X_{l-1}]$. Let $\mathcal{B}^l = \{0,1\}^l$ be the l-dimensional boolean hypercube. The evaluations of a multilinear polynomial $\phi(X_0,\ldots,X_{l-1}) \in \mathbb{F}_q^{<N}[X_0,\ldots,X_{l-1}]$ over $\mathcal{B}^l$ is called the Lagrange basis of ϕ. We denote the set $\{0,\ldots,n-1\}$ as $[n]$ and we use the basis-2 logarithm.

2.2 Expander Graphs

A graph is a tuple (V,E) with vertex set V and edge set E wherein each edge connects two vertices. The degree of a vertex is the number of edges connected to the vertex. We call a c-left-regular bipartite graph (L,R,E) with left vertex set L, right vertex set R and edges E an *expander graph*, if $|R| = \alpha|L|$ for some $0 < \alpha < 1$. The degree of each vertex in L is c and $|L| = k$. In addition, each edge has an assigned weight $w \in \mathbb{F}_q\backslash\{0\}$. As presented in [GLS+23], expander graphs G used in Spielman codes can be randomly generated using Algorithm 1. Therein, the graph structure G is a set of tuples

Algorithm 1 Expander Graph Initialization [GLS+23]

```
1:  function ExpGrInit($k$, $\alpha$, $c$): Returns a randomly sampled expander graph $G$ with
    $k$ left nodes and $\alpha k$ right nodes where $G$ and $\tilde{G}$ are in left-node and right-node major
    order, respectively. Each left node has degree $c$.
2:      $G \leftarrow \{\}$, $\tilde{G} \leftarrow \{\}$
3:      for $l$ from 0 to $k-1$ do
4:          for $i$ from 0 to $c-1$ do
5:              $r \xleftarrow{\$} [\alpha k]$
6:              $w_{l,r} \xleftarrow{\$} \mathbb{F}_q\backslash\{0\}$
7:              PushBack($G, (l, r, w_{l,r})$)            ▷ $G$ is stored in left-node major order
8:      for $i$ from 0 to $\alpha k - 1$ do
9:          for each $(l, r, w_{l,r}) \in G$ do        ▷ Transpose $G$ into right-node major order
10:             if $i == r$ then
11:                 PushBack($\tilde{G}, (r, l, w_{l,r})$)     ▷ $\tilde{G}$ is stored in right-node major order
12:     return $G$, $\tilde{G}$
```

Algorithm 2 Left-to-Right Expander Graph Evaluation (L2R)

```
1: function ExpGr_{L2R}($\mathbf{m}$, $G$, $\alpha$): Maps vector $\mathbf{m} \in \mathbb{F}_q^k$ to vector $\mathbf{y} \in \mathbb{F}_q^{\alpha k}$ with graph $G$
2:     Initialize $\mathbf{y} \in \mathbb{F}_q^{\alpha k}$ as the $\alpha k$-dimensional zero vector.
3:     for each $(l, r, w_{l,r}) \in G$ do
4:         $y_r \leftarrow y_r + w_{l,r} \cdot m_l$        ▷ Linear accesses to $m_l$, random accesses to $y_r$.
5:     return $\mathbf{y}$
```

Algorithm 3 Right-to-Left Expander Graph Evaluation (R2L)

```
1: function ExpGr_{R2L}($\mathbf{m}$, $\tilde{G}$, $\alpha$): Maps vector $\mathbf{m} \in \mathbb{F}_q^k$ to vector $\mathbf{y} \in \mathbb{F}_q^{\alpha k}$ with transposed graph $\tilde{G}$
2:     for $i$ from 0 to $\alpha k - 1$ do
3:         $y_i \leftarrow 0$
4:         for each $(r, l, w_{l,r}) \in \tilde{G}$ where $r == i$ do
5:             $y_i \leftarrow y_i + w_{l,r} \cdot m_l$        ▷ Linear accesses to $y_r$, random accesses to $m_l$.
6:     return $\mathbf{y} = (y_0, \ldots, y_{\alpha k-1})$
```

$(l, r, w_{l,r}) \in [k] \times [\alpha k] \times \mathbb{F}_q \backslash \{0\}$. Each tuple represents an edge connecting the l-th left vertex to the r-th right vertex with weight $w_{l,r}$.

We define the operation *expander graph evaluation* as the multiplication of the graph's adjacency matrix by an input vector $\mathbf{m}$, resulting in vector $\mathbf{y}$. Due to the sparse adjacency matrix of expander graphs, Algorithms 2 and 3 are more efficient than straightforward matrix-vector multiplication. Therein, Algorithm 2 evaluates the graph from left nodes to right nodes (L2R), meaning that it iterates linearly over the left nodes m_l and distributes their values to right nodes y_r. This requires numerous random memory accesses to $\mathbf{y}$. In contrast, Algorithm 3 proposed by [HKPSR25] uses a right-to-left approach (R2L) by iterating over the right nodes y_r and collecting the assigned values from random left nodes m_l. Consequently, the left nodes are randomly accessed, and the overall memory accesses are reduced. Still, in both cases, the random accesses are a critical burden to performance due to ineffective caching and degraded memory bandwidth. We remark that L2R and R2L evaluations yield the same encoding result. Yet, R2L requires transposing of the graph G in left-node major order to get $\tilde{G}$ in right-node major order. Transposing is done during an offline precomputation phase shown in Algorithm 1.

2.3 Linear Error Correcting Codes

A $[n, k, d]$ linear error correcting code $\mathbf{c} = \textsc{Enc}_{n,k,d}(\mathbf{m})$ maps a message $\mathbf{m} \in \mathbb{F}_q^k$ to a longer codeword $\mathbf{c} \in \mathbb{F}_q^n$ with $n > k$ (we also write $\textsc{Enc}()$ if n, k, and d are implicit). The code distance d is defined as $d = \min\{\|\textsc{Enc}(\mathbf{m}) - \textsc{Enc}(\mathbf{m}')\|_0 : \forall \mathbf{m}, \mathbf{m}' \in \mathbb{F}_q^k, \mathbf{m} \neq \mathbf{m}'\}$. In essence, d is the minimal number of elements in which two distinct codewords differ. Other central parameters of an $[n, k, d]$ code are the relative distance $\delta = d/n$ and the rate r^{-1} with $r = n/k$. A code is said to be linear if any linear combination of two codewords is again a valid codeword. This fact is particularly important for Brakedown and related PCSs. Therefore, we focus on the linear Spielman codes in the next section.

2.3.1 The Spielman Code

Spielman codes initially proposed in [Spi96] are attractive building blocks due to their linearity, field-agnosticism, and $\mathcal{O}(n)$ encoding time for length-n codewords. The Spielman encoding in Brakedown is systematic and uses a recursive strategy shown in Algorithm 4. Therein, $0 < \alpha < 1$ and $k_0 > 0$ are scheme-specific parameters.

Algorithm 4 $[n, k, d]$ Spielman Encoding

```
1: function $\textsc{Enc}_{n,k,d}(\mathbf{m})$: Maps message $\mathbf{m} \in \mathbb{F}_q^k$ to codeword $\mathbf{c} \in \mathbb{F}_q^n$. $G_1$ and $G_2$ are
   code-specific expander graphs. Either use $\textsc{ExpGr}_{L2R}$ or $\textsc{ExpGr}_{R2L}$ for $\textsc{ExpGr}$.
2:     if $k < k_0$ then
3:         return $\mathbf{c} = \textsc{Enc}'_{n,k,d}(\mathbf{m})$        ▷ $\textsc{Enc}'$ can be any code with sufficiently high $\delta$
4:     else
5:         $\mathbf{y} \leftarrow \textsc{ExpGr}(\mathbf{m}, G_1, \alpha)$        ▷ $\mathbf{y} \in \mathbb{F}_q^{\alpha k}$
6:         $\mathbf{z} \leftarrow \textsc{Enc}_{\alpha n, \alpha k, \alpha d}(\mathbf{y})$        ▷ Recursive encoding; yields $\mathbf{z} \in \mathbb{F}_q^{r\alpha k}$
7:         $\mathbf{v} \leftarrow \textsc{ExpGr}(\mathbf{z}, G_2, \frac{r-1-r\alpha}{r\alpha})$        ▷ $\mathbf{v} \in \mathbb{F}_q^{(r-1-r\alpha)k}$
8:         return $\mathbf{c} = (\mathbf{m} \parallel \mathbf{z} \parallel \mathbf{v})$        ▷ Concatenate $\mathbf{m}$, $\mathbf{z}$, and $\mathbf{v}$
```

During recursive steps (i.e., the termination condition in line 2 is not yet met), Algorithm 4 starts with line 5, where expander graph G_1 is evaluated over the input message $\mathbf{m}$ using either $\textsc{ExpGr}_{L2R}$ (Algorithm 2) or $\textsc{ExpGr}_{R2L}$ (Algorithm 3). This expander graph evaluation yields the vector $\mathbf{y}$, which is by a factor α smaller than $\mathbf{m}$. Subsequently, $\mathbf{y}$ serves as input for the recursive encoding $\textsc{Enc}_{\alpha n,\alpha k,\alpha d}$ which yields $\mathbf{z}$. Finally, another expander graph evaluation G_2 results in the vector $\mathbf{v}$, and the concatenation $(\mathbf{m} \parallel \mathbf{z} \parallel \mathbf{v})$ is the final codeword. The recursion terminates if the message size k is lower than k_0, as shown in lines 2 to 3. In this case, a dedicated encoding function $\textsc{Enc}'_{n,k,d}$ is used, which must have the same or higher relative distance than $\textsc{Enc}_{n,k,d}$. Depending on the scheme and parameters, $\textsc{Enc}'_{n,k,d}$ can be an RS encoding or a simple repetition code.

2.4 The Brakedown Polynomial Commitment Scheme

Modern zero-knowledge proof systems heavily rely on polynomial commitment schemes (PCSs) [Tha22]. In these constructions, the PCS enables the computationally constrained verifier to securely outsource polynomial evaluations to the untrusted but powerful prover.

Brakedown [GLS+23] is one instance of PCS and combines post-quantum security with a transparent setup. This makes Brakedown highly relevant and motivates follow-up research like Orion [XZS22], Scorpius [dHS24], or other schemes [DP23, YZRM24]. The core operations in the Brakedown framework are COMMIT, PROVE, and VERIFY as shown in Algorithm 5. Given a multilinear polynomial $\phi(X_0, \dots, X_{l-1})$ with at most $N = 2^l$ coefficients, COMMIT parses the Lagrange basis of the polynomial into a square matrix $\mathbf{U} \in \mathbb{F}_q^{k \times k}$ with $k = \sqrt{N}$ (lines 2-3 in Algorithm 5). Subsequently, all rows of $\mathbf{U}$ are encoded via the linear error correcting code ENC. The choice of the encoding procedure ENC is the dominating factor for the prover runtime. Throughout this paper, we follow prior work [dHS24, YZRM24, XZS22] and assume the linear-time Spielman code from Section 2.3.1 for ENC. The result of ENC is the matrix $\mathbf{U} \in \mathbb{F}_q^{k \times n}$. Since ENC is systematic, $\hat{\mathbf{U}}[:j] = \mathbf{U}[:j]$ for $0 \le j < k$ and $\hat{\mathbf{U}}[:j]$ for $k \le j < n$ is the result of recursive expander graph evaluations. At the final step of COMMIT, the prover builds a Merkle tree commitment over the columns of $\hat{\mathbf{U}}$ and sends the Merkle tree root $\mathcal{R}$ to the verifier.

