IACR Transactions on Cryptographic Hardware and Embedded Systems
ISSN 2569-2925, Vol. 2025, No. 2, pp. 681–704. DOI:10.46586/tches.v2025.i2.681-704

SimdMSM: SIMD-accelerated Multi-Scalar Multiplication Framework for zkSNARKs

Rui Jiang[1], Cong Peng[1]✉, Min Luo[1], Rongmao Chen[2] and Debiao He[1]✉

[1] School of Cyber Science and Engineering, Wuhan University, Wuhan, China, {jiangrui,cpeng,mluo,hedebiao}@whu.edu.cn
[2] National University of Defense Technology, Changsha, China, chromao@nudt.edu.cn

Abstract. Multi-scalar multiplication (MSM) is the primary building block in many pairing-based zero-knowledge proof (ZKP) systems. MSM at large scales has become the main bottleneck in ZKP implementations. Inspired by existing SIMD-accelerated work, we are focused on accelerating MSM computing efficiency using SIMD instructions in a single CPU environment. First, we propose a SIMD-accelerated MSM computing architecture with no write conflicts and constant memory overheads. This architecture utilizes multithreading to achieve task-level and loop-level parallelism and employs a three-tier buffer mechanism to maximize the utilization of the SIMD engine. Instanced with AVX512-IFMA instructions, we implement six SIMD elliptic curve arithmetic engines for different point addition in three coordinate systems and two groups. Moreover, we integrate our AVX-MSM implementation into the libsnark library, naming it AVX-ZK. In more detail, point deduplication and "Three-Stage" memory optimization are proposed to address problems existing in practical applications. Based on the RELIC library, our performance results on the BLS12-381 curve show that our AVX-MSM achieves up to 27.86× speedup over the most popular Pippenger algorithm. Compared with libsnark, our AVX-ZK implementation achieves over 11.53× (up to 20.26×) speedup under standard benchmarks.

Keywords: Multi-scalar Multiplication · Zero-knowledge Proof · SIMD Parallel Implementation

1 Introduction

Zero-knowledge proof (ZKP) is a cryptographic primitive that enables the prover to demonstrate the truth of a specific statement to the verifier without revealing any other confidential information [GMR89, GK96]. The emergence of ZKP is a significant breakthrough in the field of cryptography. In recent years, increased investment in research has not only driven substantial theoretical advancements [Gro10, PHGR13, Gro16, BBB+18] but also revealed its vast potential in practical applications, such as Zcash [zca22] and zk-cred [RWGM23]. In particular, *zero-knowledge Succinct Non-Interactive Arguments of Knowledge* (zkSNARKs) have drawn much attention. Among these, pairing-based zkSNARKs, such as Groth16 [Gro16], Plonk [GWC19], and Nova [KST22], are widely used in current applications.

The generation of pairing-based ZKP systems face two main bottlenecks: multi-scalar multiplication (MSM) and number theoretic transform (NTT). Among these, MSM stands out as the most time-consuming operation, comprising approximately 70% to 80% of the total runtime [LWY+23]. It calculates the inner products of scalar vectors and point vectors, following the polynomial computation phase in proof generation. MSM is to compute $Q = \sum_{i=1}^{n} k_i P_i$, where k_i is a scalar of λ bits and P_i is the point in an elliptic curve. Although the Pippenger algorithm can accelerate MSM to some degree, the performance

Received: 2024-10-15 Accepted: 2024-12-15 Published: 2025-03-04

of current MSM implementations remains inadequate, especially when handling extensive data sets. As is well-known, the Pippenger algorithm [Pip76] performs outstandingly in accelerating MSM, particularly at large data scales [BDLO12]. However, there remains significant potential for further optimization in state-of-the-art implementations based on Pippenger's method.

Nowadays, it is imperative to enhance the implementation approaches across various software and hardware platforms, due to the widespread adoption of ZKP. Previous work has demonstrated implementations of ZKP on hardware platforms such as FPGAs [RDQY24, Xav22, ZDW+22, ABC+22], GPUs [LWY+23, MXS+23, JZXJ24, CPD+24], and ASICs [ZWZ+21]. However, the high costs and inconvenience associated with hardware implementations limit their accessibility to a broad user base. For the vast majority of average users, access to computational resources primarily revolves around machines based on CPU platforms. Also, we believe that ZKP implementations should be diverse, whether on high-performance GPUs, FPGAs, or accessible CPUs. ZKPs on mobile devices greatly influence applications such as private transactions, self-sovereign identity, and scalable computation, playing an important role in advancing the field. Mobile users may prefer leveraging their local CPU resources. That is why the ZPrize [zpr] competition retains a track to accelerate MSM on mobile. ZKP acceleration on CPUs can make ZKPs more user-friendly. Consequently, achieving fast ZKP implementations on CPU platforms is of considerable practical importance to advance cryptographic research and applications.

In conventional computing modes, each instruction typically handles only one data element at a time. However, Single Instruction Multiple Data (SIMD) introduces vector registers and corresponding instructions, allowing for the simultaneous processing of multiple data elements of the same type within a single instruction cycle. Many chip manufacturers or architecture, like Intel [Int], AMD [AMD], RISC-V [ris], and ARM [ARM], now support SIMD and similar instruction sets. SIMD facilitates parallel computation of data, and previous work[CFG+21, CGT+20] has demonstrated the use of AVX-512 to accelerate point operations on elliptic curves. Therefore, it can be considered to be utilized to accelerate large-scale computations of MSM. Specifically, we aim to address the following question:

Is it possible to use the built-in parallelism of modern CPUs to improve the efficiency of MSM computations?

1.1 Difficulties

MSM is to compute $Q = \sum_{i=1}^{n} k_i P_i$, where k_i is a λ-bit scalar and P_i is an elliptic curve point. In the Pippenger algorithm [Pip76], it is decomposed into $\lceil \frac{\lambda}{s} \rceil$ subtasks G_j by choosing s as the window size. The application of AVX-512 to accelerate MSM operations presents numerous challenges. The difficulties for this work are as follows:

Difficulty-1: Data-level Parallel Architecture. In a multi-core CPU platform, both multi-threading and SIMD can be employed for parallel data processing, and their performance improvements can be independent of each other. Processing multiple subtasks G_j in parallel could be an effective approach. However, SIMD strictly requires that all input data follow the exact same instructions, which can pose challenges when trying to parallelize across subtasks. For example, within the SIMD framework's parallel processing of multiple subtasks, sequential point additions cannot be directly executed due to the potential for point copy. Also, $\lceil \frac{\lambda}{s} \rceil$ is often not a multiple of the parallelism degree, leaving some subtasks to be handled individually, which underutilizes the parallel capacity. Therefore, we propose handling parallelism between subtasks using multithreading, while designing a SIMD-based parallel architecture for operations within each subtask.

Difficulty-2: Operation Non-independence within a Single Subtask. We shift our focus to parallelism within individual subtasks. Due to the construction of the Pippenger

algorithm, it is necessary to perform a reasonable structural segmentation before utilizing SIMD instruction sets acceleration. Although constructing 8-way parallel point additions intuitively during the bucket accumulation phase in the subtask computation could expedite the operations, potential conflicts may arise. Specifically, two or more points points might be placed into the same bucket, leading to conflicts that need to be addressed. Therefore, we develop a rational scheduling mechanism that connects the underlying 8-way operators, thereby achieving SIMD acceleration of MSM.

Difficulty-3: Large Memory Overhead. While MSM carries a relatively large amount of computing data, the runtime memory overhead is a crucial concern. Moreover, in CPU environments, the available memory capacity is limited, and the space occupied by the elliptic curve points to be computed may need to reach up to GBs. Hence, it's imperative to minimize the runtime memory overhead. First, additional memory overhead from parallel architectures should be minimized as much as possible. In addition, we should consider it whether there is a way to prevent the storage space of elliptic curve points from growing linearly with the increase in operation scale.

1.2 Our Contributions

In this paper, we propose a high-speed CPU-based implementation for MSM and ZKP using SIMD and multi-threads. The proposed methods are applicable to most pairing-based zk-SNARKs and are scalable to other SIMD instruction sets. Our contributions are summarized as follows:

Low-cache Parallel Scheduling Mechanism. We overcome potential problems in parallelism by designing a novel parallel SIMD-accelerated MSM architecture. Our implementation overall uses three-level parallelism to make full use of parallel resources on CPUs. For the SIMD data-level, a three-tier buffer mechanism is designed to accommodate different types of operations, ensuring the parallel workflow proceeds without issues. To deal with the writing conflict, we implement a state transition mechanism. In terms of low-cache, we make it sure that the buffer designed in the structure minimizes additional memory overhead. And we store addresses instead of values to avoid redundant space occupation. Furthermore, the drawbacks of other suboptimal methods are discussed, highlighting the advantages of the proposed approach.

SIMD Elliptic Curve Acceleration Engines. Using SIMD, we present a three-level heterogeneous computing approach to compute MSM, thus speeding up the implementation of ZKP on CPU platforms. The "Tree-Like" structure optimized based on coordinate systems can be seamlessly adapted to our architecture. Specifically, using AVX-512IFMA, we instantiate six SIMD elliptic curve engines for the bottom 8-way point addition operations for both $\mathbb{G}_1$ and $\mathbb{G}_2$ groups.

Point Deduplication and "Three-Stage" Memory Optimization. We address the issue of errors in point addition operations due to the lack of strongly unified arithmetic in practical ZKP implementations. For large-scale MSM scenarios, a "three-stage" storage approach are proposed to alleviate memory pressure caused by excessive points. To maximize the performance of the Pippenger algorithm, we not only calculate the optimal window size for different scales but also empirically test the optimal window values. During runtime, the window size is dynamically adjusted based on the number of points.

AVX-ZK Implementation. We present a high-speed implementation of AVX-MSM not only over $\mathbb{G}_1$ but also over $\mathbb{G}_2$ on the BLS12-381 curve. Compared to the Pippenger algorithm, it achieves acceleration speedup range from 20.25× to 27.86×. We integrate our MSM implementation into the `libsnark` library, naming it AVX-ZK. Using libsnark as a baseline, we evaluate the performance of AVX-ZK, observing over 11.53× (up to 20.26×) speedup improvements. Additionally, several zero-knowledge proof use cases were tested, achieving performance gains of up to 5.00×.

2 Preliminaries

Notation. Let $\mathbb{N}$ denote the set of positive integers. Let $[n, m]$ denote an integer set $\{n, n+1, \cdots, m\}$ and $[m]$ denote an integer set $\{1, 2, \cdots, m\}$ for any positive integer $n < m \in \mathbb{N}$. Let $\mathbb{F}_{q^k}$ to denote a finite field for any prime q and small positive integer k [1].

2.1 Elliptic Curve Groups and Arithmetic

Elliptic curves over finite fields offer efficient group structures well-suited for cryptography. For brevity, an *elliptic curve* is a plane curve over a finite field (i.e., $\mathbb{F}_q$ or $\mathbb{F}_{q^2}$) which consists of the points satisfying the curve equation and a distinguished point at infinity, denoted $\mathcal{O}$. Let $\mathbb{G}_1$ and $\mathbb{G}_2$ denote the elliptic curve group over $\mathbb{F}_q$ and $\mathbb{F}_{q^2}$ respectively.

