UniMSM: An Efficient and Flexible Hardware Accelerator for Multi-Scalar Multiplication

Kaixuan Wang, Yifan Yanggong, Chenti Baixiao, Xiaoyu Yang and Lei Wang

***Abstract*—Multi-scalar multiplication (MSM) is a central kernel in cryptographic systems, which evaluates large linear combinations of elliptic-curve points. Practical MSMs couple millions of terms with hundreds-of-bit modular arithmetic, while Pippenger's bucket flow introduces irregular memory updates that can severely degrade utilization under deep pipelines.**

In this paper, we present UniMSM, an efficient and flexible hardware accelerator for MSM across practical problem sizes and diverse curve parameters. First, we design a pipelined point adder based on the extended Jacobian coordinate system and employ a time-multiplexed datapath to reduce modular multiplier cost while sustaining high throughput. Second, we introduce a conflict-aware scheduling scheme to address bucket-update conflicts and preserve utilization under irregular accesses. Third, we develop a hardware-friendly variant of the Pippenger algorithm to reduce intermediate storage overhead and serial dependencies in aggregation. Compared with prior FPGA accelerators, UniMSM achieves up to 2.12× improvement in area-time product. Furthermore, UniMSM in ASIC achieves up to a 3.85× improvement in ATP compared to the SOTA accelerator.

***Index Terms*—Multi-scalar multiplication, Elliptic-curve point adder, Hardware acceleration.**

I. Introduction

Elliptic-curve cryptography (ECC) is a foundational building block for modern cryptography due to its strong security and efficiency. A fundamental operation in ECC is scalar multiplication: given a scalar k and a curve point P, compute kP. In applications, many cryptographic constructions require multiple scalar multiplications, where the results are aggregated. This results in multi-scalar multiplication (MSM), which evaluates a large linear combination of curve points:

$$Q = \sum_{i=0}^{N-1} k_i P_i, \tag{1}$$

where k_i are scalars and P_i are elliptic-curve points.

MSM is a fundamental computational kernel in modern cryptographic protocols, such as zero-knowledge proofs (ZKPs) and polynomial commitment schemes (PCS). In ZKP protocols (e.g., Groth16 [1], Plonk [2], and Nova [3]), MSM often dominates the prover runtime, accounting for more than 70% of the total cost [4], [5]. In PCS (e.g., KZG [6] and Dory [7]), committing to a polynomial reduces to an MSM. The computational burden of MSM grows along two independent dimensions: the security requirements determine operand bit-width (e.g., 256-bit or 381-bit); application scaling drives the MSM size to millions of terms or more. For example, the privacy mechanism of the Zerocash protocol involves an MSM on more than a million points to prove membership in a 64-layer Merkle tree [8]. Moreover, these protocols are instantiated over different elliptic curves, whose parameters and coordinate systems lead to different datapaths and underlying arithmetic. Taken together, the combination of wide-word modular arithmetic, large MSM instances, and curve-dependent point operations makes MSM a critical performance bottleneck in practical deployments. This motivates dedicated MSM hardware acceleration that delivers low latency for large problem sizes and wide-word modular arithmetic while remaining practical across diverse curve parameters and deployment scenarios.

Among MSM algorithms, the Pippenger algorithm [9] is widely adopted for large instances. It splits scalars into fixed-size chunks and accumulates points that share the same chunk value into the same running sum (bucket accumulation), reducing the total number of elliptic-curve point additions. Nevertheless, MSM with large N is still dominated by a large number of point operations. Each operation expands into many modular multiplications and additions in the prime field. Moreover, the datapath of point operations depends on the curve parameters and coordinate formulas, and thus a curve-specialized accelerator does not generalize to other curve instances. Meanwhile, the Pippenger algorithm introduces irregular updates: points are mapped to buckets indexed by subscalar values, and the accelerator must repeatedly perform updates to bucket states, which can cause conflicts and dependencies under deep pipelines.

This observation motivates two design principles when realizing a resource-efficient and flexible MSM hardware accelerator. First, although the Pippenger algorithm reduces the operation count, the accelerator still processes a massive number of point operations whose cost varies across curve instances and coordinate formulas. Consequently, the arithmetic datapath must provide high-throughput, curve-parameterizable point arithmetic under wide-word modular arithmetic, typically requiring deep pipelining and careful formula/coordinate choices to balance throughput against resource cost. On the systems side, bucket accumulation exhibits read–update–write semantics with highly skewed access patterns: issuing multiple in-flight updates to the same bucket (or the same memory bank) can create hazards that either violate accumulation semantics or force stalls and arbitration, substantially degrading pipeline utilization. These challenges have motivated two lines of MSM accelerator research: optimizing arithmetic modules including modular multipliers and point adders, and developing conflict-aware scheduling techniques to deal with irregular bucket updates.

Several designs optimize modular arithmetic and point

Kaixuan Wang, Yifan Yanggong, and Lei Wang are with the Shanghai Jiao Tong University, Shanghai, China. Chenti Baixiao and Xiaoyu Yang are with the Chipltech, Shanghai, China.

operations to increase the throughput of point addition with less resource utilization. This line of work aims for curve portability through the parameterizable modular arithmetic module and retargetable curve parameters, enabling MSM across commonly used curves such as BN254 and BLS12-381 (e.g., Falic [10], if-ZKP [11]). Complementary to pipeline designs, MSMAC [12] exposes the MSM execution as configurable primitives to accommodate different execution strategies. Beyond microarchitecture, OPTIMSM [13] improves the Pippenger-based workflow under a fixed on-chip memory budget, and FusionMSM [14] reduces implementation overhead via hybrid design choices. Despite these advances, these approaches often face a portability–efficiency tradeoff: curve-agnostic datapaths reduce resource efficiency, whereas heavily optimized pipelines tend to be formula-specific.

Scheduling schemes address a different bottleneck: the conflicts due to irregular point additions in the Pippenger algorithm. When multiple points are added to the same bucket, stalls quickly decrease utilization and further increase the latency. BSTMSM [15] resolves collisions by merging conflicting updates, PriorMSM [16] mitigates contention through conflict-aware buffering and arbitration, and Gypsophila [17] emphasizes scalable multi-lane organization and bandwidth-oriented data movement. FusionMSM [14] further improves bucket accumulation with contention-aware access strategies. While effective, these techniques are typically optimized around a largely fixed MSM configuration chosen at design time (e.g., window/bucket provisioning and arbitration policy). As the MSM size or pipeline depth changes, utilization can degrade unless hazard handling and buffering are jointly tuned.

Prior MSM accelerators substantially improve performance, but often optimize one bottleneck at a time—either pushing arithmetic throughput or mitigating bucket irregularity. What remains missing is a unified approach that jointly (i) provides a resource-efficient, flexible datapath for point operations, (ii) sustains high utilization under irregular bucket updates with predictable conflict handling, and (iii) reduces algorithm-level data movement and dependency overheads in the Pippenger workflow for deeply pipelined hardware.

In this paper, we present UniMSM, a hardware MSM accelerator that co-designs elliptic-curve arithmetic, bucket update scheduling, and a hardware-friendly Pippenger variant to achieve resource-efficient performance at practical scales (i.e., 2^{20} to 2^{26}). UniMSM further employs hardware-aware parameter optimization to balance latency and on-chip RAM footprint under different device constraints. Our contributions are summarized below:

- **An efficient and flexible pipelined point adder.** We propose a pipelined EC point adder based on the extended Jacobian coordinate system, supporting both point addition and doubling across a broad class of short Weierstrass curves. The proposed five-stage design sustains one point pair per cycle in mixed addition mode used in the bucket accumulation phase, while requiring only 10 modular multipliers, fewer than existing designs. In addition, it supports general addition mode used in the bucket aggregation phase at one point pair every two cycles. To the best of our knowledge, this is the first pipelined EC point adder design built on the extended Jacobian coordinate system.
- **A conflict-aware scheduling mechanism.** We propose a bucket-level arbitration mechanism for the bucket accumulation phase that resolves update conflicts to the same bucket, preventing in-flight hazards and sustaining high utilization under deep pipelines.
- **Hardware-friendly optimizations of the Pippenger algorithm.** We develop a hardware-friendly Pippenger variant that fuses the original *bucket aggregation* and *subtask aggregation* into the unified *general aggregation*, thereby eliminating intermediate materialization and reducing cross-stage data movement. We further introduce *bucket folding* to reduce serial dependencies within each window and improve pipeline utilization during aggregation.
- **An accelerator architecture with hardware-aware parameter optimization.** We design UniMSM as a flexible hardware architecture that decouples arithmetic, control, and memory. We further perform hardware-aware parameter analysis to identify favorable configurations that balance latency and on-chip memory consumption across practical deployment constraints.

TABLE I: Comparison with representative FPGA-based MSM accelerators. The table highlights the efficiency of the point adder design (#MM), DSP usage (#DSP), on-chip memory footprint, and area-time product (ATP). **#MM** denotes the number of modular multipliers in one point adder, and **On-chip Memory** denotes the total on-chip storage capacity. **ATP** is computed using the latency of a 2^{20}-point MSM. Parenthesized factors are normalized to this work.

Work	#MM	#DSP	On-chip memory	ATP (Speedup)
Curve: BN254				
MSMAC [12]	12	6480	28.45 MB	44.20 (1.66×)
UniMSM	**10**	**2693**	**0.44 MB**	**26.67**
Curve: BLS12-381				
Ingonyama [18]	20	9011	4.7 MB	109.63 (2.12×)
UniMSM	**10**	**4690**	**3.78 MB**	**51.54**

a) Implementation and evaluation: We implement UniMSM on an FPGA platform and evaluate it across a wide range of MSM sizes. As summarized in Table I, UniMSM achieves an efficiency trade-off in point-adder complexity, DSP cost, on-chip memory footprint, and area-time product (ATP). In particular, our design uses only 10 modular multipliers in the point adder, while significantly reducing DSP usage and on-chip memory compared with prior FPGA MSM accelerators. As a result, UniMSM achieves the best reported ATP among the compared BN254 implementations, improving ATP by 1.66× over MSMAC [12] under BN254 and by 2.12× over Ingonyama [18] under BLS12-381. To further demonstrate the versatility and scalability of our architecture, we also synthesize UniMSM using the TSMC 12nm library. When synthesized for ASIC, UniMSM demonstrates an improvement

in ATP efficiency. Specifically, it outperforms PipeZK [19] by up to 73.36× and yields a 3.85× improvement over the design in [20].