After the prover has completed the commitment phase, prover and verifier engage in an interactive protocol by executing PROVE and VERIFY from Algorithm 5, respectively. The PROVE procedure mainly consists of matrix-vector products (line 10) and Merkle tree openings (line 13). Conversely, the VERIFY procedure computes the vectors $\mathbf{q_1}$ and $\mathbf{q_2}$ from the evaluation point $\mathbf{x}$ (line 16). Thereafter, the verifier sends $\mathbf{q_1}$ and a random vector $\mathbf{r}$, and receives $\mathbf{u}'$ and $\mathbf{v}'$ from the prover. Then, the verifier queries the prover for random columns of $\hat{\mathbf{U}}$ and locally performs two Spielman encodings ENC with messages $\mathbf{u}'$ and $\mathbf{v}'$ (line 22). The result of these encodings is checked against the columns revealed by the prover for consistency and proximity [XZS22]. If all checks succeed, the verifier is convinced and computes the genuine evaluation result $y = \phi(\mathbf{x})$.

Algorithm 5 Brakedown's Commitment, Proving, and Verification Phases [GLS+23]

```
1:  function Commit($\phi(X_0, \ldots, X_{l-1}) \in \mathbb{F}_q^{<N}[X_0, \ldots, X_{l-1}]$)
2:      $\mathbf{u} \leftarrow \big(\phi(x_0, \ldots, x_{l-1})\big)_{(x_0,\ldots,x_{l-1}) \in \mathcal{B}^l}$          ▷ $l = \log N$, $\mathbf{u} \in \mathbb{F}_q^N$
3:      Let $k = \sqrt{N}$. Parse $\mathbf{u}$ as $k \times k$ matrix and call it $\mathbf{U} \in \mathbb{F}_q^{k \times k}$
4:      Let $\hat{\mathbf{U}} \in \mathbb{F}_q^{k \times n}$ be a matrix with $n = rk$ and $1/r$ the rate of the linear code Enc.
5:      for $\forall i \in [k]$ do
6:          $\hat{\mathbf{U}}[i\,:] \leftarrow$ Enc($\mathbf{U}[i\,:]$)          ▷ Enc is a $[n, k, d]$ linear encoding procedure
7:      $\mathcal{R} \leftarrow$ MerkleCommit$\big((\hat{\mathbf{U}}[:j])_{j \in [n]}\big)$          ▷ Merkle-commit to columns of $\hat{\mathbf{U}}$
8:      Send the commitment $\mathcal{R}$ to the verifier.

9:  function Prove($\phi(X_0, \ldots, X_{l-1})$, sent by verifier: $\mathbf{q_1} \in \mathbb{F}_q^k, \mathbf{r} \in \mathbb{F}_q^k$)
10:     $\mathbf{u}' \leftarrow \langle \mathbf{r}, \mathbf{U} \rangle$,  $\mathbf{v}' \leftarrow \langle \mathbf{q_1}, \mathbf{U} \rangle$          ▷ $\mathbf{u}', \mathbf{v}' \in \mathbb{F}_q^k$
11:     Send $\mathbf{u}', \mathbf{v}'$ to the verifier
12:     Receive $P, Q \subseteq [n]$ from the verifier. $P, Q$ contain random column indices.
13:     $\hat{P} \leftarrow \{(j, \hat{\mathbf{U}}[:j], \text{MT\_PATH})\}_{j \in P}$,  $\hat{Q} \leftarrow \{(j, \hat{\mathbf{U}}[:j], \text{MT\_PATH})\}_{j \in Q}$
14:     Send the sets of columns $\hat{P}, \hat{Q}$ with their Merkle tree paths to the verifier.

15: function Verify($\mathcal{R}$)
16:     Select evaluation point $\mathbf{x} = (x_0, \ldots, x_{l-1}) \in \mathbb{F}_q^l$
17:     Compute $\mathbf{q_1}, \mathbf{q_2} \in \mathbb{F}_q^k$ such that $\mathbf{q_1} \otimes \mathbf{q_2} = \bigotimes_{i \in [l]} (x_i, 1 - x_i)$
18:     Send $\mathbf{q_1}$ and randomly chosen $\mathbf{r} \overset{\$}{\leftarrow} \mathbb{F}_q^k$ to the prover.
19:     Receive $\mathbf{u}', \mathbf{v}'$ from the prover.
20:     Build sets $P, Q \subseteq [n]$ by randomly sampling column indices. Send $P, Q$ to prover.
21:     Receive $\hat{P}, \hat{Q}$ with the requested columns and Merkle tree paths from prover.
22:     $\hat{\mathbf{u}}' \leftarrow$ Enc($\mathbf{u}'$), $\hat{\mathbf{v}}' \leftarrow$ Enc($\mathbf{v}'$)
23:     for each $(j, \mathbf{p}, \text{MT\_PATH}) \in \hat{P}$ do
24:         Check if $\hat{u}'_j == \langle \mathbf{r}, \mathbf{p} \rangle$ and MT_PATH complies with $\mathcal{R}$
25:     for each $(j, \mathbf{q}, \text{MT\_PATH}) \in \hat{Q}$ do
26:         Check if $\hat{v}'_j == \langle \mathbf{q_1}, \mathbf{q} \rangle$ and MT_PATH complies with $\mathcal{R}$
27:     If all checks pass, output $y = \langle \mathbf{q_2}, \mathbf{v}' \rangle$ and the verifier is convinced that $y == \phi(\mathbf{x})$.
```

Complexity of Operations. In the Brakedown PCS, the encoding procedure Enc clearly causes the major performance bottleneck. Considering the commitment phase, linear encoding consumes between 60% and 80% of the latency for $N = 2^{16}$ to $N = 2^{28}$. In addition, the encoding is not only performed by the prover but also by the verifier, although using significantly smaller datasets. Nevertheless, the linear encoding strongly influences the verification cost. Therefore, a high-performance software implementation applies to both the prover and the verifier-side encoding.

Another important operation in the prover's commitment phase is Merkle tree construction (line 7 in Algorithm 5). Yet, compared to linear encoding, Merkle tree construction is lightweight and causes less than 25% of the commitment latency. We also note that Merkle tree construction is well researched, and existing implementations [Off25, ZRW25] can be applied. Hence, we mainly focus on the more interesting Spielman encoding in this work.

2.5 SIMD and AVX

Single instruction multiple data (SIMD) is a computing paradigm tailored for high data throughput. SIMD provides data-parallel computing by concurrently performing the same operation on all involved data sets. Modern mobile, desktop, and high-end CPUs offer

SIMD support via vectorized instruction set extensions such as AVX in x86 [Int25] or ARM Neon [Arm20]. In this work, we use AVX-512 as a case study without limiting our concepts to this platform. The AVX-512 instruction set extension offers 32 512-bit wide general-purpose registers. An AVX register or vector is partitioned into *lanes* with 64, 32, 16, or 8 bits per lane. Each operation targets all lanes in parallel while executing a single instruction.

In addition to SIMD processing, AVX offers dedicated functionality such as the IFMA extension. The IFMA extension provides instructions for fused multiply-and-accumulate operations (MAC) for 52-bit wide integers. In particular, the `_mm512_madd52lo_epu64(a,b,c)` and `_mm512_madd52hi_epu64(a,b,c)` intrinsics perform 52-bit integer multiplication between lanes of `a` and `b` and either add the multiplication result's lower 52 bits (`madd52lo`) or higher 52 bits (`madd52hi`) to a third, 64-bit wide operand `c`. Due to the larger bit-width of operand `c` (64 versus 52 bits), carry bits caused by repeated accumulation do not cause an integer overflow in `c`. Given this functionality, IFMA instructions allow more efficient MAC computations. Note that similar instructions also exist for ARM Neon [ARM16].

Although AVX-512 leverages parallelism by performing 8 parallel 64-bit operations, the gained speedup from AVX-512 implementation is far from 8× [JPL$^+$25, CFGR22, GBB20]. This is due to the high latency of AVX instructions, which easily exceeds 10 clock cycles [Int24]. Therefore, data-dependent instructions cause long pipeline stalls in the CPU to avoid hazards. This requires a careful software design to achieve optimal performance.

3 Our Proposed Software Architecture

In this section, we present our optimizations to boost the performance of the critical Spielman encoding in PCS. First, we elaborate on our selection of R2L graph evaluation. Building upon this selection, we propose an AVX-512-IFMA implementation of Spielman codes and a lazy reduction approach to avoid frequent modular reduction invocations. Subsequently, we enhance the cache efficiency through our cache-friendly memory layout and our *slicing* technique. Finally, we map our design to multicore CPUs and present our approach to multithreading, including our encoding-level and expander-level parallelism. The combination of these methods boosts the performance while effectively avoiding memory access bottlenecks in the data-centric PCS.

3.1 L2R and R2L Graphs in Software

As discussed in the background Section 2.2, the left-to-right (L2R) graph evaluation and the right-to-left (R2L) graph evaluation are two options for the ExpGr operation. In this work, we choose the R2L evaluation as the default approach since it reduces the data transfers and memory pressure [HKPSR25]. In addition – and even more importantly – the R2L approach enables our lazy reduction method presented in Section 3.2, which would require substantial memory overheads in the L2R paradigm. Finally, multithreading is more effective when using R2L, due to fewer data and control flow dependencies, as we present in Section 3.5.3. Yet, although R2L is our primary selection, we discuss opportunities and limitations of the L2R evaluation in each subsection.

3.2 Efficient Spielman Codes using AVX IFMA

The Spielman encoding within the commitment phase of PCS benefits from the SIMD paradigm since the same linear code is computed over multiple rows (see Algorithm 5, lines 5-6). Thus, we pack the coefficients of 8 consecutive rows into AVX-512 vectors. Yet, to reach sufficient security, the coefficients in $\mathbb{F}_q$ are 128 bits wide, whereas AVX-512 only provides up to 64-bit lanes. Moreover, AVX-512 does not allow full-precision 64-bit multiplication. This requires splitting the $\mathbb{F}_q$ elements into machine word-sized limbs followed by limb-based multiplications and modular reduction.