Group Law. For any two points P and Q in the elliptic curve group $\mathbb{G}_1$ (or $\mathbb{G}_2$), the commutative group operation involves adding two points to obtain another point. This operation can be divided into two cases depending on the equality of the input points: *point addition* (denoted **PADD**) for different points ($P \neq Q$) and *point doubling* (denoted **PDBL**) for equal points ($P = Q$). Also, the operation defines $P + \mathcal{O} = \mathcal{O} + P = P$ to make $\mathcal{O}$ as the group identity. Moreover, the most frequent operator PMUL is *scalar multiplication* (also called point multiplication), which refers to the accumulation of k identical points, i.e., $Q = kP$ where k is a positive integer.

Table 1: The modular operation cost for point addition and doubling in different curves and coordinates. $\mathbf{PADD}^*$ denote point addition where $Z_1 = Z_2 = 1$, $\mathbf{PADD}^+$ denote mixed point addition where $Z_1 \neq Z_2 = 1$, $\mathbf{PADD}^\#$ denote point addition where $Z_1 \neq Z_2 \neq 1$, $\mathbf{PDBL}^*$ denote point double where $Z = 1$ and $\mathbf{PDBL}^\#$ denote point double where $Z \neq 1$.

Curve	Coordinate	$\mathbf{PADD}^*$	$\mathbf{PADD}^+$	$\mathbf{PADD}^\#$	$\mathbf{PDBL}^*$	$\mathbf{PDBL}^\#$
Short Weierstrass	Projective	$5\mathbf{M} + 2\mathbf{S}$	$9\mathbf{M} + 2\mathbf{S}$	$12\mathbf{M}$	$3\mathbf{M} + 5\mathbf{S}$	$5\mathbf{M} + 6\mathbf{S}$
Short Weierstrass	Jacobian	$4\mathbf{M} + 2\mathbf{S}$	$7\mathbf{M} + 4\mathbf{S}$	$11\mathbf{M} + 5\mathbf{S}$	$1\mathbf{M} + 5\mathbf{S}$	$2\mathbf{M} + 5\mathbf{S}$
Twisted Edwards	Extended	$7\mathbf{M}$	$8\mathbf{M}$	$9\mathbf{M}$	$3\mathbf{M} + 4\mathbf{S}$	$4\mathbf{M} + 4\mathbf{S}$
Twisted Edwards	Extended ($a = -1$)	$6\mathbf{M}$	$7\mathbf{M}$	$8\mathbf{M}$	$3\mathbf{M} + 4\mathbf{S}$	$4\mathbf{M} + 4\mathbf{S}$

Point Coordinates. Geometrically, the point in *affine coordinates* can be represented by two finite field elements $(x, y) \in \mathbb{F}_q^2$ which satisfy one curve equation, e.g., Weierstrass form, Edwards form, Twisted Edwards form. The affine coordinate system has minimal storage cost but requires modular inversion operations. To improve efficiency, various coordinate systems have been developed., including *Projective coordinates* represented as $\{X, Y, Z\}(x = X/Z,\ y = Y/Z)$ and *Jacobian coordinates* represented as $\{X, Y, Z\}(x = X/Z^2,\ y = Y/Z^3)$. In this work, we use **M**, **S**, **I** to represent modular multiplication, squaring and inversion, respectively. We briefly review the cost of different point operations from the EFD database [2] and show some related results in Table 1. It is worth stating that the computational costs of point operations may depend on the equality of two Z-coordinates for the added points.

Also, elliptic curve pairing is an important primitive in cryptography, which is a bilinear map between points on elliptic curves and a target group, preserving linearity across inputs. It plays a crucial role in advanced cryptographic protocols like identity-based encryption and zk-SNARKs.

[1] The basic prime field in this work are mainly $\mathbb{F}_q$ and $\mathbb{F}_{q^2}$.

[2] Explicit-Formulas Database: http://hyperelliptic.org/EFD/

2.2 Multi-scalar Multiplication and Pippenger Algorithm

For computing $Q = \sum_{i=1}^{n} k_i P_i$, the most direct approach is to utilize the binary modular exponentiation algorithm for aggregated scalar multiplication [Riv11] as shown in Figure 1. For one scalar multiplication $k_i P_i$, it costs $(\lambda - 1)\mathbf{PDBL} + \mathsf{HW}(k_i)\mathbf{PADD}$ operations where $\mathsf{HW}(k_i)$ represents the Hamming weight of the scalar k_i, approximately $\frac{\lambda}{2}$. For computing $\sum_{i=1}^{n} k_i P_i$, it costs $(\lambda - 1)\mathbf{PDBL} + \sum_{i=1}^{n} \mathsf{HW}(k_i)\mathbf{PADD}$ operations.

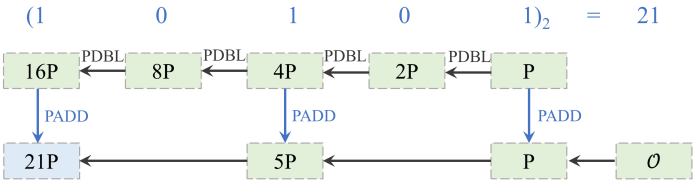

Figure 1: Computing kP by the binary scalar multiplication algorithm

Pippenger Algorithm. A well-known and widely-used method to implement multi-scalar multiplication is due to Pippenger. It works as follows:

- **Task Decomposition:** The first step is to break down the main task into several subtasks that can be executed independently. In Eq. 1, it shows that each subtask is to compute $G_\alpha = \sum_{i=1}^{n} k_{i\alpha} P_i$ $(1 \le \alpha \le \lceil \frac{\lambda}{s} \rceil)$, where $k_{i\alpha}$ denotes the α-th bit-block of the scalar k_i divided by the window size s. The original MSM task is converted into $\lceil \frac{\lambda}{s} \rceil$ subtasks.

$$Q = \sum_{i=1}^{n} k_i P_i = \sum_{i=1}^{n} \sum_{\alpha=1}^{\lceil \frac{\lambda}{s} \rceil} \left(2^{(\alpha-1)s} k_{i\alpha}\right) P_i = \sum_{\alpha=1}^{\lceil \frac{\lambda}{s} \rceil} 2^{(\alpha-1)s} G_\alpha \tag{1}$$

- **Bucket Accumulation:** The second step is to compute the sum of points with the same scalar. For each subtask, it sets $2^s - 1$ buckets, denote each bucket point as B_ℓ where ℓ is the bucket index and $\ell \in [1, 2^s - 1]$, and puts each point P_i into the corresponding bucket $B_{k_{i\alpha}}$ indexed by $k_{i\alpha}$. For example, it adds P_i to B_5 if $k_{i\alpha} = 5$ and adds P_i to B_9 if $k_{i\alpha} = 9$. This process needs $(n - 2^s + 1)$ **PADD** operations.

$$B_\ell = \sum_{k_{i\alpha} = \ell} P_i, \forall \ell \in [2^s - 1] \tag{2}$$

- **Bucket Aggregation:** The third step is to compute the sum of all bucket points weighted by their bucket indexes, i.e., $G_\alpha = \sum_{i=1}^{2^s-1} \ell B_\ell$. It iteratively computes the point $M_\ell = M_{\ell+1} + B_\ell$ from $M_{2^s-1} := B_{2^s-1}$ to M_1, then compute the sum $G_\alpha = \sum_{\ell=1}^{2^s-1} M_\ell$. It can be easily checked that B_ℓ has been added ℓ times for any $\ell \in [2^s - 1]$. This process needs $(2^{s+1} - 4)$ **PADD** operations.

$$G_\alpha = \sum_{i=1}^{2^s-1} \ell B_\ell = B_{2^s-1} + \sum_{\ell=2^s-2}^{2^s-1} B_\ell + \sum_{\ell=2^s-3}^{2^s-1} B_\ell + \cdots + \sum_{\ell=1}^{2^s-1} B_\ell = \sum_{\ell=1}^{2^s-1} M_\ell \tag{3}$$

- **Final Combination:** The final step is to compute the subtask results as $Q = \sum_{\alpha=1}^{\lceil \frac{\lambda}{s} \rceil} 2^{(\alpha-1)s} G_\alpha$ via double-and-add method. This process needs $(s \lceil \frac{\lambda}{s} \rceil - s)$ **PDBL**+ $(\lceil \frac{\lambda}{s} \rceil - 1)$ **PADD** operations.

Total Costs. In summary, the total computational overhead of MSM by the Pippenger algorithm is approximately $\lceil \frac{\lambda}{s} \rceil (n + 2^s - 2)$ **PADD**+ $(s \lceil \frac{\lambda}{s} \rceil - s)$**PDBL**. Only during the bucket accumulation phase is $\mathbf{PADD}^+$ used, whereas $\mathbf{PADD}^\#$ is employed in all other stages. The main memory overhead is $2^s - 1$ buckets per subtask.

2.3 Single Instruction Multiple Data

Single Instruction Multiple Data (SIMD) is a computer instruction set architecture and processor microarchitecture technology that allows multiple data elements to be processed under a single instruction, enabling parallel computing on the CPU. It has seen extensive application in elliptic curve cryptography, serving as a common method for accelerating computations on CPUs. Numerous works [CFG+21, CGT+20] have demonstrated the use of SIMD to implement foundational elliptic curve operations, significantly enhancing performance in cryptographic protocols.

The AVX-512 (Advanced Vector Extensions 512) instruction set is an extended instruction set for the x86 architecture released by Intel. It can be regarded as an extension of the AVX2 instruction set. Compared with previous generations of instruction sets (MMX, SSE, AVX, AVX2), the register width and number of registers available for AVX-512 have doubled. The register width has increased from 256 bits to 512 bits, and the number of registers has risen from 16 to 32.

In addition to the basic AVX-512F instructions (F for foundation), AVX-512 also introduces a new extension called AVX-512IFMA (Integer Fused Multiply-Add), which is capable of supporting fused multiply and add operations for integers within the Intel AVX-512 instruction set.

3 Our SIMD-Accelerated MSM Architecture

As described in Section 2.2, bucket accumulation requires significant computational costs (decided by the MSM size n) and can be naturally parallelized. Unlike GPUs and FPGAs, the SIMD parallelism mechanism strictly requires all data to execute the same operation and is limited by the width of the supported registers, which restricts its achievable parallelism. Therefore, utilizing SIMD to accelerate MSM computation presents a challenging problem. Upon initial observation, it seems that there are several ways to apply SIMD to MSM computation. However, each of these methods has notable and considerable drawbacks. In the upcoming sections, we will compare our multi-level parallelism approach with these alternatives, detailing how we systematically tackled these challenges and crafted a completely new framework.

Inspired by previous SIMD-accelerated implementations [BS12, CFGR22, CFG+21, ZHZ+24, CCC+09], we can utilize the SIMD instruction set for constructing multiplexed parallel finite field arithmetic and subsequently for building elliptic curve arithmetic, such as (8×1)-way or (4×2)-way or (2×4)-way. Formally, SIMD can be thought of as providing a pointwise addition for two vectors of elliptic curve points, e.g. $\{R_i\}_{i\in[\chi]} = \{P_i\}_{i\in[\chi]} \oplus \{Q_i\}_{i\in[\chi]} = \{P_i + Q_i\}_{i\in[\chi]}$. Let χ denote the parallelism number of SIMD point arithmetic engines.