II. Preliminary

A. Elliptic Curve Arithmetic

Let $\mathbb{F}_q$ be a finite field of a prime order q. An elliptic curve (EC) E defined over $\mathbb{F}_q$ is typically expressed in short Weierstrass form $y^2 = x^3 + ax + b$, where $a, b \in \mathbb{F}_q$ and $4a^3 + 27b^2 \neq 0$. The set of points on the curve, including a distinguished point at infinity $\mathcal{O}$, forms an abelian group denoted by $E(\mathbb{F}_q)$. In cryptographic applications (e.g., digital signatures, key exchange and zero-knowledge proofs), computations are performed in a subgroup $\mathbb{G} \subseteq E(\mathbb{F}_q)$ of large prime order r described as follows.

Two fundamental operations defined in EC arithmetic are point addition (PADD) and point doubling (PDBL). Most EC points are represented in affine coordinates by using two field elements in the base field $\mathbb{F}_q$, such as $P = (x, y)$. For two points in affine coordinates $P_1 = (x_1, y_1)$ and $P_2 = (x_2, y_2)$ where $P_1, P_2 \in \mathbb{G}$ and $P_1 \neq P_2$, the PADD is defined as $P_3 = (x_3, y_3) = P_1 + P_2$ in elliptic curve E. The formula for PADD is as follows:

$$x_3 = \frac{(y_2 - y_1)^2}{(x_2 - x_1)^2} - x_1 - x_2, \tag{2}$$

$$y_3 = \frac{(y_2 - y_1)(x_1 + 2x_2)}{x_2 - x_1} - \frac{(y_2 - y_1)^3}{(x_2 - x_1)^3} - y_1 \tag{3}$$

To compute PDBL $P_3 = 2P_1$, the formula is as follows:

$$x_3 = \frac{(3x_1^2 + a)^2}{(2y_1)^2} - 2x_1, \tag{4}$$

$$y_3 = \frac{3x_1^2 + a}{2y_1}(x_1 - x_3) - y_1 \tag{5}$$

While representing EC points in affine coordinates can aid in reducing storage consumption, formulas for point operations in these coordinates require the field inverse which is a time-consuming operation in the finite field arithmetic. Thus, several coordinate systems and the related formulas [21] have been developed to improve the efficiency of PADD and PDBL. Beside affine coordinates $P = (x, y)$, the efficient coordinate systems for fast computations include Projective coordinates $P = (X, Y, Z)$,where $x = X/Z$, $y = Y/Z$, Jacobian coordinates $P = (X, Y, Z)$, where $x = X/Z^2$, $y = Y/Z^3$, and extended Jacobian coordinates $P = (X, Y, Z, W)$, where $x = X/Z$, $y = Y/W$, $Z^3 = W^2$. In Table II, PADD is categorized into three commonly used cases depending on the input representations. Affine PADD denotes adding two affine points. General PADD denotes adding two points represented in the same coordinate system (e.g., projective, Jacobian, and extended Jacobian). Mixed PADD denotes adding a projective-type point with an affine point.

TABLE II: Field operation costs of point operations under different coordinate systems. "**M**" denotes the modular multiplication. "**S**" denotes the modular square.

Coordinate System	Mixed PADD	Affine PADD	General PADD	PDBL
Projective	**9M + 2S**	**5M + 2S**	**12M + 2S**	**5M + 6S**
Jacobian	**7M + 4S**	**4M + 2S**	**11M + 5S**	**1M + 8S**
Extended Jacobian	**8M + 2S**	**4M + 2S**	**12M + 2S**	**6M + 4S**

B. Multi-Scalar Multiplication and Pippenger Algorithm

Multi-scalar multiplication (MSM) is the computation of a vector inner product between a scalar vector and a point vector. Given N λ-bit scalars $[k_0, k_1, \cdots, k_{N-1}]$ and N points $[P_0, P_1, \cdots, P_{N-1}]$ on the elliptic curve, MSM calculates the point Q as follows:

$$Q = k_0P_0 + k_1P_1 + \cdots + k_{N-1}P_{N-1} = \sum_{i=0}^{N-1} k_iP_i. \tag{6}$$

MSM requires N expensive scalar multiplications. The scalar multiplication k_iP_i denotes repeated application of the elliptic-curve group law, i.e., adding the same point P_i to itself $k_i - 1$ times. In practice, scalar multiplication is implemented using double-and-add style methods by expanding k_i in binary form, and then evaluating the corresponding sequence of group operations.

To reduce the number of the point operations for MSM, the Pippenger algorithm [9] was proposed. This algorithm can be decomposed into three steps: Bucket accumulation, bucket aggregation and subtask aggregation.

a) Bucket accumulation: In this step, the scalar for each point is split as c-bit chunks called windows, where the j-th window of the i-th scalar is denoted as $k_{i,j}$. Each point P_i is assigned to the l-th bucket in j-th bucket accumulation based on the subscalar $k_{i,j}$ for a given window, resulting in the following computation:

$$B_{l,j} = \sum_{\{i|k_{i,j}=l\}} P_i,\ l \in \{1, \cdots, 2^c - 1\}. \tag{7}$$

An optimization in this step is to recode each window value into a signed-digit representation [4]. Specifically, instead of using the unsigned c-bit value $k_{i,j} \in \{0, \ldots, 2^c - 1\}$, this optimization adopts a signed recoding to obtain $k'_{i,j} \in [-2^{c-1}+1, 2^{c-1}]$. With this representation, bucket indices can be defined by $l = |k'_{i,j}|$, while the sign is absorbed by negating the input point $-P_i = (x_i, -y_i)$ from $P_i = (x_i, y_i)$. As a result, the number of non-zero buckets is reduced from $2^c - 1$ to 2^{c-1}, which directly lowers the on-chip RAM footprint.

b) Bucket aggregation: The second step (bucket aggregation) computes the weighted sum of the buckets to obtain the partial result for each window:

$$G_j = \sum_{l=1}^{2^c-1} l \cdot B_{l,j}. \tag{8}$$

c) Subtask aggregation: Finally, all partial results G_j are combined to produce the final output:

$$Q = \sum_{j=0}^{\lambda/c-1} 2^{jc} \cdot G_j. \tag{9}$$

Notably, the MSM in ZKPs differs from its use in other contexts such as pairing-based signature schemes. First, the

size of MSM N is very large, which ranges from 2^{18} to 2^{26}. Second, the bit length of the scalar field λ of MSM in zkSNARKs is also greater. For example, elliptic curves widely used in zkSNARKs feature scalar fields with bit lengths of up to 256 bits, such as BN254, BLS12-377, BLS12-381.

C. Previous MSM Accelerators and Revisiting

Existing MSM accelerators can be categorized into two main lines: (i) efficient designs of arithmetic module, e.g., modular multiplier and point adder; and (ii) conflict-aware memory and scheduling techniques that tackle irregular bucket updates.

a) Efficient designs of arithmetic module: This line of MSM hardware aims to optimize finite-field arithmetic and point operations. The main technique is adopting an efficient coordinate systems and pipelined modular multipliers to design high-throughput point addition/doubling datapaths. Most prior accelerators [14], [22], [23] are optimized for one specific curve, BLS12-377, which benefit from highly efficient point addition formula requiring 7 modular multiplication in a point addition. However, these accelerators depending on specific curve parameters does not generalize to other widely used curves. Furthermore, several FPGA-based MSM designs emphasize flexibility across curve instances by relying on parameterizable field-arithmetic processing elements and efficient curve formulas, thereby retargeting MSM to multiple widely used curves without redesigning the entire datapath. Among these MSM designs, Falic [10] evaluates MSM acceleration across multiple elliptic curves (including BN254 and BLS12-381 among others), suggesting a curve-agnostic microarchitecture enabled by retargetable field arithmetic and curve parameters. Similarly, if-ZKP [11] reports FPGA acceleration for MSM over curves such as the BN254 and BLS12-381 families, leveraging a generic curve form in Jacobian coordinates and optimized modular-arithmetic IP to improve portability across curve instances. Complementary to purely datapath-driven designs, MSMAC [12] proposes an instruction set architecture that exposes MSM execution as schedulable primitives, enabling the accelerator to realize different execution strategies through programmable control rather than a fully hard-wired pipeline. While promising for accommodating algorithmic variation, these approaches must carefully manage the tension between control overhead and peak arithmetic efficiency, especially under irregular memory behaviors. OPTIMSM [13] focuses on improving upon the Pippenger algorithm by proposing a iteration technique to decouple the required buckets from the window size, resulting in fewer EC computations by increasing the window size to 19 bits under a fixed on-chip memory budget. FusionMSM [14] uses a hybrid multiplication method to reduce LUT overhead. Despite these advances, arithmetic-centric designs often face a difficult trade-off between portability and peak efficiency: parameterizable datapaths that retarget across curves may sacrifice resource optimality, while highly optimized pipelines are frequently tuned to specific formulas or microarchitectural assumptions. In particular, achieving a resource-efficient yet high-throughput pipelined point adder—without inflating the number of large-integer multipliers or control overhead—remains underexplored, especially when the same datapath must serve both point addition and doubling patterns required by Pippenger MSM.

b) Irregular memory access and scheduling for bucket accumulation: Beyond arithmetic design, MSM performance is frequently dominated by irregular bucket updates, where multiple points contend for the same bucket or the same memory bank, causing stalls and underutilization. BSTMSM [15] handles conflicts between two bucket updates by detecting the collision, first adding the two input points, and then updating the bucket with the merged result. PriorMSM [16] addresses this bottleneck with conflict-aware on-chip memory organization and scheduling, employing multi-FIFO and multi-bank structures together with priority-based arbitration to mitigate bucket access conflicts and improve sustained throughput. Gypsophila [17] further emphasizes scalable multi-PE organization and bandwidth-optimized data movement to sustain utilization as parallelism increases, highlighting that MSM bottlenecks often migrate from point arithmetic to memory/interconnect when scaling the number of compute lanes. FusionMSM [14] optimizes the memory access for bucket accumulation by introducing a greedy collision resolution technique. Although effective, these designs typically optimize around a fixed algorithmic configuration (e.g., a windowing/bucket structure chosen at design time), leaving limited room for adapting the memory and scheduling policies to changing MSM sizes or bandwidth regimes. Meanwhile, conflict-aware scheduling and memory techniques are frequently optimized around a fixed algorithmic configuration chosen at design time (e.g., window size, bucket provisioning, PE allocation, and arbitration policy). As a result, these accelerators may suffer utilization loss when MSM scales or bandwidth regimes change, or when deeper arithmetic pipelines amplify in-flight hazards. A principled mechanism to co-optimize hazard resolution with bounded buffering and predictable control—so that high utilization can be maintained without substantial resource overhead—remains insufficiently addressed.