There are two main options to perform the limb-based multiplication in AVX. First, the limb size can be set to 32 bits, and the `_mm512_mul_epu32` intrinsic can be used to compute the 64-bit limb multiplication result. Using the schoolbook multiplication, 16 `_mm512_mul_epu32` executions are required. After the 16 limb multiplication results are available, dedicated addition instructions are required to accumulate the 256-bit integer multiplication result. The second option for limb-based multiplication is the fused MAC instruction introduced in the AVX-512-IFMA extension (see Section 2.5). With IFMA, the 128-bit operands can be split into three 52, 52, and 14-bit-wide limbs. The three limbs are multiplied and accumulated in a total of 9 `_mm512_madd52lo_epu64` executions and 8 `_mm512_madd52hi_epu64` executions. Note that multiplying the two 14-bit limbs only requires the execution of the `_mm512_madd52lo_epu64` instruction.

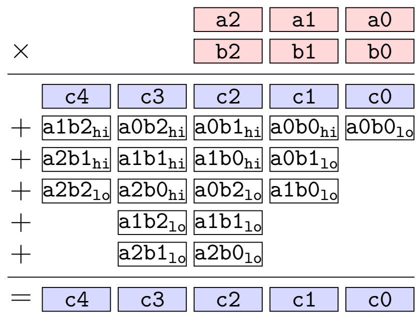
Compared to the first option, the IFMA approach requires one more multiplication instruction while entirely avoiding additional instructions for result accumulation. Moreover, the graph evaluation has a multiply-and-accumulate nature, as shown in line 5 of Algorithm 3. This nature allows to perform the multiplication of left node m_l and weight $w_{l,r}$ as well as the accumulation to the right node y_r using IFMA instructions only. Therefore, we pursue the IFMA approach, which has been applied to other cryptosystems like Multi-Scalar Multiplication [JPL+25], and detail its benefits for Spielman encoding in the next section.

3.2.1 IFMA-based Graph Evaluation

The expander graphs in the Spielman encoding compute the right nodes $y_r = \sum_l m_l \cdot w_{l,r}$ via accumulating multiple left nodes m_l multiplied by a weight $w_{l,r}$. The straightforward approach of computing this MAC operation is to store each 128-bit wide element (y_r, m_l, or $w_{l,r}$) in the 64-bit lanes of two distinct AVX vectors. Yet, for each multiplication, this would require limb-splitting m_l and $w_{l,r}$ to 52-bit limbs before performing the limb-based multiplication using IFMA instructions. Thereafter, an expensive modular reduction, e.g., using Montgomery reduction, yields the 128-bit result – again stored in two AVX vectors. Thus, for computing a single right node y_r, up to 42 limb-splittings and modular reductions are required for our parameter set.

Instead, we propose a lazy reduction approach where no limb-splitting at runtime and just one modular reduction per right node is necessary. We therefore introduce two data formats, namely the *unpacked* and *extended unpacked* data formats.

- **Unpacked Data Format:** The unpacked data format avoids repeated limb-splitting at runtime by storing the 128-bit left nodes and weights in 52/52/14-bit limb representation. Therefore, after initially converting all left nodes and graph weights to the unpacked data format, the values remain in unpacked data format and can be directly used in limb-based multiplication. The AVX vectors representing $a \in \mathbb{F}_q$ hold bits 0 to 51 in vector `a0`, 52 to 103 in vector `a1`, and 104 to 127 in vector `a2`. We store the whole matrix $\mathbf{U}$ (respectively $\hat{\mathbf{U}}$) and the graph weights $w_{l,r}$ in the unpacked data format to avoid splitting the coefficients into limbs for every MAC computation.
- **Extended Unpacked Data Format:** The second data format is the extended unpacked format. The extended unpacked format contains 5 AVX vectors `c0` to `c4` and stores an up to 272-bit wide *unreduced* right node y_r. The right node y_r in extended unpacked format is the result of repeated MAC operations without modular reduction. Therein, limb-based multiplication of 128-bit operands yields 256-bit wide intermediate results. Subsequently, we accumulate several 256-bit intermediate results during MAC operations, and the resulting carry bits are covered in the 272-bit


a2 a1 a0
× b2 b1 b0
c4 c3 c2 c1 c0
+ a1b2hi a0b2hi a0b1hi a0b0hi a0b0lo
+ a2b1hi a1b1hi a1b0hi a0b1lo
+ a2b2lo a2b0hi a0b2lo a1b0lo
+ a1b2lo a1b1lo
+ a2b1lo a2b0lo
= c4 c3 c2 c1 c0

```
input: a0,a1,a2, b0,b1,b2, c0,...,c4
output: c0,...,c4
c0 = _mm512_madd52lo_epu64(c0,a0,b0)
c1 = _mm512_madd52hi_epu64(c1,a0,b0)
c1 = _mm512_madd52lo_epu64(c1,a1,b0)
c1 = _mm512_madd52lo_epu64(c1,a0,b1)
c2 = _mm512_madd52hi_epu64(c2,a1,b0)
c2 = _mm512_madd52hi_epu64(c2,a0,b1)
c2 = _mm512_madd52lo_epu64(c2,a0,b2)
c2 = _mm512_madd52lo_epu64(c2,a1,b1)
c2 = _mm512_madd52lo_epu64(c2,a2,b0)
c3 = _mm512_madd52hi_epu64(c3,a0,b2)
...
```

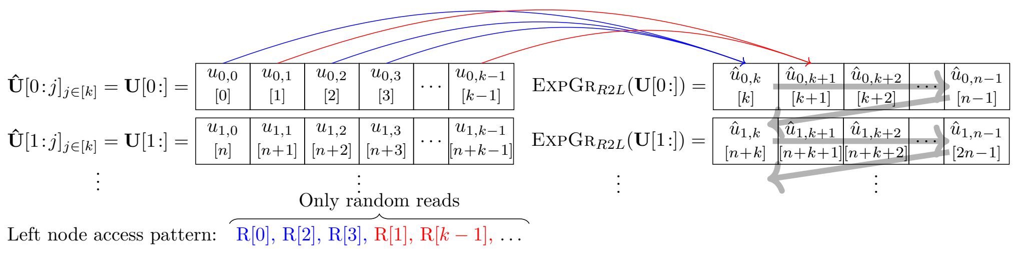
Figure 1: Illustration (left) and code (right) of our IFMA-based graph evaluation approach. Red elements are in the unpacked format using 3 vectors per element, and blue elements are in the extended unpacked format using 5 vectors per element. This operation efficiently computes $c \leftarrow c + a \cdot b$ via IFMA, without addition instructions.

extended unpacked data format. In the extended unpacked format, the vector ci stores bits $52 \cdot i$ to $52 \cdot i + 63$ of the intermediate MAC result. This is due to the functionality of IFMA instructions operating over 52-bit multiplication operands and 64-bit accumulation results.

Using the two data formats, we illustrate the resulting right node accumulation during IFMA-based graph evaluation in Figure 1. Initially, the right nodes in extended unpacked format (represented as ci in blue in Figure 1) are zeroed while the left nodes (represented as ai) and the graph weights (bi) are in the unpacked data format. When computing the repeated MAC operations $c \leftarrow c + a \cdot b$, the IFMA instructions are used to compute the limb-multiplication between vectors ai and bj. Furthermore, the IFMA instructions directly add the multiplication result to c$(i + j)$ in the extended unpacked format. Since the extended unpacked format provides enough space for carry propagation, all MAC operations of a graph evaluation can be computed using the extended unpacked format without performing a modular reduction. Finally, after all accumulations of a right node are done, the right node in extended unpacked format gets modular reduced. The reduction result is stored in unpacked format since it serves as a left node within the next graph recursion. Note that Brakedown is field-agnostic, and any modular reduction may be applied [XZS22, GLS+23, Wah21]. We therefore do not detail specific reduction methods.

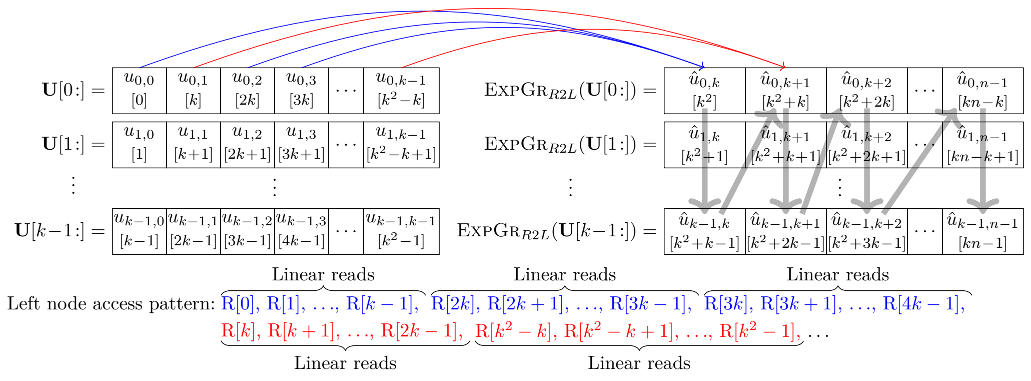
The presented expander graph evaluation significantly reduces the computational effort due to the efficient use of AVX IFMA during node accumulation. In addition, up to 42× fewer modular reductions are required. However, the described technique introduces data dependencies for c1 to c4, as can be seen in the listing in Figure 1. To avoid CPU pipeline stalls, we interleave the computation of all right nodes of a column $\hat{\mathbf{U}}[:j]$. This relaxes pipeline hazards but slightly increases the memory consumption, since all right nodes of one column need to be stored in extended unpacked format. Fortunately, in the R2L case, the size of this auxiliary memory is small enough (at most 640kB for $N = 2^{28}$) to fit into cache, and only linear read/write accesses are performed, thereby not substantially impacting the DDR bandwidth.

Considering the L2R approach, the auxiliary memory must be significantly larger than in R2L. This is because each left node may connect to any right node due to the random graph structure. Therefore, just storing one column of right nodes in the auxiliary memory is insufficient, and all αk columns must be kept in auxiliary memory. The resulting high memory overhead of up to 3.2GB for $N = 2^{28}$ is not desirable in our memory bandwidth-critical setting.



$\hat{\mathbf{U}}[0\!:\!j]_{j\in[k]} = \mathbf{U}[0\!:]=$
$\hat{\mathbf{U}}[1\!:\!j]_{j\in[k]} = \mathbf{U}[1\!:]=$
$\text{ExpGr}_{R2L}(\mathbf{U}[0\!:])=$
$\text{ExpGr}_{R2L}(\mathbf{U}[1\!:])=$
Only random reads
Left node access pattern: R[0], R[2], R[3], R[1], R[k − 1], …

Figure 2: Straightforward memory layout for R2L graph evaluation. The element located in memory location x is denoted as $[x]$.