3.1 Challenges for SIMD Implementations

Attempt-1: Performing Multiple Subtasks Using SIMD Parallelism. Intuitively, each subtask can be run independently, allowing multiple subtasks to be calculated in parallel. In GPU, $\lceil \frac{\lambda}{s} \rceil$ threads can be executed to compute each subtask simultaneously. However, parallelism by SIMD instruction sets is different from GPU threads, since SIMD requires all data processed in parallel to perform the same operation, and conditional branching statements are generally avoided. When points are added to buckets, there are two scenarios, one is to copy P_i to B_ℓ directly if $B_\ell = \mathcal{O}$ and another is to add P_i to B_ℓ if $B_\ell \neq \mathcal{O}$. Unfortunately, both scenarios can occur when adding a point P_i to χ buckets of different subtasks.

Though setting a different point index for each subtask will solve the problem, it is easy to assume that the frequency of every subtask's **PADD** is different when n is relatively small. In addition, parallelizing across subtasks needs more memory and access time. More importantly, $\lceil \frac{\lambda}{s} \rceil$ is not necessarily a multiple of parallelism such as 8-way. Considering the above reasons, we do not consider using parallelism between subtasks.

In a multi-core CPU environment, both multi-threading and SIMD can be employed together to accelerate computations by assigning each core to handle a subtask and parallelizing the computation within each subtask. Therefore, our approach introduces multi-threading to achieve task-level parallelism for sub-MSM processing, while utilizing SIMD for data-level parallelism within each thread.

Attempt-2: Accelerating Bucket Accumulation via SIMD Engines. Easily, we can use the SIMD engine to directly add χ points into corresponding bucket points during the bucket accumulation process. It is essential to ensure that the same address does not exist at the output point of the SIMD engines to avoid writing conflicts on buckets. So, this necessarily involves opening up a cache to wait for the state that satisfies the condition to trigger the engine.

Open Buffers for Each Bucket. Preparing a set of buffers for each bucket enables parallel computation of bucket points, as illustrated in the Figure 2. For example, processing eight-way point additions for each bucket means that once the buffer is filled with eight pairs of elliptic curve points, an eight-way point addition is performed. However, how to store these points still remains a problem. Adding these points directly into buckets or continuing pairwise addition using a tree-like structure are two approaches. But the former cannot achieve 8-way parallelism due to writing conflicts, while the latter requires large additional memory to store the results under the premise that it already constitutes a large memory overhead.

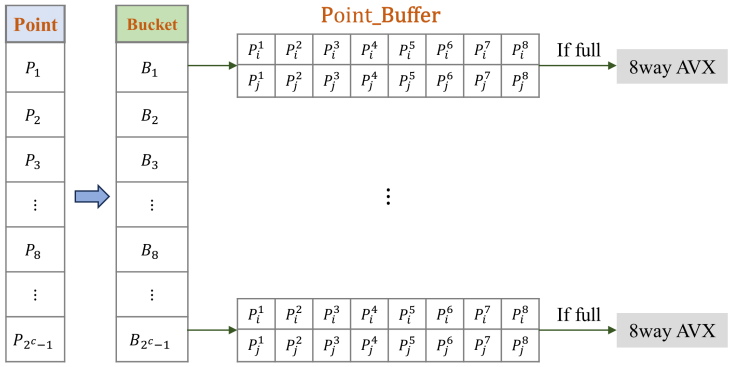

Figure 2: A simple but not optimal method

Open a Global Buffer for all Buckets. To reduce memory overhead, we can consider allocating a single buffer for all buckets instead of creating a separate buffer for each one. This approach significantly decreases the overall memory usage. Grouping χ **PADD** operations into a single batch and triggering the SIMD engine to perform them simultaneously can significantly enhance computational efficiency. However, in this process, there is a high likelihood of encountering writing conflicts—where two or more points are intended to be added to the same bucket in the same round of SIMD-PADD operations. This type of conflict can be mitigated using a deferred handling approach—if a writing conflict arises, the extra conflicting points can be placed at the end of the buffer and processed in the next round. However, this method may cause SIMD acceleration to fail in extreme cases where all scalars are identical. Similarly, efficiency will be significantly reduced when a large portion of scalars overlap. In addition, even if the scalars do not exhibit extreme

behavior, using this method requires the buffer to be dynamic rather than fixed-length. It would need to be structured as a dynamic array, which adjusts continuously based on the actual situation. At certain points, the array could grow significantly in length, leading to potential inefficiencies.

Not Using Double-Point Buffer $\mathsf{buf}^{(1)}_{\alpha\beta}$. While these conflicts can likely be resolved using the mechanism we proposed later, this approach still presents some notable shortcomings. To address writing conflicts, we propose a new solution where conflicting points are first added in pairs, and the resulting points are deferred to the next round for further processing. This method ensures that the buffer length will not exceed 2χ, and even in cases where all scalars are identical, it guarantees that SIMD acceleration can continue to function efficiently. But for a single-level buffer, adding two conflicting points and adding points back to the bucket both require invoking the same SIMD point addition operation. In this case, not all point additions can be handled by a $\mathbf{PADD}^{+}$ operation. To ensure the correctness of the computations, we must use $\mathbf{PADD}^{\#}$ requiring more multiplications globally, which increases the overhead of the **PADD** unit.

All in all, after addressing all the aforementioned limitations, we propose a novel scheduling mechanism that accelerates MSM by utilizing multi-threading and SIMD across three levels: Task-level, Loop-level, and Data-level. At the Data-level, we introduce a two-level buffer system specifically designed for the SIMD engine and address the issue of bucket point writing conflicts to handle bucket accumulation efficiently. Our approach fully leverages the parallel resources available on the CPU, with a fixed memory footprint that avoids irregular or fragmented storage patterns. Additionally, it ensures that the system operates efficiently even in cases where all scalars are identical.

3.2 Overview

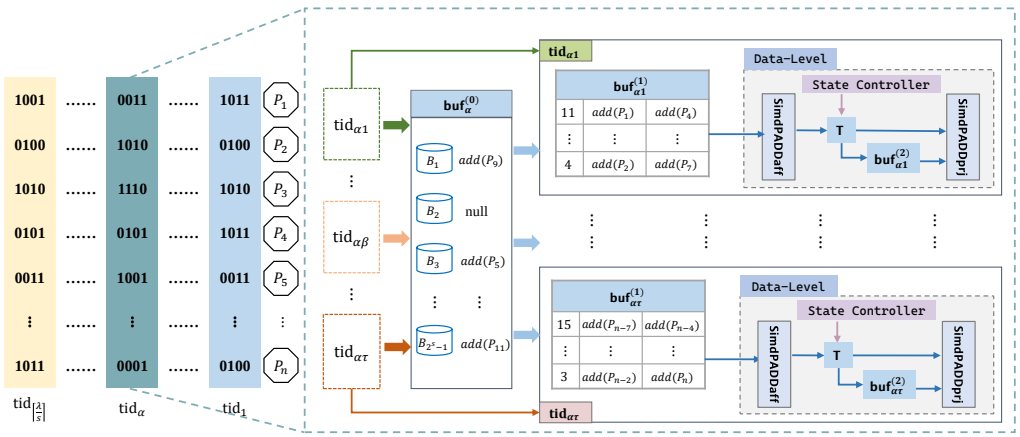

Figure 3: SIMD scheduling mechanism

In Figure 3, the entire process of the SIMD-accelerated architecture is illustrated. First, the computation of MSM is divided into $\lceil \frac{\lambda}{s} \rceil$ Sub-MSM according to the Pippenger algorithm's windows, with each Sub-MSM assigned to a tid_α. Second, each Sub-MSM maintains a bucket buffer to store the value of the bucket points and the addresses of the points waiting for matching to be added to the bucket. To fully use threads on the CPU, we divide n k_iP_i into τ parts as Loop-MSM and open a new thread $\mathsf{tid}_{\alpha\beta}$ for each of them. Then, for each $\mathsf{tid}_{\alpha\beta}$, point buffer $\mathsf{buf}^{(1)}_{\alpha\beta}$ is set to record addresses of pairs of points with the same sub-scalar. Once $\mathsf{buf}^{(1)}_{\alpha\beta}$ reaches its capacity of χ, the SIMD-based data-level

parallelism is triggered. Lastly, a tail task handles the remaining values in all buffers, ensuring correctness during the bucket accumulation stage. This part is also parallelized and processed using SimdPADDmix.

In data-level parallelism, the SimdPADDaff(defined in Section 4) processes χ way **PADD*** to compute the sum of paired points in $\mathsf{buf}^{(1)}_{\alpha\beta}$ and store the result into buffer T. These results are elliptic curve points to be added back to the corresponding bucket in parallel by the SimdPADDprj but writing conflicts will occur during this process due to different points mapped to one bucket. To solve this problem, we set a control mechanism for handling conflict points. Five states for T are designed and the state diagram is shown in Figure 4. The **State-2** is a conflict state, it will jump to other states until the conflict is resolved.

We present our process of scheduling mechanism in Algorithm 1. All SIMD-ECEngine mentioned above will be introduced in Section 4. It is worth noting that our increased memory overhead compared to the original Pippenger algorithm is very small. The $\mathsf{buf}^{(1)}_{\alpha\beta}$ and T are constant length and no more than χ and 2χ respectively. In addition to the necessary bucket points and result points stored in T, other elliptic curve points are just stored as addresses.

3.3 Main Bucket Accumulation Mechanism

Here, we provide a detailed description of the designed bucket accumulation architecture to demonstrate how to leverage the SIMD engine's parallelism advantages.

Task-level Parallelism for Sub-MSM Processing. For each sub-MSM task $G_\alpha = \sum_{i=1}^{n} k_{i\alpha}P_i$ $(\alpha \in [1, \lceil \frac{\lambda}{s} \rceil])$, it is straightforward to assign a thread tid_α to process it. In total, $\lceil \frac{\lambda}{s} \rceil$ threads are needed to compute all sub-MSM $\{G_\alpha\}_{\alpha\in[1,\lceil \frac{\lambda}{s} \rceil]}$ simultaneously. In each thread tid_α, it setups up with a bucket buffer $\mathsf{buf}^{(0)}_\alpha$ with empty tuples $\{\ell, B_\ell := \mathcal{O}, \mathsf{addrPA}_\ell := null\}_{\ell\in[1,2^s-1]}$ indexed by the scalar ℓ. In the follow-up, B_ℓ stores the temp result of bucket point accumulation, and addrPA_ℓ stores the address of an initial point.