III. An Efficient and Flexible Point Adder

In this section, we introduce a mode-dependent pipelined elliptic-curve (EC) point adder, which serves as a core compute module in MSM accelerators. This design is motivated by two considerations. First, to support a broad class of curves commonly used in cryptographic systems (e.g., short Weierstrass curves such as BN254 and BLS12-381), we adopt a unified arithmetic formulation in extended Jacobian coordinates that covers both point addition (PADD) and point doubling (PDBL). This choice avoids relying on curve-specific formulas that exploit special properties of twisted Edwards curves and therefore do not generalize well to more curve families. Second, compared with such specialized formulas, a unified formula typically incurs more field multiplications, increasing the pressure on multiplier resources inside the point adder. To balance generality and efficiency, our adder is mode-dependent: it selects the appropriate point addition mode and

the corresponding operation schedule based on the requested EC operation and the input configuration. This enables a flexible MSM backend across different curves and protocol settings.

A. Point Addition with Extended Jacobian Coordinates

In MSM, the bucket accumulation step consumes the most PADDs. In typical scenarios, the input points are provided in affine coordinates, while buckets are maintained in extend Jacobian coordinates. Therefore, bucket updates are performed via mixed PADD. To compute the mixed PADD, we adopt a formula [21] that requires 10 field multiplications (8 multiplications and 2 squares). Although the underlying formula is standard, directly implementing it yields a long dependency chain that limits the initiation interval of a fully pipelined datapath. To enable a high-throughput point adder, we reorganize the formula into five pipeline stages (Table III) with two goals. First, we aim to remove intra-stage data dependencies so that field operations within a stage can be issued in parallel. Second, we place register boundaries along the critical dependency path to achieve one-point-per-cycle initiation. The five-stage mixed PADD schedule serves as the baseline pipeline organization, while the other operation modes reuse the same arithmetic skeleton with mode-specific operand selection and multiplier scheduling.

Stage	Cost	Computations
1	2**M** + 2**S**	$U = X_2 * Z_1, S = Y_2 * W_1$ $P = U - X_1, R = S - Y_1$
2	2**M**	$PP = P^2, RR = R^2$
3	3**M** + **A** + **S**	$Q = X_1 * PP, PPP = P * PP,$ $Z_3 = Z_1 * PP$ $2Q = Q + Q, D = RR - PPP$
4	2**M** + 2**S**	$K = Y_1 * PPP, W_3 = W_1 * PPP$ $X_3 = D - 2Q, M = Q - X_3$
5	**M** + 1**S**	$Y_3 = M * R - K$

TABLE III: Five-stage processing procedure for mixed PADD. **M**, **A**, and **S** denote modular multiplication, addition, and subtraction, respectively.

Compared with a direct pipeline design, this staging exposes parallelism inside each stage and reduces the cross-stage dependency depth, which directly guides the pipelined field-arithmetic datapath design described next.

B. Point Adder Design

Our point adder shown in Figure 1 supports four operational modes: point addition (Figure 1a), mixed point addition (Figure 1b), point doubling (Figure 1c), and mixed point doubling (Figure 1d). A mode selection module determines the operational mode based on the input point type. When integrated into the MSM hardware accelerator, the operational mode is also determined by the current step in Pippenger algorithm. The most frequently used mode is mixed point addition, which facilitates the bucket accumulation step by adding an affine coordinate point to the corresponding bucket in the extended Jacobian coordinate system. During the bucket aggregation step, the adder performs point addition and point doubling over points in the extended Jacobian coordinate system. Although the fourth mode, for point doubling across two different coordinate systems, is not utilized in our hardware, it may be applicable in other scenarios. This mode-dependent organization optimizes the dominant MSM case, mixed addition during bucket accumulation for full throughput, while reusing arithmetic resources in less frequent modes to reduce total hardware cost. As a result, we adopt the common datapath without fully duplicating arithmetic units for all point-operation modes.

C. A Modular Multiplier with Adaptive Recursive Karatsuba Algorithm

We propose a parameterized modular multiplier as the arithmetic core of the point adder. The data flow of the modular multiplier is illustrated in Figure 2. The design combines recursive Karatsuba multiplication [24] with Barrett reduction [25], while its key feature lies in an adaptive decomposition mechanism.

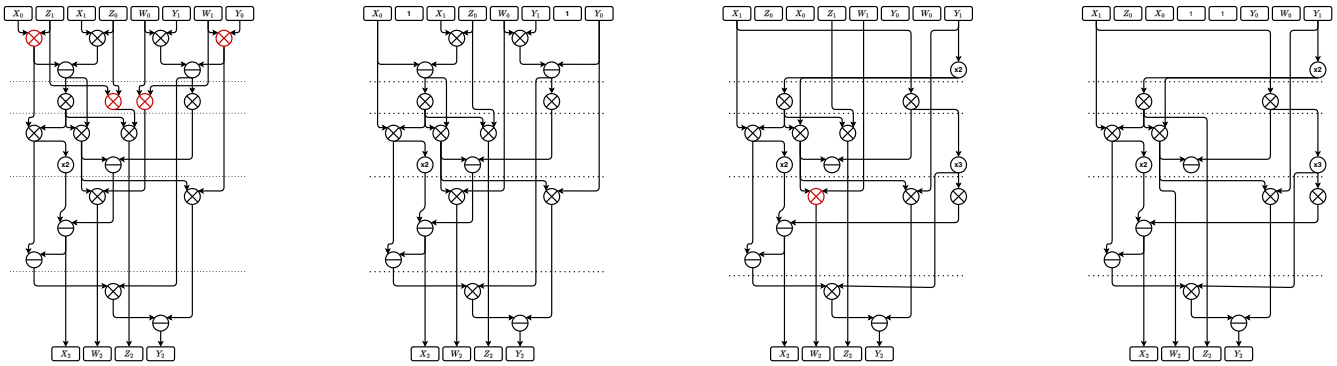
The main observation is that the efficiency of a recursive multiplier is ultimately determined by the bit width of the leaf multiplications at the lowest recursion level. Therefore, instead of using a fixed recursion template, the proposed design supports configurable operand widths and configurable base-case multiplier widths, and adaptively selects the recursive partition accordingly. In particular, it can choose between alternative decomposition patterns, such as two-way and three-way partitioning, so that the resulting leaf multiplications better match the native multiplier granularity of the target platform. This adaptive strategy improves hardware mapping efficiency and avoids the resource loss caused by mismatched operand partitioning.

After multiplication, Barrett reduction is adopted to avoid costly division and maintain a regular hardware-friendly modular reduction flow. Together, these techniques enable a modular multiplier that supports operand widths up to 381 bits while remaining adaptable to different implementation configurations.

Within the overall architecture, this multiplier is reused across the four point-operation modes and serves as the basis for the mode-dependent time-sharing strategy. By using a single configurable arithmetic core instead of separate dedicated multipliers, the design reduces hardware duplication while preserving the arithmetic flexibility required by different modes. The detailed resource and timing overhead of the multiplier are reported in the implementation results.

D. The Architecture of Point Adder

In this subsection, we provide an in-depth explanation of the hardware implementation of the EC point adder. The data flow of the EC point adder, as illustrated in Figure 3, operates through a pipelined architecture comprising five computational stages. The process begins with the *Input Control* module, which receives the input point pair. These points are prepared according to the selected operational mode, as determined by the mode control signal. If one of the input points is an



(a) General PADD. (b) Mixed PADD. (c) General PDBL. (d) Affine PDBL.

Fig. 1: Four modes of EC Point Adder. Stages are separated by dashed lines in those figures. The red multipliers in Figure 1a and Figure 1c indicate that operations are executed by other multipliers through a time-sharing multiplexing scheme.

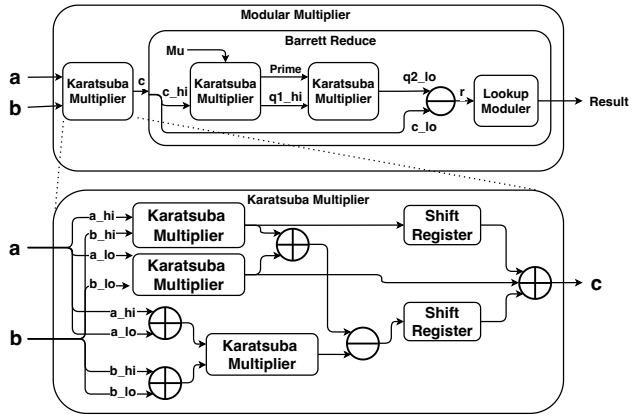


Fig. 2: Overview of the Modular Multiplier

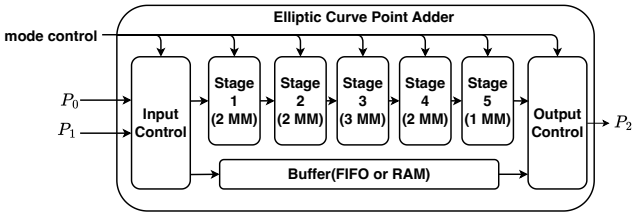


Fig. 3: Overview of EC Point Adder Architecture

infinite point, the other point is temporarily stored in a buffer, implemented using FIFO or RAM, before being processed by the *Output Control* module. This mechanism is designed to handle the special case of infinite points efficiently.