$\mathbf{U}[0\!:]=$
$\mathbf{U}[1\!:]=$
$\mathbf{U}[k-1\!:]=$
$\text{ExpGr}_{R2L}(\mathbf{U}[0\!:])=$
$\text{ExpGr}_{R2L}(\mathbf{U}[1\!:])=$
$\text{ExpGr}_{R2L}(\mathbf{U}[k-1\!:])=$
Linear reads
Left node access pattern: R[0], R[1], …, R[k − 1], R[2k], R[2k + 1], …, R[3k − 1], R[3k], R[3k + 1], …, R[4k − 1],
R[k], R[k + 1], …, R[2k − 1], R[k² − k], R[k² − k + 1], …, R[k² − 1], …

Figure 3: Cache-friendly memory layout for R2L graph evaluation. $[x]$ denotes that the element is located in memory location x.

3.3 Cache-Friendly Memory Layout

After introducing our IFMA-based graph evaluation, we now focus on leveraging the cache hierarchy in modern CPUs to boost performance. In Brakedown and related PCS, the linear encoding encodes the rows of the Lagrange basis matrix $\mathbf{U}$, as presented in Algorithm 5, line 6. Specifically, each row vector $\mathbf{U}[i\!:]$ serves as input for the Spielman encoding procedure Enc, which yields the respective row vector of the encoded matrix $\hat{\mathbf{U}}[i\!:]$. Since Enc is systematic, $\hat{\mathbf{U}}[i\!:\!j]$ can be initialized with $\mathbf{U}[i\!:\!j]$ for $0 \leq j < k$ and we use $\mathbf{U}$ to denote the input to multi-row expander graph evaluation.

Within Enc, the numerous expander graph evaluations (ExpGr from Algorithm 3) are highly critical due to their random, non-linear memory access pattern. These random memory accesses violate spatial locality and thus prevent caching. Figure 2 illustrates this inefficiency and shows the straightforward memory access pattern in sequential row-wise encodings. Therein, the matrix $\hat{\mathbf{U}}$ is stored in memory in row-major order (i.e., element $\hat{u}_{i,j}$ is stored in memory location $[in + j]$, where $[x]$ refers to the memory location at address x). The row vectors $\hat{\mathbf{U}}[i\!:]$ are processed sequentially, as indicated using gray arrows in Figure 2: Each element $\hat{u}_{i,r}$ is computed via IFMA-based MAC operations involving weights $w_{l,r}$ and random left elements $u_{i,l}$. Thus, caching can only be exploited for writing the resulting elements $\hat{u}_{i,r}$, which is performed in a linear order. However, caching is ineffective for the numerous random read operations to $u_{i,l}$.

We therefore improve the cache efficiency by introducing a different memory layout and processing flow. Specifically, we store the matrix $\hat{\mathbf{U}}$ in *column-major order* as shown in Figure 3, where element $u_{i,j}$, respectively $\hat{u}_{i,j}$ is stored in memory location $[i + kj]$. In addition, we compute the row-wise encodings by completing the whole right-side column

$\hat{\mathbf{U}}[:j]$ $(k \leq j < n-1)$ before advancing to the next column $\hat{\mathbf{U}}[:j+1]$. These two optimizations combined increase the locality of memory accesses.

To illustrate our approach, consider the expander graph evaluation indicated by the blue arrows in Figure 3. The evaluation starts with loading $u_{0,0}$ from memory, which simultaneously caches the elements $u_{1,0}, u_{2,0}, \ldots$ due to prefetching of successive memory locations. Then, $u_{0,0}$ is multiplied by a weight $w_{0,k}$ and accumulated to $\hat{u}_{0,k}$, which is initialized to zero. The execution flow now advances to $u_{1,0}$ and accumulates it to $\hat{u}_{1,k}$ using the same weight $w_{0,k}$. In this case, all operands are already cached, leading to low memory access latency. Additionally, further column elements are prefetched by the CPU. After iterating over the whole column k, the weight is updated to $w_{2,k}$ according to the graph structure. Subsequently, $\hat{u}_{0,k}$ is again accessed and accumulated by $u_{0,2} \cdot w_{2,k}$. Therein, $\hat{u}_{0,k}$ is still in cache and only reading $u_{0,2}$ causes a cache miss. This cache miss prefetches the remaining column elements of $\mathbf{U}[:2]$. Once all computations involving column $\hat{\mathbf{U}}[:k]$ are completed (blue arrows), we advance to $\hat{\mathbf{U}}[:k+1]$ (red arrows) and apply the same strategy.

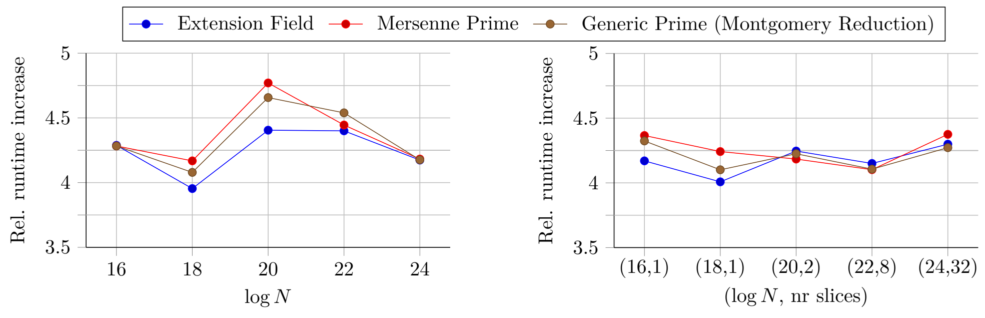
As presented above, we iterate over a column $\hat{\mathbf{U}}[:k]$ multiple times. Each time, we perform IFMA-based MAC operations involving one column $\mathbf{U}[:j]$. Considering the example in Figure 3, we iterate over column $\hat{\mathbf{U}}[:k]$ three times, once for each incoming blue arrow. Thereby, we avoid CPU pipeline stalls caused by read-after-write hazards. This method of instruction-level pipelining was already mentioned in Section 3.2.1. In addition, we only require loading each weight w once per column instead of once per element.

Compared to the initial approach, our cache-friendly memory layout offers a substantial reduction of cache misses and thus increases the performance of the implementation. In addition, we emphasize that restructuring the memory content does not negatively affect other operations of the PCS: The input matrix $\mathbf{U}$ contains the Lagrange basis of the polynomial, which can be efficiently computed in our proposed order. After the linear encoding, a Merkle tree is built over the columns of $\hat{\mathbf{U}}$, where the column hash computation also benefits from data locality provided by our layout. The presented memory layout naturally extends to the AVX case, where multiple consecutive column elements are packed into AVX-512 vectors, as explained in Section 3.2. In addition, the L2R graph evaluation also benefits from the presented memory layout due to increased data locality.

3.4 Effects of LLC Size and the Slicing Technique

As discussed in the previous section, caching is a crucial method to improve the performance of the linear encoding. However, the data involved in the Spielman encoding of PCS easily reaches gigabytes, whereas the available cache size only ranges in megabytes – even on high-end CPUs. For example, our used EPYC 9754 CPU features 16MB of LLC shared across 8 physical cores. In this section, we investigate the influence of LLC size on the linear encoding runtime and introduce our graph slicing technique to improve performance for the large polynomials.

We first examine the baseline runtime of linear encoding using our IFMA-MAC graph evaluation and efficient memory layout. The experimental results on the EPYC 9754 CPU are shown in Figure 4a for different modular reduction methods. Therein, we observe that quadrupling the polynomial degree N causes a $4\times$ to $4.25\times$ increase in runtime for $2^{16} \leq N \leq 2^{24}$. This runtime increase is slightly higher than the expected $4\times$ increase, which is explained by non-idealities of the CPU. Similar effects occur in other works as well [CHL+24, Su, HKPSR25]. Yet, for $2^{20} \leq N \leq 2^{22}$, the runtime increase in Figure 4a is significantly higher, reaching up to $4.75\times$. Interestingly, this effect can be explained by the LLC size: At $N = 2^{20}$, the array of left nodes of the largest graph evaluation (i.e., $u_{i,j}$ in Figure 3) reaches 24MB and thus no longer fits into the 16MB of our LLC. Therefore, the repeated random loads of left nodes cause cache conflicts, leading to high memory pressure and hence lower performance.



(a) Without slicing (Baseline).

(b) With slicing.

Figure 4: Relative increase of encoding runtime for our IFMA-based MAC implementation. Average latency over 120 runs using different fields on EPYC 9754 with 16MB LLC.

We mitigate this performance overhead through our slicing method. Therein, we horizontally slice the matrix $\hat{\mathbf{U}}$ into $s \geq 1$ slices. Each slice consists of k/s rows of the matrix $\hat{\mathbf{U}}$ and fits entirely into the LLC. By sequentially encoding each slice at a time, all involved left nodes are kept in LLC, thereby avoiding frequent DDR accesses. The benefit of slicing is confirmed in Figure 4b, where the relative runtime increase is kept close to 4.25× for different reduction methods. For larger degrees of $N \geq 2^{24}$, slicing becomes less effective, as indicated in Figure 4b. This is because the length of rows $\hat{\mathbf{U}}[i:]$ increases for larger N, while the slice must not exceed 16MB. Thus, each slice must contain fewer rows, and more slices are required overall. Yet, with fewer rows per slice, our cache-friendly memory layout from Section 3.3 becomes less effective and the benefit of slicing decreases. In addition, the entire graph structure needs to be loaded for each slice, which is a drawback if the number of slices increases.

Nevertheless, slicing is an effective method to improve the performance by up to 21%, especially for polynomials with $2^{20} \leq N \leq 2^{24}$. Smaller polynomials with $N < 2^{20}$ do not need slicing since they directly fit into the cache. For larger polynomials, slicing is less efficient, but fine-tuning the number of slices still leads to performance improvements, as discussed in Section 4.1. Note that optimally fine-tuning our slicing technique requires knowledge of the LLC size, which may be obtained from CPU or OS functionality [Ell20]. Otherwise, if the LCC size is not available, empirical timing measurements with different slicing parameters will reveal good estimations.

3.5 Multithreaded Linear Encoding

Multithreading is a prominent option to speed up computations on multi-core CPUs. While multithreading reduces the computation latency via parallel computations, it does not alleviate the limited memory bandwidth. Therefore, it is crucial to analyze the CPU architecture and adapt the software implementation accordingly. We choose the EPYC 9754 CPU architecture as a case study and analyze it in the next subsection. Thereafter, we present our optimized multithreading approach based on the CPU analysis.

We emphasize that we do not rely on any specific features of the EPYC CPU. Instead, we discuss general state-of-the-art CPU features like the cache hierarchy in a multi-core setting. Based on this discussion, we demonstrate a CPU-aware software design to optimally exploit caching and multithreading, which readily translates to other CPU architectures.