Loop-level Parallelism for Initial Point Caching. In each thread tid_α, individually adding a single point P_i into its bucket $B_{k_{i\alpha}}$ is unsuitable for SIMD arithmetic engines. Before performing a SIMD-based point addition, gathering and caching multiple points is more efficient. Thus, we open τ sub-thread $\mathsf{tid}_{\alpha\beta}$ $(\beta \in [1, \tau])$ to independently caching points $\{P_{(\beta-1)\lceil \frac{n}{\tau} \rceil+1}, \cdots, P_{\beta\lceil \frac{n}{\tau} \rceil}\}$ in the thread tid_α. For each sub-thread $\mathsf{tid}_{\alpha\beta}$, it setups up with a point buffer $\mathsf{buf}^{(1)}_{\alpha\beta}$ with the counter $\mathsf{cnt}^{(1)}_{\alpha\beta} := 0$, and processes the scalar $k_{i\alpha}$ and the point P_i $(i \in [(\beta-1)\lceil \frac{n}{\tau} \rceil + 1, \beta\lceil \frac{n}{\tau} \rceil])$ as follows:

- *Case 1:* If $\mathsf{addrPA}_{k_{i\alpha}}$ is null, store the address of P_i into $\mathsf{addrPA}_{k_{i\alpha}} := \mathsf{addrOf}(P_i)$.
- *Case 2:* If $\mathsf{addrPA}_{k_{i\alpha}}$ is non-null, store a new tuple $(\ell := k_{i\alpha}, \mathsf{addrPB} := \mathsf{addrPA}_{k_{i\alpha}}, \mathsf{addrPC} := \mathsf{addrOf}(P_i))$ into the buffer $\mathsf{buf}^{(1)}_{\alpha\beta}$ and set $\mathsf{cnt}^{(1)}_{\alpha\beta} := \mathsf{cnt}^{(1)}_{\alpha\beta} + 1$ and $\mathsf{addrPA}_{k_{i\alpha}} := null$. If $\mathsf{cnt}^{(1)}_{\alpha\beta}$ is larger than the parallelism number of SIMD point arithmetic, the data-level parallelism mechanism for bucket accumulation is triggered.

Since the bucket buffer $\mathsf{buf}^{(0)}_\alpha$ is shared by different sub-threads $\{\mathsf{tid}_{\alpha\beta}\}_{(\beta\in[1,\tau])}$, the thread locking mechanism needs to be established on each tuple to prevent conflicts.

Data-level Parallelism for Bucket Accumulation. Let χ denote the parallelism number of SIMD point arithmetic engines. When this engine is triggered, it requires χ tuples $\{\ell_j, \mathsf{addrPB}_j, \mathsf{addrPC}_j\}_{j\in[\chi]}$ as inputs. The SIMD engine is designed to add two points with the same scalar and then accumulate the result into bucket points through a dynamic controller mechanism. To avoid writing conflicts at bucket points B_ℓ, we design a point buffer $\mathsf{T} := \{\ell_j, T_j^{(\alpha\beta)}\}_{j\in[2\chi]}$ to cache the output of processing $\mathsf{buf}^{(1)}_{\alpha\beta}$. This parallel mechanism works in the following way:

Algorithm 1 SIMD-Accelerated MSM Computation Mechanism

Input: The MSM size n, the λ-bit scalars $\tilde{\mathbf{k}} := \{k_i\}_{i\in[n]}$, the points $\tilde{\mathbf{P}} := \{P_i\}_{i\in[n]}$, the windows size s and the parallelism number χ
Output: The MSM result $Q = \sum_{i=1}^{n} k_i P_i$

1: #pragma omp parallel
2: **for** $\alpha = 1$ to $\lceil \frac{\lambda}{s} \rceil$ **do**
3: $G_\alpha \leftarrow \mathsf{SubMSM}(\tilde{\mathbf{k}}, \tilde{\mathbf{P}}, n, \lambda, \alpha)$ ▷ Open new thread tid_α
4: **end for**
5: **for** $Q := G_{\lceil \frac{\lambda}{s} \rceil}, \alpha = \left\lceil \frac{\lambda}{s} \right\rceil - 1$ to 1 **do**
6: $Q \leftarrow \mathsf{SeqPDBL}(Q, s)$ ▷ Sequential s-times point double
7: $Q \leftarrow Q + G_\alpha$
8: **end for**
9: **return** Q

10: **procedure** $\mathsf{SubMSM}(\tilde{\mathbf{k}}, \tilde{\mathbf{P}}, n, \lambda, \alpha)$
11: $\mathsf{buf}_\alpha^{(0)} \leftarrow \{\ell, B_\ell := \mathcal{O}, \mathsf{addrPA}_\ell := null\}_{\ell\in[1,2^s-1]}$
12: #pragma omp parallel
13: **for** $\beta = 1$ to τ **do**
14: $\beta_l := (\beta-1)\lceil \frac{n}{\tau} \rceil + 1$, $\beta_r := \beta\lceil \frac{n}{\tau} \rceil$
15: $\mathsf{LoopMSM}(\tilde{\mathbf{k}}, \tilde{\mathbf{P}}, \mathsf{buf}_\alpha^{(0)}, \alpha, \beta_l, \beta_r)$ ▷ Open new thread $\mathsf{tid}_{\alpha\beta}$
16: **end for**
17: $\{B_\ell\}_{\ell\in[2^s-1]} \leftarrow \mathsf{TailTask}(\mathsf{buf}_\alpha^{(0)})$
18: $G_\alpha \leftarrow \mathsf{BucketAggregation}(\{B_\ell\}_{\ell\in[2^s-1]})$
19: **return** G_α
20: **end procedure**

21: **procedure** $\mathsf{LoopMSM}(\tilde{\mathbf{k}}, \tilde{\mathbf{P}}, \mathsf{buf}_\alpha^{(0)}, \alpha, \beta_l, \beta_r)$
22: $\mathsf{buf}_{\alpha\beta}^{(1)} \leftarrow \emptyset, \mathsf{T}_{\alpha\beta} \leftarrow \emptyset, \zeta := 1$
23: **for** $i = \beta_l$ to β_r **do**
24: $k_{i\alpha} \leftarrow \mathsf{BitsExtract}(k_i, \alpha, s)$ ▷ $k_i = \sum_{i=1}^{\lceil \frac{\lambda}{s} \rceil} 2^{(\alpha-1)s} k_{i\alpha}$
25: **if** $\mathsf{addrPA}_{k_{i\alpha}} == null$ **then**
26: $\mathsf{addrPA}_{k_{i\alpha}} := \mathsf{addrOf}(P_i)$
27: **else**
28: $\mathsf{buf}_{\alpha\beta}^{(1)}[\zeta] \leftarrow (\ell_\zeta := k_{i\alpha}, \mathsf{addrPB}_\zeta := \mathsf{addrPA}_{k_{i\alpha}}, \mathsf{addrPC}_\zeta := \mathsf{addrOf}(P_i))$
29: $\mathsf{addrPA}_{k_{i\alpha}} := null$, $\zeta := \zeta + 1$
30: **end if**
31: **if** $\zeta > \chi$ **then**
32: $\mathsf{SimdPADD}(\mathsf{buf}_\alpha^{(0)}, \mathsf{buf}_{\alpha\beta}^{(1)}, \mathsf{T}_{\alpha\beta})$
33: $\zeta := 1$
34: **end if**
35: **end for**
36: **end procedure**

37: **procedure** $\mathsf{SimdPADD}(\mathsf{buf}_\alpha^{(0)}, \mathsf{buf}_{\alpha\beta}^{(1)}, \mathsf{T}_{\alpha\beta})$
38: $\{\ell_j, \mathsf{PB}_j, \mathsf{PC}_j\}_{j\in[\chi]} \leftarrow \mathsf{buf}_{\alpha\beta}^{(1)}$
39: $\{T_j\}_{j\in[\chi]} \leftarrow \mathsf{SimdPADDaff}(\{\mathsf{PB}_j, \mathsf{PC}_j\}_{j\in[\chi]})$ ▷ $T_j = \mathsf{PB}_j + \mathsf{PC}_j$
40: Push $\{\ell_j, T_j\}_{j\in[\chi]}$ into $\mathsf{T}_{\alpha\beta}$
41: **if** $\mathsf{len}(\mathsf{T}_{\alpha\beta}) == \chi$ **then**
42: **if** $\exists a, b \in [\chi]$ s.t. $\ell_a == \ell_b$ **then return** ▷ **State-2**
43: **else**
44: $\{B_{\ell_j}\} \leftarrow \mathsf{SimdPADDprj}(\{B_{\ell_j}, T_j\}_{j\in[\chi]})$ and pop the first χ tuples in $\mathsf{T}_{\alpha\beta}$ ▷ **State-1**
45: **end if**
46: **else** ▷ **State-3** or **State 4**
47: $S \leftarrow \emptyset$, $t = 1$
48: **for** $a = 1$ to 2χ **do**
49: **if** $\exists b > a$ & $b \notin S$ s.t. $\ell_a == \ell_b$ **then**
50: $\mathsf{buf}_{\alpha\beta}^{(2)}[t] := (T_a, T_b, T_{\chi+t})$, $S = S \cup \{a, b\}$ ▷ Refer to $T_{\chi+t} = T_a + T_b$
51: **else**
52: $\mathsf{buf}_{\alpha\beta}^{(2)}[t] := (B_{\ell_a}, T_a, B_{\ell_a})$, $S = S \cup \{a\}$ ▷ Refer to $B_{\ell_a} = B_{\ell_a} + T_a$
53: **end if**
54: $t = t + 1$
55: **if** $t > \chi$ **then**
56: $\mathsf{SimdPADDprj}(\mathsf{buf}_{\alpha\beta}^{(2)})$ and update $\mathsf{buf}_{\alpha\beta}^{(1)}$ and $\mathsf{T}_{\alpha\beta}$ ▷ **State-5**
57: Pop the first χ tuples in $\mathsf{T}_{\alpha\beta}$ and goto Line 41
58: **end if**
59: **end for**
60: **end if**
61: **end procedure**

(1) For χ tuples $\{\ell_j, \mathsf{addrPB}_j, \mathsf{addrPC}_j\}_{j\in[\chi]}$ in $\mathsf{buf}^{(1)}_{\alpha\beta}$, it use $\mathsf{SimdPADDaff}$ to compute $\mathsf{PD}_j := \mathsf{PB}_j + \mathsf{PC}_j\}$ for all $j \in [\chi]$ in parallel [3] and store the results into $\{T^{(\alpha\beta)}_\gamma\}_{\gamma\in[2\chi]}$.

(2) When $\mathsf{SimdPADDaff}$ pushes new results into the buffer T, it triggers the following state control mechanism:

- **State-0:** All cache points are empty. In this state, T receives χ points outputted by $\mathsf{SimdPADDaff}$, and changes the state to **State-1** if scalars w.r.t. outputted points are different or **State-2** otherwise.
- **State-1:** The first χ cache points store points with different scalars while others are empty. This state triggers $\mathsf{SimdPADDprj}$ to add χ cache points into corresponding bucket points.
- **State-2:** The first χ cache points store points with duplicate scalars while others are empty. This state continues to wait for the output of $\mathsf{SimdPADDaff}$.
- **State-3** & **State-4:** The first χ cache points are used and the next χ cache points store χ points outputted by $\mathsf{SimdPADDaff}$. Then, this state uses $\mathsf{SimdPADDprj}$ to pairwise add points into the buckets or the buffer T and changes the state to **State-5**.
- **State-5** The first χ cache points are null. Then, this state shifts T by χ points to the left. It be changed to **State-1** or **State-2**.