In general, the computation proceeds through the two stages for input and output buffer and five pipelined stages for computing. In computing stages, each performing specific modular arithmetic operations using *Modular Multipliers* (MMs). In Stage 1 and Stage 2, each stage employs two modular multipliers to execute preliminary computations, such as field multiplications required for intermediate values. Stage 3, equipped with three modular multipliers, handles more complex modular arithmetic operations, including intermediate products and additions. Finally, Stage 4 and Stage 5 complete the point addition or doubling computations, with Stage 4 utilizing two modular multipliers and Stage 5 using one.

This pipelined structure allows computations in different stages to execute concurrently, thereby maximizing throughput. Moreover, the modular multipliers are efficiently time-shared across the pipeline stages, optimizing resource utilization while maintaining high computational performance.

TABLE IV: Resource cost and pipeline depth of the point adder

Designs	LUT	FF	DSP	BRAM	Pipeline depth (cycles)
Modular Multiplier (254 bits)	39985	48537	262	0	52
Modular Multiplier (381 bits)	56877	67629	308	0	52
Point Adder (BN254)	414067	574661	2584	0	262
Point Adder (BLS12-381)	696755	943780	4790	168	262

E. Resource Cost and Throughput Analysis

Table IV reports the post-synthesis resource cost and pipeline depth of the proposed modular multiplier and point adder for two representative field sizes, 254 and 381 bits, at a target frequency of 300 MHz. As expected, increasing the field size from 254 to 381 bits raises the cost of a single modular multiplier from 39.9K to 56.9K LUTs and from 262 to 308 DSPs. At the point adder level, the resource increase is more pronounced: the BN254 configuration consumes 414K LUTs, 575K FFs, and 2,584 DSPs, whereas the BLS12-381 configuration requires 697K LUTs, 944K FFs, and 4,790 DSPs. The BLS12-381 point adder additionally uses 168 BRAMs, reflecting the larger operand width and buffering requirements in the datapath.

Notably, both point adder configurations preserve the same pipeline depth (262 cycles) . This indicates that the proposed five-stage pipelining strategy scales to larger field sizes without

increasing latency in cycles, and that the main scaling cost lies in arithmetic resources rather than in deeper control logic or a longer critical path. Importantly, the reported pipeline depth reflects latency, whereas steady-state throughput is determined by the operating mode and the associated multiplier-sharing policy.

The proposed point adder supports two main operating modes in UniMSM: mixed addition and general addition. Mixed addition is the dominant case in bucket accumulation, and this mode is fully pipelined with an initiation interval of one cycle, i.e., one point result per cycle. In contrast, general addition is used in aggregation and reuses modular multipliers through time-sharing, increasing the initiation interval to two cycles. This mode dependent organization is the key resource–throughput trade-off in our design: it optimizes the common mixed-addition path for full throughput, while avoiding full arithmetic duplication for less frequent modes.

Overall, the proposed point adder serves as the arithmetic backbone of UniMSM. It combines full-throughput mixed addition, resource-efficient general addition, and portability across representative short Weierstrass curves such as BN254 and BLS12-381. The post-synthesis results in Table IV therefore confirm that the proposed design achieves a favorable balance between arithmetic cost and throughput.

IV. Conflict-aware Dual-FIFO Scheduling for Bucket Accumulation

In bucket accumulation, each input point is mapped to a bucket index by the current scalar window. Points mapped to the same bucket must update the same bucket state in order. Therefore, temporally adjacent updates may target the same bucket, creating write hazards in the bucket memory read-modify-write path. These hazards either reduce throughput (stalls) or require expensive state replication. To address this, we design a conflict-aware scheduler that tracks in-flight bucket updates and schedules the conflict bucket updates.

A. Problem Statement

In j-th bucket accumulation, each update performs a point addition of $B_l + P_i$, where B_l is the current state of bucket l, and P_i is an incoming point whose scalar $k_{i,j}$ equals l. We model the bucket hazard duration as H cycles, where H denotes the number of cycles from issuing an update to the point at which the updated bucket value becomes safe to reuse. This hazard distance is determined by the end-to-end latency, including the pipeline of the point adder and the associated scheduling stages. Each bucket update follows a read-compute-write sequence. First, the current bucket state B_l is read from bucket memory. Then, the operand pair is loaded into the point-adder pipeline. After H cycles, the point adder writes back the result to bucket memory. An issued update to bucket l occupies the state for H cycles before the new value becomes usable.

We define the *in-flight bucket set* at cycle c as $\mathcal{I}(c)$. This set contains the indices of buckets for which an update has been issued but has not yet been written back at cycle c. A conflict occurs if a new update to the same bucket is issued while the previous one is still in flight, i.e., $l \in \mathcal{I}(c)$. Such an in-flight overlap violates the intended read-modify-write semantics: the later update may read a stale B_l (RAW hazard) or overwrite the earlier writeback (WAW hazard), leading to incorrect bucket accumulation results.

Accordingly, for each bucket l, the scheduler must enforce that two issues to the same bucket are separated by at least H cycles:

$$\mathsf{issue}(l, c) = 1 \Rightarrow \forall c' \in (c, c + H - 1),\ \mathsf{issue}(l, c') = 0. \tag{10}$$

B. Proposed Scheduling Strategy

We leverage the fixed pipeline latency to track only the buckets that are currently in flight. With a single-issue pipelined point adder, at most one update can be injected per cycle, and thus at most H bucket updates can be simultaneously in flight. To prevent reissuing updates to a bucket that is still hazardous, the scheduler maintains a short in-flight tracking structure for recently issued bucket indices. At the algorithmic level, the required tracking horizon is determined by the hazard distance H, i.e., the number of cycles during which a newly issued bucket remains unavailable.

We maintain a shift-register of length H as a in-flight bucket FIFO:

$$\mathbf{Q} \triangleq [q_0, q_1, \ldots, q_{H-1}], \tag{11}$$

where each entry q_i stores one bucket index using $(s-1)$ bits. Hence, the storage cost is $\mathrm{Cost}_{\mathsf{track}} = L \cdot (s-1)$ bits.

At a high level, conflict detection checks whether the target bucket index already appears in the current hazard-tracking structure. Specifically, this check is augmented with two conditions: (i) a lock-stage comparison against the request currently entering the tracker, and (ii) an unlock-stage bypass that allows a request to proceed if it matches the oldest entry being released in the same cycle.

For a successful issue in cycle c to bucket l, we push l into $\mathbf{Q}$ to update the in-flight bucket FIFO. Using a shift-register realization, the update rule is

$$q_0 \leftarrow q_1,\ q_1 \leftarrow q_2,\ \ldots,\ q_{H-2} \leftarrow q_{H-1},\ q_{H-1} \leftarrow l. \tag{12}$$

Equivalently, a circular FIFO implementation can be used to reduce wiring overhead.

Previous works such as Hardcaml [23] and CycloneMSM [26] also buffer colliding requests. In contrast, our scheduler couples retry buffering with explicit hazard tracking and arbitration, so that conflicting requests are reissued as soon as their hazards clear rather than being postponed to a later pass.

C. Effective Analysis

a) Correctness.: The scheduler guarantees that a bucket update is issued only when its target bucket is not currently hazardous. Concretely, the effective hazard-tracking state consists of the tracked in-flight bucket indices together with the current-cycle lock state. A request is blocked if its target bucket matches any active tracked entry, except for the special

case in which the only match is the oldest entry being released in the same cycle. This same-cycle unlock case is safe because the previous update has completed its hazard window. Therefore, any issued update reads the latest committed bucket value, and the resulting execution preserves the intended read–modify–write semantics for each bucket.

b) Hardware cost and timing: Compared with a bitmap that requires $2^{(s-1)}$ bits, the proposed design only stores a short window of recently issued bucket indices. Ignoring small control overheads (e.g., valid bits), the dominant tracking storage scales as $(H+\delta)\cdot(s-1)$ bits, where δ accounts for the internal stages of the check logic. The conflict detection logic comprises L equality comparators of width $(s-1)$ and an OR-reduction tree implementing. This logic is independent of the total number of buckets and scales linearly with the pipeline latency L.

The above derivation assumes a single-issue point-adder pipeline (one update injected per cycle). If the accelerator supports issuing $I>1$ updates per cycle (e.g., multiple point-adder pipelines), the same principle applies by tracking up to $I\cdot L$ in-flight bucket indices, resulting in a tracking storage of $(I\cdot L)\cdot(s-1)$ bits and a membership test over $I\cdot L$ entries.

V. Hardware-Friendly Optimizations of the Pippenger Algorithm

In this section, we introduce two optimizations for the Pippenger algorithm: the general aggregation and the bucket folding algorithm. After applying these two techniques, the Pippenger algorithm consists of two steps: *bucket accumulation* and *general aggregation*. The first optimization is a general bucket aggregation algorithm that consolidates the original algorithm's *bucket aggregation* step and *subtask aggregation* step into a single step *general aggregation*. This optimization eliminates data transfer between the *bucket aggregation* and *subtask aggregation*. Building on this enhancement, the second optimization further minimizes latency by executing *general aggregation* with minimal data dependency. Since the core operations in *general aggregation* are PADD and PDBL operations, we can leverage fully pipelined point adders (Section III) for hardware optimization. This approach improves efficiency compared to the original serial bucket aggregation method and [4]'s parallel variant of the bucket aggregation.

Figure 4 summarizes the workflows of the Pippenger algorithm and our optimized variant. The Pippenger algorithm processes each window independently to obtain subtask results Q_j, and then combines $\{Q_j\}$ through a separate subtask aggregation, which introduces a redundant memory cost for storing subtask results and latencies for computing the subtask aggregation. In contrast, our variant fuses this subtask aggregation into each bucket aggregation by injecting the term $2Q_{prev}$ into a designated bucket before aggregation, referred to as general aggregation. Meanwhile, bucket folding replaces the serial bucket aggregation with an iterative folding procedure, reducing intra-window dependencies and making the aggregation structure more amenable to pipelining.