3.5.1 The EPYC 9754 CPU

To demonstrate our CPU-aware software design, we choose the server-grade AMD EPYC 9754 CPU platform [AMD23] as a case study. We especially focus on EPYC's multi-die

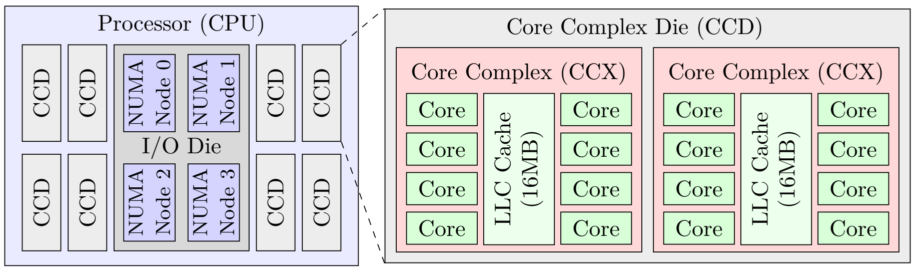


Figure 5: CPU architecture of the EPYC 9004 family with 128 cores [AMD23].

Table 1: Throughput improvement over single-threaded execution using encoding-level parallelism with different types of resource sharing.

lg *N*	Single Thread	Shared SMT	Shared LLC	Shared NUMA	Independent
22	1.00×	0.98×	1.41×	1.60×	1.99×
24	1.00×	1.00×	1.38×	1.52×	1.99×

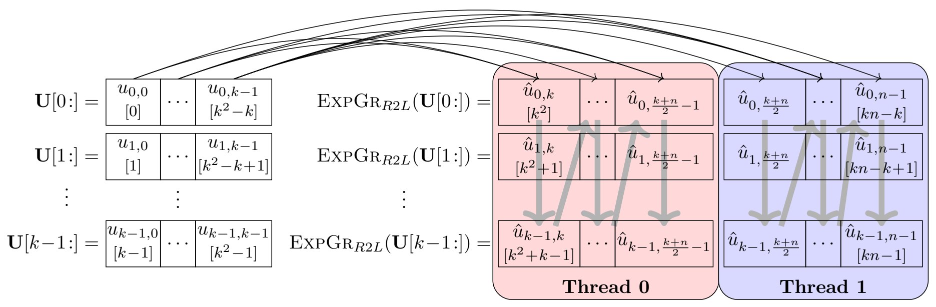
CPU architecture and cache hierarchy, shown in Figure 5. The CPU consists of 9 individual dies: one I/O Die responsible for off-chip communication, and 8 Core Complex Dies (CCDs). Each CCD instantiates two Core Complexes (CCX) with 8 Zen4c cores per CCX. The CCX provides a 16MB LLC, which is shared across the 8 cores. In addition, each core instantiates a 1MB L2 cache and a 32kB L1 data cache. One Zen4c core features two SMT threads. Given this modular architecture, each CCX can be seen as an individual CPU with 16 logical threads and an exclusive LLC.

The I/O Die incorporates four Non-Uniform Memory Access (NUMA) nodes. Each NUMA node serves two CCDs and allows faster DDR accesses to certain ranges of the DDR memory. In essence, the DDR is partitioned into four memory regions, each assigned to one NUMA node. The latency of a NUMA node accessing its assigned memory region is lower than accessing other memory regions [AMD23]. On the other hand, a NUMA node poses a bottleneck when the two connected CCDs are concurrently accessing the DDR memory. This effect is part of the memory bottleneck and needs careful approaches when increasing thread-level parallelism.

3.5.2 Encoding-Level Parallelization

To illustrate the memory bandwidth problem, we first consider encoding-level parallelism on the EPYC 9754. Therein, we assume multiple *independent* linear encodings of matrices $\mathbf{U}_i$ $(i > 1)$, where the encoding of matrix $\mathbf{U}_i$ is executed using a single thread. Table 1 shows the throughput increase compared to single-threaded execution for encoding-level parallelism of $i = 2$ encodings. In an ideal multithreading scenario, the throughput increase would be 2×. Yet, due to shared resources such as SMT threads, LLC, or the NUMA node, the throughput increase is significantly lower.

According to Table 1, forcing both encodings to run on the same SMT thread does not yield any throughput increase. For example, executing a single-threaded encoding of $N = 2^{22}$ lasts 0.19ms, while executing two parallel encodings on the same SMT thread lasts 0.39ms, which obviously does not provide a throughput advantage over a single-threaded execution. More interestingly, running two parallel encodings on one CCX (different cores but shared LLC) leads to a throughput increase of about 1.4× − far from the ideal 2× increase. This is caused by frequent cache conflicts in the shared LLC, since the amount of data required by the cores is doubled. In addition, sharing the NUMA node also impacts the throughput. As shown in Table 1, two parallel encodings with independent LLC but shared NUMA node only reach 1.52× and 1.6× throughput improvement over single



$\mathbf{U}[0:] =$
$u_{0,0}$ [0]
$u_{0,k-1}$ $[k^2-k]$
$\mathbf{U}[1:] =$
$u_{1,0}$ [1]
$u_{1,k-1}$ $[k^2-k+1]$
$\mathbf{U}[k-1:] =$
$u_{k-1,0}$ $[k-1]$
$u_{k-1,k-1}$ $[k^2-1]$
$\textsc{ExpGr}_{R2L}(\mathbf{U}[0:]) =$
$\textsc{ExpGr}_{R2L}(\mathbf{U}[1:]) =$
$\textsc{ExpGr}_{R2L}(\mathbf{U}[k-1:]) =$
$\hat{u}_{0,k}$ $[k^2]$
$\hat{u}_{0,\frac{k+n}{2}-1}$
$\hat{u}_{0,\frac{k+n}{2}}$
$\hat{u}_{0,n-1}$ $[kn-k]$
$\hat{u}_{1,k}$ $[k^2+1]$
$\hat{u}_{1,\frac{k+n}{2}-1}$
$\hat{u}_{1,\frac{k+n}{2}}$
$\hat{u}_{1,n-1}$ $[kn-k+1]$
$\hat{u}_{k-1,k}$ $[k^2+k-1]$
$\hat{u}_{k-1,\frac{k+n}{2}-1}$
$\hat{u}_{k-1,\frac{k+n}{2}}$
$\hat{u}_{k-1,n-1}$ $[kn-1]$
Thread 0
Thread 1

Figure 6: Workload distribution of our expander-level multithreading for two threads.

threaded execution. To reach the desired 2× throughput increase, the two encodings must run on independent CCDs without sharing a NUMA node or LLC. Only in this case, the encodings do not interfere, and throughput impacts are avoided.

Naturally, the reported interferences are amplified when running more than two threads since more data needs to be streamed to and from memory. This clearly shows the memory bottleneck issue of multithreaded linear encodings, where the memory bandwidth is insufficient to supply the CPU. We therefore keep encoding-level parallelism as an option to run independent encodings on CCDs or CCXs (which can be seen as an independent CPU, as mentioned above). In addition, we present a more memory-efficient multithreading approach, exploiting the multiple cores within a CCX, in the next subsection.

3.5.3 Expander-Level Parallelism

As demonstrated in the previous section, improvements in computational power through multithreading are limited due to increased memory bandwidth demands. We therefore focus on *expander-level* parallelism, where multiple threads collaboratively compute a single encoding within one CCX. Each EPYC CCX can be seen as an individual CPU featuring an LLC shared across 8 cores and 16 SMT threads. Thus, our expander-level parallelism technique also applies to other CPUs.

As a first step, we apply our cache-friendly memory layout to fit the data into the LLC of the CCX. In addition, we use up to 16 SMT threads to compute the right nodes during the expander graph evaluation. The details of our expander-level parallelism are shown in Figure 6 for two threads. Therein, we split the array of right-side nodes (i.e., the result of expander graph evaluation) into two equally sized parts, one for each thread. Each thread operates over its assigned memory region and computes a subset of right nodes. For example, thread 0 in Figure 6 operates over the red-colored memory region and computes the right nodes $\hat{\mathbf{U}}[:j]$ for $j < \frac{k+n}{2}$. Concurrently, thread 1 computes the right nodes $\hat{\mathbf{U}}[:j]$ for $\frac{k+n}{2} \leq j < n$ colored in blue. To compute the right nodes, both threads read the left nodes $\mathbf{U}[:j]$, $j \in [k]$, which are kept in LLC due to our slicing method. Since we use the R2L expander graph evaluation, no data dependencies across threads occur: Only left nodes, which are the *constant* input of the graph evaluation, are simultaneously accessed by multiple threads. This shared data fits into the LLC and, therefore, DDR accesses are minimized during the expander-level parallel computation.

Finally, expander-level parallelism operates within a CCX of the EPYC CPU. Since 16 CCXs are available, we can use encoding-level parallelism (Section 3.5.2) to distribute independent encodings of up to 16 matrices $\mathbf{U}_{i\in[16]}$ across the CCXs. Within each CCX, expander-level parallelism computes the encoding performantly.

4 Results and Comparison

In this section, we discuss the speedup contributions of each implementation technique. Thereafter, we present concrete runtimes and comparisons with Brakedown and Orion software implementations on different CPUs and on an FPGA. For all experiments, we implemented our concepts in C++, compiled with GNU g++, version 12.2.0, and the -O3 flag on Debian 12.2.0-14.

4.1 Speedup Gain by Technique

The charts in Figure 7 show the individual speedup contributions of each optimization, including our cache-friendly memory layout, lazy reduction, and slicing method. For this comparison, we instantiate the field-agnostic Spielman encoding with a Mersenne prime reduction and with Montgomery reduction for arbitrary primes. The baseline is our single-threaded AVX-512 implementation without lazy reduction on the EPYC 9754 CPU.

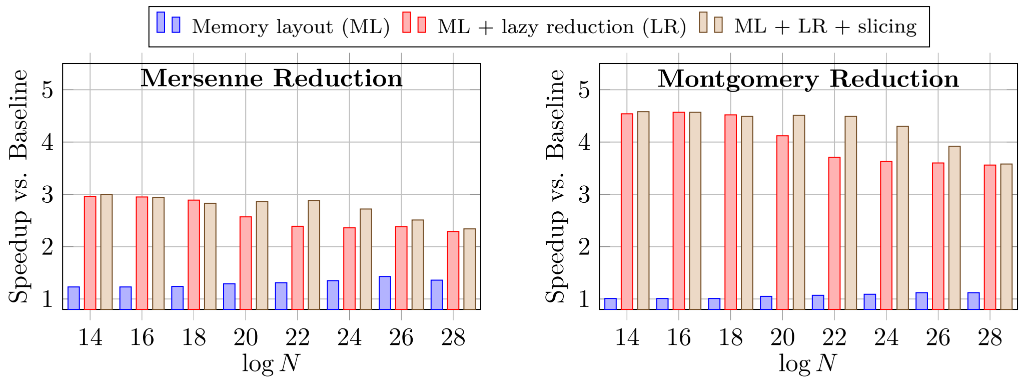


Figure 7: Contribution of optimizations to the overall speedup for Mersenne (left) and Montgomery reduction (right).