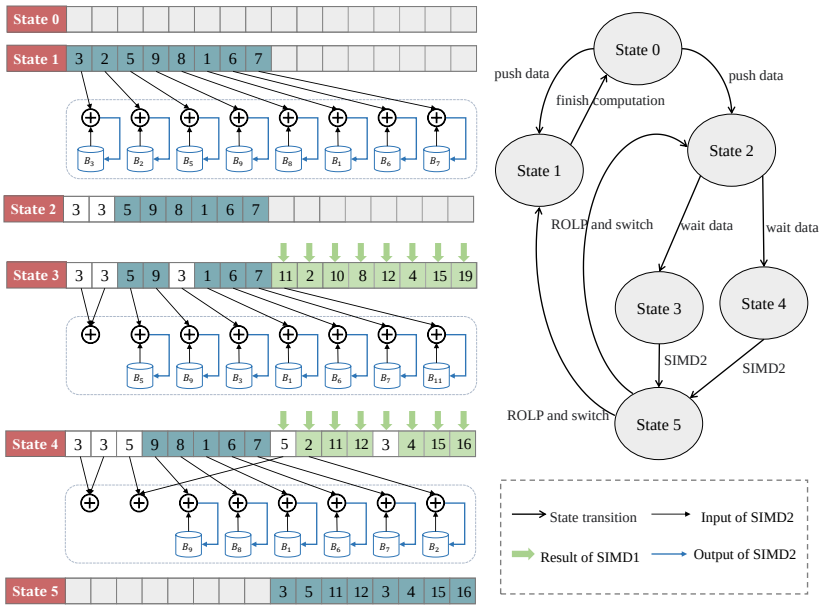

Figure 4: The state diagram of the point buffer T

Combine with Other Algorithmic MSM Optimizations. Luo's method [LFG23] reduces the time overhead of bucket aggregation but introduces the time and space overhead of precomputation at the same time. Using the Twisted Edwards curve form is also a way to speed up **PADD**s in MSM, as mentioned in Section 5.2. We mentioned that this approach can address the issue of the same points in a non-strongly unified arithmetic leading to calculation errors. However, in our implementation, we adopted the HashMap

[3] PB_j and PC_j are two elliptic curve points pointed by address addrPB_j and addrPC_j.

deduplication method. Chen et al. [CPD+24] also proposed using scalar processing and point precomputation to reduce the number of bucket aggregation calculations to a quarter of the original. These algorithmic optimizations are compatible with our proposed parallel framework and can be integrated to achieve superior performance.

3.4 Tail Task for Unaccumulated Cache Points

After processing all n points, some finalization work remains. Both $\mathsf{buf}^{(1)}_{\alpha\beta}$ and $\mathsf{buf}^{(2)}_{\alpha\tau}$ are non-empty, but the number of elements does not meet the threshold χ required to trigger the SIMD engine. We take all the remaining tuples in $\mathsf{buf}^{(1)}_{\alpha\beta}$ as input and compute them once using SimdPADDaff. Since the number of tuples is less than χ, the empty slots are simply neglected, which does not affect obtaining the valid results we want. Upon receiving the final result from SimdPADDaff, new tuples are added to T. We then iteratively invoke SimdPADDprj until T becomes null. Similarly, if the buffer cannot fill χ entries, the remaining slots will idle without affecting the computation.

For $\mathsf{buf}^{(0)}_{\alpha}$, there exist some entries with no-NULL addrPA_ℓ, indicating that there are still some unpaired points that need to be added to the corresponding buckets. Since these points are the initial ones, the addition operation falls under the type **PADD***. These points are added back to different buckets, allowing for parallel processing. Therefore, we introduced an additional SIMD engine, SimdPADDmix, specifically designed to handle χ-way **PADD*** efficiently. This engine focuses on processing remaining points, ensuring that the computation remains fully optimized and capable of handling even the final stages of the bucket accumulation process.

3.5 Writing Conflict

In the data-level parallelism stage, after triggering SimdPADDaff, the ideal scenario is for all results in buffer T to be written back to their respective buckets in χ-way parallelism. However, when multiple points need to be added to the same bucket simultaneously, writing conflicts occur. Simply moving conflict points to the next round of operations can ensure the current round has no conflict but does not guarantee that conflicts will not occur again in the next round. Moreover, this approach requires a significant amount of space to record the points awaiting computation, and it also takes a considerable amount of time to filter conditions that prevent conflicts. Once in the case where a large number or even all points encounter conflicts, this method cannot function properly. To handle this, we introduce a state transition mechanism.

As shown in Figure 4, **State-0** represents the initial state where T is empty. After populating the results of one round of SimdPADDaff, **State-1** is an ideal state where SimdPADDprj can be directly triggered. However, **State-2** represents a conflict state. It waits for new points to be pushed into T, after which it transitions to the next state and handles the conflict points. At this time, the effective length of T is 2χ. We iterate through, setting up a temporary buffer $\mathsf{buf}^{(2)}_{\alpha\beta}$ to both cache points that will be directly added back to the bucket and check if there are any cases where $\ell_a == \ell_b$. If both T_a and T_b are meant to be added to $B_{\ell_a=\ell_b}$, this round will first compute $T_a + T_b$. When the number of tuples in $\mathsf{buf}^{(2)}_{\alpha\beta}$ reaches χ, SimdPADDprj is triggered once. The resulting points are divided into two categories: one is directly written back to B_{ℓ_j}, and the other is returned to buffer T to be processed in the next stage. There is a small trick here: the number of returned result points, when added to the number of remaining tuples in T, still sums to χ. Therefore, the result points can directly fill the empty slots in the last χ tuple.

State-3 and **State-4** are essentially the same, with the only difference being whether any writing conflicts remain after one invocation of SimdPADDprj. If conflicts persist, after shifts the system transitions back to **State-1**; otherwise, it moves to **State-2**.

4 SIMD Elliptic Curve Arithmetic Engines

In this section, we instantiated SIMD elliptic curve arithmetic engines based on *Intel AVX-512IFMA (Integer Fused Multiply-Add)* instructions, which support multiple 52-bit integer multiplication and addition operations through one instruction. Specifically, we implement three engines SimdPADDaff, SimdPADDprj, SimdPADDmix for 8-way $\mathbf{PADD}^*$, $\mathbf{PADD}^{\#}$, and $\mathbf{PADD}^+$ respectively. For compatibility with $\mathbb{G}_1$ and $\mathbb{G}_2$ group operations, we actually need six AVX engines, that is

- SimdPADDaff: $G_1\_8w\_PADD^*$ and $G_2\_8w\_PADD^*$.
- SimdPADDprj: $G_1\_8w\_PADD^{\#}$ and $G_2\_8w\_PADD^{\#}$.
- SimdPADDmix: $G_1\_8w\_PADD^+$ and $G_2\_8w\_PADD^+$.

4.1 SIMD Finite Field Arithmetic

For the underlying finite field operations, we follow Cheng et al.'s work [CFG+21] to build the "(8×1)-Way Prime-Field Arithmetic" with some modified primes [4]. Building upon this foundation, we modified the elliptic curve to BLS curves and extended the operations to the $\mathbb{G}_1$ and $\mathbb{G}_2$ group.

(8×1)-way Data Structure. Note that AVX-512IFMA supports multiply-add instructions of packed unsigned 52-bit integers within each 64-bit lane of two registers. The main data structure in our (8×1)-way parallel computation is the vector set V, which consists of 8 vectors v_i $(0 \leq i < 8)$. Every v_i is composed of eight radix-2^{52} elements, and it can be stored in a 512-bit register for computation during the execution. For eight 381-bit integers $a, b, c, d, e, f, g, h \in \mathbb{F}_q$, the vector set V is defined as:

$$V = \langle a, b, c, d, e, f, g, h \rangle = \left\{ \begin{matrix} [a_0, b_0, c_0, d_0, e_0, f_0, g_0, h_0] \\ [a_1, b_1, c_1, d_1, e_1, f_1, g_1, h_1] \\ \vdots \\ [a_7, b_7, c_7, d_7, e_7, f_7, g_7, h_7] \end{matrix} \right\} = (v_0, v_1, \ldots, v_7) \tag{4}$$

where each vector $v_i = [a_i, b_i, c_i, d_i, e_i, f_i, g_i, h_i]$ and $a = \sum_{i=1}^{8} 2^{52 \cdot i} a_i$ for all $0 \leq a_i \leq 2^{52}$. Thus, we can implement the (8×1)-way fundamental finite field operations such as modular addition, subtraction, double, multiplication, square, and Montgomery domain transformation using AVX-512IFMA.

(8×1)-way PADD arithmetic. Subsequently, we implement the (8×1)-way **PADD** arithmetic over $\mathbb{F}_q$ based on the elliptic curve point addition formulas. Also, We implement $\mathbb{G}_2$ arithmetic and utilize the Karatsuba algorithm [KO62] to accelerate the $\mathbb{F}_{q^2}$ field operations.

4.2 Coordinate System and "Tree-Like" Structure

Accumulating corresponding elliptic curve points into buckets in the Pippenger algorithm requires calculations of $B_l = P_{l0} + P_{l1} + \ldots + P_{lj}$, involving a large number of **PADD**s. $\mathbf{PADD}^+$ is the main part of accumulation when adding elliptic curve points P_i with $Z = 1$ into buckets with $Z \neq 1$ and $11n$ **M** is needed in *projective coordinates* according to the Table 1.

[4] Cheng et al.'s work [CFG+21] is based on the CSIDH protocol on CSIDH-512 curve over a finite field $\mathbb{F}_q$ with a 512-bit prime.

Following the work of Chen et al. [CPD+24], we employed a "Tree-Like" **PADD** structure. It means first summing pairs of points both with $Z = 1$ and then adding these results pairwise whose Z-coordinate are not 1, and finally adding the value into the bucket B_l. In this way, **PADD**s only need $7 \times \lceil \frac{n}{2} \rceil + 12 \times \lceil \frac{n}{2} \rceil = 9.5n\mathbf{M}$. Theoretically, utilizing a "Tree-Like" structure can approximately save 13.6% of computational overhead. Taking eight-point additions as an example, under *projective coordinates*, adding eight points into the bucket B_l one by one requires $11 \times 8 = 88\mathbf{M}$. However, applying the "Tree-Like" structure to point addition as illustrated in Figure 5 only requires $7 \times 4 + 12 \times 4 = 76\mathbf{M}$.

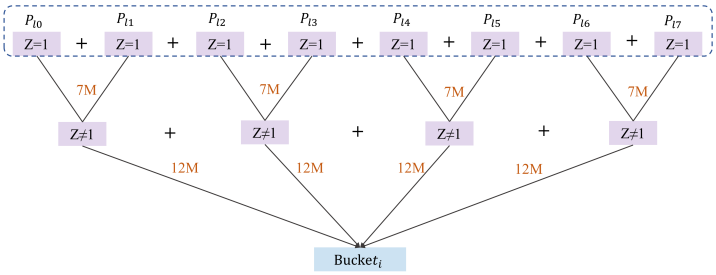

Figure 5: "Tree-Like" structure

This structure echoes perfectly with the previously discussed two distinct engines, SimdPADDaffand SimdPADDprj, which hierarchically compute **PADD** in the bucket accumulation phase, thereby our SIMD-accelerated architecture also demonstrates performance improvements in terms of coordinate systems.