A. General Aggregation Algorithm

In the original subtask aggregation, the computation can be viewed as a MSM computation with a scalar vector and a point vector. Unlike the outer MSM, however, the scalars in this step are powers of two, rather than consisting of arbitrary input scalars. After aggregation of the j-th bucket sets, we can get the result Q_j of the j-th bucket aggregation. Instead of performing a separate subtask aggregation, we inject the term $2Q_{j+1}$ into the highest-weight bucket of the next aggregation round. For the j-th subtask result Q_j, the scalar corresponding to this point is 2^{jc}, while the scalar corresponding to Q_{j-1} is $2^{(j-1)c}$. Thus, the correctness of the merging comes from the following equation:

$$\begin{aligned} Q_{j-1}+2^cQ_j &= \sum_{i=1}^{2^{c-1}} iB_{i,j-1}+2^cQ_j \\ &= \sum_{i=1}^{2^{c-1}-1} iB_{i,j-1}+2^{c-1}B_{2^{c-1},j-1}+2^cQ_j \\ &= \sum_{i=1}^{2^{c-1}-1} iB_{i,j-1}+2^{c-1}(B_{2^{c-1},j-1}+2Q_j). \end{aligned} \tag{13}$$

The final result of the bucket aggregation can be computed using Algorithm 1. The third step is to accumulate the results of the subtask for $Q=\sum_{j=0}^{\lambda/c-1}2^{jc}Q_j$. The calculation can be completed in the bucket aggregation phase in the second step.

Algorithm 1 General Bucket Aggregation Algorithm

Require: λ/c bucket sets $[B_{1,0},B_{2,0},\cdots,B_{2^{c-1},0}],\cdots,$ $[B_{1,\lambda/c-1},B_{2,\lambda/c-1},\cdots,B_{2^{c-1},\lambda/c-1}]$
Ensure: $Q=\sum_{j=0}^{\lambda/c-1}2^{jc}Q_j=\sum_{j=0}^{\lambda/c-1}2^{jc}\sum_{t=1}^{2^{c-1}}tB_{t,j}$
$Q_{\lambda/c}\leftarrow\mathcal{O}$
for $j\leftarrow\lambda/c-1$ to 0 **do**
  $Q_j\leftarrow\mathcal{O}$
  $T\leftarrow\mathcal{O}$
  $B_{2^{c-1},j}\leftarrow 2\cdot Q_{j+1}$
  for $t\leftarrow 2^{c-1}$ to 1 **do**
    $T\leftarrow T+B_{t,j}$
    $Q_j\leftarrow Q_j+T$
  end for
end for
$Q\leftarrow Q_0$
return Q

By merging subtask aggregation into bucket aggregation, the general aggregation removes the additional $\lambda-c$ PDBLs and λ/c PADDs incurred by the separate aggregation method, leaving only λ/c PDBLs. Consequently, the Pippenger algorithm with the general bucket aggregation requires $\lambda(n+2^c-2)/c$ PADDs and λ/c PDBLs in total. This optimization also eliminates the extra storage of $\lambda/c\cdot\mathcal{M}_{\mathsf{point}}$, where $\mathcal{M}_{\mathsf{point}}$ denotes the memory footprint of one point.

B. Bucket Folding Algorithm

Although our proposed general bucket aggregation algorithm (Algorithm 1) simplifies the original process, it remains

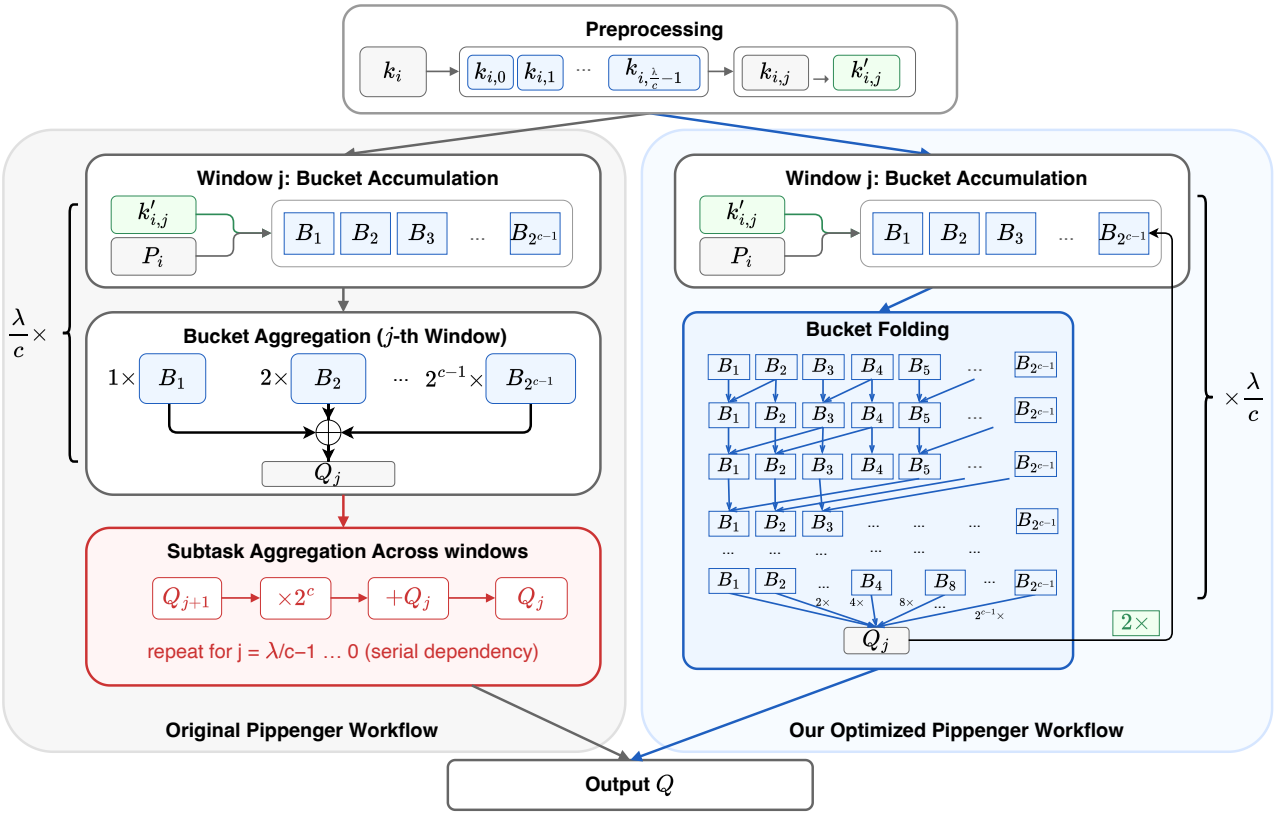


Fig. 4: Workflow comparison between the baseline and the optimized Pippenger algorithms

inherently serial and therefore cannot leverage hardware computational resources to accelerate the aggregation process. To reduce the pipeline stalls when computing the bucket aggregation in hardware, [4] proposed a parallel variant of bucket aggregation. This algorithm breaks the bucket aggregation into p sections, each of which can run in parallel to reduce the pipeline stalls caused by the serial computation of bucket aggregation. This algorithm requires p additional PADDs and $c - \log_2(p)$ more PDBLs compared to the original bucket aggregation algorithm. The additional overhead from the point operations helps reduce overall latency by minimizing pipeline stalls, making it more suitable for hardware accelerator design.

Inspired by the motivation behind [4]'s work, we introduce a bucket folding algorithm. The Bucket Folding Algorithm (Algorithm 2) introduces an iterative folding process to optimize the bucket aggregation, eliminating redundant dependencies and improving scalability. The algorithm operates as follows: First, the buckets are repeatedly folded in pairs, with each pair combined by summing their corresponding values. This process gradually reduces the total number of buckets while preserving the cumulative weight for each folded group. The folding continues iteratively until only a single set of buckets remains. Subsequently, the buckets are processed using Horner's method to produce the final result.

a) Compared to the parallel bucket aggregation algorithm: The Bucket Folding Algorithm offers several advantages over the Parallel Bucket Aggregation Algorithm [4]. It reduces data dependencies more effectively by progres-

Algorithm 2 Bucket Folding Algorithm

Require: n buckets $[B_1, B_2, \cdots, B_n]$
Ensure: $Q = \sum_{t=1}^{n} tB_t$
$Q \leftarrow \mathcal{O}$
$s_i \leftarrow 1$
while $s_i < n$ **do**
  $i \leftarrow 1$
  while $i + s_i < n$ **do**
    $s_j \leftarrow 0$
    while $s_j < s_i$ **and** $i + s_i + s_j < n$ **do**
      $B_{i+s_j} = B_{i+s_j} + B_{i+s_i+s_j}$
      if $s_j = 0$ **then**
        $s_j = 1$
      else
        $s_j = 2s_j$
      end if
    end while
    $i \leftarrow i + 2s_i$
  end while
  $s_i = 2s_i$
end while
for $k \leftarrow 0$ to $c - 1$ **do**
  $Q \leftarrow Q + 2^k B_{2^k}$
end for
return Q

sively folding and combining buckets, which simplifies the computation process and enhances scalability. By folding buckets in place, it also minimizes memory overhead compared to the parallel approach, which often requires additional memory for intermediate results. Furthermore, the iterative nature of the Bucket Folding Algorithm, with its dynamically adjusted step sizes, ensures improved scalability for large-scale computations, making it more efficient when dealing with a high number of buckets. The folding mechanism minimizes simultaneous memory accesses and optimizes the reuse of on-chip resources, making it particularly well-suited for hardware implementations. Additionally, the algorithm eliminates redundant operations by folding buckets iteratively, whereas the parallel approach often involves additional operations during its multi-stage summation process. By distributing computation more evenly across iterations, the Bucket Folding Algorithm avoids the potential load imbalances that can arise in parallel execution, further enhancing its efficiency and overall performance.

C. Putting Two-level Optimization Together

Algorithm 3 combines the two optimizations as follows: bucket accumulation remains unchanged, each round uses general aggregation to absorb the carry-over term from the previous round, and the per-round aggregation is implemented by bucket folding instead of the classical serial sweep. This combination removes the separate subtask-aggregation pass while shortening the dependency chain within each aggregation round, yielding a more hardware-friendly Pippenger workflow.