We start by adding our cache-friendly memory layout (ML) to the baseline implementation. This allows a speedup of up to 12% and 43% for Montgomery and Mersenne reduction, respectively (blue bars in Figure 7). Next, we enable our lazy reduction method (LR), which avoids repeated modular reductions. Combining LR with ML significantly improves the speedup by up to $3\times$ for Mersenne reduction and $4.6\times$ for Montgomery reduction (red bars). We observe that ML+LR yields a higher speedup in Montgomery reduction than in Mersenne reduction. This is attributed to the comparably simple Mersenne reduction, whereas the Montgomery reduction is more complex and slower. Therefore, the Montgomery case benefits more from avoiding frequent modular reductions through our lazy reduction method. Finally, we introduce our slicing technique. Slicing improves the LLC cache efficiency and contributes a speedup of up to 21% for both Mersenne and Montgomery reduction (brown bars). Note that slicing is most effective for polynomials with $2^{20} \leq N \leq 2^{26}$.

Furthermore, Figure 7 shows a decreasing overall speedup with increasing polynomial degree N. For example, in the Mersenne case, the overall speedup reached for $N = 2^{14}$ is $3\times$ while the overall speedup for $N = 2^{28}$ is $2.3\times$. This is caused by the increased data streams in our AVX-512 implementation, including the growing graph structure and the increased number of slices. After this performance analysis, we compare our results to related work in the next section.

Table 2: Comparison to the encoding runtime of related Brakedown implementations.

	lg N	**Our Latency** (ms)		**LCPC** (ms, our speedup)				**PolyCommit** (ms, our speedup)			
		1 thread	8 threads	1 thread		8 threads		1 thread		8 threads	
EPYC 9754	16	**1.45**	**0.48**	24.9	17.2×	5.3	10.9×	25.8	17.8×	5.0	10.3×
	18	**5.96**	**1.81**	91.1	15.3×	20.3	11.2×	97.1	16.3×	18.2	10.1×
	20	**25.18**	**7.24**	363.1	14.4×	61.1	8.4×	386.2	15.3×	70.1	9.7×
	22	**103.37**	**20.77**	1,466.9	14.2×	205.5	9.9×	1,638.2	15.8×	279.4	13.4×
	24	**441.54**	**87.26**	5,944.7	13.5×	784.7	9.0×	6,823.1	15.5×	1,269.0	14.5×
	26	**2,007.44**	**392.05**	25,651.0	12.8×	3,329.2	8.5×	26,707.0	13.3×	6,075.2	15.5×
	28	**8,880.82**	**2,205.57**	97,657.0	11.0×	18,135.0	8.2×	-	-	-	-
i7-11800H	16	**1.00**	**0.40**	18.7	18.7×	4.4	11.0×	19.3	19.3×	4.0	10.1×
	18	**4.25**	**1.22**	68.7	16.2×	18.3	15.1×	73.7	17.3×	15.2	12.5×
	20	**18.88**	**4.90**	287.4	15.2×	65.5	13.4×	296.2	15.7×	58.5	11.9×
	22	**79.30**	**23.29**	1,158.8	14.6×	248.4	10.7×	1,234.4	15.6×	272.9	11.7×
	24	**331.37**	**101.44**	4,699.8	14.2×	996.0	9.8×	5,153.4	15.6×	1,174.1	11.6×
	26	**1,636.78**	**607.49**	20,259.0	12.4×	4,338.4	7.1×	20,810.0	12.7×	5,593.7	9.2×
Xeon Gold 6530	16	**1.02**	**0.46**	25.0	24.4×	8.7	18.7×	27.4	26.7×	7.4	15.8×
	18	**5.31**	**1.40**	91.5	17.2×	21.4	15.3×	104.0	19.6×	27.0	19.4×
	20	**24.65**	**4.88**	371.5	15.1×	64.7	13.3×	409.5	16.6×	100.8	20.6×
	22	**105.27**	**24.24**	1,444.6	13.7×	223.4	9.2×	1,696.7	16.1×	334.8	13.8×
	24	**474.57**	**94.93**	5,858.3	12.3×	810.0	8.5×	7,477.5	15.8×	1,312.0	13.8×
	26	**2,032.54**	**368.60**	25,327.0	12.5×	3,649.6	9.9×	32,158.0	15.8×	5,734.1	15.6×
	28	**9,914.76**	**1,629.92**	104,680.0	10.6×	13,664.0	8.4×	-	-	-	-

4.2 Comparison to Related Works

We now compare our Spielman encoding latency to related works. For that, we choose the state-of-the-art LCPC [Wah21] and PolyCommit [CHL+24] libraries. Both implement the Spielman encoding as used in the Brakedown PCS in Rust using Montgomery reduction. To ensure a fair comparison, we choose the same 127-bit prime and reduction method and collect all timing results on an EPYC 9754 @2.25GHz, Intel i7-11800H @2.30GHz, and Xeon Gold 6530 @2.1GHz CPU. The results for 1 and 8 threads are presented in Table 2. We apply our expander-level parallelism on 8 cores sharing the LLC. We do not use SMT threading in all experiments.

The work LCPC [Wah21] shows a runtime between 24.94ms and 97.7s for $N = 2^{16}$ to $N = 2^{28}$ and single-threaded execution on the EPYC CPU. Compared to this, our AVX implementation in C++ reaches a speedup between 17.2× and 12.8× for $N = 2^{16}$ to $N = 2^{26}$. Even for large polynomials, our speedup is one order of magnitude (11× for $N = 2^{28}$). In the multi-threaded scenario with 8 threads, our speedup over LCPC reaches from 8.2× to 11.2× for the EPYC CPU. Considering the Intel i7 CPU, we observe up to 50% lower runtimes than in the EPYC CPU for both LCPC and our work. This effect is more prominent for single-threaded execution and lower N. The primary reason for this may be microarchitectural differences, such as the larger LLC of the i7 CPU with 24MB [Inta], while the EPYC CPU only features 16MB LLC per CCX. Our setup for the i7 CPU only provides 16GB of RAM, which is insufficient to run experiments with $N = 2^{28}$. Moreover, our precomputed graph structure stores the weights as AVX-512 vectors. This consumes more memory than in LCPC, where weights are stored as scalars. Therefore, our polynomials with $N \geq 2^{26}$ exhaust the available RAM memory on i7. This leads to swapping of the graph structure and lowers our speedup to 12.4× and 7.1× for 1 and 8 threads, respectively. However, when RAM size does not impact performance, we reach a speedup of 14.2× to 18.7× and 9.8× to 15.1× for single and multi-threaded execution on i7. On the Xeon Gold CPU, we reach similar runtimes as on EPYC, and our speedup ranges from 10.6× to 24.4× and 8.4× to 18.7× for single and multi-threaded experiments, respectively.

Next, we focus on PolyCommit [CHL+24], which also provides Spielman encodings as used in Brakedown for $N \leq 2^{26}$. Compared to PolyCommit, our design on EPYC

Table 3: Comparison to the encoding runtime of related Orion design (single-threaded).

lg N	**[Su]'s Latency in ms**			**Our Latency in ms and Speedup**					
	EPYC	i7	Xeon	EPYC		i7		Xeon	
16	27.0	18.4	20.0	2.31	11.7×	1.46	12.6×	1.39	14.4×
18	121.2	83.3	112.2	10.10	12.0×	6.40	13.0×	7.15	15.7×
20	540.3	364.3	490.3	48.67	11.1×	33.50	10.9×	34.36	14.3×
22	3,057.7	2,149.3	2,097.7	292.39	10.5×	214.63	10.0×	205.80	10.2×
24	15,763.7	-	9,653.7	2,221.47	7.1×	1,626.56	-	1,302.59	7.4×
26	91,794.3	-	53,952.0	10,962.71	8.4×	8,563.58	-	8,546.77	6.3×

reaches 13.3× to 17.8× and 9.7× to 15.5× performance improvements for single-threaded and multi-threaded execution, respectively. Considering the i7 CPU, we lower the single-threaded runtime by 19.3× to 15.6× for $N = 2^{16}$ to $N = 2^{24}$. For $N = 2^{26}$, our speedup is 12.7× due to the limited RAM on i7 and less effective slicing. The multithreaded scenario shows similar speedups in the range of 9.2× to 12.5×. Finally, our design on the Xeon Gold CPU outperforms PolyCommit by 15.8× to 26.7× and by 13.8× to 20.6× for single- and multi-threaded scenarios, respectively.

Although we use 8-lane AVX-512 vectors, achieving a speedup of 8× and more in Spielman encodings of PCSs is not trivial. First, the memory pressure is 8-fold higher, requiring a careful software design to avoid memory bottlenecks. Second, the EPYC 9754 CPU only features a 256-bit processing datapath [AMD23]; the used 512-bit AVX registers are split into two 256-bit registers, which are processed sequentially in the CPU pipeline. Thus, EPYC's 265-bit pipeline has lower performance than a fully unrolled 512-bit pipeline. Nevertheless, our implementation offers speedups of one order of magnitude for most configurations. This confirms the effectiveness of our optimizations on both high-end EPYC and Xeon CPUs, as well as on the smaller i7 CPU.

4.2.1 Comparison to Orion

In addition to Brakedown implementations, we also present benchmarks for the closely related Orion scheme [XZS22, Su]. The main difference between Orion and Brakedown is the shape of the Lagrange basis matrix $\mathbf{U}$. While Brakedown uses $\mathbf{U}$ in a square shape with $\sqrt{N}$ rows and columns, Orion parses $\mathbf{U}$ as a 128-row and $N/128$-column matrix. Moreover, the Orion software in [Su] uses an 122-bit extension field $\mathbb{F}_{p^2}$ over the Mersenne prime $p = 2^{61} - 1$. We adopt the same parameters for this experiment.

Table 3 shows our latency comparison with Orion's Spielman encoding. Therein, we observe a 10.5× to 12× speedup for $N \leq 2^{22}$ on the EPYC CPU. Yet, for $N = 2^{24}$ and 2^{26}, the speedup is lower. The reason for this effect is the matrix shape of Orion with only 128 rows. This counteracts the slicing technique since at most $128/8 = 16$ slices with very few rows per slice can be established. Combined with reduced cache efficiency due to short matrix columns, this causes the lower speedup. This effect also manifests on the Xeon Gold 6530 CPU, where our speedup is higher for smaller polynomials. Specifically, our design on Xeon reaches a speedup between 6.3× and 15.7× for the reported parameter set. Considering the i7 CPU, our speedup is between 10× and 13× for $N \leq 2^{22}$. For larger N, the 16GB RAM of the i7 setup is insufficient to execute the reference implementation [Su].