4.3 Performance for AVX Engines

In the single CPU environment [5], our MSM implementation leverages the RELIC cryptographic library [AGM+], a versatile cryptographic meta-toolkit written in C, which provides comprehensive arithmetic operations over finite fields and point operations on elliptic curves. Note that our architecture and methods are broadly applicable and can be integrated with any cryptographic library and SIMD-based operations. For instance, we selected the pairing-friendly BLS12-381 curve [Sea17], widely used in efficient zk-SNARKs constructions.

Table 2: Benchmark of IFMA **PADD**s (in microseconds)

	PADD	**RELIC1** (easy)	**RELIC2** (asm)	**IFMA**	**Speedup1**	**Speedup2**
$\mathbb{G}_1$	**PADD***	7.4465	2.9055	1.0733	6.94×	2.71×
	PADD$^+$	9.7062	3.6063	1.3135	7.39×	2.75×
	PADD$^\#$	10.4719	4.0349	1.4709	7.12×	2.74×
$\mathbb{G}_2$	**PADD***	18.8610	8.0341	2.9279	6.44×	2.74×
	PADD$^+$	22.3596	9.7903	3.6978	6.05×	2.65×
	PADD$^\#$	24.8143	10.6880	4.3326	5.73×	2.47×

Benchmark of AVX Engines. We choose two implementation modes in the RELIC library for comparion, the `easy` mode built on standard C language and the `asm` mode built on the `-DARITH=x64-asm-6l` assembly-optimized arithmetic. Table 2 displays benchmark

[5] An AMD Ryzen 9 7950X3D 16-Core Processor that supports AVX-512F and AVX-512IFMA instructions.

results for eight **PADD**s, showing the time consumed in $\mathbb{G}_1$ and $\mathbb{G}_2$ respectively. Our AVX engines are about 2.7× faster than the `asm` mode and almost 7.0× faster than the `easy` mode. SIMD instructions can speed up parallel processing, but the RELIC's specific optimizations for certain elliptic curves reduce the speedup ratio in `asm` mode.

The Scalability for CPUs without IFMA Compatibility. If a user's machine lacks support for AVX-512IFMA and even AVX-512 instruction set, our implementation can surely be adapted to AVX2. AVX2 is more widely supported compared to AVX-512, with almost all CPUs released since Haswell (2013) providing AVX2 support. Transitioning to AVX2 requires additional engineering efforts: in the finite field operation layer and point addition operation layer, the 8×1 way parallel operation can be modified to a 4×1 way parallel operation, adjusting the level of parallelism from 8 to 4 accordingly. The upper-layer scheduling mechanism and other optimization methods proposed in this article still remain reusable.

In the longer term, our approach is not limited to the AVX instruction set; we believe that it can be extended to any suitable SIMD instructions.

5 Linking to zkSNARK and Applications

In this section, we aim to apply the novel AVX-accelerated architecture for MSM designed above to zkSNARKs and even achieve specific practical applications. To address the issues encountered in practical applications, we propose the following optimizations: (1) MSM support of group $\mathbb{G}_2$, (2) points deduplicated by HashMap or transformed to Twisted Edwards form, (3) "Three-Stage" memory optimization, and (4) dynamic windows for different scales of points.

5.1 AVX-ZK and Workloads

With the rise of zero-knowledge proof theory, numerous open-source library [SCI17, ZKC22, ZoK23, BPH+23, ac22] implementations have emerged, bridging the gap between the latest theoretical advances in zero-knowledge proof technology and their practical engineering implementation.

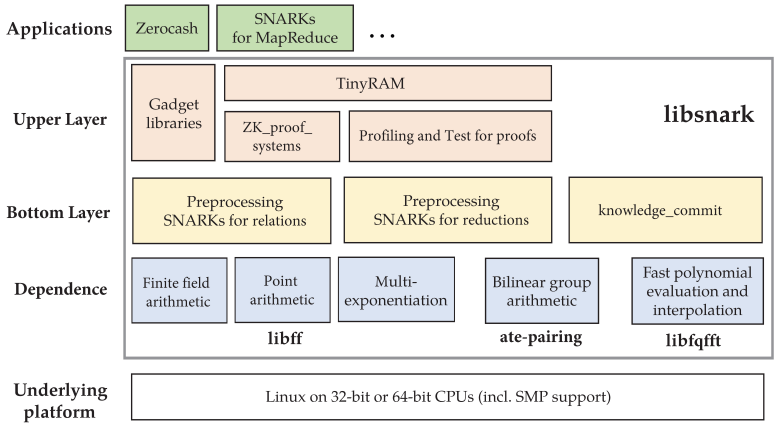

Figure 6: Libsnark's schematic diagram[6]

[6]This figure is adapted from the "Overview of the libsnark stack" in the doctoral dissertation of Madars Virza, the primary contributor to the libsnark codebase at MIT.

The `libsnark` library [SCI17] is a C++ code library used to develop applications that use zk-SNARKs, developed and maintained by the SCIPR Lab project and contributors. It has played an important role in facilitating and advancing the widespread adoption of zk-SNARKs technology. In Figure 6, we provide a schematic diagram of its structure. The dependence relies on the libff, atepairing, and libfqfft libraries for computation. At an upper layer, many kinds of general-purpose proof systems are supported in the library and we utilize `r1cs_gg_ppzksnark` for our implementation, commonly known as the famous Groth16 protocol.

Libsnark's finite field arithmetic, point arithmetic, and multi-exponentiation are all provided by the `libff` library, which is also developed by SCIPR Lab and its contributors. The core of the multi-exponentiation component is the computation of MSM, with its interface located in the `scalar_multiplication` module. Given that many zk-SNARKs are based on pairings, which require MSM calculations not only over the finite field $\mathbb{G}_1$ but also over $\mathbb{G}_2$, libsnark introduces the concept of *pair*. It is designed within the `knowledge_commit` component and is defined as (g, h) where $g \in \mathbb{G}_1$ and $h \in \mathbb{G}_2$.

To better evaluate performance in real-world applications, we assess the overall performance of AVX-ZK using multiple real-world workloads. The `jsnark` library [Ahm21], a Java library for constructing circuits for preprocessing zk-SNARKs, uses libsnark as its backend. It provides some circuit examples for cryptographic primitives such as hashes, block ciphers, key exchange, public key encryption, and signatures. There is an executable interface to run the libsnark's algorithms on the circuit, which allows us to run the proof systems `r1cs_gg_ppzksnark`. This comprehensive setup allows for an extensive evaluation of zero-knowledge proof instances using `jsnark`.

5.2 Duplicate Points into the Same Bucket

Current **PADD*** calculation formulas do not have a strongly unified [ST06] arithmetic, which means that the formula of **PADD** cannot be used to calculate **PDBL**. So, the error would occur if only two same points are PADDed in L1-AVX. In the implementation of libsnark, it cannot be guaranteed that there are no identical elliptic curve points in the running part of the MSM in the proof. If during the running process these two points happen to be placed in the same bucket, we cannot obtain the correct running result. Both of the following methods are viable for the problem. However, in our specific implementation, we choose the first one for convenience.

Deduplicated by HashMap. To address this problem, we choose to use HashMap [Goe06] to avoid duplicate points appearing. HashMap is a kind of data structure used to store key-value pairs, in which each key is unique and corresponds to exactly one value, facilitating rapid data retrieval by mapping keys to values. By using HashMap, we can achieve consolidation by aggregating the scalar values of identical points. This way not only allows deduplication with little time overhead but also helps reduce the final scale of the MSM calculation to some extent.

But not all cases require deduplication. In some proofs, the MSM calculation doesn't encounter the situation where the same point is put into the same bucket. You can determine whether to enable the HashMap deduplication mechanism based on the results of running once.

Using Twisted Edwards Form. BLS curve families are widely used in zkSNARKs due to their pairing-friendly properties. Previous works [Xav22, ABC$^+$22, RDQY24, BH23] have demonstrated the feasibility of converting BLS curves into other curve forms and coordinate systems. Using these methods, the computational overhead of the elliptic curve point operations can be reduced, thereby accelerating MSM. Costs of arithmetic in Twisted Edwards curves with $a = -1$ in extended coordinates are as Table 1 shows. It is a good choice to get more efficient **PADD** operations by transformations onto a Twisted Edwards

curve with $a = -1$. The cost of point conversion can be incorporated into the **Setup** phase of the proof, not affecting the time required for proof generation. In addition, the most important reason for using the Twisted Edwards form is its strongly unified point addition formula, which can fundamentally address the issue of point addition when two identical points are placed in the same bucket.

5.3 "Three-Stage" Memory Optimization

The original Pippenger algorithm splits the large scalar into smaller scalars and processes the data column by column. Each column represents a subtask, and during the overall processing, P_i must be accessed repeatedly. As a result, the points P_1, P_2, ..., P_n can only be out of use after the final column is processed.This also means that all points must be stored in memory until the computation of the last column is completed, and only then can this memory be released. When the number of points n is small, the issue of memory usage may be negligible. However, when n is large, it becomes crucial to consider reducing memory overhead. Taking the BLS12-381 curve as an example, with a base field of 381 bits[Sea17], storing these points would require at least 2 GB of data space, a quantity obtained through computation. For some personal computers, a 2GB memory footprint is unacceptable.

Therefore, inspired by the hash function's implementation[The20, MIR13, Rel22], we propose a "three-stage" MSM memory optimization method. We have improved it to the idea of "processing line by line, releasing point by point". For each P_i, after put into $\lceil \frac{\lambda}{s} \rceil$ sets of buckets, this point will not be used again in the processing of $k_{(i+1)}$, we can release it after P_i is processed, achieving "use, store, and release immediately". As a result, the memory space allocated to the points is greatly reduced.

To achieve the "processing line by line, releasing point by point" process described above, we use a "three-stage" MSM implementation. That is, the original MSM function is decomposed into three stages: ***start***, ***update***, and ***finish***. We set P^τ as a container for storing points whose size τ is much smaller than n. i) In the ***start*** stage, the $\lceil \frac{\lambda}{s} \rceil$ sets of buckets are initialized. ii) In the ***update*** stage, it carries out points distribution into buckets points. And values in P^τ are replaced with new points sequential selected from $\{P_1, P_2, \ldots P_n\}$ in every update. iii) In the ***finish*** stage, buckets aggregations are done on all buckets and MSM result computation is processed to calculate the total result Q.

5.4 Setting up Dynamic Windows

In the section 2.2, we have discussed and calculated the total cost required for the Pippenger algorithm. Naturally, we define a function $F(s) = \lceil \frac{\lambda}{s} \rceil (n + 2^s - 4)$.

For different scales of computation n, the choice of window size s should also vary. By calculating the minimum points of the function $F(s)$ on different scales, we can obtain the optimal s. However, the optimal values observed during the actual execution may differ from the calculated values. We have recorded all obtained data in the Table 3.

Similarly, the optimal window size for pair-MSM ($\mathbb{G}_1$, $\mathbb{G}_2$) can be calculated and obtained, and results are also shown in the Table 3.