Algorithm 3 The optimized Pippenger algorithm

Require: λ-bit scalars $k_0, k_1, \ldots, k_{n-1}$ and EC points $P_0, P_1, \ldots, P_{n-1}$
Ensure: $Q = \sum_{i=0}^{n-1} k_i P_i$
Convert each scalar k_i into its binary form
Partition each scalar into λ/c windows to obtain $\{k_{i,j}\}_{0 \le j \le \frac{\lambda}{c}-1}$
Preprocess these values to obtain $k'_{i,j} \in [-2^{c-1}+1, 2^{c-1}]$
// First subtask ($j = \lambda/c - 1$)
Initialize 2^{c-1} buckets to $\mathcal{O}$
Accumulate the points into corresponding buckets based on $k'_{i,\lambda/c-1}$
Invoke Algorithm 2 to aggregate 2^{c-1} buckets into a subtask result
// Subsequent subtasks
for $j \leftarrow \lambda/c - 2$ to 0 **do**
    Reset each bucket to infinity, except for the bucket indexed at $t = 2^{c-1}$
    Set $B_{2^{c-1}} \leftarrow 2 \cdot Q_{prev}$ according to Algorithm 1
    Accumulate points and aggregate buckets for the current subtask
end for
return The final aggregation result Q

VI. A Flexible and Decoupled Hardware Architecture

In this section, we present UniMSM, a flexible MSM architecture that decouples arithmetic computation, bucket storage, and dataflow control. UniMSM is organized into three units: a compute unit for EC arithmetic, a memory unit for bucket buffering and updates, and a control unit for preprocessing, conflict-aware scheduling, and execution flow control. This separation allows the compute pipeline to remain reusable across curve parameters and operation modes, while the control and memory units handle the irregularity of the bucket updates. Figure 5 shows the overall architecture and the separation of computation and dataflow control.

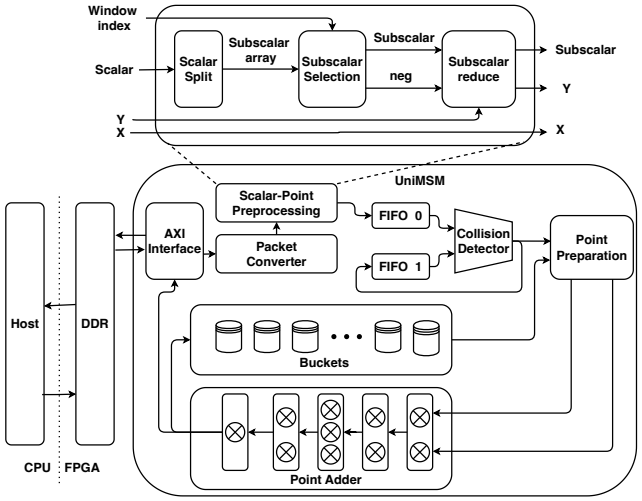


Fig. 5: Overview of the MSM architecture

a) Compute unit: Once a work item is issued, the compute unit executes the corresponding EC operation for either bucket accumulation or aggregation. The *Point Adder* is a flexible and pipelined core (Section III), that supports multiple operation modes. The compute unit is decoupled from memory addressing and collision management: it consumes operands from the control and the memory units, and produces one point in extended Jacobian coordinates as the result.

b) Memory unit: The memory unit implements the *Buckets* module, which buffers all bucket states for bucket accumulation. Buckets are stored in extended Jacobian form to avoid frequent coordinate conversions during repeated updates. For each issued bucket-update work item, the memory unit reads the current bucket value at the specified bucket address and supplies it to the compute unit as one operand; after computation, it writes the updated point back to the same bucket address. To sustain a high-throughput pipeline, the bucket memory exposes a structured access interface to the control unit, supporting read/write arbitration and hazard-aware scheduling under deep pipelining. Across phases, the memory unit also supports streaming bucket states or accumulated results via the AXI/DDR path: during accumulation it serves fine-grained bucket accesses, while after completing the final window it allows the final accumulator (or reduced bucket set) to be drained and returned to the host.

c) Control unit: The control unit orchestrates MSM execution by performing input preprocessing, conflict-aware scheduling, and phase-level flow control. As shown in Figure 5, it consists of five modules: an AXI interface module, a packed converter, a scalar–point preprocessing module, a collision detector, and a point preparation module. The *AXI Interface* module provides functional interconnection and synchronization among the control, memory-access, and compute clock domains in the MSM system. It conveys start and configuration information and bridges streaming data inputs and outputs across domains. It supplies the cross-domain communication foundation that enables coordinated MSM algorithm execution in a multi-clock environment. The *Packed Converter* module normalizes the memory interface to a uniform internal format (e.g., for BLS12-381, a 256-bit scalar and a point under affine coordinates consisting of two 384-bit coordinates).

The *Scalar-Point Preprocessing* module is responsible for preprocessing input pairs of scalar and point, generating a reduced-length subscalar based on the current window index and the corresponding processed points as output. In this module, the input scalar is divided into 16 subscalars by the *Scalar Split* module based on the specified window size. The *Subscalar Selection* module selects the corresponding subscalar for each input window index. Additionally, this module determines a flag, referred to as neg, based on the specific value of the subscalar, indicating whether the Y-coordinate of the input point should be negated in subsequent operations. The operations of negating the Y-coordinate and reducing the subscalar length from 16 bits to 15 bits are performed by the *Subscalar Reduce* module. Finally, the *Scalar-Point Preprocessing* module outputs a 15-bit subscalar and processed point coordinates, forming a preprocessed subscalar-point pair.

Before sending a processed point to the *Point Adder* module, collision detection is essential. To facilitate this, the pair from the *Scalar-Point Preprocessing* module is first stored in a FIFO buffer. Simultaneously, another FIFO buffer is used to cache colliding pairs, as identified by the *Collision Detector* module. The *Collision Detector* module dynamically selects a preprocessed pair from the two FIFOs and issues work items in an order that avoids in-flight collisions on the same target bucket, as determined by the selected subscalar in the current window. The selected pair is forwarded to the *Point Preparation* module, which forms the operands according to the current execution phase, and the resulting operand pair is then issued to the *Point Adder*.

For the phase orchestration, the control unit schedules the MSM execution across windows by tracking completion of issued updates and draining buffered collisions. After all windows are processed, the control unit triggers result draining through the AXI/DDR path, and the host retrieves the output through PCIe.

A. Analysis of Parameters

In this section, we analyze the impact of the key hardware-tunable parameter, namely the window size c, on the performance UniMSM and resource cost of UniMSM. We consider two representative curves, BN254 and BLS12-381, whose scalar-field bit lengths determine λ, while the MSM size N is treated as the workload parameter.

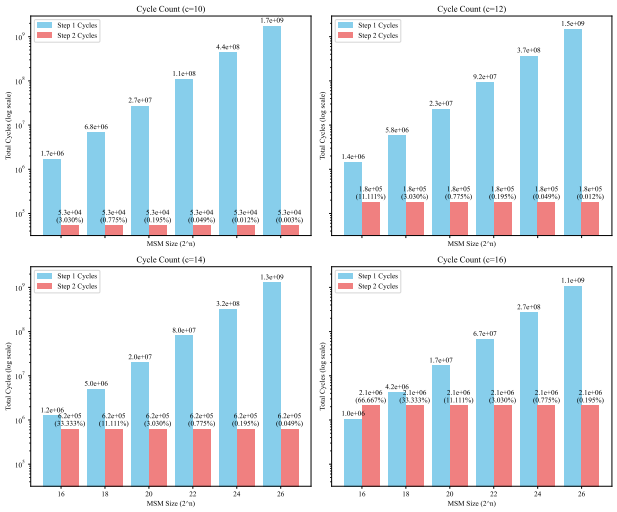


Fig. 6: Cycle Count with Different Window Sizes

To understand how these parameters influence performance, we first examine the breakdown of the MSM procedure. The entire process consists of two steps: bucket accumulation and bucket aggregation. The bucket accumulation invokes a pipelined point adder in mixed PADD mode, thus, the cycles needed in this stage are $\lambda N/c$ cycles. The cycle count of bucket aggregation using the point adder in general PADD mode in each round is equal to $2 \cdot \frac{\lambda}{c} \cdot 2^c$ (two cycles for each general PADD), which depends on the number of buckets 2^{c-1}. Thus, the total cycles C_{total} of a MSM task can be represented by the following formula:

$$C_{total} = C_{step1} + C_{step2} \approx \frac{\lambda}{c} \cdot N + \frac{\lambda}{c} \cdot 2^{c+1} \tag{14}$$

We conducted experiments to analyze the impact of different window sizes on the cycle counts across varying MSM sizes, as shown in Figure 6. As the window size increases from 10 bits to 16 bits, the number of cycles in step 1 decreases, while the cycles in step 2 increase. However, the proportion of step 2 cycles relative to the total decreases, resulting in an overall reduction in total cycles.

However, larger window sizes lead to a rapid increase in the RAM resources required by the accelerator, resulting in a larger overall area (ASIC). To identify a parameter c that achieves a favorable balance between cycles and resource cost, we define the area-cycle product (ACP). The experimental results (ASIC synthesis) for the BLS12-381 curve with $c = 10$ serve as the baseline area. The change in total area (where RAM area increases with c, while the area occupied by logic resources remains unaffected by c) is expressed as a proportion relative to the baseline area, allowing us to compute and compare ACP. The results are shown in Figure 7. Based on the experimental results, we set the window size $c = 10$ to balance the total cycles C_{total} and hardware resource consumption.