4.2.2 Comparison to FPGA

The work in [HKPSR25] presents an FPGA accelerator for the commitment and proving procedures in the Orion [XZS22] scheme. Therefore, [HKPSR25] also supports Spielman codes on a high-end Alveo U280 FPGA with High Bandwidth Memory. We compare this FPGA design to our software implementation running on the EPYC CPU. To boost the

Table 4: Throughput comparison to FPGA accelerator for Orion.

lg N		**16**	**18**	**20**	**22**	**24**	**26**
FPGA Throughput [HKPSR25]	(Enc/s)	3,144.7	803.2	202.2	50.5	12.6	3.2
Our Software Throughput	(Enc/s)	12,048.2	3,467.7	554.5	139.5	17.2	2.9
Our Improvement		**3.8**×	**4.3**×	**2.7**×	**2.8**×	**1.4**×	**0.9**×

Table 5: Performance comparison to NTT-based Reed-Solomon encoding.

lg N	**18**	**22**	**26**
NTT size	2^{10}	2^{12}	2^{14}
Latency single NTT in ms [BKS+21]	0.0081	0.0375	0.2135
Lantency RS encoding of $\mathbf{U}$ in ms [BKS+21]	4.1	76.8	1,749.4
Our Spielman encoding latency for $\mathbf{U}$ in ms	6.0	103.4	2,007.4
Overhead of Spielman encoding	**1.44**×	**1.35**×	**1.15**×

throughput of our implementation, we apply encoding-level parallelism (Section 3.5.2) and expander-level parallelism (Section 3.5.3). Specifically, we perform 8 concurrent encodings using encoding-level parallelism, one on each CCD. Within every CCD, we use expander-level parallelism involving 8 threads to collaboratively compute one linear encoding. Thus, a total of 64 threads are used in this experiment.

Table 4 presents the achieved throughput results in Spielman encodings per second (Enc/s). Our parallel execution of Spielman encodings achieves a throughput of about 12k to 2.9 Enc/s for $N = 2^{16}$ to 2^{26}. Compared to this, the FPGA design reaches about 3k to 3.2 Enc/s for the same polynomial sizes. Hence, our software on the high-end EPYC CPU achieves a notable throughput increase of up to 4.3× over the powerful FPGA accelerator. For very large polynomials with $N = 2^{26}$, the FPGA slightly outperforms our throughput.

4.2.3 Comparison to Reed-Solomon Codes

In this section, we compare the concrete performance of our Spielman encoding with the Reed-Solomon (RS) encoding. Since RS encoding utilizes the Number-Theoretic Transform for the row-wise encoding of $\mathbf{U}$, we choose the Intel HEXL library [BKS+21] for comparison. HEXL is a highly performant implementation of the NTT for FHE applications relying on AVX-512 and the IFMA extension – the same CPU features we use in our implementation. Yet, HEXL uses 50-bit moduli for its IFMA implementation. We therefore scale up the HEXL latency reported in [BKS+21] for 128-bit primes, assuming a quadratic dependency between prime size and latency [2]. Table 5 shows the scaled latency of HEXL's NTT on a server-grade Intel Xeon Platinum 8360Y @2.4GHz [Intb] with 54MB cache, which makes the results comparable to our EPYC CPU.

To RS-encode the matrix $\mathbf{U} \in \mathbb{F}_q^{k \times n}$ with $k = \sqrt{N}$ rows and columns, a total of k NTTs is required, one for each row. For a code rate $r^{-1} = 2$, each NTT involves $2k$ points. Therefore, for $N = 2^{18}$ / 2^{22} / 2^{26}, NTTs with 2^{10} / 2^{12} / 2^{14} points are required. One such row-wise NTT takes 0.0081ms / 0.0375ms / 0.2135ms, and the overall latency to RS-encode $\mathbf{U}$ is 4.1ms / 76.8ms / 1,749.4ms. Compared to RS-codes, our Spielman encoding has a small runtime overhead of 1.44× / 1.35× / 1.15×, as shown in Table 5. While NTT implementations have benefited from decades of optimization and highly regular memory access patterns, Spielman encodings inherently involve irregular memory accesses that hinder performance. Despite this disadvantage and the lack of prior software optimization efforts, our work reduces the latency of Spielman encoding to within a small margin of

[2]The best method for 128-bit integer multiplication using IFMA's 52-bit multiplication instructions is the Schoolbook method with a quadratic complexity. Hence, we use the scaling factor $(128/50)^2 = 6.5$.

Table 6: Latency comparison of the verifier-side encoding runtime (single-threaded).

	lg N	**Our Latency**	**LCPC** [Wah21]		**PolyCommit** [CHL+24]	
		μs	μs	Our Speedup	μs	Our Speedup
EPYC 9754	16	**39**	195	5.0×	202	5.2×
	18	**78**	356	4.6×	379	4.9×
	20	**160**	709	4.4×	754	4.7×
	22	**323**	1,433	4.4×	1,600	5.0×
	24	**649**	2,903	4.5×	3,332	5.1×
	26	**1,316**	6,262	4.8×	6,520	5.0×
i7-11800H	16	**25**	146	5.9×	151	6.0×
	18	**51**	268	5.3×	288	5.6×
	20	**107**	561	5.2×	579	5.4×
	22	**213**	1,132	5.3×	1,205	5.7×
	24	**430**	2,295	5.3×	2,516	5.9×
	26	**894**	4,946	5.5×	5,081	5.7×
Xeon Gold 6530	16	**45**	195	4.3×	214	4.7×
	18	**83**	357	4.3×	406	4.9×
	20	**159**	726	4.6×	800	5.0×
	22	**357**	1,411	4.0×	1,657	4.6×
	24	**602**	2,860	4.8×	3,651	6.1×
	26	**1,166**	6,183	5.3×	7,851	6.7×

state-of-the-art RS code implementations. Moreover, the results from Table 5 show a *decreasing* runtime gap between RS and Spielman codes with increasing N, reflecting the asymptotically lower $\mathcal{O}(N)$ complexity of Spielman codes compared to the $\mathcal{O}(N \log N)$ complexity of RS codes.

We note that both encoding approaches have distinct benefits, such as field agnosticism in Spielman codes or the higher distance of RS codes, allowing smaller proof sizes. Nevertheless, our results show that Spielman codes can reach the performance of RS codes and thus do not have a substantial performance drawback.

4.2.4 Results for Verifier-Side Encoding

The Brakedown scheme requires Spielman encoding for both the prover-side polynomial commitment and verifier-side proof verification (see Algorithm 5, lines 6 and 22). Although this work focuses on the heavyweight prover-side encoding, our SIMD optimizations, including the IFMA-based graph evaluation and lazy reduction, also apply to the verifier-side encoding. Therefore, we present verifier-side encoding results in this section.

The verifier in Brakedown encodes two vectors, $\mathbf{u}'$ and $\mathbf{v}'$, each having $k = \sqrt{N}$ elements (see Algorithm 5, line 22). Hence, we pack the elements of $\mathbf{u}'$ and $\mathbf{v}'$ into the two lanes of AVX-128 vectors and apply our IFMA-based expander graph evaluation and lazy reduction approach. Compared to the prover-side Spielman encoding, the data sizes on the verifier-side are small and easily fit into the LLC, even without our slicing technique. Specifically, even for polynomials with degree $N = 2^{28}$, the data requirement for the verifier is just 1.5MB, whereas the prover operates over several gigabytes.

Table 6 shows our runtime for verifier-side encoding on the EPYC, i7, and Xeon CPUs. Compared to LCPC [Wah21] and PolyCommit [CHL+24], we reach a performance improvement between 4.0× and 6.7× across all CPUs and polynomial degrees N. We also note a more uniform speedup on the verifier side than on the prover side. In particular, the prover's speedup tends to decrease for increasing N due to memory bandwidth limitations (see Table 2). Yet, this is not the case for the verifier with their relaxed memory demands.

5 Discussion

In this paper, we present novel implementation techniques for high-performance Spielman encoding used in Brakedown-like PCS. Furthermore, our contributions extend to related PCS schemes, such as [dHS24, YZRM24, XZS22], where Spielman encoding is used.

5.1 Performance on Different Microarchitectures

We demonstrate the performance of our software optimizations on two server-grade CPUs (AMD EPYC and Intel Xeon) and a consumer CPU (Intel i7-11800H) in Section 4. While all three investigated CPUs show clear performance improvements, it is difficult to extrapolate the concrete runtime to other CPUs. This is due to the large microarchitectural variety, different execution backends, and undocumented features of modern CPUs. Nevertheless, we discuss the performance impacts of prominent microarchitectural features like caches, SIMD support, SMT threading, and multicore systems.

Caches: We first focus on caches, which are a particularly important microarchitectural component for the memory-centric Spielman encoding. The LLC size clearly affects the performance of our design, since larger LLCs hold more data and require fewer slices (see Section 3.4). This improves performance due to reduced accesses to the graph structure and enhanced memory layout efficiency. Yet, concrete cache architectures differ substantially across CPUs. For example, Non-Uniform Cache Access (NUCA) provides faster access to core-local LLC slices compared to slices of other cores [RHT^{+}25, FRMK19]. Such variations in LLC access latency and slice configuration also influence the overall performance.

SIMD: Another key feature of modern CPUs is SIMD support. In this work, we utilize AVX-512 with the IFMA extension for SIMD computation, but our concepts are not limited to AVX-512. Instead, our lazy reduction, efficient memory layout, and expander-level parallelism techniques also apply to other SIMD extensions such as AVX-256 or Arm Neon. Yet, the microarchitectural implementation of the SIMD execution pipeline will influence the achievable latency of Spielman encoding. For instance, our EPYC CPU provides a 256-bit AVX pipeline as discussed in Section 4.2. Smaller pipeline configurations will have reduced throughput, while larger execution units may improve the encoding latency.

SMT and multicore systems: Next, we consider SMT threading, which enables concurrent execution of threads on a single core. In our experiments, SMT provides a 3% to 9% throughput improvement compared to non-SMT execution for $N = 2^{22}$ to 2^{24}. However, parallelism across physical cores yields significantly higher throughput than SMT. This makes additional cores more beneficial than SMT support for high-performance implementations. Finally, the number of CPU cores and the associated LLC size determine the achievable degree of parallelism and the resulting encoding throughput. As discussed in Section 3.5, cores with shared LLC should utilize expander-level parallelism to leverage the LLC and to reduce memory streams. Yet, cores with independent LLC can combine expander-level and encoding-level parallelism. Overall, CPUs such as our EPYC 9754 with many cores will reach higher throughput due to encoding-level parallelism, whereas latency remains similar to CPUs with fewer cores.