Table 3: Optimal window size under different conditions (for MSM and pair-MSM)

Scale	MSM		pair-MSM		Scale	MSM		pair-MSM	
	Com.	**Prac.**	**Com.**	**Prac.**		**Com.**	**Prac.**	**Com.**	**Prac.**
2^{15}	12	9	12	9	2^{20}	17	12	17	12
2^{16}	13	10	13	10	2^{21}	17	13	17	16
2^{17}	13	10	13	10	2^{22}	17	15	17	16
2^{18}	15	10	15	10	2^{23}	17	15	17	16
2^{19}	15	11	15	11	2^{24}	20	15	20	16

6 Evaluation

First, we test the performance of the AVX-MSM and compare it to the MSM calculated using the Pippenger algorithm. Second, we compare the performance of the improved AVX-ZK with that of libsnark across different data scales. Finally, we evaluate the performance of AVX-ZK under various real-world workloads. To better demonstrate the impact of SIMD acceleration in our implementation, we provide the performance results of AVX-MSM and AVX-ZK under two scenarios: without multi-threading and with multi-threading enabled(Multi core). All implementations follow the architecture proposed in Section 3.3. However, due to the limited number of CPU threads in the experimental environment, only one thread was used for the Loop-Level parallelism implementation. `The source code of this paper is available at` https://github.com/JR-account/SimdMSM.

6.1 Overall AVX-MSM Performance

Benchmark of AVX-MSM. We evaluated our AVX-MSM implementation under RELIC's `x64-asm` optimized conditions. The Table 4 below presents the speedup ratios achieved for the $\mathbb{G}_1$ group. The benchmark results for AVX-MSM in BLS12-381 demonstrate significant performance improvements compared to the Pippenger algorithm, the baseline implemented by RELIC. This baseline controls variables and can better demonstrate the acceleration effect of our SIMD framework. In no multi-threading situations, speedups range from 2.17× to 2.45×, and in multi-threading enable situations, they range from 22.21× to 27.86×. These results underscore the effectiveness of SIMD optimizations for parallelism.

We implement AVX-MSM of pair($\mathbb{G}_1$, $\mathbb{G}_2$) instead of point in $\mathbb{G}_2$ group. This is

Table 4: Benchmark of AVX-MSM in BLS12-381 $\mathbb{G}_1$ (in seconds)

Size	RELIC				[LFG23]		
	Pip.	AVX-MSM	Pippenger (Multi cores)	AVX-MSM (Multi cores)	Constr.	AVX-MSM	AVX-MSM (Multi cores)
2^{15}	0.39	0.16 (2.44×)	0.040 (9.25×)	0.014 (27.86×)	0.22	0.16 (1.38×)	0.040 (15.71×)
2^{16}	0.74	0.31 (2.39×)	0.076 (9.74×)	0.027 (27.40×)	0.43	0.31 (1.39×)	0.076 (15.93×)
2^{17}	1.47	0.62 (2.37×)	0.14 (10.50×)	0.053 (27.74×)	0.82	0.62 (1.32×)	0.053 (15.47×)
2^{18}	2.69	1.24 (2.17×)	0.28 (9.61×)	0.10 (26.9×)	1.49	1.24 (1.20×)	0.10 (14.90×)
2^{19}	5.30	2.36 (2.26×)	0.55 (9.64×)	0.21 (25.24×)	2.83	2.36 (1.20×)	0.21 (13.48×)
2^{20}	10.13	4.50 (2.25×)	1.13 (8.96×)	0.43 (23.56×)	5.63	4.50 (1.25×)	0.43 (13.09×)
2^{21}	20.01	8.62 (2.32×)	2.28 (8.78×)	0.84 (23.82×)	10.7	8.62 (1.24×)	0.84 (12.74×)
2^{22}	39.85	16.55 (2.41×)	4.43 (9.00×)	1.63 (24.45×)	-	-	-
2^{23}	80.95	30.03 (2.45×)	8.85 (9.15×)	3.31 (24.46×)	-	-	-
2^{24}	146.36	65.41 (2.24×)	18.25 (8.02×)	6.59 (22.21×)	-	-	-

[1] "Pip." is the baseline implemented by RELIC. AVX-MSM is our framework implementation.
[2] "Multi cores" indicates multi-threading implementations, while others are not.
[3] "Constr." is the optimal implementation from [LFG23]. When the size reaches 2^{21}, the precomputation table size of $\mathbb{G}_1$ becomes 6.5 GB, and further scaling is no longer feasible.

Table 5: Benchmark of AVX-MSM in BLS12-381 pair ($\mathbb{G}_1$, $\mathbb{G}_2$) (in seconds)

Size	RELIC				[LFG23]		
	Pip.	AVX-MSM	Pippenger (Multi cores)	AVX-MSM (Multi cores)	Constr.	AVX-MSM	AVX-MSM (Multi cores)
2^{15}	1.31	0.63 (2.08×)	0.12 (10.92×)	0.048 (27.29×)	0.74	0.63 (1.17×)	0.048 (15.42×)
2^{16}	2.51	1.21 (2.07×)	0.23 (10.91×)	0.095 (26.42×)	1.48	1.21 (1.22×)	0.095 (15.58×)
2^{17}	4.69	2.29 (2.05×)	0.46 (10.20×)	0.18 (26.05×)	2.81	2.29 (1.23×)	0.18 (15.61×)
2^{18}	8.60	4.18 (2.06×)	0.93 (9.25×)	0.37 (23.24×)	5.10	4.18 (1.22×)	0.37 (13.78×)
2^{19}	16.65	7.79 (2.14×)	1.82 (9.15×)	0.74 (22.50×)	9.66	7.79 (1.24×)	0.74 (13.05×)
2^{20}	32.89	15.15 (2.17×)	3.62 (9.09×)	1.36 (24.18×)	19.13	15.15 (1.26×)	1.36 (14.06×)
2^{21}	60.35	29.29 (2.06×)	7.41 (8.15×)	2.98 (20.25×)	-	-	-
2^{22}	116.38	55.12 (2.11×)	12.23 (9.51×)	5.35 (21.75×)	-	-	-
2^{23}	232.89	111.35 (2.09×)	23.24 (10.02×)	10.57 (22.03×)	-	-	-
2^{24}	444.10	214.35 (2.07×)	42.96 (10.34×)	19.73 (22.50×)	-	-	-

[1] "Constr." is the optimal implementation from [LFG23]. When the size reaches 2^{20}, the precomputation table size of $\mathbb{G}_2$ becomes 6.76 GB, and further scaling is no longer feasible.

because, for real-world zk-SNARK implementations, efficient computation of pairings is more suitable. The performance improvements achieved by AVX-MSM in BLS12-381 pair over the Pippenger algorithm are between 2.05× and 12.17× in no multi-threading situations. While in multi-threading enable situations, they range from 20.25× to 27.29×.

Furthermore, to showcase the experimental results better, we compare our work with the work [LFG23] which is based on state-of-the-art library [bls]. In the no multi-threading case, our AVX-MSM achieves a speedup of between 1.20× and 1.38× for the $\mathbb{G}_1$ group. Since the work does not currently support thread-level parallelism based on task decomposition, our AVX-MSM demonstrates a speedup of between 12.74× and 15.71× when using multi-threading. Additionally, based on the results for the $\mathbb{G}_2$ group from their work, we can obtain the pair($\mathbb{G}_1$, $\mathbb{G}_2$) values. The comparison results are shown in the Table 5. As [LFG23] relies on precomputation with fixed points, its approach becomes impractical for large-scale computations due to the size of the precomputation tables. For example, for a size of 2^{21}, the precomputation table for $\mathbb{G}_1$ is 6.75 GB, while the table for $\mathbb{G}_2$ is double.

Overall, AVX-MSM significantly enhances the performance of multi-scalar multiplication operations, offering substantial speedups across different input sizes. This makes AVX-MSM a highly efficient and scalable solution for accelerating zk-SNARK computations on modern CPUs, suitable for practical applications requiring high-performance operations.

6.2 AVX-ZK vs Libsnark

We integrated our AVX-MSM implementation into the `libsnark` library, naming it AVX-ZK. Our AVX-ZK implementation primarily targets the `r1cs_gg_ppzksnark` protocol, utilizing libsnark as a baseline for comparison. We conducted tests using profiling files provided by libsnark for benchmarking purposes. The results are shown in the Table 6. The AVX-ZK is about 16× faster than libsnark. When the computational scale reaches more than 2^{22}, libsnark automatically terminates the current process due to excessive

runtime. In libsnark, the profiles have specific characteristics because the constraints are generated under non-random conditions. There are relatively few special cases where polynomial coefficients are 0 or 1, which results in a higher proportion of MSM computations throughout the process. Our performance advantage is more noticeable under these test cases.

Table 6: Performance Comparison between libsnark and AVX-ZK (in seconds)

Size	libsnark	AVX-ZK	AVX-ZK (Multicores)	Size	libsnark	AVX-ZK	AVX-ZK (Multicores)
2^{15}	6.28	1.29 (4.87×)	0.31 (20.26×)	2^{20}	151.25	36.18 (4.18×)	11.41 (13.25×)
2^{16}	12.00	2.48 (4.84×)	0.61 (19.67×)	2^{21}	284.85	71.79 (3.97×)	23.72 (12.01×)
2^{17}	22.09	4.81 (4.59×)	1.25 (17.67×)	2^{22}	560.91	140.45 (3.99×)	48.64 (11.53×)
2^{18}	41.18	9.23 (4.46×)	2.50 (16.47×)	2^{23}	-	274.69	97.88
2^{19}	83.91	17.66 (4.75×)	5.23 (16.04×)	-	-	-	-

6.3 AVX-ZK Workloads Performance

Due to test cases in the `libsnark` library being specialized, they may not fully represent real-world applications. Therefore, we selected six different zero-knowledge proof (ZKP) use cases for testing, and the results are presented in Table 7. Although the vector size appears large, the actual scale of MSM computations and their proportion in the overall proof calculation are relatively small. This results in a reduction in speedup compared to the profiling comparisons with libsnark in the previous subsection. Nevertheless, our speedup ratios remain between 2.98× and 5.00× in this situation.

Table 7: Performance Comparison of Different Workloads (in seconds)

Application	Vector size	libsnark	AVX-ZK	AVX-ZK(Multi cores)
AES	14240	0.60	0.24 (2.50×)	0.12 (5.00×)
SHA-256	25656	0.91	0.36 (2.53×)	0.20 (4.55×)
RSAEnc	93658	3.10	1.40 (2.21×)	0.89 (3.48×)
Merkle-Tree	98902	3.53	1.66 (2.13×)	1.02 (3.46×)
RSASigVer	117666	3.64	1.70 (2.14×)	1.06 (3.43×)
Auction	540878	14.06	7.31 (1.92×)	4.72 (2.98×)

7 Conclusion

In this paper, we proposed a new SIMD-accelerated MSM architecture that can be applied to accelerate the computation of MSM in various scales, thereby enhancing the performance of pairing-based ZKPs on CPUs. First, we designed a three-level parallelism making full use of parallel resources on the CPUs and there are six different SIMD elliptic curve arithmetic engines to adapt it at the bottom. Our method is in a low cache and would not incur significant additional memory overhead. Second, we resolved the issue of concurrent writings to the same bucket in data-level hierarchical parallel computing. We also discussed

some other methods and the reasons why they are not feasible. Third, we implemented AVX-MSM and thereby achieved AVX-ZK at the ZKP level based on the `libsnark` library, applying our AVX-MSM to practical applications.