VII. Performance Evaluation

To evaluate the efficiency and generality of the proposed MSM accelerator, we benchmarked its performance on a Xil-

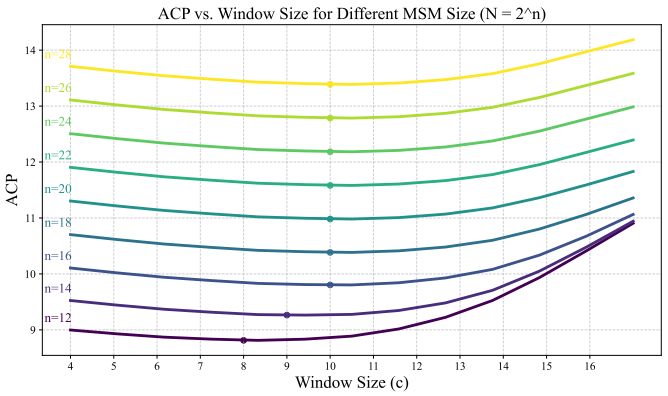


Fig. 7: Optimal window size under different MSM Size N

inx Alveo U280 FPGA and via ASIC synthesis using TSMC 12nm technology. The evaluation highlights the effectiveness of the proposed co-design across arithmetic, scheduling, and algorithm flow, together with hardware-aware parameter tuning. Results demonstrate the practical effectiveness and favorable trade-offs in performance and resource usage compared to state-of-the-art designs.

A. Experimental Setup

For the FPGA implementation, the accelerator architecture was developed in SystemVerilog and deployed on a Xilinx Alveo U280 board. This platform offers substantial programmable resources, including 9024 DSP slices, 1304K lookup tables (LUTs), 2547 36 kbits BRAMs, and 1280 288 Kb URAMs. Resource utilization and clock frequency were obtained after synthesis and implementation using Vivado 2023 with default optimization strategies. This FPGA-based evaluation showcases our design's efficiency in leveraging reconfigurable hardware resources while achieving high performance. Complementing the FPGA analysis, we performed ASIC synthesis to evaluate the design's scalability and potential for mass production. This synthesis utilized the TSMC 12nm technology library, providing insights into area, power, and latency. Combined with the FPGA results, the ASIC analysis provides a comprehensive view of the proposed MSM hardware accelerator. It highlights both its performance characteristics and adaptability across diverse hardware platforms.

a) Baselines and data sources: We compare UniMSM against representative MSM accelerators over FPGA and ASIC platforms that report latency and hardware cost. For FPGA, we include MSMAC [12] over VP1502, OPTIMSM [13] over U55C, and FusionMSM [14] over VU13P, which collectively cover short Weierstrass and twisted Edwards curves and reflect recent design points on high-end Xilinx devices. For ASIC, we compare with PipeZK [19] and two recent ASIC accelerators [20], [27].

b) Metrics and comparison methodology: We report (i) MSM latency from $N = 2^{20}$ to $N = 2^{26}$, and (ii) hardware cost in terms of DSP, LUT, FF, and on-chip RAM for FPGA, and area and frequency for ASIC. Because FPGA baselines target different devices and resource budgets, absolute resource counts are compared to highlight the implementation footprint, while latency is compared at the reported operating frequencies. Besides, we adopt the Area-Time Product (ATP) metric to summarize FPGA cost-performance trade-offs. We aggregate heterogeneous FPGA resources into an equivalent area $\mathcal{A}$ using weighting ratios reported in prior work [28], [29]:

$$\mathcal{A} = (\frac{\mathrm{N_{LUT}}}{16} + \frac{\mathrm{N_{FF}}}{8} + 102.4 \times \mathrm{N_{DSP}} + 56 \times \mathrm{N_{RAM}})/1000, \tag{15}$$

where $\mathrm{N_{LUT}}$, $\mathrm{N_{FF}}$, $\mathrm{N_{DSP}}$, and $\mathrm{N_{RAM}}$ denote the counts of LUTs, FFs, DSPs, and BRAM-equivalents, respectively.[1] We then compute $\mathrm{ATP} = \mathcal{L} \times \mathcal{A}$, where a smaller ATP indicates better efficiency. For ASIC, we additionally use the area–time product (ATP) at $N = 2^{20}$ as a unified efficiency metric. We report ATP directly from each work's area and latency.

B. Performance and Resource Utilization on FPGA

Table V compares UniMSM with representative FPGA MSM accelerators in terms of latency and implementation cost. A key strength of UniMSM is its small implementation footprint, especially in on-chip memory. For BN254, UniMSM uses 2693 DSPs and 436K LUTs, compared with 6480 DSPs and 1290K LUTs in MSMAC, while reducing on-chip memory from 6474 BRAM to 99 BRAM. This corresponds to a 2.4× reduction in DSPs, a 3.0× reduction in LUTs, and a 65× reduction in on-chip memory blocks, reflecting the efficiency of our decoupled organization and bucket buffering design. For BLS12-381, UniMSM achieves competitive latency across prover-scale sizes (e.g., 78.5 ms at 2^{20} and 4935.9 ms at 2^{26}) while significantly relieving BRAM pressure. Compared with OPTIMSM, UniMSM reduces BRAM usage from 2032 to 861, which improves deployability on RAM-constrained FPGA devices.

An important reason behind the reduced on-chip memory footprint is our window-size selection. In UniMSM, the on-chip bucket storage is implemented in the *memory unit* and scales with the number of buckets, which grows exponentially with the window size c. Concretely, each window requires approximately 2^{c-1} running buckets, and each bucket stores an EC point state; therefore, the memory requirement of buckets is $\Theta(2^{c-1})$. As the parameter analysis in Sec. VI-A, UniMSM adopts $c = 10$ as a balanced design point for prover-scale MSM. This reduces the bucket count by $2^{13-10} = 8\times$ relative to larger window sizes (e.g., $c = 13$ adopted by designs [12], [14], and $c = 19$ adopted by designs [13]). This directly translates to an order-of-magnitude reduction in bucket buffering demand and thus BRAM/URAM usage.

Overall, UniMSM delivers a favorable cost–performance trade-off. In particular, UniMSM achieves the lowest reported ATP among the compared BN254 implementations, indicating that our architecture improves efficiency not by over-provisioning resources, but by co-designing compute, scheduling, and memory to sustain scalable MSM execution.

TABLE V: Performance and resource utilization comparison across FPGA implementations.

Design	Freq. (MHz)	DSP	LUT (K)	FF (K)	BRAM*	Off-chip memory	Latency (ms)				ATP
							2^{20}	2^{22}	2^{24}	2^{26}	
Curve: BN254											
MSMAC [12]	250	6480	1290	1500†	6474	1.49 GB	34.15	114.48	373.77	1420	44.20 (1.66×)
UniMSM	300	2693	436	528	99	1.49 GB	71.2	280.95	1119.8	4475.25	26.67
Curve: BLS12-381											
OPTIMSM [13]	260	2147	721	885	2032	10.92 GB	60	231	914	–	29.36‡
Ingonyama [18]	250	9011	849	946	1069	1.99 GB	95	202	631	–	109.63 (2.12×)
UniMSM	272	4690	597	724	861	1.99 GB	78.5	309.9	1235.1	4935.9	51.54

- –: this metric is not reported.
- *: the BRAM-equivalent count including URAM.
- †: MSMAC does not report FF usage and its ATP is estimated based on the FF-to-LUT ratios of comparable accelerators.
- ‡: OPTIMSM benefits from fixed-base precomputation, whereas other designs target generic MSM with arbitrary input points.

TABLE VI: Performance, area, power, and area-time product (ATP) comparison across ASIC implementations.

Design	Freq. (GHz)	Area (mm^2)	Power (W)	Latency (ms)					ATP ($mm^2 \cdot ms$)
				2^{18}	2^{20}	2^{22}	2^{24}	2^{26}	
Curve: BN254									
PipeZK [19]	0.3	35.34	5.05	16	61	–	–	–	2155.74 (73.36×)
SZKP [27]	0.3	26.00	–	8.3	31.9	–	–	–	829.4 (28.23×)
Our work	1.5	1.61	1.94	4.62	18.25	72.78	290.88	1163.30	29.39
Curve: BLS12-381									
PipeZK [19]	0.3	33.72	4.75	46	184	–	–	–	6204.48 (43.94×)
[20]	0.8	15.04	–	10.7	36.1	137.8	544.6	2171.5	542.94 (3.85×)
Our work	1.0	6.61	5.43	5.63	21.36	84.28	335.94	1342.57	141.19

- "–" denotes this metric is not reported.

C. Performance and Area on ASIC

To evaluate the competitiveness of our MSM accelerator in ASIC implementations, we compare our design with the first ASIC-based MSM accelerator PipeZK [19] and two state-of-the-art ASIC-based MSM accelerators: the design in [20] and SZKP [27]. Table VI summarizes the key comparison metrics, including frequency, area, latency across different input sizes, and area-time product (ATP).

Our implementation targets both BN254 and BLS12-381 curves, with the BN254 design running at 1.5 GHz and the BLS12-381 design running at 1 GHz. For BN254, our design achieves 18.25 ms at $N = 2^{20}$, compared with 61 ms for PipeZK and 31.9 ms for SZKP, corresponding to 3.34× and 1.75× lower latency, respectively. For BLS12-381, our implementation occupies only 6.61 mm^2, compared to 33.72 mm^2 for PipeZK and 15.04 mm^2 for the design in [20], achieving a 2.3× to 5× area reduction despite delivering higher throughput.

In terms of latency for BLS12-381, our MSM accelerator consistently outperforms existing designs at all comparable points where results are reported. For $N = 2^{20}$, our latency is 21.36 ms, versus 184 ms for PipeZK and 36.1 ms for the accelerator in [20]—yielding an 8.6× and 1.7× speedup, respectively. At the largest evaluated size up to 2^{26}, our implementation finishes in 1342.57 ms, while this design in [20] requires 2171.5 ms, again demonstrating strong scalability.

A particularly illustrative metric is ATP, which captures the efficiency of a design in both area and latency. At $N = 2^{20}$, our ATP is 29.39 $mm^2 \cdot ms$ for BN254 and 141.19 $mm^2 \cdot ms$ for BLS12-381. This translates to a 73.36× and 28.23× improvement over PipeZK and SZKP on BN254, and a 43.94× and 3.85× improvement over PipeZK and [20] on BLS12-381, respectively.

[1]When UltraRAM is used, we report it in BRAM-equivalents based on memory capacity. On AMD/Xilinx FPGAs, one UltraRAM block provides 288 Kb storage, which is eight times the capacity of a 36 Kb Block RAM [30].