5.2 Merkle Tree Construction

The commitment phase in Brakedown involves Merkle tree construction and linear encoding. In this paper, we focus on the unexplored linear encoding since optimizations for Merkle tree computations have been covered in prior work [Off25, ZRW25]. Moreover, Merkle tree construction is less critical regarding memory pressure due to its linear access pattern. Future software integrations can combine existing Merkle tree optimization techniques with our optimized Spielman encoding to realize a fully optimized PCS.

5.3 Timing Side-Channel Resistance

Since the expander graph is public, and the implementation avoids any secret-dependent memory accesses or branching, it does not expose a surface for timing attacks. Further work may address advanced power side-channel attacks (e.g., profiling-based methods) that could attempt to infer input data from evaluations of the public graph. Masking prevents such attacks and benefits from the linearity of the Spielman encoding.

6 Conclusion

This work presented the first comprehensive software optimization of Spielman codes for Brakedown-like polynomial commitment schemes. Spielman codes are attractive for their linear-time complexity and field agnosticism, but their irregular memory access patterns and lack of prior optimization have limited their practical performance.

We addressed these challenges and designed a high-performance SIMD software with several optimizations targeting computation and memory overheads. Our IFMA-based expander graph evaluation and lazy reduction significantly reduced computational complexity. In addition to SIMD acceleration, we proposed a cache-friendly data layout and a slicing technique to reduce the memory access overhead. Combining these techniques with a CPU- and LLC-specific fine-tuning improved the overall performance. Finally, we proposed expander-level parallelism to boost throughput while enhancing cache locality.

Compared to related Brakedown implementations, we improved the encoding latency by up to 17.8× and 15.5× on an EPYC 9754 CPU, for single- and multi-threaded cases, respectively. On Xeon Gold 6530 and i7-11800H CPUs, we reached similar speedups, which confirms the effectiveness and portability of our techniques.

Our Spielman encoding achieved a competitive performance compared to Reed-Solomon encoding. Overall, our cache-aware SIMD optimizations for Spielman encoding advanced the state of the art in zero-knowledge proofs and contributed to efficient, scalable, and field-agnostic polynomial commitments on general-purpose CPUs.

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 performant software implementations of zero-knowledge proofs and was not supported by any ZKP-specific funded project.

We thank the anonymous reviewers of TCHES 2026 for their valuable feedback.

References

[AMD23] AMD Inc. 4th gen amd epyc processor architecture, 2023. https://www.amd.com/content/dam/amd/en/documents/epyc-business-docs/white-papers/221704010-B_en_4th-Gen-AMD-EPYC-Processor-Architecture---White-Paper_pdf.pdf.

[APKP24] Kasra Abbaszadeh, Christodoulos Pappas, Jonathan Katz, and Dimitrios Papadopoulos. Zero-Knowledge Proofs of Training for Deep Neural Networks. In Bo Luo, Xiaojing Liao, Jun Xu, Engin Kirda, and David Lie, editors, *Proceedings of the 2024 on ACM SIGSAC Conference on Computer and Communications Security, CCS 2024, Salt Lake City, UT, USA, October 14-18, 2024*, pages 4316–4330. ACM, 2024.

[ARM16] ARM Limited. ARM Compiler armasm User Guide Version 5.06, 2016. https://developer.arm.com/documentation/dui0473/m/neon-instructions/vmlal--by-scalar-?lang=en, Accessed on October 2025.

[Arm20] Arm Limited. Introducing neon, version 1.0, 2020. https://developer.arm.com/documentation/102474/0100/?lang=en, Accessed on October 2025.

[BKS+21] Fabian Boemer, Sejun Kim, Gelila Seifu, Fillipe D.M. de Souza, and Vinodh Gopal. Intel hexl: Accelerating homomorphic encryption with intel avx512-ifma52. In *Proceedings of the 9th on Workshop on Encrypted Computing & Applied Homomorphic Cryptography*, WAHC '21, page 57–62, New York, NY, USA, 2021. Association for Computing Machinery.

[CCC+25] Ignacio Cascudo, Anamaria Costache, Daniele Cozzo, Dario Fiore, Antonio Guimarães, and Eduardo Soria-Vazquez. Verifiable computation for approximate homomorphic encryption schemes. In Yael Tauman Kalai and Seny F. Kamara, editors, *Advances in Cryptology – CRYPTO 2025*, pages 643–677, Cham, 2025. Springer Nature Switzerland.

[CFGR22] Hao Cheng, Georgios Fotiadis, Johann Großschädl, and Peter Y. A. Ryan. Highly vectorized sike for avx-512. *IACR Transactions on Cryptographic Hardware and Embedded Systems*, 2022(2):41–68, Feb. 2022.

[CHL+24] Alessandro Chiesa, Yuncong Hu, William Lin, Mary Maller, Pratyush Mishra, Noah Vesely, and Nicholas Ward. Poly-commit: A rust library for polynomial commitments, 2024. https://github.com/arkworks-rs/poly-commit/tree/master/.

[CYY+21] Xiangren Chen, Bohan Yang, Shouyi Yin, Shaojun Wei, and Leibo Liu. Cfntt: Scalable radix-2/4 ntt multiplication architecture with an efficient conflict-free memory mapping scheme. *IACR Transactions on Cryptographic Hardware and Embedded Systems*, 2022(1):94–126, Nov. 2021.

[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.

[Ell20] Daniel Ellis. Obtaining cpu cache size, 2020. https://medium.com/cemac/obtaining-cpu-cache-size-bafd9607e61a, Accessed on December 2025.

[Fou] The Zcash Foundation. Zcash digital currency. https://z.cash, Accessed on October 2025.

[FRMK19] Alireza Farshin, Amir Roozbeh, Gerald Q. Maguire, and Dejan Kostić. Make the most out of last level cache in intel processors. In *Proceedings of the Fourteenth EuroSys Conference 2019*, EuroSys '19, New York, NY, USA, 2019. Association for Computing Machinery.

[GBB20] Mathias Gottschlag, Peter Brantsch, and Frank Bellosa. Automatic core specialization for avx-512 applications. In *Proceedings of the 13th ACM International Systems and Storage Conference*, SYSTOR '20, page 25–35, New York, NY, USA, 2020. Association for Computing Machinery.

[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.

[HKPSR25] Florian Hirner, Florian Krieger, Constantin Piber, and Sujoy Sinha Roy. Accelerating hash-based polynomial commitment schemes with linear prover time. *IACR Transactions on Cryptographic Hardware and Embedded Systems*, 2025(4):341–385, Sep. 2025.

[Inta] Intel Corp. Intel® core™ i7-11800h processor. https://www.intel.com/content/www/us/en/products/sku/213803/intel-core-i711800h-processor-24m-cache-up-to-4-60-ghz/specifications.html, Accessed on October 2025.

[Intb] Intel Corp. Intel® xeon® platinum 8360y processor. https://www.intel.com/content/www/us/en/products/sku/212459/intel-xeon-platinum-8360y-processor-54m-cache-2-40-ghz/specifications.html, Accessed on October 2025.

[Int24] Intel Corp. Intel intrinsics guide, 2024. https://www.intel.com/content/www/us/en/docs/intrinsics-guide/index.html#.

[Int25] Intel Corp. Intel advanced vector extensions 10.1 - architecture specification, revision 3.1, 2025. https://cdrdv2.intel.com/v1/dl/getContent/784267.

[JPL+25] Rui Jiang, Cong Peng, Min Luo, Rongmao Chen, and Debiao He. Simdmsm: Simd-accelerated multi-scalar multiplication framework for zk-snarks. *IACR Transactions on Cryptographic Hardware and Embedded Systems*, 2025(2):681–704, Mar. 2025.

[KDHR26] Florian Krieger, Christian Dobrouschek, Florian Hirner, and Sujoy Sinha Roy. Public source code, 2026. https://github.com/flokrieger/SIMD-Spielman-for-ZKPs.

[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.

[LCH14] Sian-Jheng Lin, Wei-Ho Chung, and Yunghsiang S. Han. Novel polynomial basis and its application to reed-solomon erasure codes. In *2014 IEEE 55th Annual Symposium on Foundations of Computer Science*, pages 316–325, 2014.

[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.

[Off25] OffchainLabs. hashtree, 2025. https://github.com/OffchainLabs/hashtree.

[OZZ+24] Yi Ouyang, Yihong Zhu, Wenping Zhu, Bohan Yang, Zirui Zhang, Hanning Wang, Qichao Tao, Min Zhu, Shaojun Wei, and Leibo Liu. Falconsign: An efficient and high-throughput hardware architecture for falcon signature generation. *IACR Transactions on Cryptographic Hardware and Embedded Systems*, 2025(1):203–226, Dec. 2024.

[PR21] Somnath Panja and Bimal Roy. A secure end-to-end verifiable e-voting system using blockchain and cloud server. *J. Inf. Secur. Appl.*, 59:102815, 2021.

[Pro] The Monero Project. Monero - secure, private, untraceable. https://www.getmonero.org/, Accessed on October 2025.

[RHT+25] Mikka Rainer, Lorenz Hetterich, Fabian Thomas, Tristan Hornetz, Leon Trampert, Lukas Gerlach, and Michael Schwarz. Rapid reversing of non-linear cpu cache slice functions: Unlocking physical address leakage. In *2025 IEEE Symposium on Security and Privacy (SP)*, pages 3497–3515, 2025.

[RRG+25] Tiago B. Rodrigues, Alexandre Rodrigues, Manuel Goulão, Pedro Tomás, and Leonel Sousa. Accelerating ntt with risc-v vector extension for fully homomorphic encryption. *IACR Transactions on Cryptographic Hardware and Embedded Systems*, 2025(4):711–736, Sep. 2025.

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

[Su] Sunblaze-ucb. sunblaze-ucb/orion. https://github.com/sunblaze-ucb/Orion. Accessed on October 2025.

[Tha22] Justin Thaler. Proofs, arguments, and zero-knowledge. *Foundations and Trends in Privacy and Security*, 4(2-4):117–660, 2022. https://doi.org/10.1561/3300000030.

[Wah21] Riad S. Wahby. lcpc: Polynomial commitment scheme from any linear code with sufficient minimum distance, 2021. https://github.com/conroi/lcpc.

[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.

[XHL+24] Runqing Xu, Debiao He, Min Luo, Cong Peng, and Xiangyong Zeng. Optimizing dilithium implementation with avx2/-512. *ACM Trans. Embed. Comput. Syst.*, 23(6), September 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.

[YZRM24] Xiao Yang, Chengru Zhang, Mark Ryan, and Gao Meng. Multivariate multi-polynomial commitment and its applications. *Cryptology ePrint Archive*, 2024.

[ZRW25] Yaoyun Zhou, Kavin Rajasekaran, and Qian Wang. Exploring parallel implementation of sphincs+ using advanced vector extensions (avx) sets. In *2025 26th International Symposium on Quality Electronic Design (ISQED)*, pages 1–8, 2025.