Compared to the ordinary Pippenger algorithm implementation, our AVX-MSM achieves a speedup of approximately 25× in the asm mode of `relic` library. And our AVX-ZK implementation achieves over 11.53× (up to 20.26×) speedup on standard benchmarks of libsnark. This means that our approach can effectively enhance the performance of zero-knowledge proofs on CPU platforms.

Acknowlegements.

The work was supported by the National Key Research and Development Program of China (No. 2022YFB3102400), the Major Program (JD) of Hubei Province (No. 2023BAA027), the National Natural Science Foundation of China (Nos. 62272350, 62325209, U23A20302) and the Fundamental Research Funds for the Central Universities (Nos. 2042024KF0002, 2042024kf1013).

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. https://eprint.iacr.org/2022/1396.

[ac22] arkworks contributors. `arkworks` zksnark ecosystem, 2022.

[AGM+] D. F. Aranha, C. P. L. Gouvêa, T. Markmann, R. S. Wahby, and K. Liao. RELIC is an Efficient LIbrary for Cryptography. https://github.com/relic-toolkit/relic.

[Ahm21] Ahmed Kosba. jsnark. https://github.com/akosba/jsnark, 2021. Accessed : 2024-05-15.

[AMD] Amd technical information portal. https://docs.amd.com/r/en-US/am004-versal-dsp-engine/Single-Instruction-Multiple-Data-SIMD-Mode.

[ARM] Arm simd instructions. https://developer.arm.com/documentation/dht0002/a/Introducing-NEON/What-is-SIMD-/ARM-SIMD-instructions.

[BBB+18] Benedikt Bünz, Jonathan Bootle, Dan Boneh, Andrew Poelstra, Pieter Wuille, and Greg Maxwell. Bulletproofs: Short proofs for confidential transactions and more. In *2018 IEEE Symposium on Security and Privacy (SP)*, pages 315–334, 2018.

[BDLO12] Daniel J. Bernstein, Jeroen Doumen, Tanja Lange, and Jan-Jaap Oosterwijk. *Faster Batch Forgery Identification*, page 454–473. Jan 2012.

[BH23] Gautam Botrel and Youssef El Housni. Faster montgomery multiplication and multi-scalar-multiplication for SNARKs. *IACR TCHES*, 2023(3):504–521, 2023.

[bls] blst: a BLS12-381 signature library focused on performance and security written in c and assembly. https://github.com/supranational/blst.

[BPH+23] Gautam Botrel, Thomas Piellard, Youssef El Housni, Ivo Kubjas, and Arya Tabaie. Consensys/gnark: v0.9.0, February 2023.

[BS12] Daniel J. Bernstein and Peter Schwabe. NEON crypto. In Emmanuel Prouff and Patrick Schaumont, editors, *CHES 2012*, volume 7428 of *LNCS*, pages 320–339. Springer, Berlin, Heidelberg, September 2012.

[CCC+09] Anna Inn-Tung Chen, Ming-Shing Chen, Tien-Ren Chen, Chen-Mou Cheng, Jintai Ding, Eric Li-Hsiang Kuo, Frost Yu-Shuang Lee, and Bo-Yin Yang. SSE implementation of multivariate PKCs on modern x86 CPUs. In Christophe Clavier and Kris Gaj, editors, *CHES 2009*, volume 5747 of *LNCS*, pages 33–48. Springer, Berlin, Heidelberg, September 2009.

[CFG+21] Hao Cheng, Georgios Fotiadis, Johann Großschädl, Peter Y. A. Ryan, and Peter B. Rønne. Batching CSIDH group actions using AVX-512. *IACR TCHES*, 2021(4):618–649, 2021.

[CFGR22] Hao Cheng, Georgios Fotiadis, Johann Großschädl, and Peter Y. A. Ryan. Highly vectorized SIKE for AVX-512. *IACR TCHES*, 2022(2):41–68, 2022.

[CGT+20] Hao Cheng, Johann Großschädl, Jiaqi Tian, Peter B. Rønne, and Peter Y. A. Ryan. High-throughput elliptic curve cryptography using AVX2 vector instructions. In Orr Dunkelman, Michael J. Jacobson Jr., and Colin O'Flynn, editors, *SAC 2020*, volume 12804 of *LNCS*, pages 698–719. Springer, Cham, October 2020.

[CPD+24] Yutian Chen, Cong Peng, Yu Dai, Min Luo, and Debiao He. Load-balanced parallel implementation on GPUs for multi-scalar multiplication algorithm. *IACR TCHES*, 2024(2):522–544, 2024.

[GK96] Oded Goldreich and Hugo Krawczyk. On the composition of zero-knowledge proof systems. *SIAM Journal on Computing*, 25(1):169–192, 1996.

[GMR89] Shafi Goldwasser, Silvio Micali, and Charles Rackoff. The knowledge complexity of interactive proof systems. *SIAM Journal on Computing*, 18(1):186–208, 1989.

[Goe06] Brian Goetz. *Java concurrency in practice*. Pearson Education, 2006.

[Gro10] Jens Groth. Short pairing-based non-interactive zero-knowledge arguments. In *Advances in Cryptology - ASIACRYPT 2010 - 16th International Conference on the Theory and Application of Cryptology and Information Security*, volume 6477 of *Lecture Notes in Computer Science*, pages 321–340. Springer, 2010.

[Gro16] Jens Groth. On the size of pairing-based non-interactive arguments. In *Proceedings, Part II, of the 35th Annual International Conference on Advances in Cryptology — EUROCRYPT 2016 - Volume 9666*, page 305–326, Berlin, Heidelberg, 2016. Springer-Verlag.

[GWC19] Ariel Gabizon, Zachary J. Williamson, and Oana Ciobotaru. PLONK: Permutations over Lagrange-bases for oecumenical noninteractive arguments of knowledge. Cryptology ePrint Archive, Report 2019/953, 2019.

[Int] Intel® intrinsics guide. https://www.intel.com/content/www/us/en/docs/intrinsics-guide/index.html.

[JZXJ24] Zhuoran Ji, Zhiyuan Zhang, Jiming Xu, and Lei Ju. Accelerating multi-scalar multiplication for efficient zero knowledge proofs with multi-gpu systems. ASPLOS '24, page 57–70, New York, NY, USA, 2024. Association for Computing Machinery.

[KO62] Anatolii Alekseevich Karatsuba and Yu P Ofman. Multiplication of many-digital numbers by automatic computers. In *Doklady Akademii Nauk*, volume 145, pages 293–294. Russian Academy of Sciences, 1962.

[KST22] Abhiram Kothapalli, Srinath Setty, and Ioanna Tzialla. Nova: Recursive zero-knowledge arguments from folding schemes. page 359–388, Berlin, Heidelberg, 2022. Springer-Verlag.

[LFG23] Guiwen Luo, Shihui Fu, and Guang Gong. Speeding up multi-scalar multiplication over fixed points towards efficient zkSNARKs. *IACR TCHES*, 2023(2):358–380, 2023.

[LWY+23] Tao Lu, Chengkun Wei, Ruijing Yu, Chaochao Chen, Wenjing Fang, Lei Wang, Zeke Wang, and Wenzhi Chen. cuZK: Accelerating zero-knowledge proof with A faster parallel multi-scalar multiplication algorithm on GPUs. *IACR TCHES*, 2023(3):194–220, 2023.

[MIR13] MIRACL. SHA256 implementation. https://github.com/miracl/MIRACL/blob/master/source/mrshs256.c, 2013. Accessed : 2024-05-15.

[MXS+23] Weiliang Ma, Qian Xiong, Xuanhua Shi, Xiaosong Ma, Hai Jin, Haozhao Kuang, Mingyu Gao, Ye Zhang, Haichen Shen, and Weifang Hu. Gzkp: A gpu accelerated zero-knowledge proof system. ASPLOS 2023, page 340–353, New York, NY, USA, 2023. Association for Computing Machinery.

[PHGR13] Bryan Parno, Jon Howell, Craig Gentry, and Mariana Raykova. Pinocchio: Nearly practical verifiable computation. In *2013 IEEE Symposium on Security and Privacy*, pages 238–252, 2013.

[Pip76] Nicholas Pippenger. On the evaluation of powers and related problems. In *17th Annual Symposium on Foundations of Computer Science (sfcs 1976)*, Oct 1976.

[RDQY24] Andy Ray, Benjamin Devlin, Fu Yong Quah, and Rahul Yesantharao. Hardcaml msm: A high-performance split cpu-fpga multi-scalar multiplication engine. FPGA '24, page 33–39, New York, NY, USA, 2024. Association for Computing Machinery.

[Rel22] Relic Contributors. SHA256 implementation. https://github.com/relic-toolkit/relic/blob/main/src/md/relic_md_sha256.c, 2022. Accessed : 2024-05-15.

[ris] riscv-p-spec. https://github.com/riscv/riscv-v-spec.

[Riv11] Matthieu Rivain. Fast and regular algorithms for scalar multiplication over elliptic curves. *IACR Cryptol. ePrint Arch.*, 2011:338, 2011.

[RWGM23] M. Rosenberg, J. White, C. Garman, and I. Miers. zk-creds: Flexible anonymous credentials from zksnarks and existing identity infrastructure. In *2023 IEEE Symposium on Security and Privacy (SP)*, pages 790–808, Los Alamitos, CA, USA, may 2023. IEEE Computer Society.

[SCI17] SCIPR Lab project and contributors. The Libsnark Library. https://github.com/scipr-lab/libsnark, 2017. Accessed : 2024-05-15.

[Sea17] Sean Bowe. Bls12-381: New zk-snark elliptic curve construction. https://electriccoin.co/blog/new-snark-curve/, Mar. 2017.

[ST06] Douglas Stebila and Nicolas Thériault. Unified point addition formulæ and side-channel attacks. In *Cryptographic Hardware and Embedded Systems-CHES 2006: 8th International Workshop, Yokohama, Japan, October 10-13, 2006. Proceedings 8*, pages 354–368. Springer, 2006.

[The20] The OpenSSL Project. MD5 implementation. https://github.com/openssl/openssl/blob/master/crypto/md5/md5_one.c, 2020. Accessed : 2024-05-15.

[Xav22] Charles. F. Xavier. Pipemsm: Hardware acceleration for multi-scalar multiplication. Cryptology ePrint Archive, Paper 2022/999, 2022. https://eprint.iacr.org/2022/999.

[zca22] Zcash corp. https://z.cash/, 2022.

[ZDW+22] Haixu Zhao, Dong Ding, Feng Wang, Pengcheng Hua, Ning Wang, Qin Wu, and Zhilei Chai. Hardware acceleration of number theoretic transform for zk-snark. *Engineering Reports*, page e12639, 2022.

[ZHZ+24] Jipeng Zhang, Junhao Huang, Lirui Zhao, Donglong Chen, and Çetin Kaya Koç. ENG25519: Faster TLS 1.3 handshake using optimized X25519 and Ed25519. In Davide Balzarotti and Wenyuan Xu, editors, *USENIX Security 2024*. USENIX Association, August 2024.

[ZKC22] ZKCrypto Corp. Bellman. https://github.com/zkcrypto/bellman, 2022. Accessed : 2024-05-15.

[ZoK23] ZoKrates Contributors. Zokrates. https://github.com/Zokrates/ZoKrates, 2023. Accessed : 2024-05-15.

[zpr] Zprize. https://www.zprize.io/.

[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.