Overall, our ASIC MSM accelerator achieves state-of-the-art efficiency with area-time product, enabled by the mode-dependent point adder and architecture-level co-design across computation, scheduling, and memory. These results demonstrate that our design is well-suited for high-throughput MSM-based cryptographic systems deployed on next-generation ASIC platforms.

VIII. Conclusion

In this work, we present UniMSM, a flexible hardware accelerator for multi-scalar multiplication that improves resource efficiency. The design introduces a pipelined EC point adder based on extended Jacobian coordinates and hardware-aware optimization of the Pippenger algorithm. Experimental results show that UniMSM achieves competitive latency while significantly reducing FPGA resource usage compared with prior designs. Furthermore, UniMSM improves efficiency, achieving up to 3.85× and 28.23× ATP improvements in area–time product over state-of-the-art MSM accelerators.

References

[1] J. Groth, "On the size of pairing-based non-interactive arguments," in *Advances in Cryptology – EUROCRYPT 2016, Part II*, ser. Lecture Notes in Computer Science, M. Fischlin and J.-S. Coron, Eds., vol. 9666. Vienna, Austria: Springer Berlin Heidelberg, Germany, May 8–12, 2016, pp. 305–326.

[2] A. Gabizon, Z. J. Williamson, and O. Ciobotaru, "PLONK: Permutations over Lagrange-bases for oecumenical noninteractive arguments of knowledge," Cryptology ePrint Archive, Report 2019/953, 2019. [Online]. Available: https://eprint.iacr.org/2019/953

[3] A. Kothapalli, S. Setty, and I. Tzialla, "Nova: Recursive zero-knowledge arguments from folding schemes," in *Advances in Cryptology – CRYPTO 2022, Part IV*, ser. Lecture Notes in Computer Science, Y. Dodis and T. Shrimpton, Eds., vol. 13510. Santa Barbara, CA, USA: Springer, Cham, Switzerland, Aug. 15–18, 2022, pp. 359–388.

[4] C. F. Xavier, "PipeMSM: Hardware acceleration for multi-scalar multiplication," Cryptology ePrint Archive, Report 2022/999, 2022. [Online]. Available: https://eprint.iacr.org/2022/999

[5] T. Lu, C. Wei, R. Yu, C. Chen, W. Fang, L. Wang, Z. Wang, and W. Chen, "cuZK: Accelerating zero-knowledge proof with A faster parallel multi-scalar multiplication algorithm on GPUs," *IACR Transactions on Cryptographic Hardware and Embedded Systems*, vol. 2023, no. 3, pp. 194–220, 2023.

[6] A. Kate, G. M. Zaverucha, and I. Goldberg, "Constant-size commitments to polynomials and their applications," in *Advances in Cryptology – ASIACRYPT 2010*, ser. Lecture Notes in Computer Science, M. Abe, Ed., vol. 6477. Singapore: Springer Berlin Heidelberg, Germany, Dec. 5–9, 2010, pp. 177–194.

[7] J. Lee, "Dory: Efficient, transparent arguments for generalised inner products and polynomial commitments," in *TCC 2021: 19th Theory of Cryptography Conference, Part II*, ser. Lecture Notes in Computer Science, K. Nissim and B. Waters, Eds., vol. 13043. Raleigh, NC, USA: Springer, Cham, Switzerland, Nov. 8–11, 2021, pp. 1–34.

[8] E. B. Sasson, A. Chiesa, C. Garman, M. Green, I. Miers, E. Tromer, and M. Virza, "Zerocash: Decentralized anonymous payments from bitcoin," in *2014 IEEE symposium on security and privacy*. IEEE, 2014, pp. 459–474.

[9] N. Pippenger, "On the evaluation of powers and related problems," in *17th Annual Symposium on Foundations of Computer Science (sfcs 1976)*. IEEE Computer Society, 1976, pp. 258–263.

[10] Y. Yang, Z. Lu, J. Zeng, X. Liu, X. Qian, and Z. Yu, "Falic: An fpga-based multi-scalar multiplication accelerator for zero-knowledge proof," *IEEE Transactions on Computers*, 2024.

[11] S. A. Butt, B. Reynolds, V. Ramamurthy, X. Xiao, P. Chu, S. Sharifian, S. Gribok, and B. Pasca, "if-zkp: Intel fpga-based acceleration of zero knowledge proofs," *arXiv preprint arXiv:2412.12481*, 2024.

[12] P. Qiu, G. Wu, T. Chu, C. Wei, R. Luo, Y. Yan, W. Wang, and H. Zhang, "MSMAC: accelerating multi-scalar multiplication for zero-knowledge proof," in *Proceedings of the 61st ACM/IEEE Design Automation Conference, DAC 2024, San Francisco, CA, USA, June 23-27, 2024*, V. De, Ed. ACM, 2024, pp. 66:1–66:6. [Online]. Available: https://doi.org/10.1145/3649329.3655672

[13] X. Pottier, T. de Ruijter, J. Bertels, W. Legiest, M. Van Beirendonck, and I. Verbauwhede, "Optimsm: Fpga hardware accelerator for zero-knowledge msm," *IACR Transactions on Cryptographic Hardware and Embedded Systems*, vol. 2025, no. 2, pp. 489–510, 2025.

[14] C. Chen, G. Yang, H. Zhou, H. Xiong, X. Ben, and Z. Wan, "Fusion-msm: A collision-free and arithmetic-optimized fpga-based accelerator for multi-scalar multiplication," *IACR Transactions on Cryptographic Hardware and Embedded Systems*, vol. 2025, no. 4, pp. 1077–1101, 2025.

[15] B. Zhao, W. Huang, T. Li, and Y. Huang, "Bstmsm: A high-performance fpga-based multi-scalar multiplication hardware accelerator," in *2023 International Conference on Field Programmable Technology (ICFPT)*. IEEE, 2023, pp. 35–43.

[16] C. Liu, H. Zhou, P. Dai, L. Shang, and F. Yang, "Priormsm: An efficient acceleration architecture for multi-scalar multiplication," *ACM Transactions on Design Automation of Electronic Systems*, vol. 29, no. 5, pp. 1–26, 2024.

[17] C. Liu, H. Zhou, L. Yang, J. Xu, P. Dai, and F. Yang, "Gypsophila: A scalable and bandwidth-optimized multi-scalar multiplication architecture," in *Proceedings of the 61st ACM/IEEE Design Automation Conference*, 2024, pp. 1–6.

[18] Ingonyama, "Deep dive into the latest msm hardware implementation," Apr 2024. [Online]. Available: https://medium.com/@ingonyama/deep-dive-into-the-latest-msm-hardware-implementation-a9739b2cd107

[19] Y. Zhang, S. Wang, X. Zhang, J. Dong, X. Mao, F. Long, C. Wang, D. Zhou, M. Gao, and G. Sun, "Pipezk: Accelerating zero-knowledge proof with a pipelined architecture," in *2021 ACM/IEEE 48th Annual International Symposium on Computer Architecture (ISCA)*. IEEE, 2021, pp. 416–428.

[20] X. Chen, B. Yang, W. Zhu, H. Wang, Q. Tao, S. Yin, M. Zhu, S. Wei, and L. Liu, "A high-performance ntt/msm accelerator for zero-knowledge proof using load-balanced fully-pipelined montgomery multiplier," *IACR Transactions on Cryptographic Hardware and Embedded Systems*, vol. 2025, no. 1, pp. 275–313, 2025.

[21] D. J. Bernstein, "Explicit-formulas database," *http://www. hyperelliptic. org/EFD*, 2007.

[22] K. Aasaraai, D. Beaver, E. Cesena, R. Maganti, N. Stalder, and J. Varela, "FPGA acceleration of multi-scalar multiplication: CycloneMSM," Cryptology ePrint Archive, Report 2022/1396, 2022. [Online]. Available: https://eprint.iacr.org/2022/1396

[23] A. Ray, B. Devlin, F. Y. Quah, and R. Yesantharao, "Hardcaml msm: A high-performance split cpu-fpga multi-scalar multiplication engine," in *Proceedings of the 2024 ACM/SIGDA International Symposium on Field Programmable Gate Arrays*, 2024, pp. 33–39.

[24] A. Weimerskirch and C. Paar, "Generalizations of the karatsuba algorithm for efficient implementations," Cryptology ePrint Archive, Report 2006/224, 2006. [Online]. Available: https://eprint.iacr.org/2006/224

[25] P. Barrett, "Implementing the Rivest Shamir and Adleman public key encryption algorithm on a standard digital signal processor," in *Advances in Cryptology – CRYPTO'86*, ser. Lecture Notes in Computer Science, A. M. Odlyzko, Ed., vol. 263. Santa Barbara, CA, USA: Springer Berlin Heidelberg, Germany, Aug. 1987, pp. 311–323.

[26] K. Aasaraai, D. Beaver, E. Cesena, R. Maganti, N. Stalder, and J. Varela, "FPGA Acceleration of Multi-Scalar Multiplication: CycloneMSM," 2022, publication info: Preprint. [Online]. Available: https://eprint.iacr.org/2022/1396

[27] A. Daftardar, B. Reagen, and S. Garg, "Szkp: A scalable accelerator architecture for zero-knowledge proofs," in *Proceedings of the 2024 International Conference on Parallel Architectures and Compilation Techniques*, 2024, pp. 271–283.

[28] Y. Tu, P. He, Ç. K. Koç, and J. Xie, "Leap: Lightweight and efficient accelerator for sparse polynomial multiplication of hqc," *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, vol. 31, no. 6, pp. 892–896, 2023.

[29] P. He, S. C. O. Madrigal, Ç. K. Koç, T. Bao, and J. Xie, "CASA: A compact and scalable accelerator for approximate homomorphic encryption," *IACR Transactions on Cryptographic Hardware and Embedded Systems*, vol. 2024, no. 2, pp. 451–480, 2024.

[30] AMD, Inc., "Alveo U280 Data Center Accelerator Card Data Sheet," https://docs.amd.com/r/en-US/ds963-u280/Summary, 2023, dS963.